Claim Rejections - 35 USC § 103

33. The following is a quotation of 35 U.S.C. 103(a) which forms the basis for all obviousness rejections set forth in this Office action:

> (a) A patent may not be obtained though the invention is not identically disclosed or described as set forth in section 102 of this title, if the differences between the subject matter sought to be patented and the prior art are such that the subject matter as a whole would have been obvious at the time the invention was made to a person having ordinary skill in the art to which said subject matter pertains. Patentability shall not be negatived by the manner in which the invention was made.

34. The factual inquiries set forth in *Graham* **v.** *John Deere Co.*, 383 U.S. 1, 148 USPQ 459 (1966), that are applied for establishing a background for determining obviousness under 35 U.S.C. 103(a) are summarized as follows:

1. Determining the scope and contents of the prior art.
2. Ascertaining the differences between the prior art and the claims at issue.
3. Resolving the level of ordinary skill in the pertinent art.
4. Considering objective evidence present in the application indicating obviousness or nonobviousness.

35. Claims 5, 11, 13, 15, 16 and 24 are rejected under 35 U.S.C. 103(a) as being unpatentable over Alcorn (US005745118A) in view of Furtner (US006778177B1).

36. With regard to Claim 5, Alcorn is relied upon for the teachings as discussed above relative to Claim 4.

However, Alcorn does teach that each of the at least two graphics pipelines further includes a scan converter. However, Furtner describes a parallel scan converter (16, Figure 23) that has a plurality of outputs for supplying data to a plurality of pixel pipelines (20, Col. 2, lines 3-16). Therefore, each of the at least two graphics pipelines further includes a scan converter.

The scan converter determines which clusters of pixel data are to be processed by which pipeline (Col. 11, lines 41-59; Col. 13, lines 54-63). Therefore, the scan converter is coupled to the back end circuitry (20), operative to determine the portion of the pixel data to be processed by the back end circuitry.

It would have been obvious to one of ordinary skill in this art at the time of invention by applicant to modify the device of Alcorn so that each of the at least two graphics pipelines further includes a scan converter as suggested by Furtner. Scan converting is the most popular method of drawing polygons because it uses only integer maths, takes up very little memory, and is simple to understand. The advantages of scan converters are well-known in the art and can be found in many publications, such as Elias' website. Furtner suggests that it is advantageous to have a parallel scan converter for two graphics pipelines because the scan conversion can be performed in parallel (Col. 17, lines 7-22), which increases the speed of processing.

37. With regard to Claim 11, Alcorn describes that the first of the at least two graphics pipelines (Col. 6, lines 33-35, 40-43) further includes circuitry (64, 72, Figure 3), coupled to the front end circuitry (60) and the back end circuitry (76), operative to provide position coordinates of the pixels within the first set of tiles (Col. 10, lines 1-8; *texel data output from the parameter interpolator circuit 64 is provided to the tiler 72, which determines the address of the four texels...checks to determine whether each is within the boundary of the texture...texel data includes the interpolated S, T coordinates as well as the map number*, Col. 11, lines 8-31) to be processed by the back end circuitry (*S, T coordinates for each display screen pixel are provided from the parameter interpolators, through the tiler, to texel interpolator 76*, Col. 12, lines 13-

20), the circuitry including a pixel identification line for receiving tile identification data indicating which of the set of tiles is to be processed by the back end circuitry (*texel data includes the interpolated S, T coordinates as well as the map number*, Col. 11, lines 15-17).

However, Alcorn does not teach a scan converter. However, Furtner describes a parallel scan converter (16, Figure 23) that has a plurality of outputs for supplying data to a plurality of pixel pipelines (20, Col. 2, lines 3-16). Therefore, the first of the at least two graphics pipelines further includes a scan converter. The scan converter receives, at its input, data which to write onto the graphic primitive to be processed (Col. 1, lines 58-62), and this data inherently comes from a front end circuitry. The output of the scan converter is connected to the pipelines (20, Col. 1, lines 62-66). Therefore, the scan converter is coupled to the front end circuitry and the back end circuitry (20). The scan converter determines which clusters of pixel data are to be processed by which pipeline (Col. 11, lines 41-59; Col. 13, lines 54-63). The scan converter has knowledge with regard to mapping the screen areas onto the memory address area (tiling) (Col. 6, lines 60-65). Therefore, the scan converter is inherently operative to provide memory addresses or position coordinates of the pixels within the first set of tiles to be processed by the back end circuitry, the scan converter inherently including a pixel identification line for receiving tile identification data indicating which of the set of tiles is to be processed by the back end circuitry. This would be obvious for the same reasons given in the rejection for Claim 5.

38. With regard to Claim 13, Claim 13 is similar in scope to Claim 11, and therefore is rejected under the same rationale.

39. With regard to Claim 15, Claim 15 is similar in scope to Claim 11, and therefore is rejected under the same rationale.

40. With regard to Claim 16, Claim 16 is similar in scope to Claim 11, and therefore is rejected under the same rationale.

41. With regard to Claim 24, Alcorn describes a graphics processing circuit (Col. 3, lines 58-63), comprising front end circuitry (32A, 32B, 32C, Figure 2) operative to generate pixel data in response to primitive data for a primitive to be rendered (*distributes 3-D primitive data evenly among the 3-D geometry accelerator chips*, Col. 6, lines 43-47; *each 3-D geometry accelerator chip processes primitive data*, Col. 6, lines 56-62; *rendering hardware interpolates the primitive data to compute the display screen pixels that are turned on to represent each primitive*, Col. 1, lines 31-33); first back end circuitry (12), coupled to the front end circuitry (Col. 7, lines 5-10), operative to receive and process a portion of the pixel data (Col. 12, lines 13-20) in response to position coordinates (Col. 12, lines 13-20); circuitry (64, 72), coupled between the front end circuitry and the first back end circuitry (76), operative to determine which set of tiles (tiler, Col. 11, lines 8-31) of a repeating tile pattern (Col. 11, lines 35-50) are to be processed by the first back end circuitry (Col. 11, lines 8-31). Alcorn discloses that the repeating tile pattern includes a horizontally and vertically repeating pattern of square regions (Col. 15, lines 44-57), as shown in Figure 6. Alcorn describes providing the position coordinates to the first back end circuitry in response to the pixel data (Col. 12, lines 13-20). Alcorn describes a memory controller (50,

Figure 2), coupled to the at least two graphics pipelines, operative to transmit and receive the processed pixel data (Col. 6, lines 33-35, 40-43; Col. 8, lines 27-40).

However, Alcorn does not teach two back end circuitries and two scan converters. However, Furtner describes a first scan converter (16, Figure 23). The first scan converter receives, at its input, data which to write onto the graphic primitive to be processed (Col. 1, lines 58-62), and this data inherently comes from a front end circuitry. The output of the scan converter is connected to the pipelines (20, Col. 1, lines 62-66). Therefore, the first scan converter is coupled to the front end circuitry and the first back end circuitry (20). The scan converter determines which clusters of pixel data are to be processed by which pipeline (Col. 11, lines 41-59; Col. 13, lines 54-63). Therefore, the first scan converter is operative to determine which set of tiles are to be processed by the first back end circuitry. The scan converter has knowledge with regard to mapping the screen areas onto the memory address area (tiling) (Col. 6, lines 60-65). Therefore, the first scan converter is inherently operative to provide the memory address area or position coordinates to the first back end circuitry in response to the pixel data. Furtner describes multiple pipelines, and each pipeline processes a cluster of pixel data (Col. 11, lines 41-59; Col. 13, lines 54-63). The scan converter is a parallel scan converter, and provides data to the multiple pipelines in parallel (Col. 2, lines 3-16), so the scan converter is considered to be similar to two scan converters, and the second scan converter performs in a similar manner as the first scan converter for the second back end circuitry. This would be obvious for the same reasons given in the rejection for Claim 5.

Allowable Subject Matter

42. Claim 19 is objected to as being dependent upon a rejected base claim, but would be allowable if rewritten in independent form including all of the limitations of the base claim and any intervening claims.

The following is a statement of reasons for the indication of allowable subject matter:

43. The prior art singly or in combination do not teach or suggest that each separate chip creates a bounding box around the polygon and wherein each corner of the bounding box is checked against a super tile that belongs to each separate chip and wherein if the bounding box does not overlap any of the super tiles associated with a separate chip, then the processing circuit rejects the whole polygon and processes a next one, as recited in Claim 19.

44. The closest prior art (Kent) teaches calculating the bounding box of the primitive and testing this against the VisRect. If the bounding box of the primitive is contained in the other P10's super tile the primitive is discarded at this stage [0129]. The method used is to calculate the distance from each subpixel sample point in the point's bounding box to the point's center and compare this to the point's radius. Subpixel sample points with a distance greater than the radius do not contribute to a pixel's coverage. The cost of this is kept low by only allowing small radius points hence the distance calculation is a small multiply and by taking a cycle per subpixel sample per pixel within the bounding box [0144]. However, Kent does not teach that each separate chip creates a bounding box around the polygon and wherein each corner of the bounding box is checked against a super tile that belongs to each separate chip and wherein if the

bounding box does not overlap any of the super tiles associated with a separate chip, then the processing circuit rejects the whole polygon and processes a next one.

Prior Art of Record

The prior art made of record and not relied upon is considered pertinent to applicant's disclosure.

1. US 20030164830A1 teaches a graphics pipeline [0006] that calculates the bounding box of the primitive in a super tile [0129].

2. Elias, Hugo. "Polygon Scan Converting." http://freespace.virgin.net/hugo.elias/graphics/x_polysc.htm.

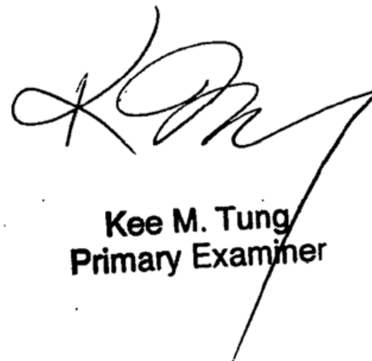
Conclusion

Any inquiry concerning this communication or earlier communications from the examiner should be directed to Joni Hsu whose telephone number is 571-272-7785. The examiner can normally be reached on M-F 8am-5pm.

If attempts to reach the examiner by telephone are unsuccessful, the examiner's supervisor, Ulka Chauhan can be reached on 571-272-7782. The fax phone number for the organization where this application or proceeding is assigned is 571-273-8300.

Information regarding the status of an application may be obtained from the Patent Application Information Retrieval (PAIR) system. Status information for published applications may be obtained from either Private PAIR or Public PAIR. Status information for unpublished applications is available through Private PAIR only. For more information about the PAIR system, see http://pair-direct.uspto.gov. Should you have questions on access to the Private PAIR system, contact the Electronic Business Center (EBC) at 866-217-9197 (toll-free).

JH



Kee M. Tung
Primary Examiner

Notice of References Cited	Application/Control No. 10/459,797	Applicant(s)/Patent Under Reexamination LEATHER ET AL.	
	Examiner Joni Hsu	Art Unit 2671	Page 1 of 1

U.S. PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Name	Classification
	A	US-5,745,118	04-1998	Alcorn et al.	345/587
	B	US-6,778,177	08-2004	Furtner, Wolfgang	345/544
	C	US-2003/0164830	09-2003	Kent, Osman	345/505
	D	US-			
	E	US-			
	F	US-			
	G	US-			
	H	US-			
	I	US-			
	J	US-			
	K	US-			
	L	US-			
	M	US-			

FOREIGN PATENT DOCUMENTS

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Country	Name	Classification
	N					
	O					
	P					
	Q					
	R					
	S					
	T					

NON-PATENT DOCUMENTS

*		Include as applicable: Author, Title Date, Publisher, Edition or Volume, Pertinent Pages)
	U	Elias, Hugo. "Polygon Scan Converting." http://freespace.virgin.net/hugo.elias/graphics/x_polysc.htm.
	V	
	W	
	X	

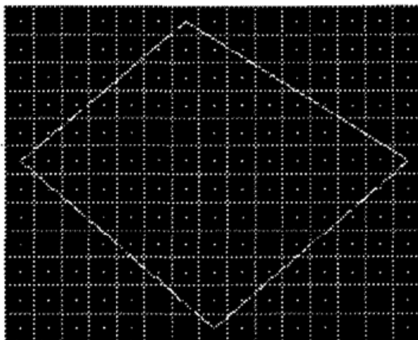
*A copy of this reference is not being furnished with this Office action. (See MPEP § 707.05(a).)
Dates in MM-YYYY format are publication dates. Classifications may be US or foreign.

Polygon Scan Converting

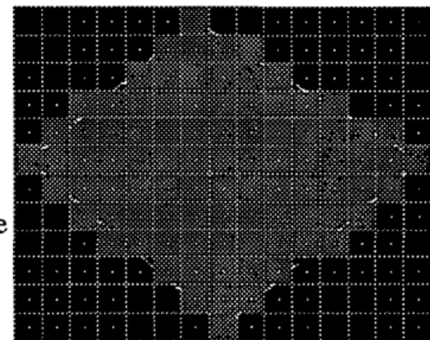
There are many ways to draw polygons. All have their uses. Some are fast, others very slow. The most popular method, used in practically every game, rendering engine, and graphics package which handles polygons, is known as scan converting.

This method uses only integer maths, takes up very little memory, and is simple to understand. The algorithm can be adapted to handle flat or gourad shaded, textured and bump mapped polygons.

It is an approximate method. You will never be able to draw a totally perfect polygon with smooth edges on a normal screen, because of the fact that the picture is divided up into pixels. It is possible to make the edges look better, but the edges will nevertheless look jaggy.



On a pixelated screen, a small polygon like this will end up with nasty edges when viewed close up.

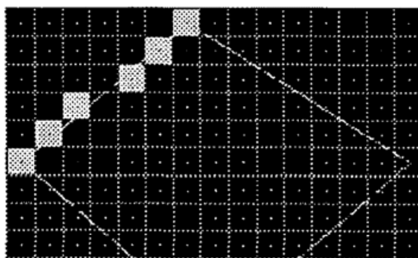


The method works by taking the polygon a line at a time, processing all the edges, then filling in the surface. If you haven't already, take a look at the page about drawing lines. You will find this very helpful.

A single scan conversion is the process of converting a polygon edge into data which can be used by the polygon filling routine. The process is essentially a single case of the line drawing algorithm. A polygon edge is calculated as if it were a line, but the line is not drawn to the screen. Instead the information is saved in a buffer for use later.
Rather than having a different routine to handle nearly horizontal or nearly vertical lines, all edges are handled as nearly vertical.

So the line algorithm travels down an edge, calculating the X-coordinate of the pixel which lies closest to the line for each Y-coordinate.

Having calculated the x-coordinates for every edge of the polygon, the next step is to loop through the Y-coordinates spanned by the top and bottom of the polygon, and draw lines between pairs of X-coordinates.

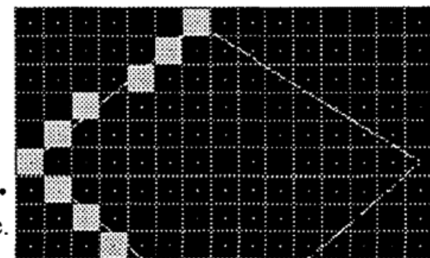


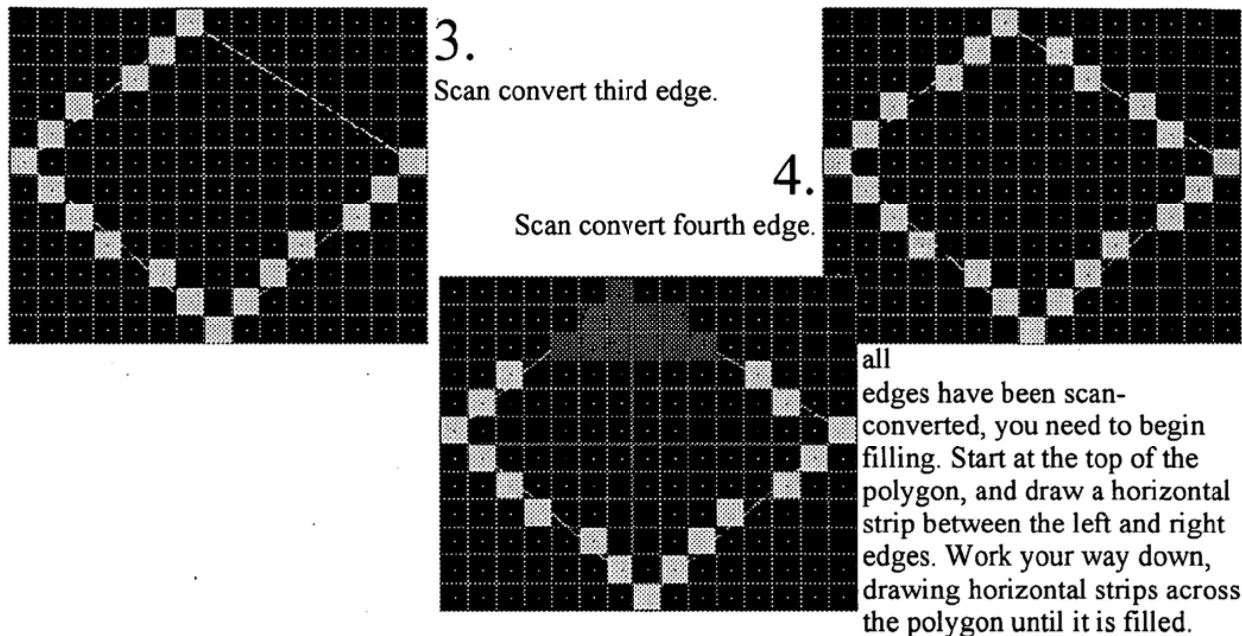
1.

Scan convert first edge.

2.

Scan convert second edge.




3.
Scan convert third edge.

4.
Scan convert fourth edge.

all
edges have been scan-converted, you need to begin filling. Start at the top of the polygon, and draw a horizontal strip between the left and right edges. Work your way down, drawing horizontal strips across the polygon until it is filled.

One filled polygon.

Now that you know the basic idea behind it, there are many things to consider.

how do you store the x values?

For convex polygons, there is a very quick and easy way to handle this. Create two arrays of integers, big enough to store an x value for each scanline of the screen. Call them Left and Right.
For example, for a 320x200 screen:

```
Left(0 to 199)  : integer
Right(0 to 199) : integer
```

Now if you always list the points of the polygon in anti-clockwise order. Then you can easily determine which lines make up the left and the right edges of the polygon. Lines who's first point is above the second make up the left edges. Lines who's first point is below the second make up the right edges. Lines who's points lie at the same y coordinate can be ignored. Store the X values in either the Left or Right arrays accordingly. Then, when you come to fill the polygon, the x coordinates are already there in the right order.

which points to fill?

If you use a simple line drawing algorithm to calculate the x-coordinates, you will find that many of the pixels drawn will actually lie slightly outside the boundaries of the polygon. This means that where polygons share the same edge, some pixels will be drawn twice. Now, this may or may not be a bad thing. It depends on how perfectly you need your polygons to be drawn. In many cases, if the polygons are flat shaded, people will never notice the fact. However, you may have transparent polygons, in which case you will get funny looking pixels at the boundaries between polygons, where the surface appears to be double thickness. It can be fatal however. If the edge of a texture mapped polygon lies very close to it's vanishing point at an oblique angle, a pixel outside the polygon may just lie past the horizon. In many perspective correct texture mapping routines, this could cause a divide by zero error.

These should be avoided at all cost.

In these cases, it is essential to write a scan-converter which guarantees that all pixels lie within the polygon's boundaries. This is, of course, easier said than done. What about pixels that lie exactly on a polygon boundary?

I now present what I believe to be a perfect scan-converting routine. It allows you to specify the vertices of the polygon to non-integer coordinates on the screen. This makes the polygon move a lot more fluidly. It also draws pixels which lie inside the edges of the polygon.

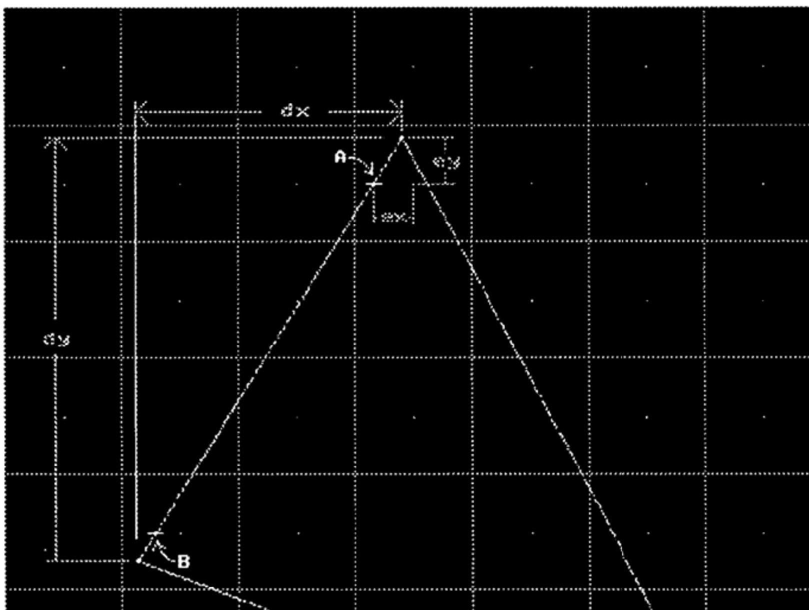
Perfect scan converting

OK, so this routine is going to use non-integer maths. That does not mean you will need slow floating point code. This can all be done with resonably fast fixed point maths, which can be handled extremely fast in assembler. Even faster, dare I say it, than the integer code I gave for drawing lines. If you don't know about fixed point maths, then you'll have to either find out for yourself, wait till I write a document on it, or just use floating point code for now.

Perfectly scanned polygons move much more smoothly than those calculated with integer maths, and so are more pleasing to the eye. Take a look at Quake, then Tomb Raider or Syndicate Wars. You will see that the cheap polygons in Tomb Raider move in a rather jittery way, making the scenery look like it's held together with selotape. Quake's smoothly rendered polygons on the other hand give the architecture a more solid feel.

There is a demo available showing the difference between integer and Fixed Point polygons. It requires DOS/4GW to run.

So, lets take a really close look at the edges of a polygon:



OK, this may get complicated, and involve a little maths, but the results are excelent.

Take a close look at the line that is to be scan converted, the yellow one. The two points that it is being drawn between (white dots (x1,y1) & (x2,y2)) do not lie exactly at the center of any pixel, (green dots). However, when the polygon comes to be rendered, it must be drawn using horizontal strips that are drawn between integer coordinates.

The way to handle this is calculate the X intersection of the line with

horizontal scanlines. Then save the X coordinate of the nearest pixel *inside* the polygon.

Firstly some variables we'll need.

```
NonIntegers:     gradient
                 ex, ey
                 x1, y1, x2, y2
                 Ax, Ay, Bx, By

Integers:        height

the function int() returns the integer part  of a number. i.e. int(5.7) = 5
```

OK, so lets break down the steps:

1. Calculate the gradient of the line:

```
dx = x2 - x1
dy = y2 - y1
gradient = dx/dy
```

2. Calculate ey:

```
ey = int(y1+1) - y1
```

3. Calculate ex:

```
ex = gradient*ey
```

4. Calculate coordinates of A:

```
Ax = x1 + ex
Ay = int(y1+1)
```

5. Calculate y coordinate of B:

```
By = int(y2)
```

You will notice that there is a divide in the calculation. Risk of a divide by zero. If dy is equal to zero, then you can simply ignore the entire line.

Right, now you have calculated all those things, the line can be scan converted. This scan converting process is actually faster than the one used for integer polygons (if you're using fixed point maths, otherwise it's slower). There are no IF's and JUMP's involved. The loop can be unrolled nicely to process at increadible speed.

So the inner loop looks something like this:

```
x = Ax

loop y from Ay to By
        YBuffer(y) = x
```

```
        x=x+gradient
end of loop
```

Again, you will have to handle Left hand lines and Right hand lines differently. The case given here is for a Left hand line. I will leave it up to you to work out how to handle the other side.

Scan converting in Assembler

Here is an example of a scan converter I wrote in Assembler. It works on 32-bit Fixed Point numbers only.

It takes as arguments; The initial x value (x), The gradient (DeltaX), the number of scan lines to calculate (length), and a pointer to the first element in the Y Buffer.

```
EAX = x
EBX = DeltaX
ECX = length
EDI = pointer to YBuffer[Y]

top:
        mov     [edi], eax              ; YBuffer[y] = x
        add     eax, ebx                ; x = x + DeltaX
        add     edi, 4                  ; y = y + 1
        dec     ecx
        jnz     top:                    ;end of loop
```

This can be unrolled and can calculate polygon edges very fast indeed. This version has been unrolled four times

```
EAX = x
EBX = DeltaX
ECX = length
EDI = pointer to YBuffer[Y]

        shr     ecx                     ; halve the number of loops
        jnc     NoSingle                ; if there are an even number of lines to do
                                        ; then don't do this odd one
        mov     [edi], eax              ; YBuffer[y] = x
        cmp     ecx, 0                  ; are there any more left?
        je      NoMore                  ; if not, then exit
        add     eax, ebx                ; x = x + 1
        add     edi, 4                  ; y = y + 1
NoSingle:
        shr     ecx                     ; halve the number of loops again
        jnc     NoSingle                ; if the number of lines is a multiple of 4
                                        ; then don't do these odd two
        mov     [edi], eax              ; YBuffer[y] = x
        add     eax, ebx                ; x = x + DeltaX
        mov     [edi+4], eax            ; YBuffer[y+1] = x
        add     eax, ebx                ; x = x + DeltaX

        cmp     ecx, 0                  ; are there any more left?
        je      NoMore                  ; if not, then exit
        add     edi, 8                  ; y = y + 2
NoDouble:
top:
```

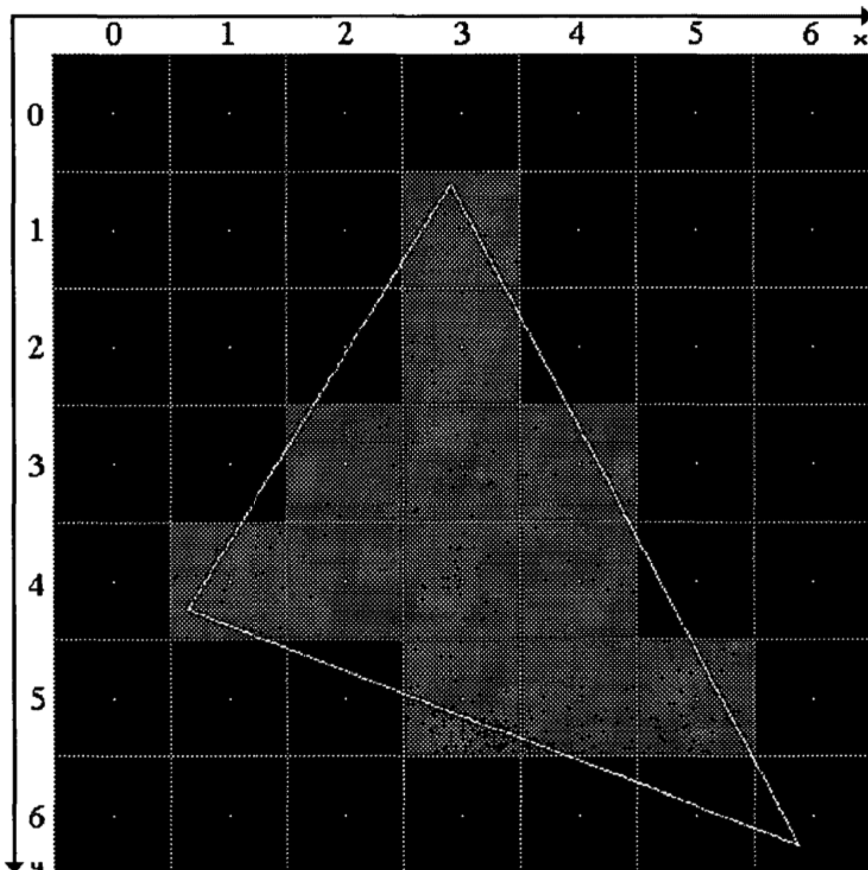
```
mov     [edi], eax          ; YBuffer[y] = x
add     eax, ebx            ; x = x + DeltaX
mov     [edi+4], eax        ; YBuffer[y+1] = x
add     eax, ebx            ; x = x + DeltaX
mov     [edi+8], eax        ; YBuffer[y+2] = x
add     eax, ebx            ; x = x + DeltaX
mov     [edi+12], eax       ; YBuffer[y+3] = x
add     eax, ebx            ; x = x + DeltaX
add     edi, 16             ; y = y + 4

dec     ecx                 ;
jnz     top                 ; end of loop
```

So, after all the edges of the polygon have been scan converted, you have an array of pairs of X coordinates where the edges cross horizontal scanlines. Assuming you are going to fill the area with a solid colour, you should loop down the height of the polygon, drawing horizontal strips from one side to the other. Remember that you only want to draw pixels that lie *inside* the polygon. So draw from the first pixel to the right of the left edge, to the first pixel to the left of the right edge. geddit?

Take a look at the polygon again, this time filled. The centres of the filled pixels all lie within the polygon. The X coordinates stored in the YBuffer would be:

```
y  x1  x2
---------
0: (-, -)
1: (3, 3)
2: (3, 3)
3: (2, 4)
4: (1, 4)
5: (3, 5)
6: (6, 5)
```



You will notice that on the last line, 6, X1 is larger then X2. This is because the polygon crosses the line, but pokes between pixels. This strip does not get drawn.

I hope I have convinced you to only ever write perfect scan converters from now on. There shouldn't really be any excuse for using tacky integer polys any more. Computers are quite fast enough to cope with the tiny extra computing overhead involved, and you as a programmer, I have no doubt, are more then capable of writing the code.

This Page is Inserted by IFW Indexing and Scanning Operations and is not part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☐ BLACK BORDERS
- ☐ IMAGE CUT OFF AT TOP, BOTTOM OR SIDES
- ☐ FADED TEXT OR DRAWING
- ☐ BLURRED OR ILLEGIBLE TEXT OR DRAWING
- ☐ SKEWED/SLANTED IMAGES
- ☑ COLOR OR BLACK AND WHITE PHOTOGRAPHS
- ☐ GRAY SCALE DOCUMENTS
- ☐ LINES OR MARKS ON ORIGINAL DOCUMENT
- ☐ REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY
- ☐ OTHER: ______________________

IMAGES ARE BEST AVAILABLE COPY.
As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.

PTO/SB/22 (12-04)
Approved for use through 07/31/2006. OMB 0651-0031
U.S. Patent and Trademark Office; U.S. DEPARMENT OF COMMERCE
Under the paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless if displays a valid OMB control number.

