AD-A228 616

DTIC
S ELECTE NOV 07 1990 D
D

# Displaying Alphanumeric Characters on Pixel-Planes 5

*TR89-049*

N00014-86-K-0680

*Andrei State*

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

University of North Carolina at Chapel Hill

Department of Computer Science

# Displaying Alphanumeric Characters on Pixel-Planes 5

*TR89-049*

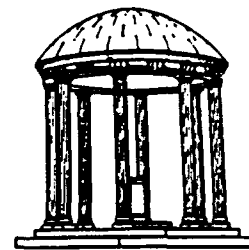
*December, 1989*

| Accession For | | |
|---|---|---|
| NTIS CRA&I | ✓ | |
| DTIC TAB | ☐ | |
| Unannounced | ☐ | |
| Justification | | |
| By per call | | |
| Distribution / | | |
| Availability Codes | | |
| Dist | Avail and / or Special | |
| A-1 | | |

*Andrei State*

Dist. "A" per telecon Dr. Ralph Wachter.
ONR/code 1133.

VHG 11/06/90



The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

*UNC is an Equal Opportunity/Affirmative Action Institution.*

# Displaying Alphanumeric Characters on Pixel-Planes 5

1. Introduction

2. The Rendering Process on Pixel-Planes 5

3. Three Methods for Character Rendering
   - 3.1 Real-Time Scan Conversion
   - 3.2 Character Bit-Blitting to Renderers
   - 3.3 Character Bit-Blitting in Graphics Processors

4. Conclusion

5. Acknowledgements

6. References

# 1. Introduction

The Pixel-Planes 5 system [Fuchs89] is an implementation of a high-performance graphics architecture designed primarily for high-speed rendering of 3-dimensional data from surface or volumetric databases. However, particular applications may require the additional display of alphanumeric text, either as superimposed (overlayed) alphanumeric text, or as true 3-dimensional text capable of interacting with other 3-dimensional elements (shading, hiding/intersecting, etc.). Different approaches that have been proposed and partially implemented so far will be described and compared.

# 2. The Rendering Process on Pixel-Planes 5

The rendering process in the Pixel-Planes 5 system is controlled by a designated graphics processor, the Master GP. The following outline describes this process (adapted from [Fuchs89] and [Brusq89]):

**Step 1**: Master GP requests new frame from other GPs.

**Step 2**: GPs interpret the database, generating Renderer commands for each graphics primitive. The commands are placed into the local bins corresponding to the screen regions where the primitive lies. Each GP has a bin for every pixel region of the screen.

**Step 3**: The GPs send bins containing commands to Renderers; these execute commands and compute intermediate results, which are then stored in each pixel's backing store memory.

**Step 4**: The GP sending the final bin to a Renderer also sends end-of-frame commands for a region. The Renderers execute these commands and compute final pixel values from the intermediate results.

**Step 5**: The Renderers send computed pixels to the Frame Buffer, one entire screen region at a time.

**Step 6**: When all regions' pixel values have been received, the Frame Buffer swaps banks and displays the newly-computed frame.

In order to display 2-dimensional antialiased alphanumeric text on a display driven by a Pixel-Planes-5 based system, these elements, which are supposedly part of the graphics database, must be passed through the graphics pipeline just as all other database elements, and must consequently be embedded in the rendering scheme described above. In the geometry subsystem, alphanumeric text strings are processed by the GPs and sorted into the bins for the various screen regions, thereby sorting into a region's bin only the characters which fall entirely or at least partially within that region. Thus, in general only single characters overlapping screen region boundaries will be sorted into more than one bin. The task of accessing and marking/updating all single pixels covered (totally or partially) by a specific character or vector (*scan conversion*) can be performed either in the Renderer units, or, alternatively, in the graphics processors. The latter alternative exists because the Pixel-Planes 5 system allows access to the video backing store memory elements from both pixel processors and graphics processors.