PETITION FOR EXTENSION OF TIME UNDER 37 CFR 1.136(a) **FY 2005** *(Fees pursuant to the Consolidated Appropriations Act, 2005 (H.R 4818).)*	Docket Number (Optional) 00100.02.0053
Application Number 10/459,797	Filed June 12, 2003
For DIVIDING WORK AMONG MULTIPLE GRAPHICS PIPELINES USING A SUPER-TILING TECHNIQUE	
Art Unit 2676	Examiner Joni Hsu

This is a request under the provisions of 37 CFR 1.136(a) to extend the period for filing a reply in the above identified application.

The requested extension and fee are as follows (check time period desired and enter the appropriate fee below):

		Fee	Small Entity Fee	
☐	One month (37 CFR 1.17(a)(1))	$120	$60	$
☑	Two months (37 CFR 1.17(a)(2))	$450	$225	$ 450.00
☐	Three months (37 CFR 1.17(a)(3))	$1020	$510	$
☐	Four months (37 CFR 1.17(a)(4))	$1590	$795	$
☐	Five months (37 CFR 1.17(a)(5))	$2160	$1080	$

☐ Applicant claims small entity status. See 37 CFR 1.27.

☐ A check in the amount of the fee is enclosed.

☐ Payment by credit card. Form PTO-2038 is attached.

☐ The Director has already been authorized to charge fees in this application to a Deposit Account.

☑ The Director is hereby authorized to charge any fees which may be required, or credit any overpayment, to Deposit Account Number 22-0259. I have enclosed a duplicate copy of this sheet.

WARNING: Information on this form may become public. Credit card information should not be included on this form. Provide credit card information and authorization on PTO-2038.

01/06/2006 SHASSEN1 00000034 220259 10459797
01 FC:1252 450.00 DA

I am the ☐ applicant/inventor.

☐ assignee of record of the entire interest. See 37 CFR 3.71. Statement under 37 CFR 3.73(b) is enclosed (Form PTO/SB/96).

☑ attorney or agent of record. Registration Number 34,414

☐ attorney or agent under 37 CFR 1.34. Registration number if acting under 37 CFR 1.34 ______

[signature] Signature — January 3, 2006 Date

Christopher J. Reckamp — Typed or printed name

312-609-7599 — Telephone Number

NOTE: Signatures of all the inventors or assignees of record of the entire interest or their representative(s) are required. Submit multiple forms if more than one signature is required, see below.

☑ Total of 1 forms are submitted.

This collection of information is required by 37 CFR 1.136(a). The information is required to obtain or retain a benefit by the public which is to file (and by the USPTO to process) an application. Confidentiality is governed by 35 U.S.C. 122 and 37 CFR 1.11 and 1.14. This collection is estimated to take 6 minutes to complete, including gathering, preparing, and submitting the completed application form to the USPTO. Time will vary depending upon the individual case. Any comments on the amount of time you require to complete this form and/or suggestions for reducing this burden, should be sent to the Chief Information Officer, U.S. Patent and Trademark Office, U.S. Department of Commerce, P.O. Box 1450, Alexandria, VA 22313-1450. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. **SEND TO: Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.**

If you need assistance in completing the form, call 1-800-PTO-9199 and select option 2.

REMARKS

Applicants respectfully traverse and request reconsideration.

Applicants with to thank the Examiner for the notice that claim 19 would be allowable if rewritten in independent form.

Claims 1-4, 6-8, 9, 10, 12, 14, 17, 18, 20-23, 25 and 26 stand rejected under 35 U.S.C. §102(b) as being anticipated by U.S. Patent No. 5,745,118 (Alcorn). The independent claims have been amended to include inherent language indicating that the tiles described in the specification and claimed correspond to screen locations and may have corresponding frame buffer memory locations as well. Alcorn is directed to different structure and operations from that claimed and instead is directed to texture space source data. Alcorn describes a 3D bypass structure for the download of textures and describes a system that receives primitive information from a host processor and passes it through a distributor 30 which then distributes 3D primitive data evenly among the 3D geometry accelerator chips. In this way, for example, three groups of primitives are operated upon simultaneously. The multiple 3D geometry accelerator chips determine object red, green and blue values and texture values for the screen space coordinates and they also perform view clipping operations. The output from these multiple 3D geometry accelerator chips are then passed to a concentrator chip 36 which combines the 3D primitive output data received from the 3D geometry accelerator chips and reorders the primitives to the original order that they had prior to being distributed by the distributor chip 30. (See for example, column 6, line 42 through column 7, line 10). As such, distribution of primitive data is done merely in a round robin type approach wherein each graphics accelerator chip receives an even distribution of primitives. The texture mapping board 12 then receives the primitives in the same order that the distributor receives them in and then processes them in that order.

In contrast, Applicants' claims are directed to a different operation – render space destination data. There is no teaching or suggestion in Alcorn of at least two graphics pipelines that process data in a corresponding set of tiles of a repeating tile pattern corresponding to screen locations, wherein the repeating tile pattern includes a horizontally and vertically repeating pattern of square regions.

It appears that the Alcorn reference actually teaches a type of round robin or sequential load balancing for texture source data in a front end. In contrast, Applicants describe, for example, a multi-pipeline system that performs pixel operations on pixels within a determined set of tiles by a corresponding one of a plurality of graphics pipelines based on a set of tiles of a repeating tile pattern corresponding to screen locations. In one embodiment, a scan converter determines, for example, whether pixels within portions of an object, such as a triangle, intersect with tiles that backend circuitry is responsible for processing. No tile based load distribution appears to be taught or suggested in the cited reference. Accordingly, the claims are believed to be in condition for allowance.

For example, the office action cites Alcorn, column 6, lines 40-43 as allegedly teaching a plurality of graphics pipelines. This portion refers to the multiple accelerator chips 32a-32c, for example. The office action then cites to column 11, lines 8-31 as teaching the processing of corresponding sets of tiles. However, this cited portion actually refers to the texture mapping board which is not part of the graphics accelerator chips. In fact, the graphics pipelines (i.e. the graphics accelerator chips) merely process data in a round robin fashion and do not process data based on tiles of a repeating tile pattern. Accordingly, the independent claims are in condition for allowance.

The office action also cites to Alcorn at column 15, lines 44-57. However again, this portion refers to the texture mapping board 12 which again processes data in the order in which

the distributor 30 received them. The portion referred to in the office action actually refers to the storage of texels in a MIP map so that the tiler 72 in the texture mapping board can access texels in the texel cache access 82. There is no teaching or suggestion that any texture tiles correspond to screen locations nor a plurality of pipelines that process data in a corresponding set of tiles of a repeating tile pattern corresponding to screen locations wherein the pattern includes a horizontally and vertically repeating pattern of the square regions. Accordingly, claims 1, 20, 24 and 25 are in condition for allowance.

The dependent claims add additional novel and non-obvious subject matter. For example, claim 3 requires that the square regions are 2-dimensional partition of memory in a frame buffer. However, the cited portion of Alcorn actually indicates that the texture map which actually comes from the texture cache 48 and not the frame buffer VRAMs, is combined in the frame buffer board to generate the final RGB values for each display screen pixel.

Also for example, with respect to claim 14, again the office action cites the 3D geometry accelerator chips of Alcorn as the claimed graphics pipelines. However, these 3D geometry accelerators do not process pixel data in the set of tiles in a repeating tile pattern as alleged in the office action. As noted above, the texture mapping board obtains texels for texture mapping and this board is not part of the front end board 10. Accordingly, the claim is in condition for allowance.

Claims 5, 11, 13, 15, 16 and 24 stand rejected under 35 U.S.C. §103(a) as being unpatentable over Alcorn in view of U.S. Patent No. 6,778,177 (Furtner). Applicants respectfully reassert the relevant remarks made above and as such, these claims are also in condition for allowance.

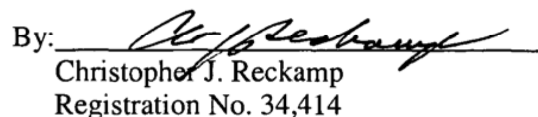
In addition, the Furtner reference is directed to a method for rasterizing a graphics component. Claim 5 requires that each of the graphics pipeline each include a scan converter.

However, the cited portion of Furtner merely describes a parallel scan converter not a scan converter for each of the pixel pipelines. Accordingly, the claim is in condition for allowance. As to claim 11, this claim requires, among other things, a scan converter for one of the graphics pipelines that provides position coordinates of pixels within the first set of tiles to be processed by the back end circuitry and that the scan converter includes a pixel identification line for receiving tile identification data indicating which of the set of tiles is to be processed by the back end circuitry. The office action cites column 10, lines 1-8 of Alcorn. However, the cited portion merely describes that there are texture map coordinates that are generated that correspond to the pixel. The cited portion actually refers to the back end circuit of Alcorn, namely the texture mapping card or board 12 (see FIG. 3). As such, the claim is in condition for allowance.

Applicants respectfully submit that the claims are in condition for allowance and respectfully request that a timely Notice of Allowance be issued in this case. The Examiner is invited to contact the below-listed attorney if the Examiner believes that a telephone conference will advance the prosecution of this application.

Respectfully submitted,

Date: 1/3/06

By: 
Christopher J. Reckamp
Registration No. 34,414

Vedder, Price, Kaufman & Kammholz, P.C.
222 N. LaSalle Street
Chicago, Illinois 60601
PHONE: (312) 609-7599
FAX: (312) 609-5005

Amendments to the Claims:

Re-write the claims as set forth below. This listing of claims will replace all prior versions and listings, of claims in the application:

Listing of Claims:

1. (currently amended) A graphics processing circuit, comprising:

at least two graphics pipelines operative to process data in a corresponding set of tiles of a repeating tile pattern corresponding to screen lccations, a respective one of the at least two graphics pipelines operative to process data in a dedicated tile,

wherein the repeating tile pattern includes a horizontally and vertically repeating pattern of square regions.

2. (original) The graphics processing circuit of claim 1, wherein the square regions comprise a two dimensional partitioning of memory.

3. (original) The graphics processing circuit of claim 2, wherein the memory is a frame buffer.

4. (original) The graphics processing circuit of claim 1, wherein each of the at least two graphics pipelines further includes front end circuitry operative to receive vertex data and generate pixel data corresponding to a primitive to be rendered, and back end circuitry, coupled to the front end circuitry, operative to receive and process a portion of the pixel data.

5. (original) The graphics processing circuit of claim 4, wherein each of the at least two graphics pipelines further includes a scan converter, coupled to the back end circuitry, operative to determine the portion of the pixel data to be processed by the back end circuitry.

6. (original) The graphics processing circuit of claim 1, wherein each tile of the set of tiles further comprises a 16x16 pixel array.

7. (original) The graphics processing circuit of claim 4, wherein the at least two graphics pipelines separately receive the pixel data from the front end circuitry.

8. (original) The graphics processing circuit of claim 4, wherein the at least two graphics pipelines are on multiple chips.

9. (previously presented) The graphics processing circuit of claim 1, further including a memory controller coupled to the at least two graphics pipelines, operative to transfer pixel data between each of a first pipeline and a second pipeline and a memory.

10. (original) The graphics processing circuit of claim 4, wherein a first of the at least two graphics pipelines processes the pixel data only in a first set of tiles in the repeating tile pattern.

11. (original) The graphics processing circuit of claim 10, wherein the first of the at least two graphics pipelines further includes a scan converter, coupled to the front end circuitry and the back end circuitry, operative to provide position coordinates of the pixels within the first

set of tiles to be processed by the back end circuitry, the scan converter including a pixel identification line for receiving tile identification data indicating which of the set of tiles is to be processed by the back end circuitry.

12. (previously presented) The graphics processing circuit of claim 1, wherein a second of the at least two graphics pipelines processes the data only in a second set of tiles in the repeating tile pattern.

13. (previously presented) The graphics processing circuit of claim 12, wherein the second of the at least two graphics pipelines further includes a scan converter, coupled to front end circuitry and back end circuitry, operative to provide position coordinates of the pixels within the second set of tiles to be processed by the back end circuitry, the scan converter including a pixel identification line for receiving tile identification data indicating which of the set of tiles is to be processed by the back end circuitry.

14. (original) The graphics processing circuit of claim 1 including a third graphics pipeline and a fourth graphics pipeline, wherein the third graphics pipeline includes front end circuitry operative to receive vertex data and generate pixel data corresponding to a primitive to be rendered, and back end circuitry, coupled to the front end circuitry, operative to receive and process the pixel data in a third set of tiles in the repeating tile pattern, and wherein the fourth graphics pipeline includes front end circuitry operative to receive vertex data and generate pixel data corresponding to a primitive to be rendered, and back end circuitry, coupled to the front end circuitry, operative to receive and process the pixel data in a fourth set of tiles in the repeating tile pattern.

15. (original) The graphics processing circuit of claim 14, wherein the third graphics pipeline further includes a scan converter, coupled to the front end circuitry and the back end circuitry, operative to provide position coordinates of the pixels within the third set of tiles to be processed by the back end circuitry, the scan converter including a pixel identification line for receiving tile identification data indicating which of the sets of tiles is to be processed by the back end circuitry.

16. (original) The graphics processing circuit of claim 14, wherein the fourth graphics pipeline further includes a scan converter, coupled to the front end circuitry and the back end circuitry, operative to provide position coordinates of the pixels within the fourth set of tiles to be processed by the back end circuitry, the scan converter including a pixel identification line for receiving tile identification data indicating which of the sets of tiles is to be processed by the back end circuitry.

17. (original) The graphics processing circuit of claim 14, wherein the third and fourth graphics pipelines are on separate chips.

18. (original) The graphics processing circuit of claim 14, further including a bridge operative to transmit vertex data to each of the first, second, third and fourth graphics pipelines.

19. (original) The graphics processing circuit of claim 17 wherein the data includes a polygon and wherein each separate chip creates a bounding box around the polygon and wherein each corner of the bounding box is checked against a super tile that belongs to each separate chip

and wherein if the bounding box does not overlap any of the super tiles associated with a separate chip, then the processing circuit rejects the whole polygon and processes a next one.

20. (currently amended) A graphics processing method, comprising:

receiving vertex data for a primitive to be rendered;

generating pixel data in response to the vertex data;

determining the pixels within a set of tiles of a repeating tile pattern corresponding to screen locations to be processed by a corresponding one of at least two graphics pipelines in response to the pixel data, the repeating tile pattern including a horizontally and vertically repeating pattern of square regions; and

performing pixel operations on the pixels within the determined set of tiles by the corresponding one of the at least two graphics pipelines.

21. (original) The graphics processing method of claim 20, wherein determining the pixels within a set of tiles of the repeating tile pattern to be processed further comprises determining the set of tiles that the corresponding graphics pipeline is responsible for.

22. (original) The graphics processing method of claim 20, wherein determining the pixels within a set of tiles of the repeating tile pattern to be processed further comprises providing position coordinates of the pixels within the determined set of tiles to be processed to the corresponding one of the at least two graphics pipelines.

23. (original) The graphics processing method of claim 20, further comprising transmitting the processed pixels to memory.

24. (currently amended) A graphics processing circuit, comprising:

front end circuitry operative to generate pixel data in response to primitive data for a primitive to be rendered;

first back end circuitry, coupled to the front end circuitry, operative to process a first portion of the pixel data in response to position coordinates;

a first scan converter, coupled between the front end circuitry and the first back end circuitry, operative to determine which set of tiles of a repeating tile pattern are to be processed by the first back end circuitry, the repeating tile pattern including a horizontally and vertically repeating pattern of square regions, and operative to provide the position coordinates to the first back end circuitry in response to the pixel data;

second back end circuitry, coupled to the front end circuitry, operative to process a second portion of the pixel data in response to position coordinates;

a second scan converter, coupled between the front end circuitry and the second back end circuitry, operative to determine which set of tiles of the repeating tile pattern are to be processed by the second back end circuitry, and operative to provide the position coordinates to the second back end circuitry in response to the pixel data; and

a memory controller, coupled to the first and second back end circuitry[[.]] operative to [[receive]] transmit and receive the processed pixel data.

25. (currently amended) A graphics processing circuit, comprising:

at least two graphics pipelines operative to process data in a corresponding set of tiles of a repeating tile pattern corresponding to screen locations, a respective one of the at least two

graphics pipelines operative to process data in a dedicated tile, wherein the repeating tile pattern includes a horizontally and vertically repeating pattern of regions.

26. (previously presented) The graphics processing circuit of claim 25 wherein the horizontally and vertically repeating pattern of regions include NxM number of pixels.

PATENT APPLICATION

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicants: Mark M. Leather et al.
Serial No.: 10/459,797
Filing Date: June 12, 2003
Confirmation No.: 4148

Examiner: Joni Hsu
Art Group: 2676
Docket No.: 00100.02.0053

Title: **DIVIDING WORK AMONG MULTIPLE GRAPHICS PIPELINES USING A SUPER-TILING TECHNIQUE**

Mail Stop Amendment
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Certificate of First Class Mailing
I hereby certify that this paper is being deposited with the United States Postal Service as first-class mail in an envelope addressed to Mail Stop Amendment, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.

1-3-06 — *Date*
Christine A. Wright

AMENDMENT AND RESPONSE

Dear Sir:

In response to the Office Action mailed August 1, 2005, Applicants submit the following Amendment and Response.

Amendments to the Claims are reflected in the listing of claims which begins on page 2 of this paper.

Remarks begin on page 9 of this paper.

CHICAGO/#1435290.1

IFW $2676

PTO/SB/21 (09-04)
Approved for use through 07/31/2006. OMB 0651-0031
U.S. Patent and Trademark Office; U.S. DEPARTMENT OF COMMERCE
Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

TRANSMITTAL FORM

(to be used for all correspondence after initial filing)

Application Number	10/459,797
Filing Date	June 12, 2003
First Named Inventor	Mark M. Leather
Art Unit	2676
Examiner Name	Joni Hsu
Attorney Docket Number	00100.02.0053
Total Number of Pages in This Submission	

ENCLOSURES *(Check all that apply)*

- [ ] Fee Transmittal Form
  - [ ] Fee Attached
- [x] Amendment/Reply
  - [ ] After Final
  - [ ] Affidavits/declaration(s)
- [x] Extension of Time Request
- [ ] Express Abandonment Request
- [ ] Information Disclosure Statement
- [ ] Certified Copy of Priority Document(s)
- [ ] Reply to Missing Parts/ Incomplete Application
  - [ ] Reply to Missing Parts under 37 CFR 1.52 or 1.53
- [ ] Drawing(s)
- [ ] Licensing-related Papers
- [ ] Petition
- [ ] Petition to Convert to a Provisional Application
- [ ] Power of Attorney, Revocation Change of Correspondence Address
- [ ] Terminal Disclaimer
- [ ] Request for Refund
- [ ] CD, Number of CD(s) ____________
  - [ ] Landscape Table on CD
- [ ] After Allowance Communication to TC
- [ ] Appeal Communication to Board of Appeals and Interferences
- [ ] Appeal Communication to TC **(Appeal Notice, Brief, Reply Brief)**
- [ ] Proprietary Information
- [ ] Status Letter
- [x] Other Enclosure(s) (please Identify below):

-return postcard

Remarks

SIGNATURE OF APPLICANT, ATTORNEY, OR AGENT

Firm Name	Vedder, Price, Kaufman & Kammholz, P.C.		
Signature	[signature]		
Printed name	Christopher J. Reckamp		
Date	January 3, 2006	Reg. No.	34,414

CERTIFICATE OF TRANSMISSION/MAILING

I hereby certify that this correspondence is being facsimile transmitted to the USPTO or deposited with the United States Postal Service with sufficient postage as first class mail in an envelope addressed to: Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on the date shown below:

Signature	[signature]		
Typed or printed name	Christine A. Wright	Date	January 3, 2006

This collection of information is required by 37 CFR 1.5. The information is required to obtain or retain a benefit by the public which is to file (and by the USPTO to process) an application. Confidentiality is governed by 35 U.S.C. 122 and 37 CFR 1.11 and1.14. This collection is estimated to 2 hours to complete, including gathering, preparing, and submitting the completed application form to the USPTO. Time will vary depending upon the individual case. Any comments on the amount of time you require to complete this form and/or suggestions for reducing this burden, should be sent to the Chief Information Officer, U.S. Patent and Trademark Office, U.S. Department of Commerce, P.O. Box 1450, Alexandria, VA 22313-1450. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. SEND TO: **Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.**

If you need assistance in completing the form, call 1-800-PTO-9199 and select option 2.

BEST AVAILABLE COPY

PATENT APPLICATION FEE DETERMINATION RECORD
Effective January 1, 2003

Application or Docket Number: 10459797

CLAIMS AS FILED - PART I

	(Column 1)	(Column 2)	SMALL ENTITY RATE	FEE	OR	OTHER THAN SMALL ENTITY RATE	FEE
TOTAL CLAIMS	24						
FOR	NUMBER FILED	NUMBER EXTRA	BASIC FEE	375.00	OR	BASIC FEE	750.00
TOTAL CHARGEABLE CLAIMS	24 minus 20=	* 4	X$ 9=		OR	X$18=	72
INDEPENDENT CLAIMS	3 minus 3 =	* 0	X42=		OR	X84=	
MULTIPLE DEPENDENT CLAIM PRESENT		☐	+140=		OR	+280=	
			TOTAL		OR	TOTAL	

* If the difference in column 1 is less than zero, enter "0" in column 2

SMALL ENTITY TYPE ☐

CLAIMS AS AMENDED - PART II

1-5-06

AMENDMENT A

	(Column 1) CLAIMS REMAINING AFTER AMENDMENT		(Column 2) HIGHEST NUMBER PREVIOUSLY PAID FOR	(Column 3) PRESENT EXTRA	SMALL ENTITY RATE	ADDITIONAL FEE	OR	OTHER THAN SMALL ENTITY RATE	ADDITIONAL FEE
Total	* 26	Minus	** 6	=	X$ 9=		OR	X$18=	
Independent	* 4	Minus	*** 7	=	X42=		OR	X84=	
FIRST PRESENTATION OF MULTIPLE DEPENDENT CLAIM				☐	+140=		OR	+280=	
					TOTAL ADDIT. FEE		OR	TOTAL ADDIT. FEE	

AMENDMENT B

	(Column 1) CLAIMS REMAINING AFTER AMENDMENT		(Column 2) HIGHEST NUMBER PREVIOUSLY PAID FOR	(Column 3) PRESENT EXTRA	SMALL ENTITY RATE	ADDITIONAL FEE	OR	OTHER THAN SMALL ENTITY RATE	ADDITIONAL FEE
Total	*	Minus	**	=	X$ 9=		OR	X$18=	
Independent	*	Minus	***	=	X42=		OR	X84=	
FIRST PRESENTATION OF MULTIPLE DEPENDENT CLAIM				☐	+140=		OR	+280=	
					TOTAL ADDIT. FEE		OR	TOTAL ADDIT. FEE	

AMENDMENT C

	(Column 1) CLAIMS REMAINING AFTER AMENDMENT		(Column 2) HIGHEST NUMBER PREVIOUSLY PAID FOR	(Column 3) PRESENT EXTRA	SMALL ENTITY RATE	ADDITIONAL FEE	OR	OTHER THAN SMALL ENTITY RATE	ADDITIONAL FEE
Total	*	Minus	**	=	X$ 9=		OR	X$18=	
Independent	*	Minus	***	=	X42=		OR	X84=	
FIRST PRESENTATION OF MULTIPLE DEPENDENT CLAIM				☐	+140=		OR	+280=	
					TOTAL ADDIT. FEE		OR	TOTAL ADDIT. FEE	

* If the entry in column 1 is less than the entry in column 2, write "0" in column 3.
** If the "Highest Number Previously Paid For" IN THIS SPACE is less than 20, enter "20."
*** If the "Highest Number Previously Paid For" IN THIS SPACE is less than 3, enter "3."
The "Highest Number Previously Paid For" (Total or Independent) is the highest number found in the appropriate box in column 1.

FORM PTO-875 (Rev. 12/02) Patent and Trademark Office, U.S. DEPARTMENT OF COMMERCE

EAST Search History

Ref #	Hits	Search Query	DBs	Default Operator	Plurals	Time Stamp
L1	355	345/506.ccls.	US-PGPUB; USPAT; USOCR; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2006/03/01 15:35
L2	273	345/530.ccls.	US-PGPUB; USPAT; USOCR; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2006/03/01 15:35
L3	294	345/505.ccls.	US-PGPUB; USPAT; USOCR; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2006/03/01 15:35
L4	40	345/588.ccls.	US-PGPUB; USPAT; USOCR; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2006/03/01 15:35
L5	92	345/544.ccls.	US-PGPUB; USPAT; USOCR; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2006/03/01 15:35
L6	478	345/545.ccls.	US-PGPUB; USPAT; USOCR; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2006/03/01 15:35
L7	72	345/532.ccls.	US-PGPUB; USPAT; USOCR; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2006/03/01 15:35
L8	740	345/501.ccls.	US-PGPUB; USPAT; USOCR; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2006/03/01 15:35

EAST Search History

L9	364	345/502.ccls.	US-PGPUB; USPAT; USOCR; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2006/03/01 15:35
L10	613	345/531.ccls.	US-PGPUB; USPAT; USOCR; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2006/03/01 15:35
L11	1322	til$3 and repeat$3 and pattern$3 and pixel$1 and pipelin$3	US-PGPUB; USPAT; USOCR; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2006/03/01 15:35
L12	1053	til$3 and repeat$3 and pattern$3 and pixel$1 and horizontal$2 and vertical$2 and pipelin$3	US-PGPUB; USPAT; USOCR; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2006/03/01 15:35
L13	115	til$3 and repeat$3 same (til$3 pattern$3) same pixel$1 and pattern$3 and horizontal$2 and vertical$2 and pipelin$3	US-PGPUB; USPAT; USOCR; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2006/03/01 15:35
L14	87	til$3 and repeat$3 with (til$3 pattern$3) same pixel$1 and pattern$3 and horizontal$2 and vertical$2 and pipelin$3	US-PGPUB; USPAT; USOCR; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2006/03/01 15:35
L15	226	scan adj conver$5 and pixel$1 and til$3 and pipelin$3	US-PGPUB; USPAT; USOCR; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2006/03/01 15:35
L16	303	(multiple plurality) same pipelin$3 and scan adj conver$5	US-PGPUB; USPAT; USOCR; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2006/03/01 15:35

EAST Search History

L17	81	(multiple plurality) near2 pipelin$3 and scan adj conver$5	US-PGPUB; USPAT; USOCR; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2006/03/01 15:35
L18	76	(multiple plurality) near2 pipelin$3 and scan adj conver$5 and pixel$1	US-PGPUB; USPAT; USOCR; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2006/03/01 15:35
L19	84	non adj square same til$3	US-PGPUB; USPAT; USOCR; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2006/03/01 15:35
L20	34	repeat$3 and til$3 and non adj square same til$3	US-PGPUB; USPAT; USOCR; EPO; JPO; DERWENT; IBM_TDB	OR	OFF	2006/03/01 15:35

Search Notes	Application/Control No. 10/459,797	Applicant(s)/Patent under Reexamination LEATHER ET AL.	
	Examiner Joni Hsu	Art Unit 2671	

SEARCHED			
Class	Subclass	Date	Examiner
345	506, 530, 505, 588, 544, 545	7/26/2005	JH
345	532, 501	7/26/2005	JH
345	502, 531	7/26/2005	JH
Above	UPDATED	3/1/2006	JH

INTERFERENCE SEARCHED			
Class	Subclass	Date	Examiner

SEARCH NOTES (INCLUDING SEARCH STRATEGY)		
	DATE	EXMR
See attached search history.	3/1/2006	JH

Index of Claims	Application/Control No.	Applicant(s)/Patent under Reexamination
	10/459,797	LEATHER ET AL.
	Examiner: Joni Hsu	Art Unit: 2671

√	Rejected	–	(Through numeral) Cancelled	N	Non-Elected	A	Appeal
=	Allowed	÷	Restricted	I	Interference	O	Objected

Claim		Date								
Final	Original	9/21/05								
	1	√								
	2	√								
	3	√								
	4	√								
	5	√								
	6	√								
	7	√								
	8	√								
	9	√								
	10	√								
	11	√								
	12	√								
	13	√								
	14	√								
	15	√								
	16	√								
	17	√								
	18	√								
	19	O								
	20	√								
	21	√								
	22	√								
	23	√								
	24	√								
	25	√								
	26	√								
	27									
	28									
	29									
	30									
	31									
	32									
	33									
	34									
	35									
	36									
	37									
	38									
	39									
	40									
	41									
	42									
	43									
	44									
	45									
	46									
	47									
	48									
	49									
	50									

Claim		Date								
Final	Original									
	51									
	52									
	53									
	54									
	55									
	56									
	57									
	58									
	59									
	60									
	61									
	62									
	63									
	64									
	65									
	66									
	67									
	68									
	69									
	70									
	71									
	72									
	73									
	74									
	75									
	76									
	77									
	78									
	79									
	80									
	81									
	82									
	83									
	84									
	85									
	86									
	87									
	88									
	89									
	90									
	91									
	92									
	93									
	94									
	95									
	96									
	97									
	98									
	99									
	100									

Claim		Date								
Final	Original									
	101									
	102									
	103									
	104									
	105									
	106									
	107									
	108									
	109									
	110									
	111									
	112									
	113									
	114									
	115									
	116									
	117									
	118									
	119									
	120									
	121									
	122									
	123									
	124									
	125									
	126									
	127									
	128									
	129									
	130									
	131									
	132									
	133									
	134									
	135									
	136									
	137									
	138									
	139									
	140									
	141									
	142									
	143									
	144									
	145									
	146									
	147									
	148									
	149									
	150									

UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
10/459,797	06/12/2003	Mark M. Leather	00100.02.0053	4148

29153 7590 03/13/2006

ATI TECHNOLOGIES, INC.
C/O VEDDER PRICE KAUFMAN & KAMMHOLZ, P.C.
222 N.LASALLE STREET
CHICAGO, IL 60601

EXAMINER	
HSU, JONI	
ART UNIT	PAPER NUMBER
2671	

DATE MAILED: 03/13/2006

Please find below and/or attached an Office communication concerning this application or proceeding.

PTO-90C (Rev. 10/03)

Office Action Summary	Application No.	Applicant(s)
	10/459,797	LEATHER ET AL.
	Examiner	Art Unit
	Joni Hsu	2671

-- The MAILING DATE of this communication appears on the cover sheet with the correspondence address --

Period for Reply

A SHORTENED STATUTORY PERIOD FOR REPLY IS SET TO EXPIRE 3 MONTH(S) OR THIRTY (30) DAYS, WHICHEVER IS LONGER, FROM THE MAILING DATE OF THIS COMMUNICATION.