Figure 1 shows the organization of the backing store memory of a single Pixel-Planes 5 Renderer unit. A total of 128 32-bit words is available at each pixel (in addition to the fast

static RAM local pixel memory of 208 bits). A *sector* contains one 32-bit word for each of the Renderer's $128^2$ pixels, arranged in 128 *scanlines* à 128 horizontal pixel-words each. When transferring data to and from the backing store via the Ring Network Port, an integer number of scanlines must be transferred, even if only the contents of a single bit of a particular pixel are to be accessed. Transfers between the fast local pixel memory and the Backing Store (done by the parallel pixel processors) involve the transfer of one 32-bit word at a time.

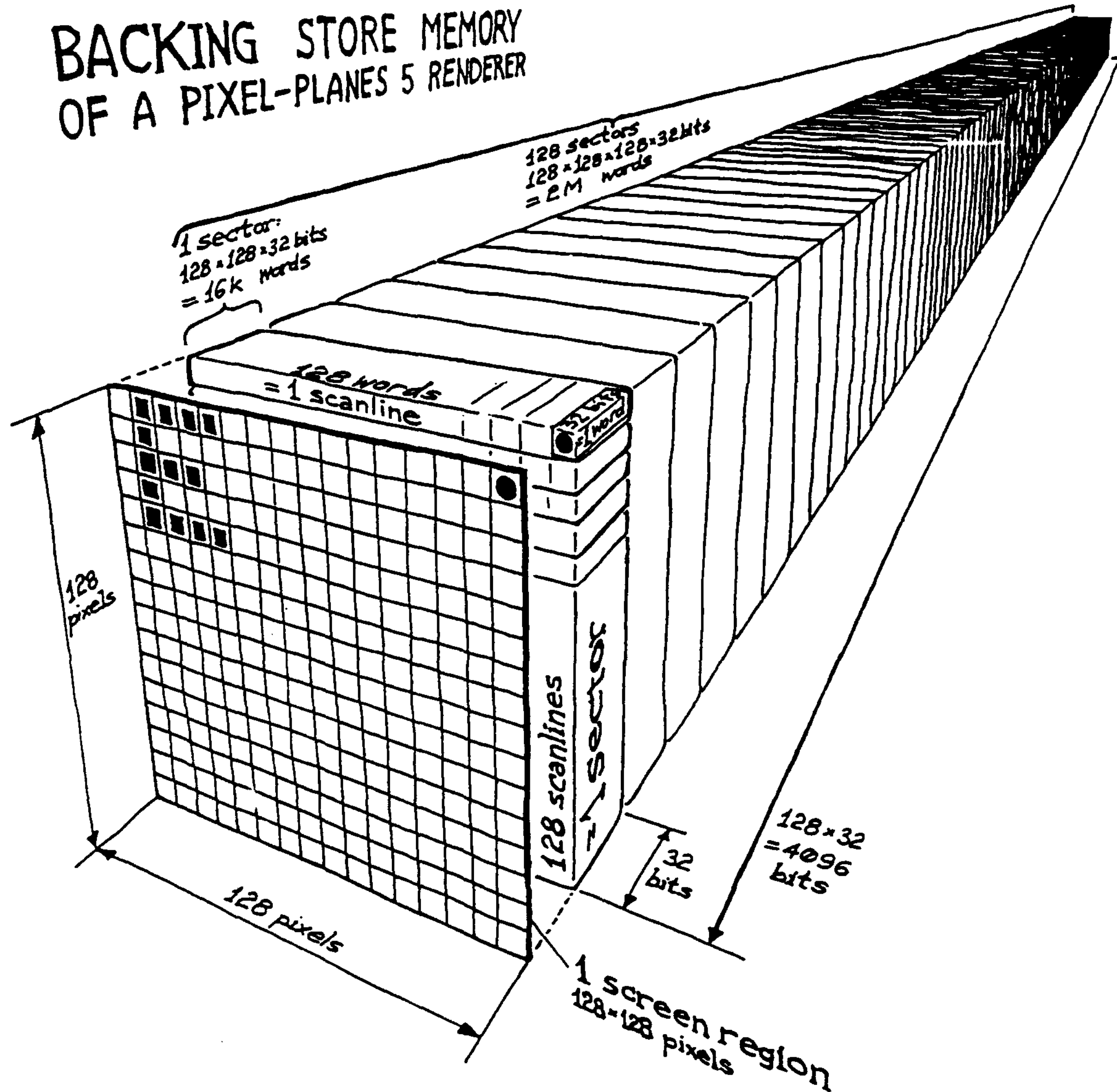



Figure 1 Backing Store memory organization for a single Pixel-Planes 5 Renderer

If the alphanumeric characters are to be overlayed over a specific background, then they