- Extensions of time may be available under the provisions of 37 CFR 1.136(a). In no event, however, may a reply be timely filed after SIX (6) MONTHS from the mailing date of this communication.
- If NO period for reply is specified above, the maximum statutory period will apply and will expire SIX (6) MONTHS from the mailing date of this communication.
- Failure to reply within the set or extended period for reply will, by statute, cause the application to become ABANDONED (35 U.S.C. § 133). Any reply received by the Office later than three months after the mailing date of this communication, even if timely filed, may reduce any earned patent term adjustment. See 37 CFR 1.704(b).

Status

1)☐ Responsive to communication(s) filed on _____.

2a)☒ This action is **FINAL**. 2b)☐ This action is non-final.

3)☐ Since this application is in condition for allowance except for formal matters, prosecution as to the merits is closed in accordance with the practice under *Ex parte Quayle*, 1935 C.D. 11, 453 O.G. 213.

Disposition of Claims

4)☒ Claim(s) *1-26* is/are pending in the application.

4a) Of the above claim(s) _____ is/are withdrawn from consideration.

5)☐ Claim(s) _____ is/are allowed.

6)☒ Claim(s) *1-18 and 20-26* is/are rejected.

7)☒ Claim(s) *19* is/are objected to.

8)☐ Claim(s) _____ are subject to restriction and/or election requirement.

Application Papers

9)☒ The specification is objected to by the Examiner.

10)☐ The drawing(s) filed on _____ is/are: a)☐ accepted or b)☐ objected to by the Examiner.

Applicant may not request that any objection to the drawing(s) be held in abeyance. See 37 CFR 1.85(a).

Replacement drawing sheet(s) including the correction is required if the drawing(s) is objected to. See 37 CFR 1.121(d).

11)☐ The oath or declaration is objected to by the Examiner. Note the attached Office Action or form PTO-152.

Priority under 35 U.S.C. § 119

12)☐ Acknowledgment is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d) or (f).

a)☐ All b)☐ Some * c)☐ None of:

1.☐ Certified copies of the priority documents have been received.

2.☐ Certified copies of the priority documents have been received in Application No. _____.

3.☐ Copies of the certified copies of the priority documents have been received in this National Stage application from the International Bureau (PCT Rule 17.2(a)).

* See the attached detailed Office action for a list of the certified copies not received.

Attachment(s)

1) ☐ Notice of References Cited (PTO-892)
2) ☐ Notice of Draftsperson's Patent Drawing Review (PTO-948)
3) ☐ Information Disclosure Statement(s) (PTO-1449 or PTO/SB/08) Paper No(s)/Mail Date _____.
4) ☐ Interview Summary (PTO-413) Paper No(s)/Mail Date. _____.
5) ☐ Notice of Informal Patent Application (PTO-152)
6) ☐ Other: _____.

DETAILED ACTION

Response to Amendment

1. In light of Applicant's amendment to Claim 24, the objection to Claim 24 has been withdrawn.

2. Applicant's arguments with respect to claims 1-18 and 20-26 have been considered but are moot in view of the new ground(s) of rejection.

3. Applicant's arguments, see page 9, filed January 5, 2006, with respect to the rejection(s) of claim(s) 1-4, 6-10, 12, 14, 17, 18, 20-23, 25, and 26 under 35 U.S.C. 102(b) and claims 5, 11, 13, 15, 16, and 24 have been fully considered and are persuasive. Therefore, the rejection has been withdrawn. However, upon further consideration, a new ground(s) of rejection is made in view of Furtner (US006778177B1).

4. Applicant argues that Alcorn (US005745118A) is directed to texture space source data and not to tiles corresponding to screen locations (page 9).

In reply, the Examiner agrees. However, new grounds of rejection are made in view of Furtner.

5. Applicant's arguments filed January 5, 2006, with respect to Claim 5 have been fully considered but they are not persuasive.

6. With regard to Claim 5, Applicant argues that Furtner describes a parallel scan converter not a scan converter for each of the pixel pipelines (pages 11-12).

In reply, the Examiner disagrees. Furtner does teach a scan converter for each of the pixel pipelines (Col. 6, lines 47-51).

Specification

7. The disclosure is objected to because of the following informalities: Paragraph [0001] states that this application is a related application to a co-pending application, but does not provide the serial number for this co-pending application.

Appropriate correction is required.

Claim Rejections - 35 USC § 102

8. The following is a quotation of the appropriate paragraphs of 35 U.S.C. 102 that form the basis for the rejections under this section made in this Office action:

> A person shall be entitled to a patent unless –
>
> (e) the invention was described in (1) an application for patent, published under section 122(b), by another filed in the United States before the invention by the applicant for patent or (2) a patent granted on an application for patent by another filed in the United States before the invention by the applicant for patent, except that an international application filed under the treaty defined in section 351(a) shall have the effects for purposes of this subsection of an application filed in the United States only if the international application designated the United States and was published under Article 21(2) of such treaty in the English language.

9. Claims 1-17 and 20-26 are rejected under 35 U.S.C. 102(e) as being anticipated by Furtner (US006778177B1).

10. With regard to Claim 1, Furtner describes a graphics processing circuit, comprising at least two graphics pipelines (20, Figure 23; Col. 2, lines 11-14) operative to process data in a corresponding set of tiles of a repeating tile pattern corresponding to screen locations, a respective one of the at least two graphics pipelines operative to process data in a dedicated tile, wherein the repeating tile pattern includes a horizontally and vertically repeating pattern of square regions (Figure 21b, Col. 1, lines 40-49).

11. With regard to Claim 2, Furtner describes that the square regions comprise a two dimensional partitioning of memory (10, Figure 21b; Col. 1, lines 40-49).

12. With regard to Claim 3, Furtner describes that the memory is a frame buffer (10, Figure 21b; Col. 1, lines 40-49).

13. With regard to Claim 4, Furtner describes that each of the at least two graphics pipelines further includes front end circuitry (102, Figure 1) operative to receive vertex data and generate pixel data corresponding to a primitive to be rendered (Col. 8, lines 38-44) , and back end circuitry (108), coupled to the front end circuitry, operative to receive and process a portion of the pixel data (Col. 8, lines 51-60).

14. With regard to Claim 5, Furtner describes that each of the at least two graphics pipelines further includes a scan converter (16, Figure 23) (Col. 2, lines 3-16; Col. 6, lines 47-51). The scan converter determines which clusters of pixel data are to be processed by which pipeline (Col. 11, lines 41-59; Col. 13, lines 54-63). Therefore, the scan converter is coupled to the back end circuitry (20), operative to determine the portion of the pixel data to be processed by the back end circuitry.

15. With regard to Claim 6, Furtner describes that each tile of the set of tiles further comprises a 16x16 pixel array (Col. 11, lines 45-48, 64-65).

16. With regard to Claim 7, Furtner describes that the at least two graphics pipelines (108, Figure 1) separately receive the pixel data from the front end circuitry (102) (Col. 8, lines 51-57).

17. With regard to Claim 8, Furtner describes that the at least two graphics pipelines are on multiple chips (Col. 6, lines 47-51).

18. With regard to Claim 9, Furtner describes a memory controller (22, Figure 23) coupled to the at least two graphics pipelines (20), operative to transfer pixel data between each of a first pipeline and a second pipeline and a memory (24) (Col. 2, lines 20-34).

19. With regard to Claim 10, Furtner describes that a first of the at least two graphics pipeline processes the pixel data only in a first set of tiles in the repeating tile pattern (Figure 21b, Col. 1, lines 41-49).

20. With regard to Claim 11, Furtner describes that the first of the at least two graphics pipelines further includes a scan converter (16; Col. 2, lines 3-16). The scan converter receives, at its input, data which to write onto the graphic primitive to be processed (Col. 1, lines 58-62), and this data inherently comes from a front end circuitry. The output of the scan converter is connected to the pipelines (20, Col. 1, lines 62-66). Therefore, the scan converter is coupled to the front end circuitry and the back end circuitry (20). The scan converter determines which clusters of pixel data are to be processed by which pipeline (Col. 11, lines 41-59; Col. 13, lines 54-63). The scan converter has knowledge with regard to mapping the screen areas onto the memory address area (tiling) (Col. 6, lines 60-65). Therefore, the scan converter is inherently operative to provide memory addresses or position coordinates of the pixels within the first set of tiles to be processed by the back end circuitry, the scan converter inherently including a pixel identification line for receiving tile identification data indicating which of the set of tiles is to be processed by the back end circuitry.

21. With regard to Claim 12, Furtner describes that a second of the at least two graphics pipelines processes the data only in a second set of tiles in the repeating tile pattern (Figure 21b, Col. 1, lines 41-49).

22. With regard to Claim 13, Claim 13 is similar in scope to Claim 11, and therefore is rejected under the same rationale.

23. With regard to Claim 14, Claim 14 is similar to Claims 4 and 10, except that Claim 14 is for a third and fourth graphics pipeline. Furtner describes four graphics pipelines (Col. 1, lines 37-49). Therefore, Claim 14 is rejected under the same rationale as Claims 4 and 10.

24. With regard to Claim 15, Claim 15 is similar in scope to Claim 11, and therefore is rejected under the same rationale.

25. With regard to Claim 16, Claim 16 is similar in scope to Claim 11, and therefore is rejected under the same rationale.

26. With regard to Claim 17, Claim 17 is similar in scope to Claim 8, and therefore is rejected under the same rationale.

27. With regard to Claim 20, Furtner describes a graphics processing method, comprising receiving vertex data for a primitive to be rendered; generating pixel data in response to the vertex data (Col. 8, lines 38-44); determining the pixels within a set of tiles of a repeating tile pattern corresponding to screen locations (Figure 21b, Col. 1, lines 41-49) to be processed by a corresponding one of at least two graphics pipelines in response to the pixel data (Col. 11, lines 41-59; Col. 13, lines 54-63), the repeating tile pattern including a horizontally and vertically

repeating pattern of square regions; and performing pixel operations on the pixels within the determined set of tiles by the corresponding one of the at least two graphics pipelines (Figure 21b, Col. 1, lines 41-49).

28. With regard to Claim 21, Furtner describes determining the pixels within a set of tiles of the repeating tile pattern to be processed further comprises determining the set of tiles that the corresponding graphics pipeline is responsible for (Col. 11, lines 41-49; Col. 13, lines 54-63).

29. With regard to Claim 22, Furtner describes that the scan converter determines which clusters of pixel data are to be processed by which pipeline (Col. 11, lines 41-59; Col. 13, lines 54-63). The scan converter has knowledge with regard to mapping the screen areas onto the memory address area (tiling) (Col. 6, lines 60-65). Furtner discloses that determining the pixels within a set of tiles of the repeating tile pattern to be processed (Col. 1, lines 41-49) inherently further comprises providing position coordinates of the pixels within the determined set of tiles to be processed to the corresponding one of the at least two graphics pipelines.

30. With regard to Claim 23, Furtner describes transmitting the processed pixels to memory (24, Figure 23; Col. 2, lines 20-34).

31. With regard to Claim 24, Furtner describes a graphics processing circuit, comprising front end circuitry (102, Figure 1) operative to generate pixel data in response to primitive data for a primitive to be rendered (Col. 8, lines 38-44); first back end circuitry (108a), coupled to the

front end circuitry (Col. 8, lines 51-57), operative to process a first portion of the pixel data in response to position coordinates; a first scan converter (16, Figure 23). The first scan converter receives, at its input, data which to write onto the graphic primitive to be processed (Col. 1, lines 58-62), and this data inherently comes from a front end circuitry. The output of the scan converter is connected to the pipelines (20, Col. 1, lines 62-66). Therefore, the first scan converter is coupled to the front end circuitry and the first back end circuitry (20). The scan converter determines which clusters of pixel data are to be processed by which pipeline (Col. 11, lines 41-59; Col. 13, lines 54-63). Therefore, the first scan converter is operative to determine which set of tiles of a repeating tile pattern (Figure 21b, Col. 1, lines 41-49) are to be processed by the first back end circuitry. The scan converter has knowledge with regard to mapping the screen areas onto the memory address area (tiling) (Col. 6, lines 60-65). Therefore, the first scan converter is inherently operative to provide the memory address area or position coordinates to the first back end circuitry in response to the pixel data. Furtner describes multiple pipelines, and each pipeline processes a cluster of pixel data (Col. 11, lines 41-59; Col. 13, lines 54-63). The scan converter is a parallel scan converter, and provides data to the multiple pipelines in parallel (Col. 2, lines 3-16), so the scan converter is considered to be similar to two scan converters, and the second scan converter performs in a similar manner as the first scan converter for the second back end circuitry. Furtner also describes that each pipeline has a scan converter (Col. 6, lines 47-51). Furtner describes a memory controller (24, Figure 23), coupled to the first and second back end circuitry (20) operative to transmit and receive the processed pixel data (Col. 2, lines 30-34).

32. With regard to Claim 26, Furtner describes that the horizontally and vertically repeating pattern of regions (Figure 21b, Col. 1, lines 41-49) include NxM number of pixels (Col. 11, lines 45-55).

33. Thus, it reasonably appears that Furtner describes or discloses every element of Claims 1-17 and 20-26 and therefore anticipates the claims subject.

Claim Rejections - 35 USC § 103

34. The following is a quotation of 35 U.S.C. 103(a) which forms the basis for all obviousness rejections set forth in this Office action:

> (a) A patent may not be obtained though the invention is not identically disclosed or described as set forth in section 102 of this title, if the differences between the subject matter sought to be patented and the prior art are such that the subject matter as a whole would have been obvious at the time the invention was made to a person having ordinary skill in the art to which said subject matter pertains. Patentability shall not be negatived by the manner in which the invention was made.

35. The factual inquiries set forth in *Graham* **v.** *John Deere Co.*, 383 U.S. 1, 148 USPQ 459 (1966), that are applied for establishing a background for determining obviousness under 35 U.S.C. 103(a) are summarized as follows:

1. Determining the scope and contents of the prior art.
2. Ascertaining the differences between the prior art and the claims at issue.
3. Resolving the level of ordinary skill in the pertinent art.
4. Considering objective evidence present in the application indicating obviousness or nonobviousness.

36. Claim 18 is rejected under 35 U.S.C. 103(a) as being unpatentable over Furtner (US006778177B1) in view of Alcorn (US005745118A).

Furtner is relied upon for the teachings as discussed above relative to Claim 14.

However, Furtner does not teach a bridge operative to transmit vertex data to each of the first, second, third and fourth graphics pipelines. However, Alcorn describes a bridge (30, Figure 2) operative to transmit vertex data to each of the first, second, third and fourth graphics pipelines (Col. 6, lines 32-35, 40-47; Col. 7, lines 28-33; Col. 5, line 65-Col. 6, line 3).

It would have been obvious to one of ordinary skill in the art at the time of invention by applicant to modify the device of Furtner to include a bridge operative to transmit vertex data to each of the first, second, third and fourth graphics pipelines as suggested by Alcorn because Alcorn suggests the advantage of being able to evenly distribute the vertex data among the graphics pipelines. In this manner, the system bandwidth is increased because the groups of vertex data are operated upon simultaneously (Col. 6, lines 43-49).

Allowable Subject Matter

37. Claim 19 is objected to as being dependent upon a rejected base claim, but would be allowable if rewritten in independent form including all of the limitations of the base claim and any intervening claims.

The following is a statement of reasons for the indication of allowable subject matter:

38. The prior art singly or in combination do not teach or suggest that each separate chip creates a bounding box around the polygon and wherein each corner of the bounding box is checked against a super tile that belongs to each separate chip and wherein if the bounding box

does not overlap any of the super tiles associated with a separate chip, then the processing circuit rejects the whole polygon and processes a next one, as recited in Claim 19.

39. The closest prior art (Kent) teaches calculating the bounding box of the primitive and testing this against the VisRect. If the bounding box of the primitive is contained in the other P10's super tile the primitive is discarded at this stage [0129]. The method used is to calculate the distance from each subpixel sample point in the point's bounding box to the point's center and compare this to the point's radius. Subpixel sample points with a distance greater than the radius do not contribute to a pixel's coverage. The cost of this is kept low by only allowing small radius points hence the distance calculation is a small multiply and by taking a cycle per subpixel sample per pixel within the bounding box [0144]. However, Kent does not teach that each separate chip creates a bounding box around the polygon and wherein each corner of the bounding box is checked against a super tile that belongs to each separate chip and wherein if the bounding box does not overlap any of the super tiles associated with a separate chip, then the processing circuit rejects the whole polygon and processes a next one.

Prior Art of Record

The prior art made of record and not relied upon is considered pertinent to applicant's disclosure.

US 20030164830A1 teaches a graphics pipeline [0006] that calculates the bounding box of the primitive in a super tile [0129].

Conclusion

Applicant's amendment necessitated the new ground(s) of rejection presented in this Office action. Accordingly, **THIS ACTION IS MADE FINAL**. See MPEP § 706.07(a). Applicant is reminded of the extension of time policy as set forth in 37 CFR 1.136(a).

A shortened statutory period for reply to this final action is set to expire THREE MONTHS from the mailing date of this action. In the event a first reply is filed within TWO MONTHS of the mailing date of this final action and the advisory action is not mailed until after the end of the THREE-MONTH shortened statutory period, then the shortened statutory period will expire on the date the advisory action is mailed, and any extension fee pursuant to 37 CFR 1.136(a) will be calculated from the mailing date of the advisory action. In no event, however, will the statutory period for reply expire later than SIX MONTHS from the date of this final action.

Any inquiry concerning this communication or earlier communications from the examiner should be directed to Joni Hsu whose telephone number is 571-272-7785. The examiner can normally be reached on M-F 8am-5pm.

If attempts to reach the examiner by telephone are unsuccessful, the examiner's supervisor, Ulka Chauhan can be reached on 571-272-7782. The fax phone number for the organization where this application or proceeding is assigned is 571-273-8300.

Information regarding the status of an application may be obtained from the Patent Application Information Retrieval (PAIR) system. Status information for published applications may be obtained from either Private PAIR or Public PAIR. Status information for unpublished applications is available through Private PAIR only. For more information about the PAIR system, see http://pair-direct.uspto.gov. Should you have questions on access to the Private PAIR system, contact the Electronic Business Center (EBC) at 866-217-9197 (toll-free).

JH

ULKA CHAUHAN
SUPERVISORY PATENT EXAMINER

PTO/SB/08A (08-03)
Approved for use through 07/31/2006. OMB 0651-0031
U.S. Patent and Trademark Office; U.S. DEPARTMENT OF COMMERCE
Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it contains a valid OMB control number.

Substitute for form 1449/PTO

INFORMATION DISCLOSURE STATEMENT BY APPLICANT

(Use as many sheets as necessary)

Sheet 1 of 2

Complete if Known	
Application Number	10/459,797
Filing Date	June 12, 2003
First Named Inventor	Mark M. Leather
Art Unit	2671
Examiner Name	Joni Hsu
Attorney Docket Number	00100.02.0053

U. S. PATENT DOCUMENTS

Examiner Initials*	Cite No.[1]	Document Number Number-Kind Code[2] *(if known)*	Publication Date MM-DD-YYYY	Name of Patentee or Applicant of Cited Document	Pages, Columns, Lines, Where Relevant Passages or Relevant Figures Appear
		US- 6,424,345 B1	07-23-2002	Smith et al.	
		US-			
		US-			
		US-			
		US-			
		US-			
		US-			
		US-			
		US-			
		US-			
		US-			
		US-			
		US-			
		US-			
		US-			
		US-			
		US-			
		US-			
		US-			

FOREIGN PATENT DOCUMENTS

Examiner Initials*	Cite No.[1]	Foreign Patent Document Country Code[3] -Number[4] -Kind Code[5] *(if known)*	Publication Date MM-DD-YYYY	Name of Patentee or Applicant of Cited Document	Pages, Columns, Lines, Where Relevant Passages Or Relevant Figures Appear	T[6]

Examiner Signature		Date Considered	

*EXAMINER: Initial if reference considered, whether or not citation is in conformance with MPEP 609. Draw line through citation if not in conformance and not considered. Include copy of this form with next communication to applicant. [1]Applicant's unique citation designation number (optional). [2]See Kinds Codes of USPTO Patent Documents at www.uspto.gov or MPEP 901.04. [3]Enter Office that issued the document, by the two-letter code (WIPO Standard ST.3). [4]For Japanese patent documents, the indication of the year of the reign of the Emperor must precede the serial number of the patent document. [5]Kind of document by the appropriate symbols as indicated on the document under WIPO Standard ST.16 if possible. [6]Applicant is to place a check mark here if English language Translation is attached.

This collection of information is required by 37 CFR 1.97 and 1.98. The information is required to obtain or retain a benefit by the public which is to file (and by the USPTO to process) an application. Confidentiality is governed by 35 U.S.C. 122 and 37 CFR 1.14. This collection is estimated to take 2 hours to complete, including gathering, preparing, and submitting the completed application form to the USPTO. Time will vary depending upon the individual case. Any comments on the amount of time you require to complete this form and/or suggestions for reducing this burden, should be sent to the Chief Information Officer, U.S. Patent and Trademark Office, P.O. Box 1450, Alexandria, VA 22313-1450. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. **SEND TO: Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.**

If you need assistance in completing the form, call 1-800-PTO-9199 (1-800-786-9199) and select option 2.

PTO/SB/08B (08-03)
Approved for use through 07/31/2006. OMB 0651-0031
U.S. Patent and Trademark Office; U.S. DEPARTMENT OF COMMERCE
Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it contains a valid OMB control number.

Substitute for form 1449/PTO

INFORMATION DISCLOSURE STATEMENT BY APPLICANT

(Use as many sheets as necessary)

Complete if Known	
Application Number	10/459,797
Filing Date	June 12, 2003
First Named Inventor	Mark M. Leather
Art Unit	2671
Examiner Name	Joni Hsu
Attorney Docket Number	00100.02.0053

Sheet 2 of 2

NON PATENT LITERATURE DOCUMENTS

Examiner Initials*	Cite No.[1]	Include name of the author (in CAPITAL LETTERS), title of the article (when appropriate), title of the item (book, magazine, journal, serial, symposium, catalog, etc.), date, page(s), volume-issue number(s), publisher, city and/or country where published.	T[2]
		European Search Report from European Patent Office; European Application No. 03257464.2; dated April 4, 2006	
		FOLEY, James et al.; Computer Graphics, Principles and Practice; Addison-Wesley Publishing Company; 1990; pages 873-899	
		CROCKETT, Thomas W.; An introduction to parallel rendering; Elsevier Science B.V.; 1997; pages 819-843	
		MONTRYM, John S. et al.; InfiniteReality: A Real-Time Graphics System; Silicon Graphics Computer Systems; 1997; pages 293-302	
		HUMPHREYS, Greg et al.; WireGL: A Scalable Grpahics System for Clusters; ACM Siggraph; 2001; pages 129-140	

Examiner Signature		Date Considered	

*EXAMINER: Initial if reference considered, whether or not citation is in conformance with MPEP 609. Draw line through citation if not in conformance and not considered. Include copy of this form with next communication to applicant.
1 Applicant's unique citation designation number (optional). 2 Applicant is to place a check mark here if English language Translation is attached.
This collection of information is required by 37 CFR 1.98. The information is required to obtain or retain a benefit by the public which is to file (and by the USPTO to process) an application. Confidentiality is governed by 35 U.S.C. 122 and 37 CFR 1.14. This collection is estimated to take 2 hours to complete, including gathering, preparing, and submitting the completed application form to the USPTO. Time will vary depending upon the individual case. Any comments on the amount of time you require to complete this form and/or suggestions for reducing this burden, should be sent to the Chief Information Officer, U.S. Patent and Trademark Office, P.O. Box 1450, Alexandria, VA 22313-1450. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. **SEND TO: Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.**

If you need assistance in completing the form, call 1-800-PTO-9199 (1-800-786-9199) and select option 2.

Electronic Acknowledgement Receipt

EFS ID:	1017094
Application Number:	10459797
Confirmation Number:	4148
Title of Invention:	Dividing work among multiple graphics pipelines using a super-tiling technique
First Named Inventor:	Mark M. Leather
Customer Number:	29153
Filer:	Christopher J. Reckamp/Christine Wright
Filer Authorized By:	Christopher J. Reckamp
Attorney Docket Number:	00100.02.0053
Receipt Date:	04-APR-2006
Filing Date:	12-JUN-2003
Time Stamp:	15:16:21
Application Type:	Utility
International Application Number:	

Payment information:

Submitted with Payment	no

File Listing:

Document Number	Document Description	File Name	File Size(Bytes)	Multi Part	Pages
1	Information Disclosure Statement (IDS) Filed	IDS_10459797.pdf	162453	no	2

Warnings:					
Information:					
This is not an USPTO supplied IDS fillable form					
2	NPL Documents	EPSearchReport_10459797.pdf	118954	no	4
Warnings:					
Information:					
3	NPL Documents	Foley_10459797.pdf	1599163	no	28
Warnings:					
Information:					
4	NPL Documents	Crockett_10459797.pdf	1355723	no	25
Warnings:					
Information:					
5	NPL Documents	Montrym_10459797.pdf	693876	no	10
Warnings:					
Information:					
6	NPL Documents	Humphreys_10459797.pdf	1144513	no	12
Warnings:					
Information:					
			Total Files Size (in bytes):	5074682	

This Acknowledgement Receipt evidences receipt on the noted date by the USPTO of the indicated documents, characterized by the applicant, and including page counts, where applicable. It serves as evidence of receipt similar to a Post Card, as described in MPEP 503.

New Applications Under 35 U.S.C. 111

If a new application is being filed and the application includes the necessary components for a filing date (see 37 CFR 1.53(b)-(d) and MPEP 506), a Filing Receipt (37 CFR 1.54) will be issued in due course and the date shown on this Acknowledgement Receipt will establish the filing date of the application.

National Stage of an International Application under 35 U.S.C. 371

If a timely submission to enter the national stage of an international application is compliant with the conditions of 35 U.S.C. 371 and other applicable requirements a Form PCT/DO/EO/903 indicating acceptance of the application as a national stage submission under 35 U.S.C. 371 will be issued in addition to the Filing Receipt, in due course.

ELSEVIER

Parallel Computing 23 (1997) 819–843

PARALLEL COMPUTING

An introduction to parallel rendering

Thomas W. Crockett *,[1]

Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23681-0001, USA

Received 2 November 1996; revised 30 November 1996

Abstract

In computer graphics, rendering is the process by which an abstract description of a scene is converted to an image. When the scene is complex, or when high-quality images or high frame rates are required, the rendering process becomes computationally demanding. To provide the necessary levels of performance, parallel computing techniques must be brought to bear. Today, parallel hardware is routinely used in graphics workstations, and numerous software-based rendering systems have been developed for general-purpose parallel architectures. This article provides an overview of the parallel rendering field, encompassing both hardware and software systems. The focus is on the underlying concepts and the issues which arise in the design of parallel renderers. We examine the different types of parallelism and how they can be applied in rendering applications. Concepts from parallel computing, such as data decomposition and load balancing, are considered in relation to the rendering problem. Our survey explores a number of practical considerations as well, including the choice of architectural platform, communication and memory requirements, and the problem of image assembly and display. We illustrate the discussion with numerous examples from the parallel rendering literature, representing most of the principal rendering methods currently used in computer graphics.

Keywords: Parallel rendering; Computer graphics; Survey

1. Introduction

In computer graphics, *rendering* is the process by which an abstract description of a scene is converted to an image. Fig. 1 illustrates the basic problem. For purposes of this

* E-mail: tom@icase.edu.

[1] This work was supported in part by the National Aeronautics and Space Administration under Contract No. NAS1-19480 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), M/S 403, NASA Langley Research Center, Hampton, VA 23681-0001.

0167-8191/97/$17.00 Copyright © 1997 Elsevier Science B.V. All rights reserved.
PII S0167-8191(97)00028-8

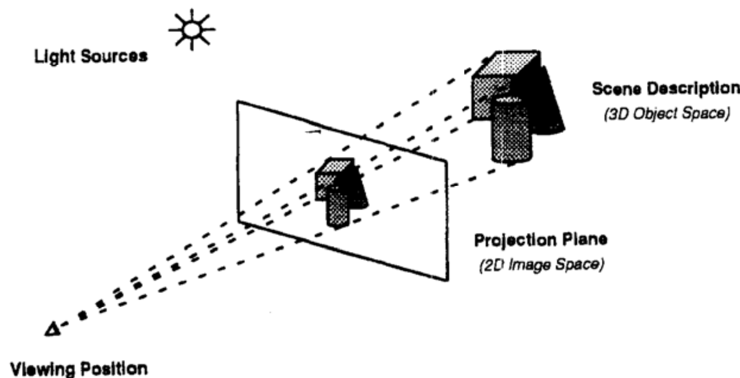


Fig. 1. The generic rendering problem. A three-dimensional scene is projected onto an image plane, taking into account the viewing parameters and light sources.

discussion, a scene is a collection of geometrically-defined objects in three-dimensional *object space*, with associated lighting and viewing parameters. The rendering operation illuminates the objects and projects them into two-dimensional *image space*, where color intensities of individual pixels are computed to yield a final image [2].

For complex scenes or high-quality images, the rendering process is computationally intensive, requiring millions or billions of floating-point and integer operations for each image. The need for interactive or real-time response in many applications places additional demands on processing power. The only practical way to obtain the needed computational power is to exploit multiple processing units to speed up the rendering task, a concept which has become known as *parallel rendering*.

Parallel rendering has been applied to virtually every image generation technique used in computer graphics, including surface and polygon rendering, terrain rendering, volume rendering, ray-tracing, and radiosity. Although the requirements and approaches vary for each of these cases, there are a number of concepts which are important in understanding how parallelism applies to the generic rendering problem.

We begin our examination of parallel rendering in Section 2 by considering the types of parallelism which are available in computer graphics applications. Section 3 then introduces a number of concepts which are central to an understanding of parallel rendering algorithms. Building on this base, Section 4 considers design and implementation issues for parallel renderers, with an emphasis on architectural considerations and application requirements. Sections 2–4 are illustrated throughout with examples from the parallel rendering literature. Section 5 completes our survey with an examination of several parallel rendering applications.

[2] For a comprehensive reference to the discipline of computer graphics, see [23].

2. Parallelism in the rendering process

Several different types of parallelism can be applied in the rendering process. These include *functional parallelism*, *data parallelism*, and *temporal parallelism*. These basic types can also be combined into hybrid systems which exploit multiple forms of parallelism. Each of these options is discussed below.

2.1. Functional parallelism

One way to obtain parallelism is to split the rendering process into several distinct functions which can be applied in series to individual data items. If a processing unit is assigned to each function (or group of functions) and a data path is provided from one unit to the next, a rendering *pipeline* is formed (Fig. 2). As a processing unit completes work on one data item, it forwards it to the next unit, and receives a new item from its upstream neighbor. Once the pipeline is filled, the degree of parallelism achieved is proportional to the number of functional units.