must be composited with it in the Renderers. Since each screen region is visited only once by a Renderer, compositing of the different layers of the final image can be accomplished by each Renderer just before it sends the final pixel values to the Frame Buffer and "moves on" to the next screen region it is assigned to. According to [Turk89], one such composition step, i. e. blending one foreground layer with one background layer, requires approximately 1,000 machine cycles (rough estimate), which corresponds to 25 microseconds or 1/1,667th of the time available for a single frame (assuming 40 MHz operation and a 24 Hz update rate).

# 3. Three Methods for Alphanumeric Character Rendering

Character fonts can be represented either as 2-dimensional, simply or multiply connected regions bounded by line segments and possibly also by conic curves (in order to take advantage of the Quadratic Expression Evaluators (QEEs) in the Pixel-Planes 5 Renderers) or, alternatively, as pixel arrays (i. e. bit patterns) with a specific number $b$ of bits for each pixel covered, thus providing $2^b$ levels of partial pixel coverage (for antialiasing). Since the earlier reprentation is perfectly adapted to scan conversion performed in the Renderers, we will take only the following three alphanumeric character drawing methods into consideration:

1. Scan conversion in Renderers,from 2-d region boundary representation
2. Bit pattern copy in Renderers,from bit pattern representation
3. Bit pattern copy in GPs,from bit pattern representation

Note that methods no. 2 and 3, i. e. using either the pixel processors or the graphics processors for character drawing from a bit pattern (i. e. already scan-converted) representation merely amount to placing the font matrices for a string of characters to be displayed at the appropriate pixel locations.

For either of the three alternatives, final composition is most efficiently performed by the pixel processors in the renderer units as outlined above, taking full advantage of the parallel architecture. The $b$-bit -per-pixel representation of the fonts in the bit pattern representation scheme (methods no. 2 and 3) is treated as a transparency mask during this process.

## 3.1 Real-Time Scan Conversion

Method no. 1 has been studied in depth at UNC by visiting researcher Hervé Tardif, who has also implemented it on the Pixel-Planes 4 system. On this system, the fonts were represented as simply or multiply connected 2-dimensional polygonal regions (polygons with zero or more holes); since the single Renderer in Pixel-Planes 4 does not have a QEE, the Bézier curve segments in the initial representations were approximated by straight line segments (using the de Casteljau algorithm). On the average, 50 straight line segments were used per character. Tardif's implementation performs at only 2,500 characters/sec [Good89].

A significant speedup is expected from the implementation of this method on the Pixel-Planes 5 system [Fuchs89]. The higher clock frequencies used throughout and the Renderer's integrated QEEs allow predictions on the order of approximately 20,000 characters per second for a single Renderer, thereby using a boundary representation consisting of straight line segments and conic sections instead of Bézier curve segments (adapted for scan conversion with quadratic expressions), but otherwise exhibiting the

same level of detail as the fonts used in the Pixel-Planes 4 implementation. The performance figure applies to characters of arbitrary size (unless the characters are large enough so that a significant fraction of all on-screen characters fall within more than one screen region, thus effectively increasing the number of characters to be scan-converted), but without antialiasing, the incorporation of which may further increase the time complexity of this method: for example, if the fonts were to be supersampled at a rate of 2x2 = 4 samples per pixel, the capacity of one Renderer unit would drop to about 20,000/4 = 5,000 characters per second..