The functional approach works especially well for polygon and surface rendering applications, where 3D geometric primitives are fed into the beginning of the pipe, and final pixel values are produced at the end. This approach has been mapped very successfully into the special-purpose rendering hardware used in a variety of commercial computer graphics workstations produced during the 1980's and 1990's. The archetypal example is Clark's Geometry System [10,11], which replicated a custom VLSI geometry processor in a 12-stage pipeline to perform transformation and clipping operations in two and three dimensions.

Despite its success, the functional approach has two significant limitations. First, the overall speed of the pipeline is limited by its slowest stage, so functional units must be designed carefully to avoid bottlenecks. More importantly, the available parallelism is limited to the number of stages in the pipeline. To achieve higher levels of performance, an alternate strategy is needed.

2.2. Data parallelism

Instead of performing a sequence of rendering functions on a single data stream, it may be preferable to split the data into multiple streams and operate on several items simultaneously by replicating a number of identical rendering units (Fig. 3).

Because the data-parallel approach can take advantage of larger numbers of processors, it has been adopted in one form or another by most of the software renderers which have been developed for general-purpose 'massively parallel' systems. Data parallelism also lends itself to scalable implementations, allowing the number of processing elements to be varied depending on factors such as scene complexity, image resolution, or desired performance levels.

Two principal classes of data parallelism can be identified in the rendering process. *Object parallelism* refers to operations which are performed independently on the geometric primitives which comprise objects in a scene. These operations constitute the first few stages of the rendering pipeline (Fig. 2), including modeling and viewing

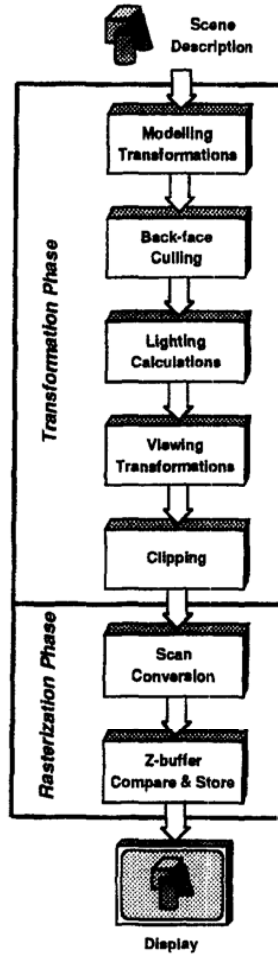


Fig. 2. A typical polygon rendering pipeline. The number of function units and their order varies depending on details of the implementation.

transformations, lighting computations, and clipping. *Image parallelism* occurs in the later stages of the rendering pipeline, and includes the operations used to compute individual pixel values. Pixel computations vary depending on the rendering method in use, but may include illumination, interpolation, composition, and visibility determination. Collectively we call the object-level stages of the pipeline the *transformation phase*; the image-level stages are grouped together to form the *rasterization phase*.

To avoid bottlenecks, most data-parallel rendering systems must exploit both object

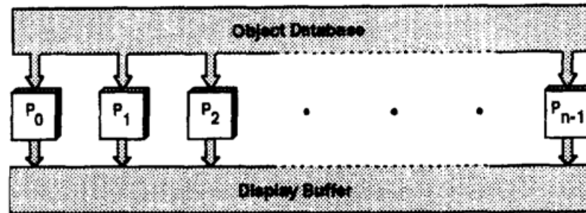


Fig. 3. A data-parallel rendering system. Multiple data items are processed simultaneously and the results are merged to create the final image.

and image parallelism. Obtaining the proper balance between these two phases of the computation is difficult, since the workloads involved at each level are highly dependent on factors such as the scene complexity, average screen area of transformed geometric primitives, pixel sampling factor, and image resolution.

2.3. *Temporal parallelism*

In animation applications, where hundreds or thousands of high-quality images must be produced for subsequent playback, the time to render individual frames may not be as important as the overall time required to render all of them. In this case, parallelism may be obtained by decomposing the problem in the time domain. The fundamental unit of work is a complete image, and each processor is assigned a number of frames to render, along with the data needed to produce those frames.

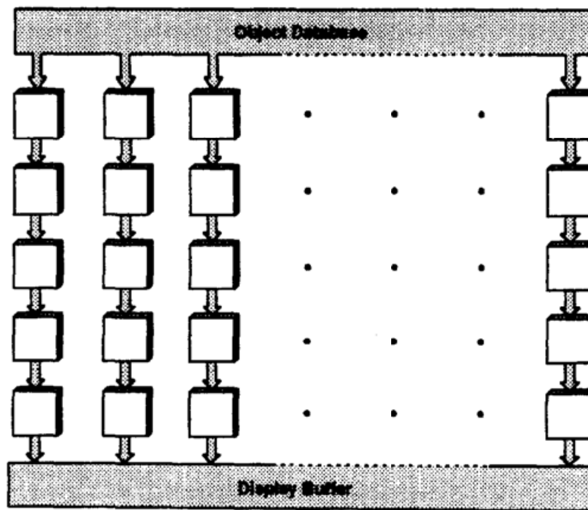


Fig. 4. A hybrid rendering architecture. Functional parallelism and data parallelism are both exploited to achieve higher performance.

2.4. Hybrid approaches

It is certainly possible to incorporate multiple forms of parallelism in a single system. For example, the functional- and data-parallel approaches may be combined by replicating all or part of the rendering pipeline (Fig. 4). This strategy was adopted for Silicon Graphics' RealityEngine [1], which combines multiple transformation and rasterization units in a highly pipelined architecture to achieve rendering rates on the order of one million polygons per second. In similar fashion, temporal parallelism may be combined with the other strategies to produce systems with the potential for extremely high aggregate performance.

3. Algorithmic concepts

Some problems can be parallelized trivially, requiring little or no interprocessor communication, and with no significant computational overheads attributable to the parallel algorithm. Such applications are said to be *embarrassingly parallel*, and efficient operation can be expected on a variety of platforms, ranging from networks of personal computers or graphics workstations up to massively parallel supercomputers. Rendering algorithms which exploit temporal parallelism typically fall into this category.

Rendering methods based on ray-casting (such as ray-tracing and direct volume rendering) also have embarrassingly parallel implementations in certain circumstances. Because pixel values are computed by shooting rays from each pixel into the scene, image-parallel task decompositions are very natural for these problems. If every processor has fast access to the entire object database, then each ray can be processed independently with no interprocessor communication required. This approach is practical for shared-memory architectures, and also performs well on distributed-memory systems when sufficient memory is available to replicate the object database on every processor.

In other cases the design of effective parallel rendering algorithms can be a challenging task. Most parallel algorithms introduce overheads which are not present in their sequential counterparts. These overheads may result from some or all of the following:
- communication among tasks or processors
- delays due to uneven workloads
- additional or redundant computations
- increased storage requirements for replicated or auxiliary data structures

To understand how these overheads arise in parallel rendering algorithms, we need to examine several key concepts. Some of these concepts (task and data decomposition, load balancing) are common to most parallel algorithms, while others (coherence, object-space to image-space mapping) are specific to the rendering problem.

3.1. Coherence

In computer graphics, *coherence* refers to the tendency for features which are nearby in space or time to have similar properties [64]. Many fundamental algorithms in the

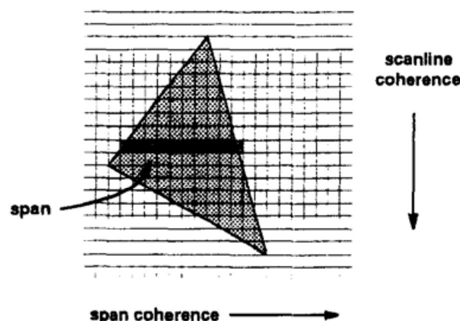


Fig. 5. Spatial coherence in image space. Pixel values tend to be similar from one scanline to the next, and from pixel to pixel within spans. Sequential rendering algorithms exploit this property to reduce computation costs during scan conversion.

field rely on coherence in one form or another to reduce computational requirements. Coherence is important to parallel rendering in two ways. First, parallel algorithms which fail to preserve coherence will incur computational overheads which may not be present in equivalent sequential algorithms. Secondly, parallel algorithms may be able to exploit coherence to reduce communication costs or improve load balance.

Several types of coherence are important in parallel rendering. *Frame coherence* is the tendency of objects, and hence resulting pixel values, to move or change shape or color slowly from one image to the next in a related sequence of frames. This property can be used to advantage in load balancing (for predicting workloads) and in image display (by reducing the number of pixels to be transmitted).

Scanline coherence refers to the similarity of pixel values from one scanline to the next in the vertical direction. The corresponding property in the horizontal direction is called *span coherence*, which refers to the similarity of nearby pixel values within a scanline (Fig. 5). Sequential rasterization algorithms rely on these two forms of *spatial coherence* for efficient interpolation of pixel values between the vertices of geometric primitives. When an image is partitioned to exploit image parallelism, coherence may be lost at partition boundaries, resulting in computational overheads. The probability that a primitive will intersect a boundary depends on the size, shape, and number of image partitions [50,69], and hence is an important consideration in the design of parallel polygon renderers [21].

A related notion in ray-casting renderers is *data* or *ray coherence*. This is the tendency for rays cast through nearby pixels to intersect the same objects in a scene. Ray coherence has been exploited in conjunction with data-caching schemes to reduce communication loads in parallel volume rendering and ray-tracing algorithms [2,48].

3.2. Object-space to image-space mapping

The key to high performance on many parallel architectures is successful exploitation of data locality to minimize remote memory references. In parallel rendering algorithms,

we also want to partition the image and object data among the available processors to achieve scalable performance and to accommodate increases in scene complexity and image resolution. Unfortunately, these two goals are in conflict.

To understand the problem, we observe that, geometrically, rendering is a mapping from three-dimensional object space to two-dimensional image space (Fig. 1). This mapping is not fixed, but instead depends on the modeling transformations and viewing parameters in use when a scene is rendered. If both the object and image data structures are partitioned among the processors, then at some point in the rendering pipeline data must be communicated among the processors. Because of the complexity and dynamic nature of the mapping function, the communication pattern is essentially arbitrary, with each processor sending data to, and receiving data from, a large number of other processors.

Managing this communication is one of the central issues for parallel renderers, particularly on distributed-memory architectures. To better understand this problem, Molnar et al. [50] developed a taxonomy of parallel rendering algorithms based on the point in the rendering pipeline at which the object-space to image-space mapping occurs. They classify algorithms as either *sort-first*, *sort-middle*, or *sort-last*, depending on whether the communication step occurs at the beginning, middle, or end of the rendering pipeline. Their analysis of the computation and communication costs of each approach concludes that none of them is inherently superior in all circumstances. Additional analysis of the three strategies can be found in [15], and a detailed study of the rarely-used sort-first method is presented in [51]. Examples of sort-middle renderers include [20,21], while the sort-last strategy is used in [14,16,28,40].

3.3. Task and data decomposition

Data-parallel rendering algorithms may be distinguished based on the way in which they decompose the problem into individual workloads or tasks. There are two main strategies. In an *object-parallel* approach, tasks are formed by partitioning either the geometric description of the scene or the associated object space. Rendering operations are then applied in parallel to subsets of the geometric data, producing pixel values which must be combined to form a final image. In contrast, *image-parallel* algorithms reverse this mapping. Tasks are formed by partitioning the image space, and each task renders those geometric primitives which contribute to the pixels which it has been assigned.

The choice of image-parallel versus object-parallel algorithms is not clear-cut. Object-parallel algorithms tend to distribute object computations evenly among processors, but since geometric primitives usually vary in size, rasterization loads may be uneven. Furthermore, the integration step needed to combine pixel values into a finished image can place heavy bandwidth demands on memory busses or communication networks.

Image-parallel algorithms avoid the integration step, but have another problem: portions of a single geometric primitive may map to several different regions in the image space. This requires that primitives, or portions of them, be communicated to

multiple processors, and the corresponding loss of spatial coherence results in additional or redundant computations which are not present in equivalent sequential algorithms.

To achieve a better balance among the various overheads, some algorithms adopt a hybrid approach, incorporating features of both object- and image-parallel methods [20,21,53,62]. These techniques partition both the object and image spaces, breaking the rendering pipeline in the middle and communicating intermediate results from object rendering tasks to image rendering tasks.

3.4. Load balancing

In any parallel computing system, effective processor utilization depends on distributing the workload evenly across the system. In parallel rendering, there are many factors which make this goal difficult to achieve. Consider a data-parallel polygon renderer [3] which attempts to balance workloads by distributing geometric primitives evenly among all of the processors. First, polygons may have varying numbers of vertices, resulting in differing operation counts for illumination and transformation operations. If back-face culling is enabled, different processors may discard different numbers of polygons, and the subsequent clipping step may introduce further variations. The sizes of the transformed screen primitives will also vary, resulting in differing operation counts in the rasterization routines. Depending on the method being used, hidden surface elimination will also produce variations in the number of polygons to be rasterized or the number of pixels to be stored in the frame buffer.

While this list may seem intimidating, we observe that if the number of input primitives is large (as it usually is) and the primitives are randomly assigned to processors, the workload variations described above will tend to even out. Unfortunately, a much more serious source of load imbalance arises due to another factor: in real scenes, the distribution of primitives in image space is not uniform, but tends to cluster in areas of detail. Thus processors responsible for rasterizing dense regions of the image will have significantly more work to do than other processors which may end up with nothing more than background pixels. To make matters worse, the mapping from object space to image space is view dependent, which means the distribution of primitives in the image is subject to change from one frame to the next, especially in interactive applications.

Strategies for dealing with this image-space load imbalance may be classified as either *static* or *dynamic*. Static load balancing techniques rely on a fixed data partitioning to distribute local variations across large numbers of processors. Fig. 6 shows several image partitioning strategies with different load balancing characteristics. Large blocks of contiguous pixels (Fig. 6a) usually result in poor load balancing, while fine-grained partitioning schemes (Fig. 6c, d) distribute the load better. However, fine-grained schemes are subject to computational overheads due to loss of spatial coherence, as discussed in Section 3.1. Analytical and experimental results [68,69]

[3] Although the causes are different, similar imbalances arise in other rendering methods as well.

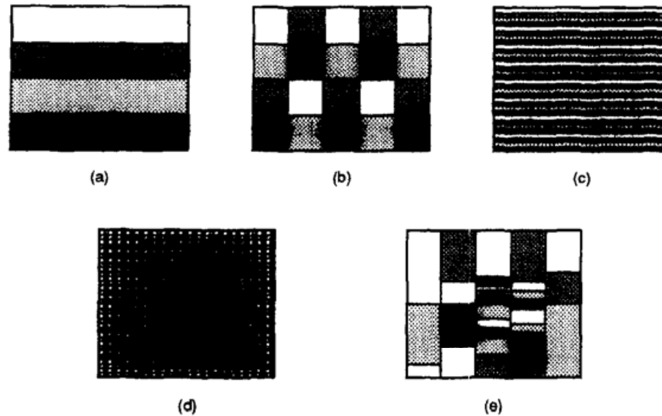


Fig. 6. Image partitioning strategies. Shading indicates the assignment of image regions to four processors. (a) Blocks of contiguous scanlines; (b) square regions; (c) interleaved scanlines; (d) pixel interleaving in two dimensions; (e) adaptive partitioning (loosely based on [70]).

indicate that square regions (Fig. 6b) minimize the loss of coherence since they have the smallest perimeter-to-area ratio of any rectangular subdivision scheme.

Dynamic load-balancing schemes try to improve on static techniques by providing more flexibility in assigning workloads to processors. There are two principal strategies. The *demand-driven* approach decomposes the problem into independent tasks which are assigned to processors one-at-a-time or in small groups. When a processor completes one task, it receives another, and the process continues until all of the tasks are complete. If tasks exhibit large variations in run time, the most expensive ones must be started early so that they will have time to finish while other processors are still busy with shorter tasks. The alternative is to use large numbers of fine-grained tasks in order to minimize potential variations, but this approach suffers increased overheads due to loss of coherence and more frequent task assignment operations.

The alternate *adaptive* strategy tries to minimize pre-processing overheads by deferring task partitioning decisions until one or more processors becomes idle, at which time the remaining workloads of busy processors are split and reassigned to idle processors. The result is that data partitioning is not predetermined, but instead adapts to the computational load (Fig. 6e). A good example is Whitman's image-parallel polygon renderer for the BBN TC2000 [70]. Whitman's renderer initially partitions the image space into a relatively small number of coarse-grained tasks, which are then assigned to processors using the demand driven model. When a processor becomes idle and no more tasks are available from the initial pool, it searches for the processor with the largest remaining workload and 'steals' half of its work. The principal overheads in the adaptive approach arise in maintaining and retrieving non-local status information, partitioning tasks, and migrating data.

While dynamic schemes offer the potential for more precise load balancing than static

schemes, they are successful only when the improvements in processor utilization exceed the overhead costs. For this reason, dynamic schemes are easiest to implement on architectures which provide low-latency access to shared memory. In message-passing systems, the high cost of remote memory references makes dynamic task assignment, data migration, and maintenance of global status information more expensive, especially for fine-grained tasks.

4. Design and implementation issues

As the above discussion suggests, the design space for parallel rendering algorithms is large and replete with trade-offs. How these trade-offs are resolved depends on a variety of factors, including application requirements and characteristics of the target architecture. In the following sections, we examine some of the issues which must be considered.

4.1. Hardware versus software systems

Perhaps the most fundamental distinction between parallel rendering designs is that of hardware-based versus software-based systems. Hardware systems, ranging from specialized graphics computers to graphics workstations and add-on graphics accelerator boards, all employ dedicated circuitry to speed up the rendering task. The hardware approach has been very successful, although commercial systems to date have been designed primarily for polygon rendering. Furthermore, the specialization which contributes to the high performance and cost-effectiveness of dedicated hardware also tends to limit its flexibility. Specialized lighting models, high-resolution imaging, and sophisticated rendering methods such as ray-tracing and radiosity must be implemented largely in software, with a corresponding degradation in performance.

One way to boost the performance of software-based renderers is to implement them on general-purpose parallel platforms, such as scalable parallel supercomputers or networks of workstations. On these systems, the processors are not specifically optimized for graphical operations, and communication networks often have bandwidth limitations and software overheads which are not found in hardware-based rendering systems. The challenge is to develop algorithms which can cope successfully with these overheads in order to realize the performance potential of the underlying hardware. Some recent examples indicate that this challenge can be met. Polygon renderers developed for Intel's Touchstone Delta and Paragon systems [21,40], Thinking Machines' CM-200 and CM-5 [28,55], and Cray's T3D [16] achieved performance levels which equalled or exceeded those of contemporary high-end graphics workstations.

Software-based renderers are of interest on massively parallel architectures for another reason: massive data. The datasets produced by large-scale scientific applications can easily be hundreds of megabytes in size, and time-dependent simulations may produce this much data for hundreds or thousands of time-steps. Visualization techniques are imperative in exploring and understanding datasets of this size, but the sheer

volume of data may make the use of detached graphics systems impractical or impossible. The alternative is to exploit the parallelism of the supercomputer to perform the visualization and rendering computations in place, eliminating the need to move the data. This has motivated recent work on software-based rendering systems which can be embedded in parallel applications to produce live visual output at run time [16,17].

Networks of workstations and personal computers provide another type of platform which can be used by software-based parallel renderers. These systems are inexpensive and ubiquitous, and their processing power and memory capacities are increasing dramatically. However, they tend to be connected by low-bandwidth networks, and suffer from high communication latencies due to operating system overheads and costly network protocols. For these reasons, they are best used in modest numbers for large granularity computations where high frame rates are not an overriding consideration. They are also well-suited for embarrassingly parallel applications which replicate the object database or exploit temporal parallelism to render entire frames locally. Examples of network-based systems include volume renderers [26,46], radiosity renderers [59,60,72], and Pixar's photorealistic NetRenderMan system [57].

4.2. Shared vs. distributed memory

While traditional shared-memory systems offer the potential for low-overhead parallel rendering, their performance scalability is limited by contention on the busses or switch networks which connect processors to memory. Adding processors does not increase the memory bandwidth, so at some point the paths to memory become saturated and performance stalls. For this reason, most parallel architectures with large numbers of processing elements employ a distributed-memory model, in which each processor is tightly coupled to a local memory. The combined processor/memory elements are then interconnected by a relatively scalable network or switch. The advantage is that processing power and aggregate local memory bandwidth scale linearly with the number of hardware units in the system. The disadvantage is that access to off-processor data may take several orders of magnitude longer to complete than local accesses.

A number of recent systems combine elements of both architectures, using physically distributed memories which are mapped into a global shared address space [13,36,41]. The shared address space permits the use of concise shared-memory programming paradigms, and is amenable to hardware support for remote memory references. The result is that communication overheads can be significantly lower than those found in traditional message-passing systems, allowing algorithms with fine-grained communication requirements to scale to larger numbers of processors.

From an algorithmic standpoint, shared-memory systems provide relatively efficient access to a global address space, which in turn reduces the need to pre-partition major data structures, simplifies processor coordination, and maximizes the range of practical algorithms. To avoid resource contention, good shared-memory algorithms must decompose the problem into tasks which eliminate memory hot spots and keep critical sections and synchronization operations to a minimum. Since most shared-memory systems are augmented with processor caches and/or local memories, algorithms intended for these

platforms still must strive for a high degree of locality in their memory reference patterns.

Distributed-memory systems offer improved architectural scalability, but generally incur higher costs for remote memory references. For this class of machines, managing communication is a primary consideration. Parallel renderers must pay special attention to this issue due to the large volume of intermediate data which must be re-mapped from object space to image space. In the absence of specialized hardware support, global operations and synchronization may be particularly expensive, and the higher cost of data migration may favor static assignment of tasks and data.

4.3. SIMD vs. MIMD

Because MIMD architectures allow processors to respond to local differences in workload, they would seem to be a good match for the highly variable operation counts and data access patterns which characterize the rendering process (see Section 3.4). Furthermore, the MIMD environment lends itself to demand-driven and adaptive load balancing schemes, where processors work independently on relatively coarse-grained tasks. Numerous MIMD renderers have been implemented, on a variety of hardware platforms, encompassing all of the major rendering methods.

Despite the apparent mismatch between the variability of the rendering process and the tight synchronization of SIMD architectures, a number of parallel renderers have demonstrated good performance on SIMD systems [29,33,45,55]. There are several reasons for this. First of all, the flexibility of MIMD systems imposes a burden on applications and operating systems, which must be able to cope with the arrival of data from remote sources at unpredictable intervals and in arbitrary order. This often results in complex communication and buffering protocols, particularly on distributed-memory message-passing systems. The lock-step operation of SIMD systems virtually eliminates these software overheads, resulting in communication costs which are much closer to the actual hardware speeds.

Secondly, it is often possible to structure algorithms as several distinct phases, each of which operates on a uniform data type. The rendering pipeline maps naturally onto this structure, and the regularity of the data structures within each phase leads to uniform operations, providing a good fit with the SIMD programming paradigm.

Finally, SIMD architectures usually contain thousands of simple processing elements. Because of their sheer numbers, good performance can often be achieved even though processors may not be fully utilized.

4.4. Communication

For renderers which exploit both image and object parallelism, a high volume of interprocessor communication is inherent in the process (see Section 3.2). Managing this communication is a central issue in renderer design, and the choice of algorithm can have a significant impact on the timing, volume, and patterns of communication [15,20,32,50,53]. There are three main factors which need to be considered: *latency*, *bandwidth*, and *contention*. Latency is the time required to set up a communication

operation, irrespective of the amount of data to be transmitted. Bandwidth is simply the amount of data which can be communicated over a channel per unit time. If a renderer tries to inject more data into a network than the network can absorb, delays will result and performance will suffer. Contention occurs when multiple processors are trying to route data through the same segment of the network simultaneously and there is insufficient bandwidth to support the aggregate demand.

Hardware latencies for sending, receiving, and routing messages are in the sub-microsecond range on many systems. However, software layers can boost these times considerably — measured send and receive latencies on message-passing systems often exceed the hardware times by a few orders of magnitude. Bandwidths exhibit similar variations, ranging from hundreds of kilobytes/second on workstation networks up to several gigabytes/second in dedicated graphics hardware. While latencies and bandwidths can usually be determined with reasonable precision, contention delays are more difficult to characterize, since they depend on dynamic traffic patterns which tend to be scene- and view-dependent.

A number of algorithmic techniques have been developed for coping with communication overheads in parallel renderers. A simple way to reduce latency is to accumulate short messages into large buffers before sending them, thereby amortizing the cost over many data items. Unfortunately, this technique does not scale well for the common case of object- to image-space sorting, since the communication pattern is generally many-to-many [20,21]. This implies that the number of messages generated per processor is $O(p)$, where p is the number of processors in the system. Assuming a fixed scene and image resolution and a p-way partitioning of the object and image data, the number of data items per processor is proportional to $1/p$, and the number of data items per message decreases as $1/p^2$. Hence overheads due to latency increase linearly with the number of processors and amortization of these overheads becomes increasingly ineffective.

One solution is to reduce the algorithmic complexity of the communication by using a multi-step delivery scheme [21,40]. With this approach, the processors are divided into approximately $\sqrt{p}$ groups, each containing roughly $\sqrt{p}$ processors. Data items intended for any of the processors within a remote group are accumulated in a buffer and transmitted together as a single large message to a forwarding processor within the destination group. The forwarding processor copies the incoming data items into a second set of buffers on the basis of their final destinations, merging them with contributions from each of the other groups. The sorted buffers are then routed to their final destinations within the local group.

While helpful in reducing latency, large message buffers can contribute to contention delays when network bandwidth is insufficient [20]. The problem arises when a large volume of data is injected into the network within a short period of time. If the traffic fails to clear rapidly enough, processors must wait for data to arrive, and performance suffers. The problem is most pronounced when workloads are evenly balanced, since processors tend to be communicating at about the same time. By using a series of intermediate-sized messages and asynchronous communication protocols, the load on the network can be spread out over time, and data transfer can be overlapped with useful computation.

4.5. Memory constraints

Memory consumption is another issue which must be considered when designing parallel renderers. Rendering is a memory-intensive application, especially with complex scenes and high-resolution images. As a baseline, a full-screen (1280×1024), full-color (24 bits/pixel), z-buffered image requires on the order of 10 MB of memory for the image data structures alone. The addition of features such as transparency and antialiasing can push memory demands into the hundreds of megabytes, a regime in which parallel systems or high-end graphics workstations are mandatory.

The structure of a parallel renderer can have a major impact on memory requirements, either facilitating memory-intensive techniques by partitioning data structures across processors, or inhibiting scalability by requiring replicated or auxiliary data structures. Sort-middle polygon rendering is one example of an approach which exhibits good data scalability, since object and image data structures can be partitioned uniformly among the processing elements. The cost of image memory in these systems is essentially fixed. By contrast, some sort-last algorithms require the entire image memory to be replicated on every processor, increasing the cost in direct proportion to the number of processing elements in the system.

The issue of memory consumption involves many tradeoffs, and system designers must balance application requirements, performance goals, and system cost. For example, replicating object data in an image-parallel renderer can reduce or eliminate overheads for interprocessor communication, a strategy which may work well for rendering moderately complex scenes in low-bandwidth, high-latency environments, such as workstation networks. On the other hand, rendering algorithms which are embedded in memory-intensive applications must be careful to limit their own resource requirements to avoid undue interference with the application [17]. In this case, data scalability may be a more important consideration than absolute performance.

Some renderers operate in distinct phases, requiring each phase to complete before the next phase begins. This implies that intermediate results produced by each phase must be stored, rather than being passed along for immediate consumption. The amount of intermediate storage needed for each phase depends on the particular data items being produced, but in general is a function of the scene complexity. For complex scenes the memory overheads may be substantial, but they do exhibit data scalability, assuming the object data is partitioned initially.

4.6. Image display

High-performance rendering systems produce prodigious quantities of output in the form of an image stream. For full-screen, full-color animation (1280×1024 resolution, 24 bits/pixel, 30 frames/s), a display bandwidth of 120 MB/s is required. Since most parallel renderers either partition or replicate the image space, the challenge is to combine pixel values from multiple sources at high frame rates. Failure to do so will create a bottleneck at the display stage of the rendering pipeline, limiting the amount of parallelism which can be effectively utilized.

The display problem is best addressed at the architectural level, and hardware

rendering systems have adopted several different techniques. One approach is to integrate the frame buffer memory directly with the pixel-generation processors [1,24,58]. Highly parallel, multi-ported busses or other specialized hardware mechanisms are then used to interface the distributed frame buffer to the video generation subsystem.

Alternatively, the rasterization engines and frame buffer may be distinct entities, with pixel data being communicated from one to the other via a high-speed communication channel. One example is the Pixel-Planes 5 system [25], which uses a 640 MB/s token ring network to interconnect system components, including the pixel renderers and frame buffer. The PixelFlow system [49] pushes transfer rates a step further, using a pipelined image composition network with an effective interstage bandwidth in excess of 4 GB/s. The frame buffer resides at the terminus of the pipeline, acting as a sink for the final composited pixel values.

With general-purpose parallel computers, sustaining high frame rates is problematic, since these systems often lack specialized features for image integration and display. There are two principal issues, assembling finished images from distributed components, and moving them out of the system and onto a display device. The bandwidth of the interprocessor communication network is an important consideration for the image assembly phase, since high frame rates cannot be sustained unless image components can be retrieved rapidly from individual processor memories.

Several current systems, including the Intel Paragon and Cray T3D, provide internal networks with transfer rates in excess of 100 MB/s, which is more than adequate for interactive graphics. The challenge on these systems is to orchestrate the image retrieval and assembly process so that the desired frame rates can be achieved [18,19]. In the absence of multi-ported frame buffers, the image stream must be serialized, perhaps with some ordering imposed, and forwarded to an external device interface.

Assuming that the internal image assembly rate is satisfactory, the next bottleneck is the I/O interface to the display. The typical configuration on current systems uses a HIPPI interface [30] attached to an external frame buffer device. While many of the existing implementations fail to sustain the 100 MB/s transfer rate of the HIPPI specification, the technology is improving, and either HIPPI or emerging technologies such as ATM [66] are likely to provide sufficient external bandwidth in the near future.

To avoid the bottlenecks associated with serial I/O interfaces, some general-purpose architectures incorporate multi-ported frame buffers which attach either directly or indirectly to the system's internal communication network [4,67]. Pixels or image segments must then be routed to the appropriate frame buffer ports and the inputs must be synchronized to ensure coherent displays.

5. Examples of parallel rendering systems

Virtually all current graphics systems incorporate parallelism in one form or another. We have illustrated the preceding discussion with a number of examples. In this section, we round out our survey by examining additional representative systems, running the gamut from specialized graphics computers to software-based terrain and radiosity renderers. Our coverage is by no means complete — many more examples can be found

in the literature. Readers are encouraged to explore the references at the end of this article for additional citations.

5.1. Polygon rendering

One of the earliest graphics architectures to exploit large-scale data parallelism was Fuchs and Poulton's classic Pixel-Planes system [24]. Pixel-Planes parallelized the rasterization and z-buffering stages of the polygon rendering pipeline by augmenting each pixel with a simple bit-serial processor which was capable of computing color and depth values from the plane equations which described each polygon. The pixel array operated in SIMD fashion, taking as input a serial stream of transformed screen-space polygons generated by a conventional front-end processor.

While Pixel-Planes provided massive image parallelism, it suffered from poor processor utilization, since only those processors which fell within the bounds of a polygon were active at any given time. The Pixel-Planes 5 architecture [25] rectifies these deficiencies. Instead of a single large array of image processors, it incorporates several smaller ones which can be dynamically reassigned to screen regions in demand-driven fashion. Pixel-Planes 5 is a classic example of a sort-middle architecture, with global communication occurring at the break between the transformation and rasterization phases. By contrast, the newer PixelFlow design [49] implements a sort-last architecture, in which each processing node incorporates a full graphics pipeline. Object parallelism is achieved by distributing primitives across the nodes, while pixel parallelism is provided by a Pixel-Planes-style SIMD rasterizer on each node. A 256-bit-wide pipelined interconnect supports the bandwidth-intensive image composition step.

Among commercially-available polygon renderers, Silicon Graphics' RealityEngine series [1] has enjoyed the most success, and is the renderer of choice in a host of demanding applications, including virtual reality, real-time simulation, and scientific visualization. The recently-introduced InfiniteReality system continues this tradition, boosting polygon rendering rates by a factor of five over the second-generation RealityEngine2.

5.2. Volume rendering

Graphics architectures have also been developed specifically for volume rendering and ray-tracing applications. In volume rendering, one of the keys to performance is providing high-bandwidth, conflict-free access to the volume data. This has prompted the development of specialized volume memory structures which allow simultaneous access to multiple data values. Kaufman and Bakalash's Cube system [35] introduced an innovative 3D voxel buffer which facilitates parallel access to cubes of volumetric data. A linear array of simple SIMD comparators simultaneously evaluates a complete shaft or 'beam' of voxels oriented along any of the three principal axes (x, y, or z). The output of the comparator network is a single voxel chosen on the basis of transparency, color, or depth values. By iterating through the other two dimensions, the complete volume can be scanned at interactive rates. The most recent version of the Cube architecture, Cube-4 [56], uses a more flexible memory organization to support a general

ray-casting model with arbitrary viewing angles, perspective projections, and trilinear interpolation of ray samples.

Knittel and Straßer [37] adopt a somewhat different approach with a VLSI-based volume rendering architecture intended for desktop implementation. Memory is organized into eight banks in order to provide parallel access to the sets of neighboring voxels which are needed for trilinear interpolation and gradient computations at sample points along rays. The basic design consists of a volume memory plus four specialized VLSI function units arranged in a pipeline. One function unit performs ray-casting and computes sample points along each ray, generating addresses into the volume memory. A second unit accepts the eight data values in the neighborhood of each sample and performs trilinear interpolation and gradient computations. A third unit computes color intensities for each sample point using a Phong illumination model, while the fourth unit composites the samples along each ray to produce a final pixel value. To obtain higher performance, the entire pipeline can be replicated, with subvolumes of the data being stored in each volume memory.

Recent developments in algorithms and computer architectures have combined to produce substantial performance increases for software-based volume renderers as well. One of the best examples is Lacroute's image-parallel renderer for shared-memory systems [39], which is capable of interactive frame rates on large datasets (256^3 and above) using commercially-available symmetric multiprocessors. Lacroute employs an optimized shear-warp rendering algorithm [38] which exploits both image-space and object-space coherence and incorporates demand-driven dynamic load balancing.

While the majority of volume rendering algorithms are designed for use with simple rectilinear grids, many scientific and engineering applications rely on more complex discretizations, including curvilinear, unstructured, and multi-block grids. This has prompted the development of several specialized volume renderers. For non-rectilinear and multi-block grids, Challinger developed an image-parallel shared-memory algorithm and tested it on the BBN TC-2000 [7]. While the rendering phase showed good speedups, a sorting step is needed to assign cell faces to image tiles, and this, along with load imbalances, tended to limit performance. More recently, Ma developed a distributed-memory volume renderer for unstructured grids [47], and implemented it on the Intel Paragon. He also noted performance limitations due to load imbalances. Together, these results suggest that additional work is needed to develop scalable volume rendering strategies for complex grids.

5.3. Ray-tracing

Due to its computational expense, its ability to produce realistic images, and its lack of support in commercial graphics architectures, ray-tracing was an early and frequently-addressed topic in parallel rendering. The SIGHT architecture [52] is one example of a system which was designed specifically to support parallel ray-tracing. The image space is partitioned across processors, with each processor responsible for tracing those rays which emanate from its local pixels. Interprocessor communication is largely avoided by replicating the object database in each processor's memory. An additional

level of parallelism is achieved through the use of multiple floating-point arithmetic units in each processing element to speed up the ray intersection calculations.

On general-purpose parallel systems, the majority of the execution time in ray tracing is spent on calculating the intersections of rays with objects in the scene. When the object data can be shared or replicated over the processors, a task distribution based on an image space subdivision will generally be very efficient. When the data size is larger and has to be distributed over the processors, either static or demand-driven task assignment can be used, both of which introduce additional communication and scheduling overheads.

In the static approach the ray tasks are allocated to the processors that contain the relevant data, and rays are communicated from one processor to another as needed. In the demand-driven approach the ray tasks are delegated to processors on request, which then have to fetch the needed object data, introducing extra communication. The amount of communication can be reduced by caching object data, in effect exploiting coherence in the scene. The static approach can handle arbitrarily large models but balancing workloads among processors is very difficult, since the cost of calculating ray/object intersections and evaluating secondary rays varies depending on the type and distribution of objects within the scene. The demand-driven approach has turned out to be rather efficient even with limited cache sizes [2,27]. With larger caches even complex models can be rendered successfully [63].

A hybrid load-balancing scheme for distributed-memory MIMD architectures was developed independently by Salmon and Goldsmith [62] and Caspary and Scherson [6]. With this approach the object data is organized using a hierarchical spatial subdivision, a well-known technique employed by sequential ray-tracers to reduce the search space for intersection testing. The upper part of the hierarchy is replicated on every processor, while the lower parts (which comprise the bulk of the object data) are distributed among the processors. This results in two distinct types of tasks: one which performs intersection calculations in the upper hierarchy (ray traversal), and another which performs the same calculations for the local data (ray-object intersection). Because the upper-level hierarchy is available everywhere, any processor in the system can perform the initial intersection tests on any ray, effectively decoupling the image-space and object-space partitionings.

A similar hybrid strategy has been adopted for use in stochastic ray tracing with explicit sampling of the diffuse reflectance [31,61]. In this method data coherence is almost completely lost, severely impacting the performance of caching schemes. However, for this application a different task distribution is needed: non-coherent ray tasks are assigned statically to provide a basic load that is adjusted by demand-driven tasks that execute the coherent ray tasks (mainly the primary rays and the shadow rays).

5.4. Radiosity renderers

Radiosity methods produce exceptionally realistic illumination of enclosed spaces by computing the transfer of light energy among all of the surfaces in the environment. Strictly speaking, radiosity is an illumination technique, rather than a complete rendering method. However, radiosity methods are among the most computationally-intensive

procedures in computer graphics, making them an obvious candidate for parallel processing. Because the quality of a radiosity solution depends in part on the resolution used to compute energy transfers, the polygons which describe objects are typically subdivided into small patches. In radiosity methods, the primary expense arises in generating the geometric *form factors* which are used to compute energy transfers among patches. Hence, parallel implementations have focused on speeding up this portion of the computation.

Although radiosity solutions can be computed directly by solving the system of equations which describes the energy transfers between surfaces, all of the form factors must be generated first, resulting in lengthy solution times which preclude interactive use. For this reason, an alternate iterative approach known as *progressive refinement* [12] has become popular. In this technique, the patch with the highest energy level at each iteration is selected as the shooting patch, and energy is transferred from it to other patches in the environment. This process repeats until the maximum level of untransmitted energy drops below some specified threshold. In this way, an initial approximation of the global illumination can be computed relatively quickly, with subsequent refinements resulting in incremental improvements to the image quality.

Many of the parallel radiosity methods described in the literature attempt to speed up the progressive refinement process by computing energy transfers from several shooting patches in parallel (i.e., several iterations are performed simultaneously) [5,8,22,54,59,60]. Because the time to complete an iteration can vary considerably depending on the geometric relationships between patches, load imbalance can seriously degrade overall performance. Several implementations compensate for this using a demand-driven strategy in which multiple worker processes independently compute form factors for different shooting patches [54,59,60]. With this strategy, the complete patch database is usually replicated on every processor, and a separate master process picks shooting patches and completes the energy transfers using vectors of form factors generated by the workers. This approach has several drawbacks, including a lack of data scalability for complex scenes and the tendency for the master process to become a bottleneck as the number of workers increases.

The alternative is to distribute the patch database and radiosity computations across all of the processors. This strategy necessitates global communication in order to compute form factors and complete the energy transfers from shooting patches. Çapın et al. [5] used a simple ring network, circulating patch data and local results from processor to processor in pipelined fashion to obtain global solutions. Because performance is limited at each step of the computation by the slowest processor, load imbalances can have a profound effect on overall performance. By ensuring that patches belonging to the same object are scattered across processors, variations in workload due to spatial locality are minimized, and a rough static load balance is maintained. Additional examples of radiosity renderers which use distributed databases can be found in [8,22].

The strategy of processing multiple shooting patches in parallel perturbs the order of execution found in the sequential version of the progressive refinement algorithm, and this can lead to slower convergence, partially offsetting the benefits of parallel execution. The effect is minimal when only a few shooting patches are active [3], but becomes more pronounced as the number of processors increases [5]. In order to exploit massive

parallelism, a different approach is needed. Varshney and Prins developed a SIMD radiosity renderer for a MasPar MP-1 with 4096 processing elements [65]. As in Çapın's algorithm, patches are distributed uniformly among the processors. At each iteration, a global reduction operation is used to find the shooting patch with the highest energy, thus maintaining the convergence properties of the sequential algorithm. All of the other patches in the environment are then scan-converted onto the shooting patch, and form factors are obtained by accumulating the resulting pixel values. Energy transfers are performed in parallel using the results of the form factor computations. While this algorithm is able to exploit the massive parallelism of its target architecture, load imbalances in the scan conversion phase are found to be significant, and further static or dynamic load balancing measures appear to be in order.

5.5. *Terrain rendering*

In terrain rendering, the problem is to generate a plausible representation of a real or imaginary landscape as viewed from some point on or above the surface. Typically the viewpoint will change over time, often under interactive control, and in some applications additional objects such as vegetation, buildings, or vehicles must be included in the scene. Terrain rendering techniques have been widely applied in areas such as flight simulation, scientific data analysis and exploration, and the creation of virtual landscapes for entertainment or artistic purposes. The need for high-quality images, high frame rates, rapid response to changes in viewpoint, and the ability to navigate through large datasets has stimulated the development of parallel terrain rendering techniques.

Although a variety of methods can be used to render terrain, most of the parallel techniques described in the literature begin with an aerial or satellite image of an actual planetary surface. This image is registered with a separate elevation dataset of the same region, typically represented by a two-dimensional grid with an associated height field. The problem, then, is to assign an elevation value to pixels in the input image and project them onto a display with hidden surfaces eliminated. This technique is known as *forward projection*, in contrast to ray-casting methods which begin at the eye point and project rays through display pixels into the scene. With the forward projection approach, care must be taken to account for the mismatch between input and output image projections, filling in gaps in the output image and compositing input pixels which map to the same location in screen space.

Kaba et al. [33,34] developed data-parallel terrain rendering techniques for the Princeton Engine, a programmable SIMD system originally designed for real-time processing of digital video [9]. Their methods utilize an object-parallel task decomposition, distributing the input image and elevation datasets among the processors by assigning complete columns of pixels to processors. Before projecting the data onto the display, it must be rotated and scaled to account for the viewing direction and altitude. This is accomplished efficiently by decomposing the necessary transformations into a sequence of shear, shear/scale, and transpose operations. Hidden surfaces are eliminated by scanning the transformed data from front-to-back, one horizontal scanline at a time. The pixels in each scanline are processed in parallel. With each pass, a horizon line is updated; only those pixels which lie above the current horizon line will be visible.

The system is capable of rendering terrain fly-overs at 30 frames/s using 512×512 resolution and 8-bit color, or 15 frames/s with 24-bit color.

At the Jet Propulsion Laboratory, Li and Curkendall have developed techniques for rendering planetary surfaces using a variety of large-scale distributed-memory architectures, including Intel's iPSC/860, Delta, and Paragon systems, and Cray's T3D. Like Kaba, they use surface images registered with elevation data, and project object-space pixels into screen space. While their initial methods [42] partitioned the input data by horizontal slices and assigned them to processors in interleaved fashion, more recent implementations use rectangular tiles with either interleaved [44] or random [43] assignment. The random strategy provides a measure of stochastic load balancing, reducing sensitivity to hot spots in the data which may occur when the view zooms in on small terrain regions.

While the two previous examples both exploited data parallelism, other approaches are certainly possible. Wright and Hsieh [71] describe a pipelined terrain rendering algorithm which has been implemented in hardware. As in the other examples, a forward projection technique is used to map from object to image space, but the surface data and objects in the scene are represented as specialized volume elements (voxels). The architecture consists of two concatenated pipelines, one for voxel processing and one for pixel processing. The voxel pipeline scans through the database, generating columns of voxels which are illuminated, transformed into viewing coordinates, and rasterized into pixels. The pixel pipeline projects pixels from polar viewing coordinates into screen space, performs haze, translucency, and z-buffering calculations, and normalizes pixel intensities. A variety of techniques are applied at different levels in the pipeline to reduce temporal and spatial aliasing. The hardware implementation is capable of rendering 10 frames/s at 384×384 resolution, a speedup of more than three orders of magnitude over a software-based sequential implementation.

6. Summary

Demanding applications such as real-time simulation, animation, virtual reality, photo-realistic imaging, and scientific visualization all benefit from the use of parallelism to increase rendering performance. Indeed, these applications have been primary motivators in the development of parallel rendering methods. We have examined many of the general principles and algorithmic approaches which apply to computer graphics rendering on parallel architectures, and surveyed representative implementations in both hardware and software.

As our discussion illustrates, the algorithm or architecture designer is faced with a wide range of implementation strategies and a complex series of tradeoffs. A successful parallel renderer must take into account application requirements, architectural parameters, and algorithmic characteristics. As the rapidly growing performance of rendering systems indicates, there have been numerous successes, but these are balanced by other attempts which have fallen short. Many challenges remain, particularly in the areas of scalability, load balancing, communication, and image assembly. Finding solutions to

these problems will motivate further explorations in parallel rendering as computer architectures advance into the teraflops regime.

Acknowledgements

The author would like to thank Tony Apodaca, Chuck Hansen, Scott Whitman, Craig Wittenbrink, Sam Uselton, and the staff of NASA Langley's Technical Library for their assistance in researching this article. Erik Jansen provided extensive editorial assistance and contributed to the section on ray-tracing. David Banks and John van Rosendale offered valuable comments on an early version of the manuscript.

References

[1] K. Akeley, RealityEngine graphics, in: Comp. Graphics Proc. Ann. Conf. Series, ACM SIGGRAPH, 1993, pp. 109–116.
[2] D. Badouel, K. Bouatouch, T. Priol, Distributing data and control for ray tracing in parallel, IEEE Comput. Graph. Appl. 14 (4) (1994) 69–77.
[3] D.R. Baum, J.M. Winget, Real time radiosity through parallel processing and hardware acceleration, in: Proc. 1990 Symp. on Interactive 3D Graphics, Comp. Graphics, Vol. 24(2), ACM SIGGRAPH, 1990, pp. 67–75.
[4] R.E. Benner, Parallel graphics algorithms on a 1024-processor hypercube, in: Proc. 4th Conf. on Hypercubes, Concurrent Computers, and Applications, Vol. I, Monterey, CA, 1989, pp. 133–140.
[5] T.K. Çapın, C. Aykanat, B. Özgüç, Progressive refinement radiosity on ring-connected multicomputers, in: Proc. 1993 Parallel Rendering Symp., ACM Press, 1993, pp. 71–76.
[6] E. Caspary, I.D. Scherson, A self-balanced parallel ray-tracing algorithm, in: P.M. Dew, R.A. Earnshaw, T.R. Heywood (Eds.), Parallel Processing for Computer Vision and Display, Addison-Wesley, 1989, pp. 408–419.
[7] J. Challinger, Scalable parallel volume raycasting for nonrectilinear computational grids, in: Proc. 1993 Parallel Rendering Symp., ACM Press, 1993, pp. 81–88.
[8] A.G. Chalmers, D.J. Paddon, Parallel processing of progressive refinement radiosity methods, in: Photorealistic Rendering in Computer Graphics, Proc. 2nd Eurographics Workshop on Rendering, Springer-Verlag, 1991, pp. 149–159.
[9] D. Chin, J. Passe, F. Bernard, H. Taylor, S. Knight, The Princeton engine: A real-time video system simulator, IEEE Trans. Consumer Electron. 34 (2) (1988) 285–297.
[10] J. Clark, A VLSI geometry processor for graphics, Computer 13 (7) (1980) 59–68.
[11] J. Clark, The geometry engine: A VLSI geometry system for graphics, Comput. Graph. 16 (3) (1982) 127–133.
[12] M.F. Cohen, S.E. Chen, J.R. Wallace, D.P. Greenberg, A progressive refinement approach to fast radiosity image generation, Comput. Graph. 22 (4) (1988) 75–84.
[13] Convex Exemplar System Overview, Convex Computer Corporation, Richardson, TX, 1994.
[14] M. Cox, P. Hanrahan, A distributed snooping algorithm for pixel merging, IEEE Parallel Dist. Tech. 2 (2) (1994) 30–36.
[15] M.B. Cox, Algorithms for parallel rendering, Ph.D. dissertation, Department of Computer Science, Princeton University, 1995.
[16] Cray Animation Theater, Cray Research, Inc., Eagan, MN, 1994.
[17] T.W. Crockett, Design considerations for parallel graphics libraries, in: Proc. Intel Supercomputer Users Group, Annual North America Users Conf., San Diego, 1994, pp. 3–14.
[18] T.W. Crockett, Parallel rendering, Rep. 95-31, NASA CR-195080, Institute for Computer Applications in Science and Engineering, Hampton, VA, 1995.
[19] T.W. Crockett, Beyond the renderer: Software architecture for parallel graphics and visualization, in: A.

Chalmers, E. Jansen (Eds.), Proc. 1st Eurographics Workshop on Parallel Graphics and Visualisation, α lpha Books, Bristol, UK, 1996, pp. 1–15.
[20] T.W. Crockett, T. Orloff, Parallel polygon rendering for message-passing architectures, IEEE Parallel Dist. Tech. 2 (2) (1994) 17–28.
[21] D. Ellsworth, A new algorithm for interactive graphics on multicomputers, IEEE Comput. Graph. Appl. 14 (4) (1994) 33–40.
[22] M. Feda, W. Purgathofer, Progressive refinement radiosity on a transputer network, in: Photorealistic Rendering in Computer Graphics, Proc. 2nd Eurographics Workshop on Rendering, Springer-Verlag, 1991, pp. 139–148.
[23] J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes, Computer Graphics, Principles and Practice, 2nd Ed. in C, Addison-Wesley, 1996.
[24] H. Fuchs, J. Poulton, Pixel-planes: A VLSI-oriented design for a raster graphics engine, VLSI Des. Q3 (1981) 20–28.
[25] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, L. Israel, Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories, Comput. Graph. 23 (3) (1989) 79–88.
[26] C. Giertsen, J. Petersen, Parallel volume rendering on a network of workstations, IEEE Comput. Graph. Appl. 13 (6) (1993) 16–23.
[27] S.A. Green, D.J. Paddon, A highly flexible multiprocessor solution for ray tracing, Visual Comput. 6 (2) (1990) 62–73.
[28] C.D. Hansen, M. Krogh, W. White, Massively parallel visualization: parallel rendering, in: Proc. 7th SIAM Conf. Parallel Proc. for Sci. Comp., SIAM, 1995, pp. 790–795.
[29] W.M. Hsu, Segmented ray casting for data parallel volume rendering, in: Proc. 1993 Parallel Rendering Symp., ACM Press, 1993, pp. 7–14.
[30] J.P. Hughes, HIPPI, in: Proc. 17th Conf. Local Comp. Networks, IEEE CS Press, 1992, pp. 346–354.
[31] F.W. Jansen, A. Chalmers, Realism in real time?, in: Proc. 4th Eurographics Workshop on Rendering (1993) 1–20.
[32] D.W. Jensen, D.A. Reed, A performance analysis exemplar parallel ray tracing , Concurrency: Pract. Exper. 4 (2) (1992) 119–141.
[33] J. Kaba, J. Matey, G. Stoll, H. Taylor, P. Hanrahan, Interactive terrain rendering and volume visualization on the Princeton Engine, in: Proc. Visualization '92, IEEE CS Press, 1992, pp. 349–355.
[34] J. Kaba, J. Peters, A pyramid-based approach to interactive terrain visualization, in: Proc. 1993 Parallel Rendering Symp., ACM Press, 1993, pp. 67–70.
[35] A. Kaufman, R. Bakalash, Memory and processing architecture for 3D voxel-based imagery, IEEE Comput. Graph. Appl. 8 (6) (1988) 10–23.
[36] R.E. Kessler, J.L. Schwarzmeier, Cray T3D: A new dimension for Cray Research, in: Digest of Papers, COMPCON Spring '93, IEEE CS Press, 1993, pp. 176–182.
[37] G. Knittel, W. Straßer, A compact volume rendering accelerator, in: Proc. 1994 Symp. on Volume Visualization, ACM SIGGRAPH, 1994, pp. 67–74.
[38] P. Lacroute, M. Levoy, Fast volume rendering using a shear-warp factorization of the viewing transformation, in: Comp. Graphics Proc., Ann. Conf. Series, ACM SIGGRAPH, 1994, pp. 451–458.
[39] P. Lacroute, Analysis of a parallel volume rendering system based on the shear-warp factorization, IEEE Trans. Visual. Comput. Graph. 2 (3) (1996) 218–231.
[40] T.-Y. Lee, C.S. Raghavendra, J.N. Nicholas, Image composition schemes for sort-last polygon rendering on 2D mesh architectures, IEEE Trans. Visual. Comput. Graph. 2 (3) (1996) 202–217.
[41] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, M.S. Lam, The Stanford Dash multiprocessor, Computer 25 (3) (1992) 63–79.
[42] P.P. Li, D.W. Curkendall, Parallel three dimensional perspective rendering, in: Proc. 2nd European Workshop on Parallel Comp., 1992, pp. 320–331.
[43] P. Li, D. Curkendall, W. Duquette, H. Henry, Interactive scientific visualization on massively parallel processors, in: CSCC Update: The Newsletter of the Concurrent Supercomputing Consortium, Vol. 13(7), Caltech CCSF, Pasadena, CA, 1994, pp. 4–6.
[44] P.P. Li, W.H. Duquette, D.W. Curkendall, RIVA: A versatile parallel rendering system for interactive scientific visualization, IEEE Trans. Visual. Comput. Graph. 2 (3) (1996) 186–201.

[45] T.T.Y. Lin, M. Slater, Stochastic ray tracing using SIMD processor arrays, Visual Comput. 7 (4) (1991) 187–199.

[46] K.-L. Ma, J.S. Painter, C.D. Hansen, M.F. Krogh, Parallel volume rendering using binary-swap compositing, IEEE Comput. Graph. Appl. 14 (4) (1994) 59–68.

[47] K.-L. Ma, Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures, in: Proc. 1995 Parallel Rendering Symp., ACM Press, 1995, pp. 23–30.

[48] P. Mackerras, B. Corrie, Exploiting data coherence to improve parallel volume rendering, IEEE Parallel Dist. Tech. 2 (2) (1994) 8–16.

[49] S. Molnar, J. Eyles, J. Poulton, PixelFlow: High-speed rendering using image composition, Comput. Graph. 26 (2) (1992) 231–240.

[50] S. Molnar, M. Cox, D. Ellsworth, H. Fuchs, A sorting classification of parallel rendering, IEEE Comput. Graph. Appl. 14 (4) (1994) 23–32.

[51] C. Mueller, The sort-first rendering architecture for high-performance graphics, in: Proc. 1995 Symp. Interactive 3D Graphics, ACM SIGGRAPH, 1995, pp. 75–84.

[52] T. Naruse, M. Yoshida, T. Takahashi, S. Naito, SIGHT — A dedicated computer graphics machine, Comput. Graph. Forum 6 (4) (1987) 327–334.

[53] U. Neumann, Communication costs for parallel volume-rendering algorithms, IEEE Comput. Graph. Appl. 14 (4) (1994) 49–58.

[54] A. Ng, M. Slater, A multiprocessor implementation of radiosity, Comput. Graph. Forum 12 (5) (1993) 329–342.

[55] F.A. Ortega, C.D. Hansen, J.P. Ahrens, Fast data parallel polygon rendering, in: Proc. Supercomputing '93, IEEE CS Press, 1993, pp. 709–718.

[56] H. Pfister, A. Kaufman, Cube-4 — a scalable architecture for real-time volume rendering, in: Proc. 1996 Symp. on Volume Visualization, ACM SIGGRAPH, 1996, pp. 47–54.

[57] PhotoRealistic RenderMan Toolkit v3.5 Reference Manual, Pixar, Richmond, CA, 1994.

[58] M. Potmesil, E.M. Hoffert, The pixel machine: A parallel image computer, Comput. Graph. 23 (3) (1989) 69–78.

[59] C. Puech, F. Sillion, C. Vedel, Improving interaction with radiosity-based lighting simulation programs, Comput. Graph. 24 (2) (1990) 51–57.

[60] R.J. Recker, D.W. George, D.P. Greenberg, Acceleration techniques for progressive refinement radiosity, Comput. Graph. 24 (2) (1990) 59–66.

[61] E. Reinhard, F.W. Jansen, Hybrid scheduling for efficient ray tracing of complex images, in: M. Chen, P. Townsend, J.A. Vince (Eds.), Proc. Intl. Workshop on High Performance Computing for Computer Graphics and Visualisation, Springer-Verlag, 1996, pp. 78–87.

[62] J. Salmon, J. Goldsmith, A hypercube ray-tracer, in: G.C. Fox (Ed.), Proc. 3rd Conf. Hypercube Concurrent Comps. and Apps., Vol. II, Applications, ACM Press, 1988, pp. 1194–1206.

[63] J.P. Singh, A. Gupta, M. Levoy, Parallel visualization algorithms: Performance and architectural implications, Computer 27 (7) (1994) 45–55.

[64] I.E. Sutherland, R.F. Sproull, R.A. Schumacker, A characterization of ten hidden-surface algorithms, Comput. Surv. 6 (1) (1974) 1–55.

[65] A. Varshney, J.F. Prins, An environment-projection approach to radiosity for mesh-connected computers, in: Proc. 3rd Eurographics Workshop on Rendering, Springer-Verlag, 1992, pp. 271–281.

[66] R.J. Vetter, ATM concepts, architectures, and protocols, Commun. ACM 38 (2) (1995) 30–38.

[67] B. Wei, G. Stoll, D.W. Clark, E.W. Felten, K. Li, Synchronization for a multi-port frame buffer on a mesh-connected multicomputer, Proc. 1995 Parallel Rendering Symp., ACM Press, 1995, pp. 81–88.

[68] D.S. Whelan, Animac: A multiprocessor architecture for real-time computer animation, Ph.D. dissertation, California Institute of Technology, 1985.

[69] S. Whitman, Multiprocessor Methods for Computer Graphics Rendering, Jones and Bartlett, Boston, 1992.

[70] S. Whitman, Dynamic load balancing for parallel polygon rendering, IEEE Comput. Graph. Appl. 14 (4) (1994) 41–48.

[71] J.R. Wright, J.C.L. Hsieh, A voxel-based forward projection algorithm for rendering surface and volumetric data, in: Proc. Visualization '92, IEEE CS Press, 1992, pp. 340–348.

[72] D. Zareski, B. Wade, P. Hubbard, P. Shirley, Efficient parallel global illumination using density estimation, in: Proc. 1995 Parallel Rendering Symp., ACM Press, 1995, pp. 47–54.

XP-002360078

InfiniteReality: A Real-Time Graphics System

John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal
Silicon Graphics Computer Systems

ABSTRACT

The InfiniteReality™ graphics system is the first general-purpose workstation system specifically designed to deliver 60Hz steady frame rate high-quality rendering of complex scenes. This paper describes the InfiniteReality system architecture and presents novel features designed to handle extremely large texture databases, maintain control over frame rendering time, and allow user customization for diverse video output requirements. Rendering performance expressed using traditional workstation metrics exceeds seven million lighted, textured, antialiased triangles per second, and 710 million textured antialiased pixels filled per second.

CR Categories and Subject Descriptors: I.3.1 [Computer Graphics]: Hardware Architecture; I.3.3 [Computer Graphics]: Picture/Image Generation

1 INTRODUCTION

This paper describes the Silicon Graphics InfiniteReality architecture which is the highest performance graphics workstation ever commercially produced. The predecessor to the InfiniteReality system, the RealityEngine™, [Akel93] was the first example of what we term a third-generation graphics system. As a third-generation system, the target capability of the RealityEngine was to render lighted, smooth shaded, depth buffered, texture mapped, antialiased triangles. The level of realism achieved by RealityEngine graphics was well-matched to the application requirements of visual simulation (both flight and ground based simulation), location based entertainment [Paus96], defense imaging, and virtual reality. However, application success depends on two areas: the ability to provide convincing levels of realism and to deliver real-time performance of constant scene update rates of 60Hz or more. High frame rates reduce interaction latency and minimize symptoms of motion sickness in visual simulation and virtual reality applications. If frame rates are not constant, the visual integrity of the simulation is compromised.

InfiniteReality is also an example of a third-generation graphics system in that its target rendering quality is similar to that of RealityEngine. However, where RealityEngine delivered performance in the range of 15-30 Hz for most applications, the fundamental design goal of the InfiniteReality graphics system is to deliver real-time performance to a broad range of applications. Furthermore, the goal is to deliver this performance far more economically than competitive solutions.