A characteristic of this method is that since fonts are represented as true 3-dimensional elements in the database, they can be rendererd using techniques available for 3-d rendering, e. g. shading with ambient, diffuse, and specular components. Furthermore, they can be positioned and oriented in 3-d space, thus possibly hiding or being hidden by other 3-d elements in the database. This method can also be extended to generate character fonts with "thickness".

```
                                                                          cycles:
/* Renderer microcode to draw a string of characters */
set number of fractional bits to 0                                          10
disable pixels outside of rectangle containing text (4 linear expressions)  40
/* Horizontal pass: */
for each of the wn pixel columns inside text rectangle
    store pixel column font data into local pixel memory                    bh
    disable pixel column (1 linear expression)                              10
/* Now each of the pixels in the text rectangle contains all pixel */
/* values for the pixel column it belongs to in its local memory */
re-enable text rectangle                                                    40
/* Vertical pass: */
for each of the h pixel rows inside text rectangle
    select row pixel data from values stored during horizontal pass
    move data to final value locations in local pixel memory                2b
    disable pixel row (1 linear expression)                                 10
/* Now each of the pixels in the text rectangle contains the correct */
/* pixel data in the final value locations in its local memory */
re-enable text rectangle                                                    40
restore number of fractional bits                                           10
discard all data stored during horizontal pass
```

**Figure 2** Implementation of character drawing in the Renderers (method no. 1)

## 3.2 Character Bit-Blitting to Renderers

The approach of method no. 2 [Turk89] requires the character fonts to be stored as bit patterns (transparency mask) somewhere in memory addressable by the GPs (or by a single GP, if the system is set up so that a particular GP is allocated the task of handling all character strings). When sorting into the bins associated with the screen regions, character groups (i. e. adjacent characters occupying a rectangular area of the screen), hereafter referred to as *strings*, are not split into single characters; instead, Renderer instructions to process entire strings at a time are generated, more exactly in the form of pixel processor instructions to write values extracted from the bit pattern arrays available to the GP(s) into the Renderers' pixel memory bits. Figure 2 outlines the core part of the pixel processor code implementing the proposed algorithm.

The time complexity of this algorithm depends on the size of the characters and on the number of bits per pixel used for the font's transparency mask in the bit pattern representation. If we denote a character's width and height (in pixels) by $w$ and $h$ respectively, and the number of bits per pixel by $b$, then, assuming a text string with $n$ characters, we obtain the following approximate formula for the number of cycles $c$ for the core portion of the algorithm [Turk89]: $c = 140+b^2+24b+wn(bh+10)+h(2b+10)$.