Author contacts: {montrym | drb | dignam | migdal}@sgi.com

Most of the features and capabilities of the InfiniteReality architecture are designed to support this real-time performance goal. Minimizing the time required to change graphics modes and state is as important as increasing raw transformation and pixel fill rate. Many of the targeted applications require access to very large textures and/or a great number of distinct textures. Permanently storing such large amounts of texture data within the graphics system itself is not economically viable. Thus methods must be developed for applications to access a "virtual texture memory" without significantly impacting overall performance. Finally, the system must provide capabilities for the application to monitor actual geometry and fill rate performance on a frame by frame basis and make adjustments if necessary to maintain a constant 60Hz frame update rate.

Aside from the primary goal of real-time application performance, two other areas significantly shaped the system architecture. First, this was Silicon Graphics' first high-end graphics system to be designed from the beginning to provide native support for OpenGL™. To support the inherent flexibility of the OpenGL architecture, we could not take the traditional approach for the real-time market of providing a black-box solution such as a flight simulator [Scha83].

The InfiniteReality system is fundamentally a sort-middle architecture [Moln94]. Although interesting high-performance graphics architectures have been implemented using a sort-last approach [Moln92][Evan92], sort-last is not well-suited to supporting OpenGL framebuffer operations such as blending. Furthermore, sparse sort-last architectures make it difficult to rasterize primitives into the framebuffer in the order received from the application as required by OpenGL.

The second area that shaped the graphics architecture was the need for the InfiniteReality system to integrate well with two generations of host platforms. For the first year of production, the InfiniteReality system shipped with the Onyx host platform. Currently, the InfiniteReality system integrates into the Onyx2 platform. Not only was the host to graphics interface changed between the two systems, but the I/O performance was also significantly improved. Much effort went into designing a graphics system that would adequately support both host platforms.

The remainder of the paper is organized as follows. The next section gives an architectural overview of the system. Where appropriate, we contrast our approach to that of the RealityEngine system. Section 3 elaborates on novel functionality that enables real-time performance and enhanced video capabilities. Section 4 discusses the performance of the system. Finally, concluding remarks are made in Section 5.

2 ARCHITECTURE

It was a goal to be able to easily upgrade Onyx RealityEngine systems to InfiniteReality graphics. Accordingly, the physical partitioning of the InfiniteReality boardset is similar to that of

RealityEngine; there are three distinct board types: the Geometry, Raster Memory, and Display Generator boards (Figure 1).

The Geometry board comprises a host computer interface, command interpretation and geometry distribution logic, and four Geometry Engine processors in a MIMD arrangement. Each Raster Memory board comprises a single fragment generator with a single copy of texture memory, 80 image engines, and enough framebuffer memory to allocate 512 bits per pixel to a 1280x1024 framebuffer. The display generator board contains hardware to drive up to eight display output channels, each with its own video timing generator, video resize hardware, gamma correction, and digital-to-analog conversion hardware.

Systems can be configured with one, two or four raster memory boards, resulting in one, two, or four fragment generators and 80, 160, or 320 image engines.

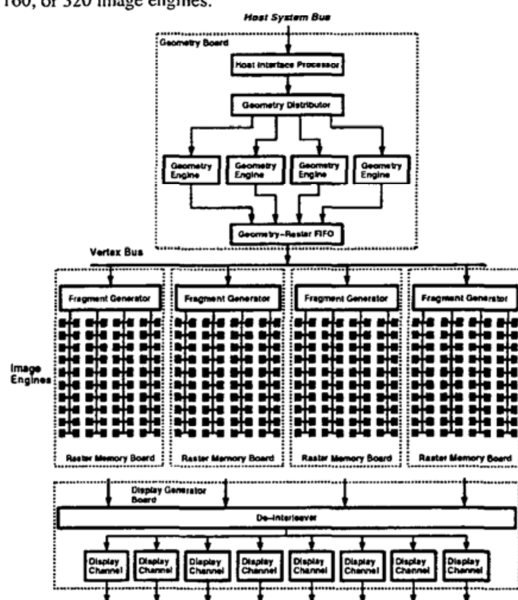


Figure 1: Board-level block diagram of the maximum configuration with 4 Geometry Engines, 4 Raster Memory boards, and a Display Generator board with 8 output channels.

2.1 Host Interface

There were significant system constraints that influenced the architectural design of InfiniteReality. Specifically, the graphics system had to be capable of working on two generations of host platforms. The Onyx2 differs significantly from the shared memory multiprocessor Onyx in that it is a distributed shared memory multiprocessor system with cache-coherent non-uniform memory access. The most significant difference in the graphics system design is that the Onyx2 provides twice the host-to-graphics bandwidth (400MB/sec vs. 200MB/sec) as does Onyx. Our challenge was to design a system that would be matched to the host-to-graphics data rate of the Onyx2, but still provide similar performance with the limited I/O capabilities of Onyx.

We addressed this problem with the design of the display list subsystem. In the RealityEngine system, display list processing had been handled by the host. Compiled display list objects were stored in host memory, and one of the host processors traversed the display list and transferred the data to the graphics pipeline using programmed I/O (PIO).

With the InfiniteReality system, display list processing is handled in two ways. First, compiled display list objects are stored in host memory in such a way that leaf display objects can be "pulled" into the graphics subsystem using DMA transfers set up by the Host Interface Processor (Figure 1). Because DMA transfers are faster and more efficient than PIO, this technique significantly reduces the computational load on the host processor so it can be better utilized for application computations. However, on the original Onyx system, DMA transfers alone were not fast enough to feed the graphics pipe at the rate at which it could consume data. The solution was to incorporate local display list processing into the design.

Attached to the Host Interface Processor is 16MB of synchronous dynamic RAM (SDRAM). Approximately 15MB of this memory is available to cache leaf display list objects. Locally stored display lists are traversed and processed by an embedded RISC core. Based on a priority specified using an OpenGL extension and the size of the display list object, the OpenGL display list manager determines whether or not a display list object should be cached locally on the Geometry board. Locally cached display lists are read at the maximum rate that can be consumed by the remainder of the InfiniteReality pipeline. As a result, the local display list provides a mechanism to mitigate the host to graphics I/O bottleneck of the original Onyx. Note that if the total size of leaf display list objects exceeds the resident 15MB limit, then some number of objects will be pulled from host memory at the reduced rate.

2.2 Geometry Distribution

The Geometry Distributor (Figure 1) passes incoming data and commands from the Host Interface Processor to individual Geometry Engines for further processing. The hardware supports both round-robin and least-busy distribution schemes. Since geometric processing requirements can vary from one vertex to another, a least-busy distribution scheme has a slight performance advantage over round-robin. With each command, an identifier is included which the Geometry-Raster FIFO (Figure 1) uses to recreate the original order of incoming primitives.

2.3 Geometry Engines

When we began the design of the InfiniteReality system, it became apparent that no commercial off-the-shelf floating point processors were being developed which would offer suitable price/performance. As a result, we chose to implement the Geometry Engine Processor as a semicustom application specific integrated circuit (ASIC).

The heart of the Geometry Engine is a single instruction multiple datapath (SIMD) arrangement of three floating point cores, each of which comprises an ALU and a multiplier plus a 32 word register

file with two read and two write ports (Figure 2). A 2560 word on-chip memory holds elements of OpenGL state and provides scratch storage for intermediate calculations. A portion of the working memory is used as a queue for incoming vertex data. Early simulations of microcode fragments confirmed that high bandwidth to and from this memory would be required to get high utilization of the floating point hardware. Accordingly, each of the three cores can perform two reads and one write per instruction to working memory. Note that working memory allows data to be shared easily among cores. A dedicated float-to-fix converter follows each core, through which one floating point result may be written per instruction.

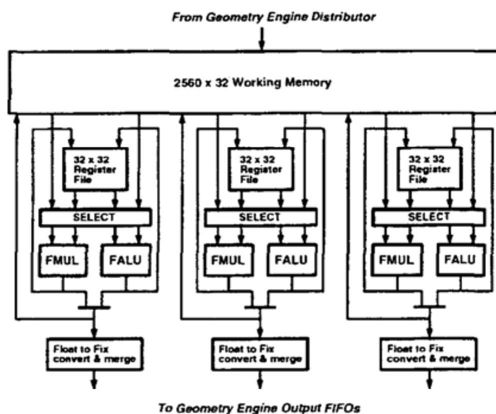


Figure 2: Geometry Engine

We used a very simple scheduler to evaluate the performance effect of design trade-offs on critical microcode fragments. One of the trade-offs considered was the number of pipeline stages in the floating point arithmetic blocks. As we increased the depth of the pipeline from one to four stages, the machine's clock speed and throughput increased. For more than four stages, even though the clock speed improved, total performance did not because our code fragments did not have enough unrelated operations to fill the added computation slots.

Quite often machine performance is expressed in terms of vertex rates for triangles in long strips whereas application performance is much more likely to be determined by how well a system handles very short strips, with frequent mode changes. The problem of accelerating mode changes and other non-benchmark operations has enormous impact on the microcode architecture, which in turn influences aspects of the instruction set architecture.

To accelerate mode change processing, we divide the work associated with individual OpenGL modes into distinct code modules. For example, one module can be written to calculate lighting when one infinite light source is enabled, another may be tuned for one local point light source, and still another could handle a single spotlight. A general module exists to handle all cases which do not have a corresponding tuned module. Similarly, different microcode modules would be written to support other OpenGL modes such as texture coordinate generation or backface elimination. A table consisting of pointers to the currently active modules is maintained in GE working memory. Each vertex is processed by executing the active modules in the table-specified sequence. When a mode change occurs, the appropriate table entry is changed. Vertex processing time degrades slowly and predictably as additional operations are turned on, unlike microcode architectures which implement hyper-optimized fast paths for selected bundles of mode settings, and a slow general path for all other combinations.

Since microcode modules tend to be relatively short, it is desirable to avoid the overhead of basic-block preamble and postamble code. All fields necessary to launch and retire a given operation, including memory and register file read and write controls, are specified in the launching microinstruction.

2.4 Geometry-Raster FIFO

The output streams from the four Geometry Engines are merged into a single stream by the Geometry-Raster FIFO. A FIFO large enough to hold 65536 vertexes is implemented in SDRAM. The merged geometry engine output is written, through the SDRAM FIFO, to the Vertex Bus. The Geometry-Raster FIFO contains a 256-word shadow RAM which keeps a copy of the latest values of the Fragment Generator and Image Engine control registers. By eliminating the need for the Geometry Engines to retain shadowed raster state in their local RAMs, the shadow RAM permits raster mode changes to be processed by only one of the Geometry Engines. This improves mode change performance and simplifies context switching.

2.5 Vertex Bus

One of our most important goals was to increase transform-limited triangle rates by an order of magnitude over RealityEngine. Given our desire to retain a sort-middle architecture, we were forced to increase the efficiency of the geometry-raster crossbar by a factor of ten. Whereas the RealityEngine system used a *Triangle Bus* to move triangle parameter slope information from its Geometry Engines to its Fragment Generators, the InfiniteReality system employs a *Vertex Bus* to transfer only screen space vertex information. Vertex Bus data is broadcast to all Fragment Generators. The Vertex Bus protocol supports the OpenGL triangle strip and triangle fan constructs, so the Vertex Bus load corresponds closely to the load on the host-to-graphics bus. The Geometry Engine triangle strip workload is reduced by around 60 percent by not calculating triangle setup information. However, hardware to assemble screen space primitives and compute parameter slopes is now incorporated into the Fragment Generators.

2.6 Fragment Generators

In order to provide increased user-accessible physical texture memory capacity at an acceptable cost, it was our goal to have only one copy of texture memory per Raster Memory board. A practical consequence of this is that there is also only one fragment generator per raster board. Figure 3 shows the fragment generator structure.

Connected vertex streams are received and assembled into triangle

primitives. The Scan Converter (SC) and Texel Address Calculator (TA) ASICs perform scan conversion, color and depth interpolation, perspective correct texture coordinate interpolation and level-of-detail computation. Up to four fragments, corresponding to 2x2 pixel regions are produced every clock. Scan conversion is performed by directly evaluating the parameter plane equations at

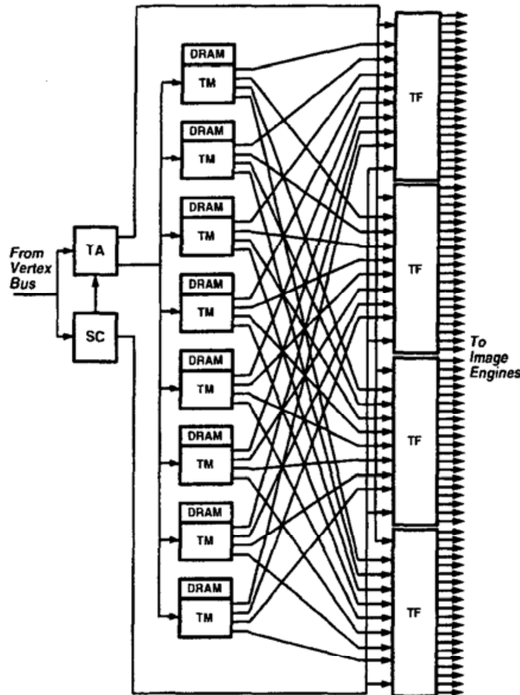


Figure 3: Fragment Generator

each pixel [Fuch85] rather than by using an interpolating DDA as was done in the RealityEngine system. Compared to a DDA, direct evaluation requires less setup time per triangle at the expense of more computation per pixel. Since application trends are towards smaller triangles, direct parameter evaluation is a more efficient solution.

Each texture memory controller (TM) ASIC performs the texel lookup in its four attached SDRAMs, given texel addresses from the TA. The TMs combine redundant texel requests from neighboring fragments to reduce SDRAM access. The TMs forward the resulting texel values to the appropriate TF ASIC for texture filtering, texture environment combination with interpolated color, and fog application. Since there is only one copy of the texture memory distributed across all the texture SDRAMs, there must exist a path from all 32 texture SDRAMs to all Image Engines. The TMs and TFs implement a two-rank omega network [Hwan84] to perform the required 32-to-80 sort.

2.7 Image Engines

Fragments output by a single Fragment Generator are distributed equally among the 80 Image Engines owned by that generator. Each Image Engine controls a single 256K x 32 SDRAM that comprises its portion of the framebuffer. Framebuffer memory per Image Engine is twice that of RealityEngine, so a single raster board system supports eight sample antialiasing at 1280 x 1024 or four sample antialiasing at 1920 x 1200 resolution.

2.8 Framebuffer Tiling

Three factors contributed to development of the framebuffer tiling scheme: the desire for load balancing of both drawing and video requests; the various restrictions on chip and board level packaging; and the requirement to keep on-chip FIFOs small.

In systems with more than one fragment generator, different fragment generators are each responsible for two-pixel wide vertical strips in framebuffer memory. If horizontal strips had been used instead, the resulting load imbalance due to display requests would have required excessively large FIFOs at the fragment generator inputs. The strip width is as narrow as possible to minimize the load imbalance due to drawing among fragment generators.

The Fragment Generator scan-conversion completes all pixels in a two pixel wide vertical strip before proceeding to the next strip for every primitive. To keep the Image Engines from limiting fill rate on large area primitives, all Image Engines must be responsible for part of every vertical strip owned by their Fragment Generator. Conversely, for best display request load balancing, all Image Engines must occur equally on every horizontal line. For a maximum system, the Image Engine framebuffer tiling repeat pattern is a rectangle 320 pixels wide by 80 pixels tall (320 is the number of Image Engines in the system and 80 is the number of Image Engines on one Raster Memory board).

2.9 Display Hardware

Each of the 80 Image Engines on the Raster Memory boards drives one or two bit serial signals to the Display Generator board. Two wires are driven if there is only one Raster Memory board, and one wire is driven if there are two or more. Unlike RealityEngine, both the number of pixels sent per block and the aggregate video band width of 1200 Mbytes/sec are independent of the number of Raster Memory boards. Four ASICs on the display board (Figure 4) de-serialize and de-interleave the 160 bit streams into RGBA10, RGB12, L16, Stereo Field Sequential (FS), or color indexes. The cursor is also injected at this point. A total of 32,768 color index map entries are available.

Color component width is maintained at 12 bits through the gamma table outputs. A connector site exists with a full 12 bit per component bus, which is used to connect video option boards. Option boards support the Digital Video Standard CCIR 601 and a digital pixel output for hardware-in-the-loop applications.

The base display system consists of two channels, expandable to eight. Each display channel is autonomous, with independent

video timing and image resizing capabilities. The final channel output drives eight-bit digital-to-analog converters which can run up to a 220Mhz pixel clock rate. Either RGB or Left/Right Stereo Field Sequential is available from each channel.

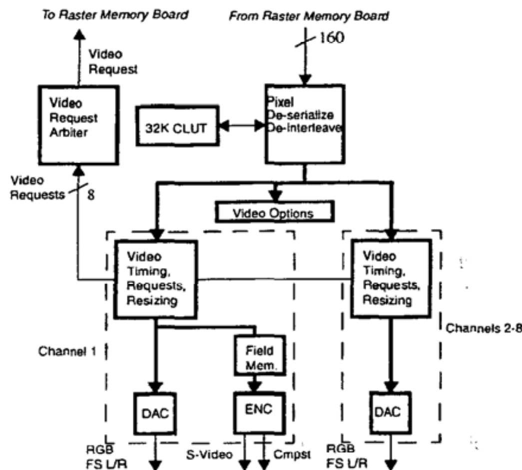


Figure 4: Display System

Video synchronization capabilities were expanded to support independent timing per channel (Figure 5). Swap events are constrained to happen during a common interval. Three different methods are used to synchronize video timing to external video sources. *Framelocking* is the ability to rate lock, using line rate dividers, two different video outputs whose line rates are related by small integer ratios. Line rate division is limited by the programmability of the phase-locked-loop gain and feedback parameters and the jitter spectrum of the input genlock source. The start of a video frame is detected by programmable sync pattern recognition hardware. Disparate source and displayed video formats which exceed the range of framelock are vertically locked by simply performing an asynchronous *frame reset* of the display video timing hardware. In this instance, the pixel clock is created by multiplying an oscillator clock. Identical formats may be *genlocked*. With frame lock or genlock, the frame reset from the pattern recognition hardware will be synchronous, and therefore cause no disturbance of the video signal being sent to the monitor.

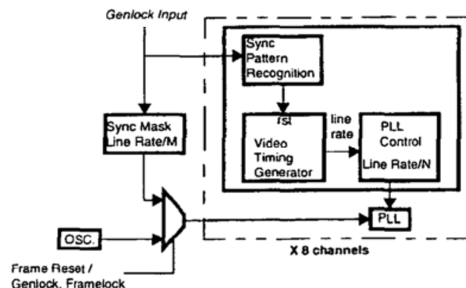


Figure 5: Video Synchronization

Certain situations require the synchronization of drawing between separate graphics systems. This is required in visual simulation installations where multiple displays are each driven by their own graphics system. If one graphics system takes longer than a frame time to draw a scene, the other graphics systems must be kept in lock step with the slowest one. InfiniteReality uses an external *swap ready* wire connecting all the graphics systems together in a wired AND configuration.

The video outputs of all the graphics systems are first locked together. Each pipe monitors the swap ready wire to determine if all the other pipes have finished drawing. A local buffer swap is only allowed to happen if all the graphics systems are ready to swap. In order to cope with slight pipe to pipe variations in video timing, a write exclusion window exists around the swap ready register to guarantee all pipes make the same decision.

Finally an NTSC or PAL output is available with any of the eight channels as the source. Resizing hardware allows for the scaling of any source resolution or windowed subset, to NTSC or PAL resolution.

3 FEATURES

3.1 Virtual Texture

The size of texture databases is rapidly increasing. Texture data that cover the entire world at one meter resolution will be commercially available in 1998. This corresponds to a texture size of 40,000,000 x 20,000,000 texels. Advanced simulation users need to be able to navigate around such large data in real-time. To meet this need, the InfiniteReality system provides hardware and software support for very large *virtual textures*, that is, textures which are too large to reside in physical texture memory.

Previous efforts to support texture databases larger than available texture memory required that the scene database modeler partition the original texture into a number of smaller tiles such that a subset of them fit into physical texture memory. The disadvantage of this approach is that the terrain polygons need to be subdivided so that no polygon maps to more than one texture tile. The InfiniteReality system, by contrast, allows the application to treat the original large texture as a single texture.

We introduce a representation called a *clip-map* which significantly reduces the storage requirements for very large textures. To illustrate the usefulness of the clip-map representation, we observe that the amount of texture data that can be viewed at one time is limited by the resolution of the display monitor. For example, using trilinear mip-map textures on a 1024x1024 monitor, the highest resolution necessary occurs just before a transition to the next coarser level of detail. In this case the maximum amount of resident texture required for any map level is no more than 2048 x 2048 for the finer map, and 1024x1024 for the coarser map, regardless of the size of the original map level. This is the worst case which occurs when the texture is viewed from directly above. In most applications the database is viewed obliquely and in perspective. This greatly reduces the maximum size of a particular level-of-detail that must be in texture memory in order to render a frame.

Recall that a mip-map represents a source image with a pyramidal set of two-dimensional images, each of which covers the full area of the source image at successively coarser resolution [Will83]. A clip-map can be thought of as a subset of the mip-map of the entire texture. It has two parts: a clip-map pyramid which is exactly the same as the coarser levels of the original mip-map, and a clip-map stack which holds a subset of the data in the original mip-map for the finest levels of detail. The clip-map stack levels all have the same size in texture memory, but each successively coarser level covers four times the source image area of the immediately finer level. Figure 6 illustrates the relationships between levels in a clip-map when viewed from above a textured database. The clip-map stack levels are centered on a common point. Each stack level represents larger and larger areas as the resolution of the data they contain becomes coarser and coarser. Figure 7 illustrates a clip-map for a 32K x 32K source image using a 2K x 2K clip-map tile size. Note that the clip-map representation requires about 1/64 the storage of the equivalent 32K x 32K mip-map.

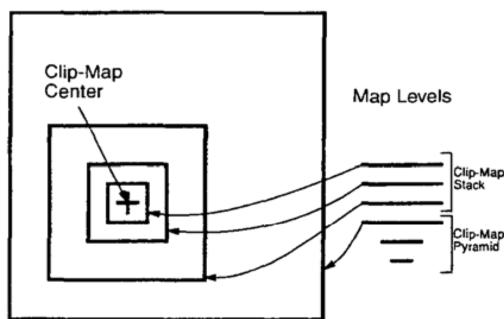


Figure 6: Clip-Map Levels

Because the clip-map stack does not contain the entire texture the position of the clip-map stack needs to be updated to track the viewer's position, or more optimally the center of the viewer's gaze. As the viewer's position or gaze moves, the contents of the clip-map stack should be updated to reflect this movement. New texture data is loaded into the texture memory to replace the texture data that is no longer required. The rate of update of texture data is highest for the finest clip-map stack level and becomes less for coarser stack levels of the clip-map. In the InfiniteReality system, it is not necessary to replace all data in a clip-map level when only a few texels actually need to be updated. The hardware loads new texture data over the old and automatically performs the correct addressing calculations using offset registers. Additionally, the Fragment Generators contain registers that define the clip-map center as it moves through the texture.

If the stack tile size is chosen correctly and the clip-map stack is updated properly as the viewpoint moves through the scene, the InfiniteReality system will produce images identical to those that would have been produced if the entire source mip-map had been resident in texture memory.

It cannot always be guaranteed that the texture data requested during triangle rendering will be available at the requested level of detail. This may occur if the size of the clip-map tile has been chosen to be too small, or the update of the stack center failed to keep pace with the motion of the viewer. The InfiniteReality texture subsystem detects when texture is requested at a higher resolution than is available in texture memory. It substitutes the best available data which is data at the correct spatial position, but at a coarser level-of-detail than requested. As a result, the rendered scene will have regions where the texture will be coarser than if the entire mip-map were resident in texture memory. However, it will otherwise be rendered correctly. This substitution mechanism limits the required clip-map tile size and reduces the required texture update rate.

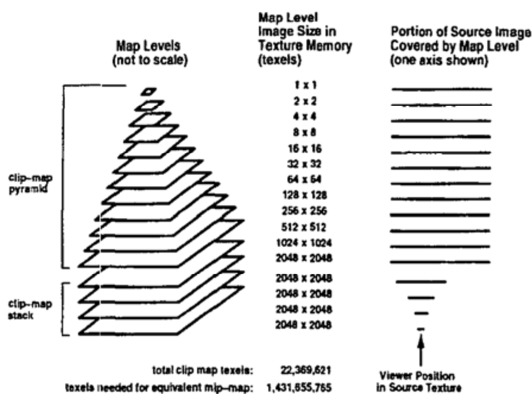


Figure 7: 32Kx32K texture represented as a 2Kx2K clip-map.

The Fragment Generator is limited to addressing a 32K x 32K clip-map. The addressability of clip-maps can be extended to arbitrary texture sizes through software. The software layer needs only to keep track of and implement a transformation from an arbitrarily large texture space into the texture space addressable by the hardware.

3.2 Texture Loading and Paging

We minimize the performance impact of large amounts of texture paging in the design of InfiniteReality system. The graphics subsystem interprets texture images directly as specified by the OpenGL programmer so no host processor translation is required. The front end of the Geometry Subsystem includes pixel unpacking and format conversion hardware; DMA hardware directly implements stride and padding address arithmetic as required by OpenGL. The Fragment Generators accept raster-order texture images at Vertex Bus-limited rates. To eliminate the need for the host computer to make and retain copies of loaded textures for context switching, the hardware supports texture image reads back to the host.

The Geometry-Raster FIFO maintains a separate path through which data bound for texture memory is routed. When the Fragment Generators are busy with fill-limited primitives, pending texture data is transferred over the Vertex Bus and routed to write queues in the TM ASICs. When a moderate amount of texture data is queued for a particular texture DRAM, the TM suspends draw

access and writes the queue contents to that DRAM. Because total bandwidth to and from texture memory is an order of magnitude greater than that of the Vertex Bus, this action only slightly impacts fill rate. For fill-limited scenes, however, this approach utilizes Vertex Bus cycles which would otherwise go unused. Synchronization barrier primitives ensure that no texture is referenced until it has been fully loaded, and conversely, that no texture loading occurs until the data to be overwritten is no longer needed.

3.3 Scene Load Management

3.3.1 Pipeline Performance Statistics

Regardless of the performance levels of a graphics system, there may be times when there are insufficient hardware resources to maintain a real-time frame update rate. These cases occur when the pipeline becomes either geometry or fill rate limited. Rather than extending frame time, it is preferable for the application to detect such a situation and adjust the load on the pipeline appropriately.

The InfiniteReality system provides a mechanism for performing feedback-based load management with application-accessible monitoring instrumentation. Specifically, counters are maintained in the Geometry-Raster FIFO that monitor stall conditions on the Vertex Bus as well as wait conditions upstream in the geometry path. If the counters indicate that there is geometry pending in the Geometry-Raster FIFO, but writes to the Vertex Bus are stalled, then the system is fill rate limited. On the other hand, if the FIFO is empty, then the system is either host or geometry processing limited. By extracting these measurements, the application can take appropriate action whenever a geometry or fill rate bottleneck would have otherwise caused a drop in frame rate.

A common approach to a geometry limited pipeline is for the application to temporarily reduce the complexity of objects being drawn starting with those objects that are most distant from the viewer [Funk93][Rohl94]. This allows the application to reduce the polygon count being sent to the pipeline without severely impacting the visual fidelity of the scene. However, since distant objects do not tend to cover many pixels, this approach is not well-suited to the case where the pipeline is fill limited. To control fill limited situations, the InfiniteReality uses a novel technique termed *dynamic video resizing.*

3.3.2 Dynamic Video Resizing

Every frame, fill requirements are evaluated, and a scene is rendered to the framebuffer at a potentially reduced resolution such that drawing completes in less than one frame time. Prior to display on the monitor, the image is scaled up to the nominal resolution of the display format. Based on the current fill rate requirements of the scene, framebuffer resolution is continuously adjusted so that rendering can be completed within one frame time. A more detailed explanation follows.

Pipeline statistics are gathered each frame and used to determine if the current frame is close to being fill limited. These statistics are then used to estimate the amount by which the drawing time should be reduced or increased on the subsequent frame. Drawing time is altered by changing the resolution at which the image is rendered in the framebuffer. Resolution is reduced if it is estimated that the new image cannot be drawn in less than a frame time. Resolution can be increased if it was reduced in prior scenes, and the current drawing time is less than one frame. The new frame may now be drawn at a different resolution from the previous one. Resolution can be changed in X or Y or both. Magnifying the image back up to the nominal display resolution is done digitally, just prior to display. The video resizing hardware is programmed for the matching magnification ratios, and the video request hardware is programmed to request the appropriate region of the framebuffer.

Finally, to ensure the magnification ratio is matched with the resolution of the frame currently being displayed, loading of the magnification and video request parameters is delayed until the next swap buffer event for that video channel. This ensures that even if scene rendering exceeds one frame time, the resizing parameters are not updated until drawing is finished.

Each channel is assigned a unique display ID, and the swap event is detected for each of these ID's. This swap forces the loading of the new resize parameters for the corresponding video channel, and allows channels with different swap rates to resize.

Note that the effectiveness of this technique is independent of scene content and does not require modifications to the scene data base.

3.4 Video Configurability

One of the goals for the InfiniteReality system was to enable our customers to both create their own video timing formats and to assign formats to each video channel.

This required that the underlying video timing hardware had to be more flexible than in the RealityEngine. Capabilities were expanded in the video timing and request hardware's ability to handle color field sequential, interlace, and large numbers of fields. The biggest change needed was an expanded capability to detect unique vertical sync signatures when genlocking to an external video signal. Since our customers could define vertical sync signatures whose structure could not be anticipated, the standard approach of simply hard-wiring the detection of known sync patterns would have been inadequate. Therefore, each video channel contains programmable pattern recognition hardware, which analyzes incoming external sync and generates resets to the video timing hardware as required.

In previous graphics systems, multi-channel support was designed as an afterthought to the basic single channel display system. This produced an implementation that was lacking in flexibility and was not as well integrated as it could have been. In the RealityEngine system, support for multiple channels was achieved by pushing video data to an external display board. The software that created multi-channel combinations was required to emulate the system hardware in order to precisely calculate how to order the video data. Ordering had to be maintained so each channel's local FIFO would not overflow or underflow. This approach was not very robust and made it impossible for our customers to define their own format combinations.

In the InfiniteReality system, every video channel was designed to be fully autonomous in that each has its own programmable pixel clock and video timing. Each video channel contains a FIFO, sized to account for latencies in requesting frame buffer memory. Video data is requested based on each channel's FIFO levels. A round robin arbiter is sufficient to guarantee adequate response time for multiple video requests.