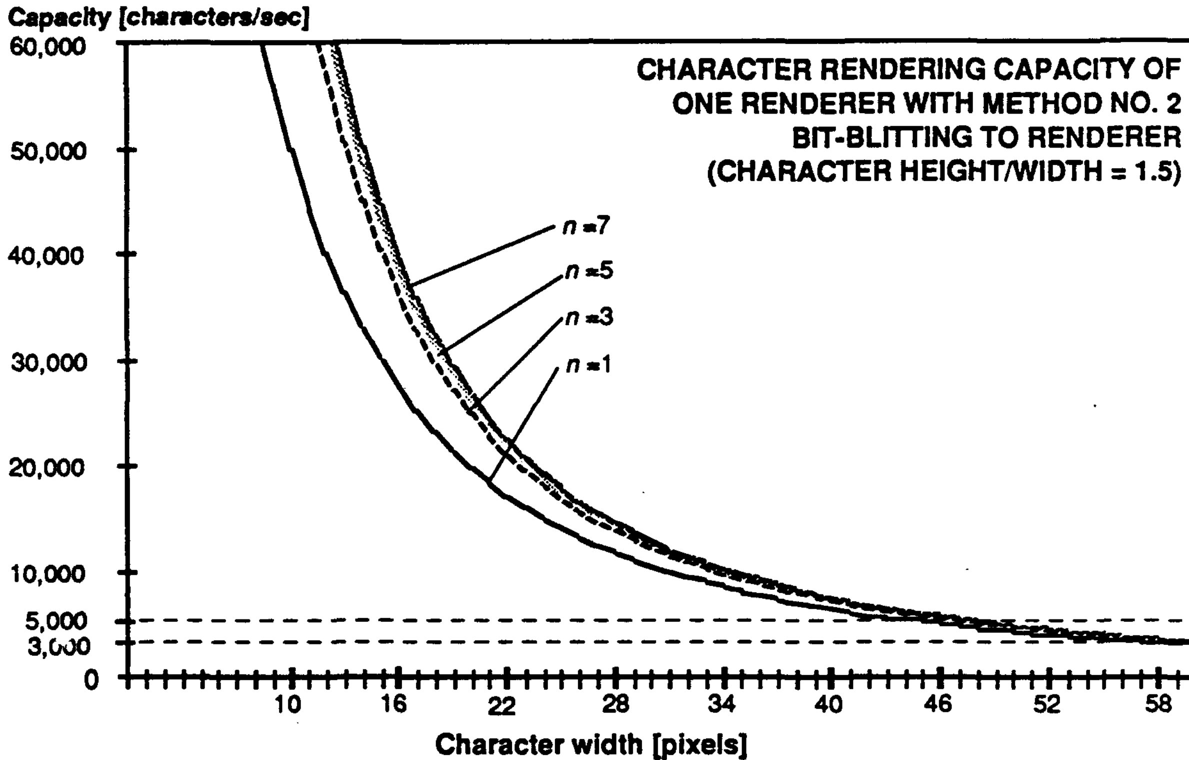



**Figure 3** Character rendering capacity of one Pixel-Planes 5 Renderer for various character sizes and numbers $n$ of characters per string, assuming 2 bits per pixel, a 3/2 character height/width ratio, and 40 MHz operation

Figure 3 is a plot showing the character rendering capacity of a single Pixel-Planes 5 Renderer for different character sizes and a character height-to-width ratio $h/w = 3/2$, thereby also assuming a representation using a 2-bit-per-pixel transparency mask providing 4 opacity levels per pixel, which we consider a minimum requirement for antialiasing. Several curves for different values of $n$ are given, showing that simultaneously rendering multiple characters (strings) eliminates a certain amount of fixed overhead; but obviously a small value of $n$ is sufficient to ensure this since there is no significant improvement in performance beyond $n = 3$. Hence assuming that characters are displayed as strings with an average length of 3, and for a character size of 20-by-30 pixels, we obtain a performance of nearly 25,000 (antialiased!) characters per second.

The graph shows that this algorithm is considerably faster than the previously discussed method, but only if the characters used are small; more precisely, for $n \geq 3$, method no.2 is superior to method no. 1 if the character matrix is smaller or equal to 48-by-72 pixels, which represents a Renderer performance of approximately 5,000 characters/second, with a quality of antialiasing comparable to the one achievable using method no. 1 in connection with 2-by-2 supersampling.

However, the algorithm of method no.2 does not make efficient use of the massively parallel structure of the Renderers: due to the SIMD concept, the pixel processors must essentially loop over every single pixel column and then over every single pixel row in the rectangle containing the text string.

### 3.3 Character Bit-Blitting in Graphics Processors

*In the approach of method no. 3* [Fuchs89a] *much of the computational burden is moved from* the Renderers to the GPs responsible for character string processing. This concept evolved from the understanding that in the previous method, character processing is performed on a bit-sequential basis in the pixel processors; since these are not very efficient in this mode, it was suggested to use one or more GPs for this task. The basic approach is very simple: transformations and screen region distribution (bin sorting) are accomplished as in method no. 2. But instead of sending specific character drawing commands to the Renderer currently in charge of the screen region the string falls into, the GPs extract the bit patterns from the font arrays and store them directly into the Backing Store memory of the Renderer, at the appropriate pixel locations. The GPs then merely send the end-of-frame composition commands to the Renderers, whose pixel processors use the pixel values from the Backing Store in order to composite the final pixel colors for the Renderers' assigned screen regions (the Renderers' only remaining alphanumerics processing task under this approach).

For the following time complexity estimations, we assume again a 2-bit-per pixel transparency mask. In the Pixel-Planes 5 system, the GPs access the Backing Store memory of the Renderers via the ring network. One communication channel has a bandwidth of 20 million 32-bit words per second, thus imposing a theoretical upper limit to the character rendering capacity of a single GP. In order to determine this limit, we must also respect the limitations imposed by the data format for transmission of pixel values to the Renderers' backing store memories. According to [Eyles89], an entire 32-bit word must be transmitted per pixel (even if we only use 2 bits for our transparency mask character font representation). Moreover, an entire Renderer scanline (i. e. 128 horizontal pixels if *the Renderers are configured as 128-by-128 pixel devices) must be transmitted if a pixel* on that scanline is to be modified. Hence the average number $m$ of characters intersected by a Renderer scanline must be taken into account. For example, in order to transfer the character "E" in figure 1, the upper 5 scanlines must be transferred to the Renderer, a total of 5x128x32 = 20480 bits, whereas the font's bit pattern requires only $5 \times 4 \times b = 40$ bits (for $b = 2$)!. In this example, $m$ has a value of 1. Note that $m$ is independent of the number $n$ of characters scan-converted simultaneously (as a string) in method no. 2, which depends on the *logical* organization of the text items on the screen, whereas $m$ depends on the *physical* distribution of text strings on the screen. The (integer) values for $m$ range from 1 to INT(128/$w$). A second example might serve to clarify this: assuming a character size of 20-by-30 pixels, a screen filled with (adjacent) alphanumeric characters corresponds to $m$=INT(128/20)=6, whereas a screen in which characters are arranged in vertical columns that are 128 pixels apart, so that a Renderer's screen region is intersected by only one column, corresponds to $m=1$. Hence for this method, the theoretical limit imposed by the communication bandwidth is $c = 20{,}000{,}000m/128h$ characters/second. Figure 4 shows plots of $c$ for different values of $w$ and $m$, assuming again a character aspect ratio $h/w = 3/2$.

As far as the actual bit pattern manipulations in the GPs are concerned, the algorithm implementing this method must copy the character bitmask from a font table into an area of memory reserved for generation of final-image pixel data. Typically, a GP might have to manage several such bitmap areas, each of which would have a size equal to that of a screen region processed by a Renderer unit; there would be a one-to-one correspondence

between these bitmap areas and the GP bins used for the sort middle process.

We assume that the character font is stored (in a wasteful way) in a table with 1 byte per pixel, even if we use only 2 bits from each byte for the transparency mask font representation. We also assume a 20-by-30 pixel character matrix. Thus a complete set of printable ASCII characters would occupy no more than than 96*20*30 = 57600 bytes, which can be stored in a single ROM circuit. This storage schema may be optimized, at the expense of higher computational overhead. However, even with the computationally most efficient font storage schema, it is unlikely that the time complexity be lower than 2 cycles per pixel copied into a bitmap region, or 1,200 cycles per character. Assuming 40 MHz operation, a single GP can then render approximately 33,000 characters per second. These can actually be transmitted to the Renderers at this rate only if $m = 6$, which is unlikely to be an average value unless the application is such that screen regions are "filled" with alphanumeric characters. Thus the communication through the Ring Network is obviously the bottleneck in this configuration. A more probable average value, $m = 3$, leads to a maximum possible transmission rate of 16,000 characters per second, comparable to the performance achievable with method no. 1.