Format combinations are limited to video formats with the same swap rate. Thus, the combination of 1280x1024@60Hz + 640x480@180Hz field sequential + 1024x768@120Hz stereo + NTSC is allowed but combining 1920x1080@72Hz and 50Hz PAL is not.

In order to achieve our design goal of moving more control of video into the hands of our customers, two software programs were developed. The first program is the Video Format Compiler or **vfc**. This program generates a file containing the microcode used to configure the video timing hardware. The source files for the compiler use a language whose syntax is consistent with standard video terminology. Source files can be generated automatically using templates. Generating simple block sync formats can be accomplished without any specific video knowledge other than knowing the width, height and frame rate of the desired video display format. More complex video formats can be written by modifying an existing source file or by starting from scratch. The Video Format Compiler generates an object file which can be loaded into the display subsystem at any time. Both the video timing hardware and the sync pattern recognition hardware are specified by the **vfc** for each unique video timing format.

The second program is the InfiniteReality combiner or **ircombine**. Its primary uses are to define combinations of existing video formats, verify that they operate within system limitations, and to specify various video parameters. Both a GUI and a command line version of this software are provided. Once a combination of video formats has been defined, it can be saved out to a file which can be loaded at a later time. The following is a partial list of **ircombine** capabilities:

- Attach a video format to a specific video channel
- Verify that the format combination can exist within system limits
- Define the rectangular area in framebuffer memory to be displayed by each channel
- Define how data is requested for interlace formats
- Set video parameters (gain, sync on RGB, setup etc.)
- Define genlock parameters (internal/external, genlock source format, horizontal phase, vertical phase)
- Control the NTSC/PAL encoder (source channel, input window size, filter size)
- Control pixel depth and size

4 PERFORMANCE

The InfiniteReality system incorporates 12 unique ASIC designs implemented using a combination of 0.5 and 0.35 micron, three-layer metal semiconductor fabrication technology.

Benchmark performance numbers for several key operations are summarized in Tables 1, 2, and 3. In general, geometry processing rates are seven to eight times that of the RealityEngine system and pixel fill rates are increased by over a factor of three. Note that the depth buffered fill rate assumes that every Z value passes the Z comparison and must be replaced which is the worst case. In practice, not every pixel will require replacement so the actual depth buffered fill rates will fall between the stated depth buffered and non depth buffered rate.

Although the benchmark numbers are impressive, our design goals focused on achieving real-time application performance rather than the highest possible benchmark numbers. Predicting application performance is a complex subject for which there are no standard accepted metrics. Some of the reasons that applications do not achieve peak benchmark rates include the frequent execution of mode changes (e.g. assigning a different texture, changing a surface material, etc.), the use of short triangle meshes, and host processing limitations. We include execution times for commonly performed mode changes (Table 4) as well as performance data for shorter triangle meshes (Table 5). Practical experience with a variety of applications has shown that the InfiniteReality system is successful in achieving our real-time performance goals.

We were pleasantly surprised by the utility of video resizing as a fill rate conservation tool. Preliminary simulations indicated that we could expect to dynamically reduce framebuffer resolution up to ten percent in each dimension without substantially degrading image quality. In practice, we find that we can frequently reduce framebuffer resolution up to 25% in each dimension which results in close to a 50% reduction in fill rate requirements.

unlit, untextured tstrips	11.3 Mtris/sec
unlit, textured tstrips	9.5 Mtris/sec
lit, textured tstrips	7.1 Mtris/sec

Table 1: Non Fill-Limited Geometry Rates

non-depth buffered, textured, antialiased	830 Mpix/sec
depth buffered, textured, antialiased	710 Mpix/sec

Table 2: Non Geometry-Limited Fill Rates (4 Raster Memory boards)

RGBA8	83.1 Mpix/sec (332 Mb/sec)

Table 3: Peak Pixel Download Rate

glMaterial	240,941/sec
glColorMaterial	337,814/sec
glBindTexture	244,537/sec
glMultMatrixf	1,110,779/sec
glPushMatrix/glPopMatrix	1,489,454/sec

Table 4: Mode Change Rates

Length 2 triangle strips	4.7 Mtris/sec

Table 5: Geometry Rates for Short Triangle Strips

Length 4 triangle strips	7.7 Mtris/sec
Length 6 triangle strips	8.6 Mtris/sec
Length 8 triangle strips	9.0 Mtris/sec
Length 10 triangle strips	11.3 Mtris/sec

Table 5: Geometry Rates for Short Triangle Strips

The above numbers are for unlit, untextured triangle strips. Other types of triangle strips scale similarly.

The performance of the InfiniteReality system makes practical the use of multipass rendering techniques to enhance image realism. Multipass rendering can be used to implement effects such as reflections, Phong shading, shadows, and spotlights [Sega92]. Figure 8 shows a frame from a multipass rendering demonstration running at 60Hz on the InfiniteReality system. This application uses up to five passes per frame and renders approximately 40,000 triangles each frame.

5 CONCLUSION

The InfiniteReality system achieves real-time rendering through a combination of raw graphics performance and capabilities designed to enable applications to achieve guaranteed frame rates. The flexible video architecture of the InfiniteReality system is a general solution to the image generation needs of multichannel visual simulation applications. A true OpenGL implementation, the InfiniteReality brings unprecedented performance to traditional graphics-intensive applications. This underlying performance, together with new rendering functionality like virtual texturing, paves the way for entirely new classes of applications.

Acknowledgments

Many of the key insights in the area of visual simulation came from Michael Jones and our colleagues on the Performer team. Gregory Eitzmann helped architect the video subsystem and associated software tools. The multipass rendering demo in Figure 8 was produced by Luis Barcena Martin, Ignacio Sanz-Pastor Revorio, and Javier Castellar. Finally, the authors would like to thank the talented and dedicated InfiniteReality and Onyx2 teams for their tremendous efforts. Every individual made significant contributions to take the system from concept to product.

REFERENCES

[Akel93] K. Akeley, "RealityEngine Graphics", *SIGGRAPH 93 Proceedings*, pp. 109-116.

[Evan92] Evans and Sutherland Computer Corporation, "*Freedom Series Technical Report*", Salt Lake City, Utah, Oct. 92.

[Funk93] T. Funkhouser, C. Sequin, "Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments", *SIGGRAPH 93 Proceedings*, pp. 247-254.

[Fuch85] H. Fuchs, J. Goldfeather, J. Hultquist, S. Spach, J. Austin, F. Brooks, J. Eyles, and J. Poulton, "Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes", *SIGGRAPH 85 Proceedings*, pp. 111-120.

[Hwan84] K. Hwang, F. Briggs, "*Computer Archetecture and Parallel Processing*", McGraw-Hill, New York, pp. 350-354, 1984.

[Moln92] S. Molnar, J. Eyles, J. Poulton, "Pixelflow: High-Speed Rendering Using Image Composition", *SIGGRAPH 92 Proceedings*, pp. 231-240.

[Moln94] S. Molnar, M. Cox, D. Ellsworth, H. Fuchs, "A Sorting Classification of Parallel Rendering", *IEEE Computer Graphics and Applications, July 94, pp. 23-32.*

[Paus96] R. Paush, J. Snoddy, R. Taylor, E. Haseltine, "Disney's Aladdin: First Steps Toward Storytelling in Virtual Reality", *SIGGRAPH 96 Proceedings*, pp. 193-203.

[Scha83] B. Schachter, "Computer Image Generation", John Wiley & Sons, New York, 1983.

[Sega92] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, P. Haeberli, "Fast Shadows and Lighting Effects Using Texture Mapping", *SIGGRAPH 92 Proceedings*, pp. 249-252.

[Rohl94] J. Rohlf, J. Helman, "IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3DGraphics, *SIGGRAPH 94 Proceedings*, pp. 381-394.

[Will83] L. Williams, "Pyramidal Parametrics", *SIGGRAPH 83 Proceedings*, pp. 1-11.

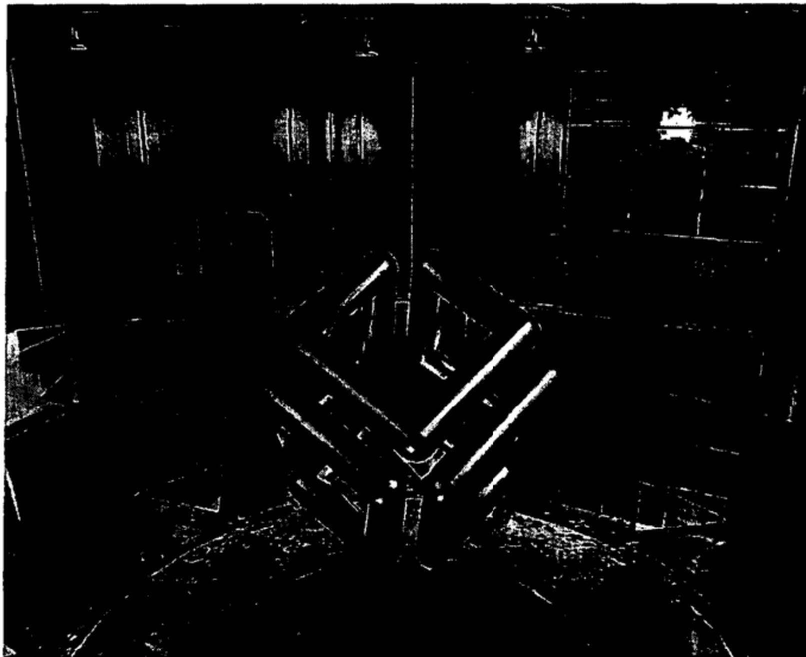


Figure 8: An example of a high quality image generated at 60 Hz using multipass rendering techniques

2001 EXPLORE INTERACTION AND DIGITAL IMAGES

XP-001049881

G06T15/00A

PD 12-0-2001
BO 269 070 p129-140=(12)

WireGL: A Scalable Graphics System for Clusters

Greg Humphreys* Matthew Eldridge* Ian Buck* Gordon Stoll† Matthew Everett* Pat Hanrahan*

*Stanford University †Intel Corporation

Abstract

We describe WireGL, a system for scalable interactive rendering on a cluster of workstations. WireGL provides the familiar OpenGL API to each node in a cluster, virtualizing multiple graphics accelerators into a sort-first parallel renderer with a parallel interface. We also describe techniques for reassembling an output image from a set of tiles distributed over a cluster. Using flexible display management, WireGL can drive a variety of output devices, from standalone displays to tiled display walls. By combining the power of virtual graphics, the familiarity and ordered semantics of OpenGL, and the scalability of clusters, we are able to create time-varying visualizations that sustain rendering performance over 70,000,000 triangles per second at interactive refresh rates using 16 compute nodes and 16 rendering nodes.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics; I.3.4 [Computer Graphics]: Graphics Utilities—Software support, Virtual device interfaces; C.2.2 [Computer-Communication Networks]: Network Protocols—Applications; C.2.4 [Computer-Communication Networks]: Distributed Systems—Client/Server, Distributed Applications

Keywords: Scalable Rendering, Cluster Rendering, Parallel Rendering, Tiled Displays, Remote Graphics, Virtual Graphics

1 Introduction

Despite recent advances in accelerator technology, many real-time graphics applications still cannot run at acceptable rates. As processing and memory capabilities continue to increase, so do the sizes of data being visualized. Today we can construct laser range scans comprised of billions of polygons [14] and solutions to fluid dynamics problems with several hundred million data points per frame over thousands of frames [8, 21]. Because of memory constraints and lack of graphics power, visualizations of this magnitude are difficult or impossible to perform on even the most powerful workstations. Therefore, the need for a scalable graphics system is clear.

The necessary components for scalable graphics on clusters of PC's have matured sufficiently to allow exploration of clusters as a reasonable alternative to multiprocessor servers for high-end visualization. In addition to graphics accelerators and processor power, memory and I/O controllers have reached a level of sophistication that permits high-speed memory, network, disk, and graphics I/O to all occur simultaneously, and high-speed general purpose networks are now fast enough to handle the demanding task of routing streams of graphics primitives.

To take advantage of these opportunities, we have designed and implemented WireGL, a software system that unifies the rendering power of a collection of graphics accelerators in cluster nodes, treating each separate framebuffer as part of a single tiled display. A high-level block diagram of WireGL's major components is shown in figure 1. WireGL provides a virtualized interface to the graphics hardware through the OpenGL API. OpenGL provides immediate-mode semantics, so we support visualizations of time-varying data that would be inconvenient to express with a retained-mode interface or in a scene graph.

In addition, WireGL provides a parallel interface to the virtualized graphics system, so each node in a parallel application can issue graphics commands directly. This helps applications overcome one of the most common performance-limiting factors in modern graphics systems: the interface bottleneck. WireGL extends the OpenGL API to allow the simultaneous streams of graphics commands to obey ordering constraints imposed by the programmer.

Another recent development is the introduction of the Digital Visual Interface (DVI) standard for digital scan-out of the framebuffer [5]. WireGL allows a flexible assignment of tiles to graphics accelerators, recombining these tiles using DVI-based tile reassembly hardware called Lightning-2 [27]. In the absence of image composition hardware, WireGL can also perform the final image reassembly in software, using the general purpose cluster interconnect. Because of this flexible assignment of tiles to accelerators, WireGL can deliver the combined rendering power of a cluster to any display, be it a multi-projector wall-sized display or a single monitor. By decoupling the number of graphics accelerators from the number of displays and allowing a flexible partitioning of the output image among the accelerators, image reassembly gives applications control over their graphics load balancing needs.

2 Design Issues and Related Work

Designing a parallel graphics system involves a number of tradeoffs and choices. In this section, we present some of the most crucial issues facing parallel graphics system designers.

2.1 Commodity Parts and Work Granularity

Parallel graphics architectures can usually be classified according to the point in the graphics pipeline at which data are redistributed [16]. This redistribution, or "sorting" step is the transition from object parallelism to image parallelism, and the location of this sort has significant implications for the architecture's communication needs. When building a new hardware architecture, the design of the communication infrastructure is flexible, and can be engineered to meet the requirements of the system.

The SGI InfiniteReality is a sort-middle architecture which uses bus-based broadcast communication to distribute primitives [18].

*{humper|eldridge|ianbuck|meverett|hanrahan}@graphics.stanford.edu
†gordon.stoll@intel.com

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACM SIGGRAPH 2001, 12-17 August 2001, Los Angeles, CA, USA © 2001 ACM 1-58113-374-X/01/08...$5.00

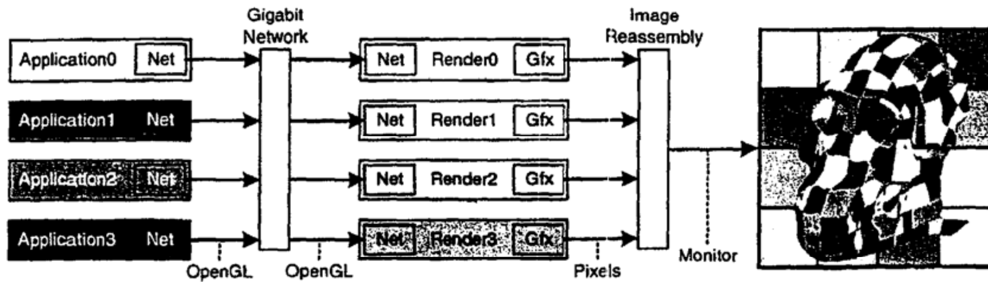


Figure 1: WireGL is comprised of application nodes, rendering nodes, and a display. In this example, each application node is performing isosurface extraction in parallel and rendering its data using the OpenGL API. Each application node is responsible for the correspondingly colored portions of the object. In the configuration shown, the display is divided into 16 tiles, each of which is managed by the correspondingly shaded rendering node. These tiles are reassembled to a single monitor after they are scanned out of the graphics accelerators.

To overcome the difficulties encountered in load-balancing image-parallel data, it uses a fine interleaving of tiles, which works well because of the available high-bandwidth broadcast bus. Pixel-Planes 5 is a sort-middle architecture with large tiles, which uses a ring network to distribute primitives from a retained-mode scene description [7].

Because such systems do not use commodity building blocks, they must be repeatedly redesigned or rebuilt in order to continue to scale as faster semiconductor technology is developed. WireGL chooses instead to unify multiple unmodified commodity graphics accelerators housed in cluster nodes. This decision has the advantage that we can upgrade the graphics cards or the network at any time without redesigning the system.

However, the choice of cluster network will greatly affect the overall performance and scalability of the resulting system. On PC clusters today, high-speed networks tend to be in the 100-200 megabyte per second range. These networks are an order of magnitude slower than that of a high-end SMP like the SGI Origin 3000, and yet another order of magnitude slower than custom on-chip networks. Although PC cluster networks are not as efficient as more custom solutions, we can still use them to provide scalable graphics performance. As high-speed commodity networks improve in bandwidth and robustness, WireGL will be able to provide better scalability in larger clusters, as well as higher peak performance.

Using commodity parts restricts our choices about communication and work granularity because we cannot modify the individual graphics accelerators. As shown in figure 2, there are only two points in the graphics pipeline where we can introduce communication: immediately after the application stage, and immediately before the final display stage. Communication after the application stage provides a redistribution of primitives to remote graphics accelerators based on those primitives' screen-space extent, which is a traditional sort-first graphics architecture. By introducing communication at the very end of the graphics pipeline, the final image can be recombined from multiple framebuffers. Although WireGL uses this stage to perform tile reassembly, communication at the end of the pipeline can also be used for image composition-based renderers.

For remote use of unmodified graphics components, GLR [13] and SGI's "Vizserver" product [26] transmit a stream of compressed images from the framebuffer of a graphics supercomputer to a low-end desktop. Image compression and streaming technology is an attractive approach to rendering at a distance, although it is not the best approach when the eventual display is local to the rendering hardware.

Although WireGL is a sort-first renderer, sort-last architectures also use a final image recombination step to produce a single image from a fragmented framebuffer. PixelFlow uses image-composition to drive a single display from a parallel host [17]. The Hewlett-Packard visualize *fx* architecture uses a custom network to composite the results of multiple graphics accelerators [4]. Sony's GSCube combines the outputs of multiple Playstation2 graphics systems using a custom network, and supports both sort-first and image composition modes of operation. The GSCube is a particularly interesting architecture because it leverages consumer technology to produce a scalable rendering technology.

To perform image reassembly on clusters, Compaq Research has developed a system called Sepia for performing image composition using ServerNet-II networking technology [9]. Blanke et al. describe the Metabuffer, a system for performing sort-last parallel rendering on a cluster using DVI to scan out color and depth [1]. The Metabuffer is similar to Lightning-2 [27], the DVI-based image reassembly network that we use to drive displays with our cluster. Unlike Sepia, Lightning-2 and the Metabuffer do not require pixel data to be transferred to the image composition network over the internal system bus, where bandwidth is often a critical resource for parallel visualization applications.

2.2 Flexible Application Support

Many applications visualize the results of a simulation as those results are calculated. In this case, the simulation usually generates data more slowly than the graphics system can accept it. Such an application is referred to as *compute-limited*. There are many compute-limited visualization applications that scale by generating geometry in parallel and communicating that geometry over a network to a single display server. This geometry communication is almost always done with custom networking code, using a custom wire protocol.

Other applications, however, make intensive use of the graphics hardware, and a single client may effectively occupy many servers. Such an application is called *graphics-limited*. For example, volume rendering with 3D textures requires high fill rates while using few primitives. In this case, a single client may submit commands to multiple servers and keep them all busy because the rendering time of each individual primitive is so large.

Many applications are limited by the rate at which they can issue geometry to the graphics system. Such an application is *interface-*

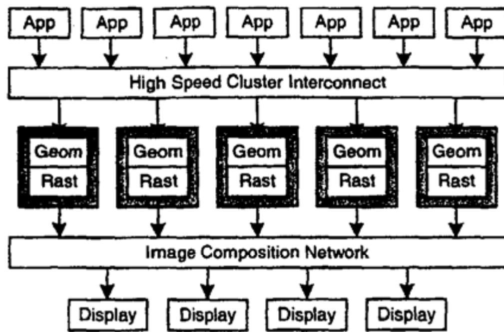


Figure 2: Communication in WireGL. Each graphics pipeline is a standalone graphics accelerator, so we cannot introduce communication between its stages. Notice that the number of application nodes, graphics pipelines, and final displays can all be changed independently according to the application's needs.

limited. For example, visualizations of large geometric data sets that have been computed off-line will tend to be interface limited. Interface limitation is the usual argument for using display lists, compiled vertex arrays, or other retained-mode interfaces. Another way to alleviate the interface bottleneck is to allow multiple processors to issue graphics commands in parallel.

Finally, some applications are not limited by performance, but they cannot effectively visualize their data due to a lack of display resolution. Such an application is called *resolution-limited.* This is typical of many scientific applications where it is important to view all the data at a macroscopic level to get an overview of a dataset, and also to examine microscopic details to fully understand the data. Such an application requires the combined resolution of multiple graphics accelerators and a high-resolution tiled display. One example of this type of display is IBM's Bertha, a 3840×2560 LCD display driven by four DVI inputs.

WireGL does not place any restrictions on the number of clients or servers. For compute-limited applications it is desirable to have more clients than servers, for graphics-limited applications it is better to have more servers than clients, and for interface-limited applications it is most effective to have an approximately equal number of clients and servers. WireGL also works well in a heterogeneous environment where the servers and the clients may be running different operating systems on different hardware.

2.3 Programming Interface

Graphics API's can provide a low-level resource abstraction such as OpenGL, or a high-level abstraction such as a scene graph library. Scene graphs and other high-level interfaces are attractive because global information can be used to automatically parallelize rendering or perform fast culling. IRIS Performer provides parallel traversal of a retained-mode scene graph, and can also take advantage of multiple graphics pipelines in a single SMP [22]. Samanta et al. describe a novel screen subdivision algorithm for load-balanced rendering of a scene graph that has been replicated across the nodes of a cluster [23, 24].

However, not all visualization tools can conveniently use a scene graph, because their data may be unstructured and time-varying. Another significant drawback of scene graphs is the lack of a standardized scene graph API. Any scene graph library that uses OpenGL for rendering can run on top of WireGL. In addition, if the scene graph has bounding-box information about primitive groups, that information can be provided to WireGL through the OpenGL hinting mechanism to speed up geometry sorting.

WireGL provides the OpenGL API to each node in a cluster. The decision to use OpenGL for specifying graphics data has several advantages over using a custom API. First, we can run an unmodified application on a single node in our cluster without recompiling it. If that application is graphics-limited, WireGL can provide an immediate speedup. Also, if we have access to a large display wall, we can easily interact with resolution-limited datasets that can take advantage of the larger display area. Portions of WireGL were first described by Humphreys et al. [10]. In that paper, we described our techniques for sorting OpenGL streams to tile servers in order to transparently support large displays. SGI also provides a library called "Multipipe" that intercepts OpenGL commands and allows unmodified applications to render across multiple graphics accelerators, providing increased output resolution [25].

Many applications, however, are not graphics-limited and must be parallelized to achieve speedup. Using WireGL, many existing serial OpenGL applications can be parallelized with minor changes to the inner drawing routines. In particular, applications that render large geometric datasets using the depth buffer to resolve visibility can simply partition their dataset across the nodes of the cluster, and have each node render its portion as before. Because such an application has almost no ordering requirements, achieving parallelism is straightforward.

For applications with more complex ordering requirements, WireGL implements extensions to OpenGL that were first proposed by Igehy, Stoll and Hanrahan [12]. Their simulations showed that scalable applications could easily be written using their extensions, results that were further verified by the Pomegranate simulations [6]. These extensions add traditional synchronization primitives (barriers and semaphores) to the graphics library. WireGL is the first implementation of this API in a hardware-accelerated (that is, not simulated) architecture.

Although OpenGL is an immediate-mode API, some OpenGL features like display lists and texture objects allow data to be stored by the graphics system and reused. WireGL supports this by storing those data on the server, so that users who want to replicate data across the nodes of the cluster can do so. In addition, texture objects can optionally be shared between multiple clients, which means that they can be specified once at the start of the application and do not need to be duplicated per-client. It would also be easy to allow similar sharing of display lists between clients, although we have not implemented this feature.

3 WireGL

A WireGL based rendering system consists of one or more clients submitting OpenGL commands simultaneously to one or more graphics servers, called *pipeservers.* The pipeservers are organized as a sort-first parallel graphics pipeline [19], and together they render a single output image. Each pipeserver has its own graphics accelerator and a high-speed network connecting it to all clients. The output image is divided into tiles, which are partitioned over the servers, each server potentially managing multiple tiles. The assembly of the final output display from the tiles is described in section 4. A high-level view of the system is shown in figure 1. In that figure, each rendering node is a pipeserver. WireGL virtualizes this architecture, providing a single conceptual graphics pipeline to the clients.

3.1 Client Implementation

This section provides an overview of WireGL's sort-first client implementation. Interested readers should refer to Humphreys et al. [10] for a more complete description of our sort-first system, the protocol efficiency, and display size scalability results. The state tracking system is described in detail in Buck, Humphreys, and Hanrahan [3].

The WireGL client library is implemented as a replacement for the system's OpenGL library on Windows, Linux, or IRIX. As the application makes calls to the OpenGL API, WireGL classifies each call into one of three categories: geometry, state, or special. Special commands, such as `SwapBuffers`, `glFinish`, and `glClear`, require individual treatment, and will not be described here.

Geometry commands are those that legally appear between a `glBegin`/`glEnd` pair, as well as commands that can generate fragments on their own, such as `glDrawPixels`. These commands are packed immediately into a global "geometry buffer". This buffer contains a copy of the arguments to the function, as well as an opcode. Each opcode is encoded in a single byte, and opcodes and data are packed into separate portions of the buffer which grow in opposite directions. This representation allows the buffer to retain each argument's memory alignment, minimizes the space overhead of the opcodes, and keeps opcodes and data contiguous in memory so that they can be sent with a single call to the networking library. Some commands that can appear legally between a `glBegin`/`glEnd` pair do not generate fragments, such as `glNormal3f`. These commands are still packed immediately into the buffer, but their state effects are also recorded. Our geometry packing code has been carefully engineered, and achieves a maximum packing performance of over 20 million vertices per second (the exact computer configuration used to perform these experiments is described in section 5).

As each vertex is specified, WireGL maintains an object-space bounding box. Each incremental update to the bounding box requires only six conditional moves, which can be implemented efficiently using a SIMD instruction set such as the Pentium III's. When geometry is sent to the servers, this bounding box is transformed into screen space, and the set of overlapped screen tiles is computed. This set is used to compute the servers that need to receive the geometry buffer. Because geometry sorting is done on groups of primitives, the overhead of bounding box transformation and extent intersection is amortized over many vertices.

State commands are those that directly affect the graphics state, such as `glRotatef`, `glBlendFunc`, or `glTexImage2D`. The effects of state commands are recorded into a graphics context data structure. Each element of state has n bits associated with it indicating whether that state element is out of sync with each of n servers. When a state command is executed, the bits are all set to 1, indicating that each server might need a new copy of that element. The OpenGL state is represented as a hierarchy, roughly mirroring the layout described in the OpenGL specification [20]. For example, `GL_LIGHT0`'s diffuse color is a member of `GL_LIGHT0`'s state, which is an element of the lighting state. Each non-leaf node in the hierarchy also has a vector of n synchronization bits which reflect the logical OR of all its children. We have shown that this representation allows for very efficient computation of the difference between two contexts [3].

Either of two circumstances can trigger the transmission of the geometry buffer. First, if the buffer fills up, it must be flushed to make room for subsequent commands. Second, if a state command is called while the geometry buffer is not empty, the geometry buffer must be flushed before the state command is recorded, since OpenGL has strict ordering semantics. However, we cannot send the geometry buffer to the overlapped servers immediately, because they might not have the correct OpenGL state. We must prepend a packed representation of the application's state before transmitting any geometry. To do this, the client library keeps a copy of each server's graphics state. Using our efficient context differencing operation, the commands needed to bring the server up to date with the application are placed in that server's outgoing network buffer. The global geometry buffer can then be copied after the state differences. By updating state lazily and bucketing geometry, we keep network traffic to a minimum.

This behavior has an important implication for the granularity of work in WireGL. Sorting individual primitives in software would be too expensive, but grouping too many primitives may result in excessive overlap and inefficient network usage. Assuming that a state call is made before a network buffer fills, WireGL's work granularity is that of groups of primitive blocks, or multiple `glBegin`/`glEnd` pairs. The optimal granularity of work will be a balance between screen-space coherency and the expense of bounding-box transformation.

It would be impractical to transform each primitive separately, but it is not always beneficial to coalesce the maximum number of primitive blocks, as this may result in partial network broadcasts if the geometry is not spatially coherent and requires a large screen-space bounding box. WireGL currently has no automatic mechanism for determining the best time to bucket geometry. Applications that are aware of their bucketing needs can optionally force a sort after a specified number of primitive blocks.

When running a parallel application, each client node behaves in the manner described above, performing a sort-first distribution of geometry and state to all pipeservers. This means that each pipeserver must be prepared to handle multiple asynchronous incoming streams of work, each with its own associated graphics context. OpenGL guarantees that commands from a serial context will appear to execute in the order they are issued. When multiple OpenGL contexts render to a single image, this restriction must be relaxed because the graphics commands are being issued in parallel. To provide ordering control for parallel rendering, WireGL adds barriers and semaphores to the OpenGL API, as proposed by Igehy et al. [12].

The key advantage of these synchronization primitives is that they do not block the application. Instead, the primitives are encoded into the graphics stream, and their implied ordering is obeyed by the graphics system when a context switch occurs. A graphics context may enter a barrier at any time by calling `glBarrierExec(name)`. Semaphores can be acquired and released with `glSemaphoreP(name)` and `glSemaphoreV(name)`, respectively. Note that these ordering commands must be broadcast, as the same ordering restrictions must be observed by all servers, and we wish to avoid a central oracle making global scheduling decisions.

When running a parallel application, WireGL does not change the semantics of any commands, even those with global effects. For example, `SwapBuffers` marks the end of the frame and causes a buffer swap to be executed by all servers. Therefore, it is important that only one client execute `SwapBuffers` per frame. Also, a parallel application with no intra-frame ordering dependencies will still need two barriers per frame. To ensure that the framebuffer clear happens before any drawing, a barrier must follow the call to `glClear`. Similarly, all nodes must have completely submitted their data for the current frame before swapping buffers, so another barrier must precede the call to `SwapBuffers`. Pseudocode for this minimal usage is shown in figure 3. More complex usage examples can be found in Igehy's original paper [12].

3.2 Pipeserver Implementation

A pipeserver maintains a queue of pending commands for each connected client. When new commands arrive over the network, they are placed on the end of their client's queue. These queues are stored in a circular "run queue" of contexts. A pipeserver continues

```
Display() {
    if (my_thread_id == 0) // I am the master
        glClear( ... );
    glBarrierExec( global_barrier );
    DrawFrame();
    glBarrierExec( global_barrier );
    if (my_thread_id == 0) // I am the master
        glSwapBuffers();
}
```

Figure 3: A minimal parallel display routine. Although the geometry itself has no intra-frame ordering dependencies, the imposition of frame semantics requires barriers following the framebuffer clear and preceding the buffer swap to ensure that the entire frame is visible.

executing a client's commands until it runs out of work or the context "blocks" on a barrier or semaphore operation. Blocked contexts are placed on wait queues associated with the semaphore or barrier they are waiting on. The pipeserver's queue structures are shown in figure 4.

Because each client has an associated graphics context, a context switch must be performed each time a client's stream blocks. Although all modern graphics accelerators can switch contexts fast enough to support several concurrent windows, hardware context switching is still slow enough to discourage fine-grained sharing of the graphics hardware. When programmatically forced to switch contexts, the fastest modern accelerators achieve a rate of approximately 12,000 times per second [3], which is slow enough that it would limit the amount of intra-frame parallelism achievable in WireGL.

To overcome this limitation, each pipeserver uses the same state tracking library as the client to maintain the state of each client in software. Just as an extremely efficient context differencing operation is the key to lazy state update between the client and the server, it is also effective for performing context switching on the server. Since nodes in a parallel application are collaborating to produce a single image, they will typically have similar graphics states, and performing context switching with our hierarchical representation has a cost proportional to the contexts' disparity. We have measured this hierarchical approach as being able to switch contexts almost 200,000 times per second for contexts that differ in current color and transformation matrix, and over 5 million times per second for identical contexts [3].

In practice, when a context blocks, the servers often have a choice of many potentially runnable contexts. Because a parallel application will almost always enter a barrier immediately before the end of the frame, it is unlikely that one context will become starved. Therefore, in choosing a scheduling algorithm, the main concerns are the expense of the context switch itself as well as the amount of useful work that can be done before the next context switch. In practice, we have found that a simple round-robin scheduler works well, for two reasons. First, clients participating in the visualization of a large dataset are likely to have similar contexts, making the expense of context switching low and uniform. Also, since we cannot know when a stream is going to block, we can only estimate the time to the next context switch by using the amount of work queued for a particular context. Moreover, any large disparity in the amount of work queued for a particular context is most likely the result of an application-level load imbalance. This load imbalance, not context switching overhead, will certainly be the main performance limitation of the application. In general, because of the low cost of context switching, and because we need to complete execution of all contexts before the end of the frame, the pipeserver's scheduling algorithm is not a significant factor in an application's performance.

Since each pipeserver may manage more than one tile, it may be necessary to render a block of geometry more than once. The arrangement of tiles in the local framebuffer is described in section 4.1. The client library inserts the bounding box for each block of geometry between the geometry itself and its preceding state commands. Each server compares this bounding box against the extents of the tiles managed by that server. For each intersection found, a translate and scale matrix is prepended to the current transformation matrix, positioning the resulting geometry with respect to the intersected tile's portion of the final output. Because of the semantics of OpenGL rasterization, this technique can lead to seaming artifacts for anti-aliased or wide lines and points. Unfortunately, not all OpenGL implementations adhere to the same rules regarding clipping of wide lines and points that are larger than one pixel, so this problem is difficult to address in general.

Calls to `glViewport` and `glScissor` are then issued to restrict the drawing to the tile's extent in the server's local framebuffer, and finally the geometry opcodes are decoded and executed. Because the geometry block also includes vertex attribute state, the graphics state may have changed by the end of the geometry block. However, the client will insert commands to restore the vertex attribute state at the beginning of the geometry buffer. Therefore, if the geometry overlaps more than one tile, the vertex attribute state will always be properly restored before the geometry is re-executed.

3.3 Network

We use a connection-based network abstraction to support multiple network types such as TCP/IP and Myrinet. Our abstraction provides a credit-based flow control mechanism to prevent servers from exhausting their memory resources when they cannot keep up with the clients. Flow control is particularly important when a context is blocked, since additional commands may come in from the client at any time even though the server cannot drain a blocked context's command queue.

Each server/client pair is joined by a connection. By making buffer allocation the responsibility of the network layer, we allow a zero-copy send. For example, the client packs OpenGL commands directly into network buffers, and the Myrinet network layer sends them over the network using DMA. In order for this to work, these buffers must be pinned (locked and unpageable), which is done by the implementation of our network abstraction for Myrinet. Receiving data on our network operates in a similar manner: the network layer allocates (possibly pinned) buffers, allowing a zero-copy receive.

The connection is completely symmetric, which means that the servers can return data such as the results of `glReadPixels` to the clients. More importantly, WireGL supports the `glFinish` call so that applications can determine when the commands they have issued have been fully executed. This is available so that applications that need to synchronize their output with some external input source can make sure the graphics system's internal buffering is not causing their output to lag behind the input. The user can optionally enable an implicit `glFinish`-like synchronization on each `SwapBuffers` call, which ensures that no client will ever get more than one frame ahead of the servers.

4 Display Management

To form a seamless output image, tiles must be extracted from the framebuffers of the pipeservers and reassembled to drive a display device. We provide two ways to perform this reassembly. For highest performance, the images may be reassembled after being scanned out of the graphics accelerator. If this is not possible, the

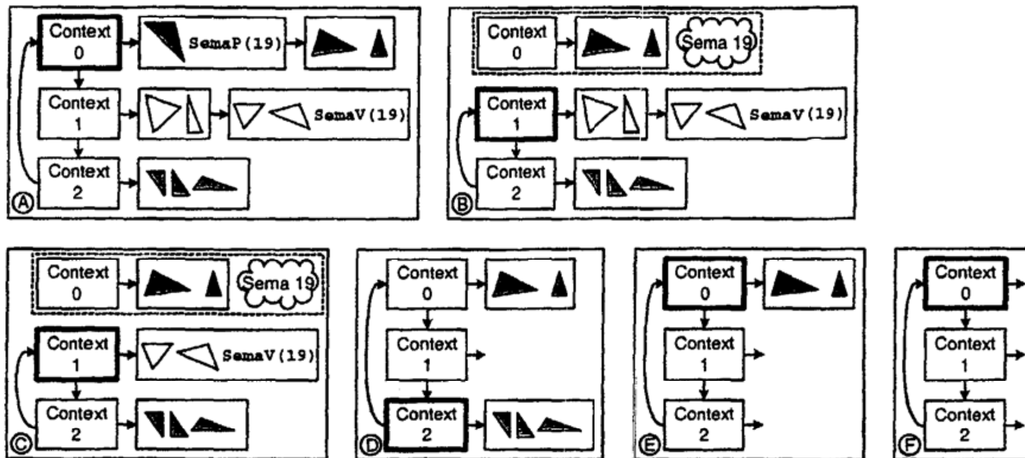


Figure 4: Inside a pipeserver. Runnable contexts will be serviced in a round-robin fashion. Graphics commands being issued by a context's application can be appended to the end of a work queue at any time, until the client consumes its allotted server-side buffer space. Blocks A-F show sequential timesteps as the pipeserver decodes command blocks; the currently executing context is shown with a heavy outline. In timestep A, the pipeserver encounters the SemaP operation in context 0, which blocks the context and removes it from the run queue. In timestep C, context 1's SemaV command will unblock context 0 and place it back on the run queue.

tiles can be extracted from the framebuffer over the host bus interface and distributed over a general purpose network, often the same one used for distributing geometry commands.

Of course, the most straightforward way to reassemble the image after scan-out is to allow each pipeserver to drive a single locally-attached display. These displays can then be abutted to form a large logical output space. This arrangement constrains each pipeserver to manage exactly one tile that is precisely the size of its local framebuffer. This limits WireGL's ability to provide an application with flexible load balancing support, but makes the final display simple to construct.

4.1 Display Reassembly in Hardware

For our experiments with hardware display assembly, we use the Lightning-2 system [27]. Each Lightning-2 board accepts 4 DVI inputs from graphics accelerators and emits up to 8 DVI outputs to displays. Multiple Lightning-2 boards can be connected in a column via a "pixel bus" to provide more total inputs. Multiple columns can also be chained by repeating the DVI inputs, providing more DVI outputs. An arbitrary number of accelerators and displays may be connected in such a two-dimensional mesh, and pixel data from any accelerator may be redirected to any location on any output display. Routing information is drawn into the framebuffer by the application in the form of two-pixel-wide (48 bit) "strip headers". Each header specifies the destination of a one-pixel-high, arbitrarily wide strip of pixels following the packet header in the frame buffer. Lightning-2 can drive a variable number of displays, including a single monitor.

Each input to Lightning-2 usually contributes to multiple output displays, so Lightning-2 must observe a full output frame from *each* input before it may swap, introducing exactly one frame of latency. However, almost no currently available graphics accelerators have external synchronization capabilities. For this reason, Lightning-2 provides a per-host back-channel using the host's serial port. When Lightning-2 has accepted an entire frame from all inputs, it then notifies all input hosts simultaneously that it is ready for the next frame. WireGL waits for this notification before executing a client's SwapBuffers command. Because the framebuffer scan-out happens in parallel with the next frame's rendering, Lightning-2 will usually be ready to accept the new frame before the host is done rendering it, unless the application runs at a faster rate than the eventual monitor's refresh rate. In this case, the application will be limited to the display's refresh rate, which is often a desirable property. Lightning-2 can also lock groups of outputs to swap together. Having synchronized outputs allows Lightning-2 to drive tiled display devices such as IBM's Bertha or a multi-projector display wall without tearing artifacts. This in turn enables stereo rendering on tiled displays.

Each pipeserver reserves space for its assigned tiles in its local framebuffer in a left-to-right, top-to-bottom pattern, leaving two-pixel-wide gaps between tiles, as shown in figure 5. A fixed pattern of strip headers is drawn into the gaps to route the tiles to their correct destination in the display space. Because Lightning-2 routes portions of a single horizontal scanline, non-uniform decompositions of the screen such as octrees or KD-trees can easily be accomplished using WireGL and Lightning-2. In general, each application will have different tiling needs which should be determined experimentally. In the future, we would like to be able to adjust the screen tiling on the fly to meet the application's needs automatically.

4.2 Display Reassembly in Software

Without special hardware to support image reassembly, the final rendered image must be read out of each local framebuffer and redistributed over a network. This network can be the same one used to distribute graphics commands, or it could be a separate dedicated network for image reassembly.

To provide this functionality, WireGL has a mode called the "visualization server". In this mode, all pipeservers read the color contents of their managed tiles at the end of each frame. Those images are then sent over the cluster's interconnect to a separate

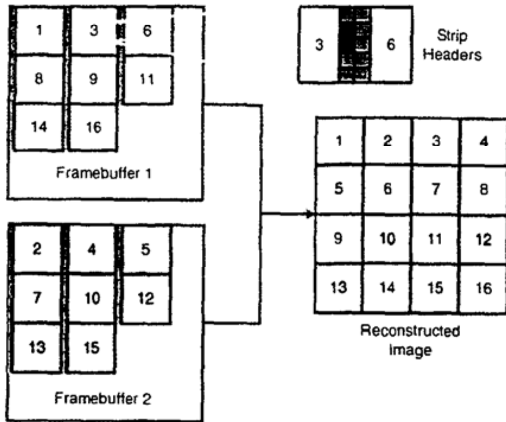


Figure 5: Allocating multiple tiles to a single accelerator with Lightning-2. In the zoomed-in region, the two-pixel wide strip headers are clearly visible.

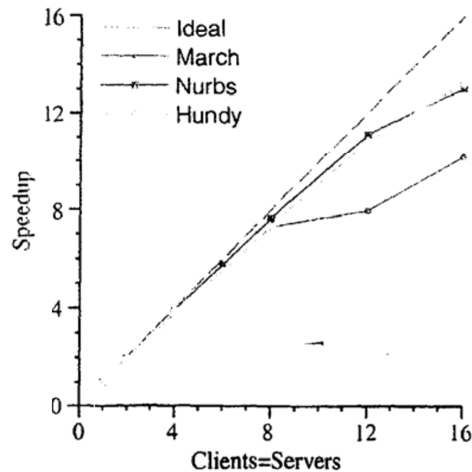


Figure 6: Speedup for March, Nurbs, and Hundy using up to 16 pipeservers. With 16 clients and 16 servers, Hundy achieves 83% efficiency, Nurbs achieves 81% efficiency, and March achieves 64% efficiency.

compositing server for reassembly, with the same protocol used by the clients to send geometry to the pipeservers. In effect, each pipeserver becomes a client in a parallel image-drawing application. The compositing server is simply another WireGL pipeserver accepting `glDrawPixels` commands and parallel API synchronization directives.

The primary drawback of this pure software approach is its potential impact on performance. Pixel data must be read out of the local framebuffer, transferred over the internal network of the cluster, and written back to a framebuffer for display. Even with the limited bandwidth available on modern cluster networks, image drawing bandwidth will tend to be the limiting factor for applications that can update at high framerates. As networks and graphics cards improve and can carry more pixel data along with the geometry data, this technique may become more attractive, but it cannot currently sustain high frame rates, as we will show in section 5.3.

5 Performance and Scalability

The cluster used for all our experiments, called "Chromium", consists of 32 Compaq SP750 workstations. Each node has two 800 MHz Intel Pentium III Xeon processors, 256 megabytes of RDRAM, and an NVIDIA Quadro2 Pro graphics adapter. The SP750 uses the Intel 840 chipset to control its I/O and memory channels, including a 64-bit, 66 MHz PCI bus, an AGP4x slot, and dual-channel RDRAM. Each SP750 is running RedHat Linux 7.0 with NVIDIA's 0.9-769 OpenGL drivers.

Each node has a Myricom high-speed network adapter [2] connected to its PCI bus. Each network card has 2MB of local memory and a 66 MHz LANai 7 RISC processor. The cluster is fully connected using two cascaded 16-port Myricom switches. Using the 1.4pre37 version of the Myricom Linux drivers, we are able to achieve a bandwidth of 101 MB/sec when communicating between two different hosts.

When rendering locally, each node can draw 21.4 million unlit points per second using an immediate mode interface (i.e., without display lists or vertex arrays). WireGL's maximum packing rate (the speed at which WireGL can construct network buffers) is 21.8 million vertices per second. When using WireGL to render remotely from one client to one server, we achieve a maximum rate of 7.5 million points per second. Since each point occupies 13 bytes (three floats plus an opcode byte), this represents a network bandwidth of 93 MB/sec, which is close to the 101 MB/sec we have measured when repeatedly resending the same packet after creation.

For our experiments with parallel applications, we partition the cluster into 16 computation nodes and 16 visualization nodes. This is done because our network does not perform well when senders and receivers are running on the same host, as shown in section 5.4.

5.1 Applications

We have analyzed WireGL's performance and scalability with three applications:

- March is a parallel implementation of the marching cubes volume rendering algorithm [15]. A $200 \times 200 \times 200$ volume is divided into subvolumes of size $4 \times 4 \times 4$ which are processed in parallel by a number of isosurface extraction and rendering processes. March draws independent triangles (three vertices per triangle) with per-vertex normal information. March extracts and renders 385,492 lit triangles per frame at a rate of 374,000 tris/sec on a single node. Our graphics accelerators can render 2.9 million lit, independent triangles with vertex normals per second.

- Nurbs is a parallel patch evaluator that uses multiple processors to subdivide curved surfaces and tessellate them for submission to the graphics hardware. For our tests, Nurbs tessellates and renders 413,100 lit, stripped triangles per frame with vertex normals, at a rate of 467,000 tris/sec on a single node.

- Hundy is a parallel application that renders a set of unorganized triangle strips. Each strip is assigned a color, but no lighting is used. Hundy is representative of many scientific visualization applications where the data are computed off-line and the visualization can be decomposed almost arbitrarily.

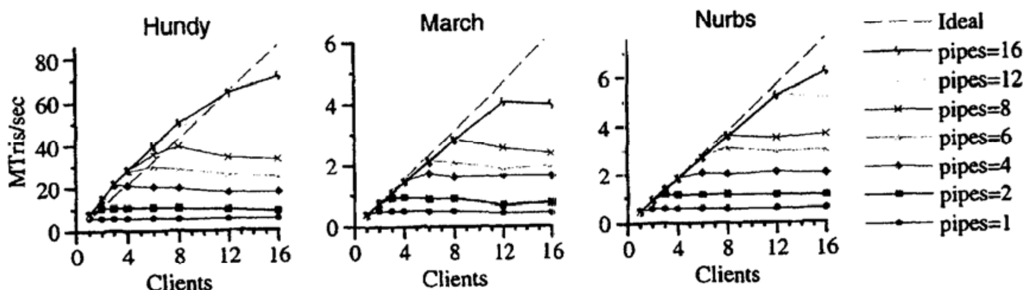


Figure 7: Scaling interface-limited applications. For each application, the number of clients and servers is varied. Hundy uses a tile size of 100 × 100, and achieves a peak rendering performance of 71 million tris/sec at a rate of 17.7 fps. Nurbs uses a tile size of 100 × 100, and achieves a peak rendering performance of 6.1 million tris/sec at a rate of 14.9 fps. March uses a tile size of 200 × 200, and achieves a peak rendering performance of 4 million tris/sec at a rate of 10.6 fps. For each run, the display is a single 1600 × 1200 monitor. As the number of clients surpasses the number of servers, the performance of the application once again becomes limited by the interface.

> Each processor is responsible for its own portion of the scene database. Each frame of Hundy renders 4 million triangles, at a rate of 7.45 million tris/sec. On a single node, Hundy is completely limited by the interface to the graphics system; it cannot submit its data fast enough to keep the graphics system busy.

Scaling March, Nurbs, and Hundy using a single system is a challenging problem. Although other useful applications could be written that pose less of a challenge for WireGL, the applications we have chosen stress our implementation. Each application has very different load balancing behavior, requires immediate mode semantics, and generates a large amount of network traffic per frame. The speedup for these applications using 16 pipeservers is shown in figure 6.

5.2 Parallel Interface

To scale any interface-limited application, it is necessary to allow parallel submission of graphics primitives. To demonstrate this, we have run our applications in a number of different configurations, shown in figure 7. In these graphs, the tile size is chosen empirically, and Lightning-2 reconstructs a final 1600 × 1200 output image.[1] Each curve represents a different number of pipeservers, from 1 to 16. As the number of clients grows greater than the number of servers, the performance flattens out, demonstrating that such a configuration is once again limited by the interface.

Some of Hundy's performance measurements show a superlinear speedup; this is because Hundy generates a large amount of network traffic per second. This traffic is spread uniformly over all the servers, and when the number of servers is greater than the number of clients, each path in the network is less fully utilized. Essentially, this shows that Hundy's performance is very sensitive to the behavior of our network under high load.

WireGL's approach provides scalable rendering to applications with a variety of graphics performance needs. To measure scalability with a compute-limited application, we have artificially limited Hundy's geometry issue rate. The number of submitting clients is then varied while only using one pipeserver. The results of this experiment are shown in figure 8. For each test, the application scales excellently until it reaches the interface limit of the single pipeserver or the size of the cluster.

[1] Currently, Lightning-2 supports input resolutions up to 1280 × 1024, so for one pipeserver we bypass Lightning-2 and drive the display directly

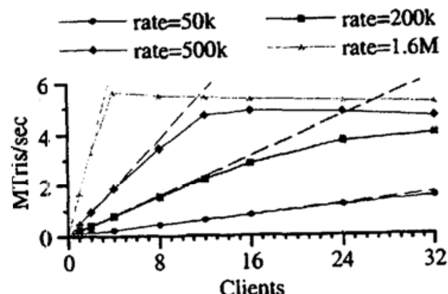


Figure 8: Scaling a compute-limited application with a single pipeserver. For each curve, Hundy's issue rate has been restricted. We achieve excellent scalability up to either the pipeserver interface limit, or the full 32 nodes of our cluster.

The results shown in figures 7 and 8 demonstrate WireGL's flexibility. Interface-limited applications can be scaled by adding servers and clients, while compute-limited applications can be scaled by adding clients only.

5.3 Hardware vs. Software Image Reassembly

The overhead of performing software image reassembly can quickly dominate the performance of an application as the output image size grows. Each node in our cluster has a pixel read performance of 28 million pixels/sec, and a pixel write performance of 64 million pixels/sec. If we can transmit 100 MB/sec of image data into a display node, this implies a maximum performance of 33 million pixels/sec for the visualization server. In practice, we achieve approximately half this rate in all-to-one communication, yielding a maximum frame rate of approximately 8 Hz at a resolution of 1600 × 1200.

To measure the overhead of the visualization server versus Lightning-2, we wrote a simple serial application that calls `SwapBuffers` repeatedly. The performance of this application represents an upper bound on the achievable framerate of any application. A serial application is a fair test because, as described in sec-

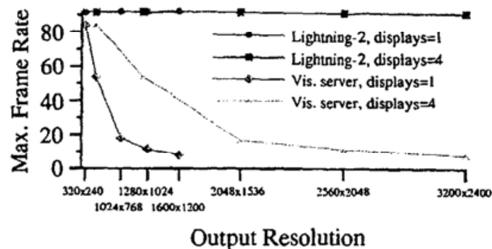


Figure 9: Maximum framerate achievable using Lightning-2 or the visualization server. As the image size increases, the expense of reading and writing blocks of pixels to the framebuffer quickly limits the visualization server to non-interactive framerates.

tion 3.1, only one node in a parallel application calls `SwapBuffers` for each frame. In each experiment, 12 pipeservers are used. The results are shown in figure 9. The "displays=4" curves are representative of a tiled display wall or a multi-input display such as IBM's Bertha.

This graph demonstrates that hardware supported image reassembly is necessary to maintain high framerates for most output image sizes. Lightning-2 is able to maintain a constant refresh rate of 90 Hz for any image size ranging from 320×240 to 3200×2400. The visualization server provides a maximum refresh rate of 8 Hz for a 1600×1200 image, which is approximately 46 MB/sec of network traffic. This is consistent with the measured bandwidth of our network under high fan-in congestion.

5.4 Load Balance

When evaluating a scalable graphics application, there are two different kinds of load balancing to consider. First, there is application-level load balance, or the amount of computation performed by each client node. This type of load balancing cannot be addressed by WireGL; it is the responsibility of the application writer to distribute work evenly among the application nodes in the cluster.

To evaluate application-level load balance, we measured the speedup of our applications in a full 32-node configuration without a network (i.e., discarding packets). In this configuration, March achieved 85% efficiency, Nurbs 98% efficiency, and Hundy 96% efficiency. From these results, we conclude that each application has a good distribution of work across client nodes.

The other type of load balancing is graphics work. For most applications, the interface to a single rendering server quickly becomes a bottleneck, and it is necessary to distribute the rendering work across multiple servers. However, the rendering work required to generate an output image is typically not uniformly distributed in screen space. Thus, the tiling of the output image introduces a potential load imbalance, which may in turn create a load imbalance on the network as well.

Because the triangles in our test applications are uniformly small, the server-side load balance can be reasonably measured by the total number of bytes sent to each server. For each application, the total incoming traffic when using one pipeserver is a lower bound on the total amount of network traffic for any number of pipeservers, since adding servers will result in some redundant communication. The overlap factor is the ratio of total traffic received by all servers to this lower bound, and the load imbalance is the ratio of the maximum traffic received by any server to the average traffic. In figure 10, the height of each curve shows the overlap factor. The error bars indicate the overlap if each server received the maximum or minimum traffic received by any server. The load imbalance is therefore the ratio of the maximum shown to the observed overlap factor for that number of servers.

As expected, the choice of tile size affects the load balance and the overlap factor. For smaller tiles, there is less variance in the total number of bytes received, resulting in a better load balance, but the overall average data transmitted has increased due to overlap. As the tiles get larger, the overlap is smaller, but longer error bars indicate a poorer load balance. At a tile size of 100×100, Nurbs has a load imbalance of 1.53 on 16 servers, while at 32 servers the load imbalance increases to 2.13. The load imbalance will continue to increase as the number of servers increases. Currently, Nurbs is sufficiently compute-limited that its load imbalance is not exposed in the speedup curve shown in figure 6. However, as cluster size increases, the increasing load imbalance will eventually limit Nurbs' scalability. Nonetheless, WireGL provides excellent scalability up to 16 pipeservers, which makes it a useful solution for many applications on many current cluster configurations.

To verify our assumption that the server load balance can be reasonably measured by simply counting network traffic, we ran all our measurements in a mode where the pipeservers discarded incoming traffic rather than decoding it. The performance measurements in this mode were almost identical to the measurements when graphics commands were actually executed. This demonstrates that the performance of interface-limited applications will largely be determined by the scalability of the network under heavy all-to-all communication, and not by the execution of the graphics commands. As networks improve, this effect will be reduced.

To fully understand our scalability results, we have measured the achievable send and receive bandwidths of our network when performing all-to-all communication. We performed this test in a partitioned configuration, in which sources and sinks run on different cluster nodes, and an unpartitioned configuration where sources and sinks run on the same cluster nodes. This test was performed with a WireGL-independent program in which each source node sends fixed-size network packets to all sink nodes in a round-robin pattern. The results are shown in figure 11. The partitioned dataset, shown with green crosses, achieves much higher overall performance, and has much less transmit bandwidth variance. For example, in an unpartitioned 18-way test, the transmit bandwidth ranges from 26.02 to 60.75 to MB/sec, while a partitioned run using 9 clients and 9 servers had bandwidths ranging from 93.92 to 96.96 MB/sec. It is interesting to note that any individual node will observe a very stable transmit bandwidth over the lifetime of its run. That is, the node achieving 26 MB/sec will always achieve 26 MB/sec, although varying the number of nodes will change which nodes perform poorly.

6 Discussion and Future Work

The real power of WireGL derives from its flexibility. Because WireGL is based on commodity parts, it is easy and inexpensive to build a parallel rendering system with a cluster. Although there can be a tradeoff between using commodity parts and parallel efficiency, the ability to reconfigure the system to meet an application's load balancing and resource needs is a large advantage for commodity-based parallel rendering solutions like WireGL on small to medium-sized clusters.

Because the techniques used to provide scalability are independent of specific graphics adapters and networking technology, any component in our system may be upgraded at any time to obtain better performance. In particular, we believe that WireGL's performance on a 16 to 32 node cluster will improve dramatically with the introduction of new server-area networking technology such as

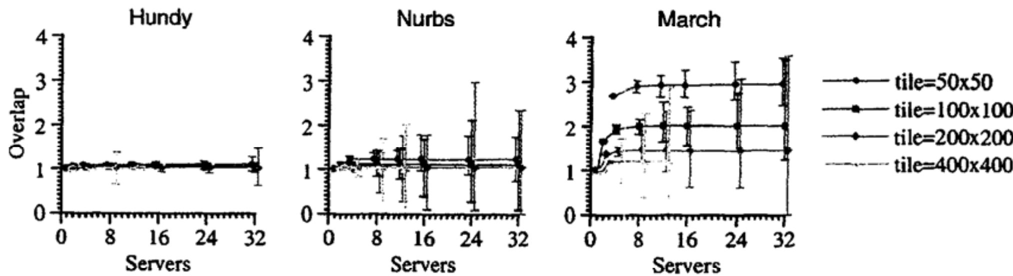


Figure 10: Overlap factor and load imbalance with various tile sizes on a 1600×1200 display. The height of each curve indicates the overlap factor, while the size of the error bars is proportional to the load imbalance. Increasing the tile size decreases the total amount of network traffic, but at the expense of load balance. Note that with a 400×400 tile size, only 12 total tiles are needed to cover the display, so no more than 12 servers can contribute to the final image.

InfiniBand. To achieve peak performance today, it is necessary to perform image reassembly after scan-out. Our Lightning-2 implementation is a large custom piece of hardware, but a smaller version could be built very cheaply and would enable the construction of a small, self-contained cluster that could act as a standalone graphics subsystem for a larger cluster.

6.1 Scalability Limits

We have demonstrated that WireGL's sort-first approach to parallel rendering on clusters provides excellent scalability for a variety of applications with a configuration of up to 16-pipeservers and 16-clients. Our experiments indicate that the system would scale well in a 32-server, 32-client setup if the cluster were bigger, or if the network had better support for all-to-all communication. However, there is a limit to the amount of screen-space parallelism available at any given output size. This limit will prevent a sort-first approach from scaling to much bigger configurations, such as clusters of 128 nodes or more. For clusters that large, the tile size becomes small enough that it is very difficult to provide a good load balance for any non-trivial application without introducing a prohibitively high overlap factor. One possible solution to this problem would be to provide dynamic screen tiling, either automatically (using frame-coherent heuristics) or with application support. We believe alternate architectures such as sort-last image composition would scale better on larger clusters, but this will likely come at the cost of ordered semantics.

6.2 Texture Management

WireGL's client implementation treats texture data as state elements, and lazily updates it to servers as needed. In the worst case, this will result in each texture being replicated on every server node in the system. This replication is a direct consequence of our desire to use commodity graphics accelerators in our cluster; it is not possible to introduce a stage of communication to remotely access texture memory.

WireGL's naive approach to parallel texture management can be a limitation for some applications. More work needs to be done in this area, and we are beginning to investigate new texture management strategies. One approach being considered will leverage our recent work in parallel texture caching [11].

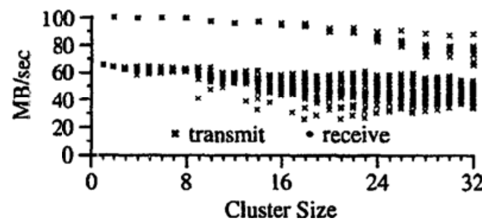


Figure 11: Transmit and receive bandwidth for Myrinet with all-to-all communication. For each cluster size, the observed send and receive bandwidth is plotted for all nodes. The top dataset represents a partitioned n-to-n run, where sources and sinks are not run on the same nodes. The bottom dataset is an unpartitioned run of all n nodes. Partitioning the cluster results in much higher bandwidth in general, as well as less transmit bandwidth variance.

6.3 Latency

There are two main sources of latency in WireGL: the display reassembly stage, and the buffering of commands on the client. When using Lightning-2, display reassembly will add exactly one frame of latency. While single-frame latency is usually acceptable for interactive applications, it can be a problem for certain virtual reality applications. The overhead of using software image reassembly will usually be much higher (on the order of 50-100 milliseconds), although it will vary with the image size.

The latency due to command buffering will depend on the size of the network buffers. WireGL's default buffer size is 128KB, which we can fill with geometry in half a millisecond, given our packing rate of 20 MTris/sec (recall that a triangle occupies 13 bytes in our protocol). Additional latency can occur due to network transmission, although the latency of most high-speed cluster interconnects is less than 20 μs. Finally, since the pipeserver cannot process the buffer until it has been completely received, we incur slightly over one millisecond of additional latency for a 128KB buffer on a network with 100 MB/sec of bandwidth.