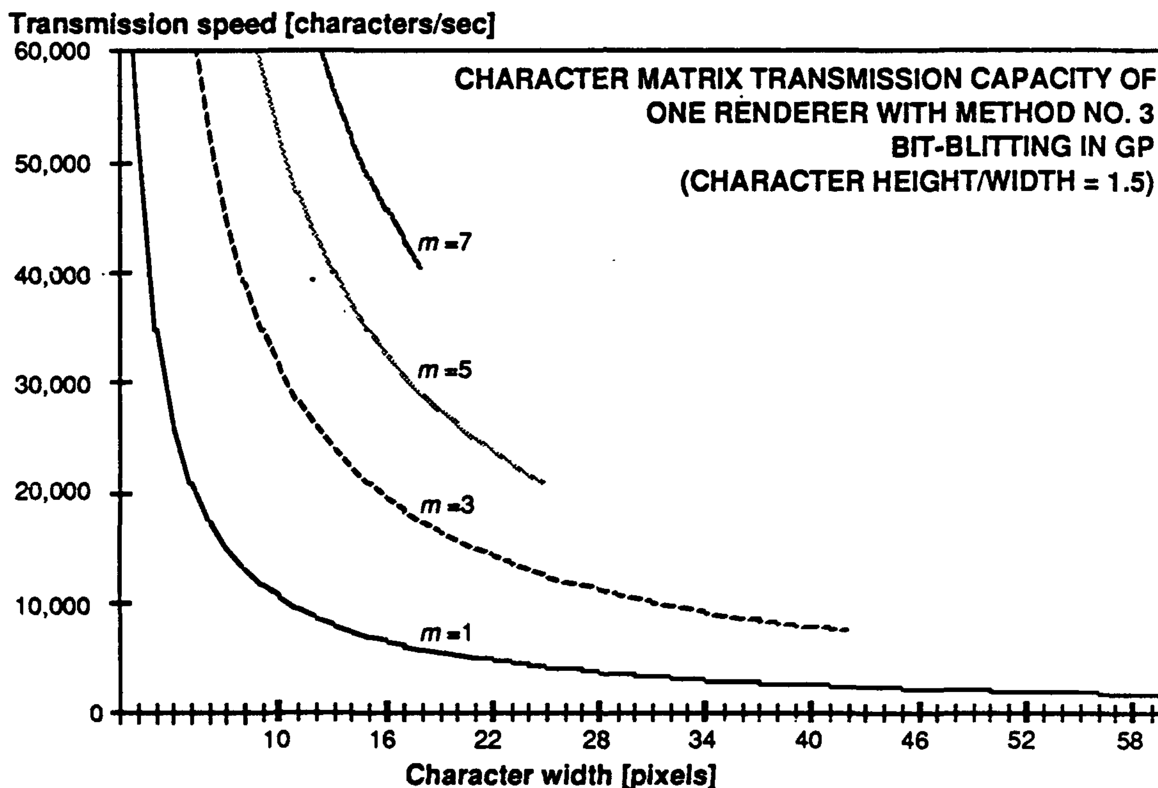



**Figure 4** Speed limit for transmission of character matrices into a Renderer's backing store. See text for definition of parameter $m$.

The transmission rate could be increased by packing the bit-pattern representations for several screen regions into a package of transmitted 32-bit words [Tebbs89]. For $b$ bits per pixel, character bit pattern data for $32/b$ screen regions could be transmitted simultaneously. Thus, for $b = 2$, the transmission rates shown in figure 4 could theoretically go up by as high a factor as 16. However, this assumes that characters are distributed on the screen in such a way that the same scanlines have to be transmitted for

each screen region, which is highly improbable except in situations when several screen regions are completely filled with alphanumerics. However, packaging the bit pattern information for several screen regions into single words requires more computation (shifts, logical operations) on a per-pixel basis in the GPs, thus increasing the cycle count to at least 4 cycles per pixel. Consequently, the character rendering capacity of a single GP would drop by a factor of approximately 2. In the example using 20-by-30 pixel character matrices, a single GP could render only about 16,000 characters per second, and thus couldn't possibly take advantage of the increased transmission capacity.

## 4. Conclusion

Table B.1 summarizes this analysis of different methods for character rendering. We conclude that method no. 1, while suitable for rendering of high-quality text of arbitrary size, and extensible to applications involving 3-d text interacting with other 3-d elements, is computationally too expensive for applications requiring relatively small 2-dimensional characters, which cannot take full advantage of a high-quality representation. Out of the other 2 methods, method no. 2, while not exhibiting any disadvantage over method no. 3, has the advantage of requiring less communication bandwidth between GPs and Renderers.

---

**1. Real-time scan conversion in Renderers from 2-d boundary representation**

- Rendering time independent of character size, efficient for large characters (above 48-by-72 pixels), but inefficient for smaller ones.
- Character font storage independent of character size on screen, efficient for very large characters, but inefficient for small and medium-sized ones.
- Can be extended to 3-d shaded characters with "thickness", interacting with other 3-d elements.
- Antialiasing by supersampling costly, but of arbitrarily high quality.

**2. Bit-blitting to Renderers from transparency mask bit pattern representation**

- Rendering time dependent of character size, efficient for small characters, but inefficient above 48-by72 pixel character size.
- Character font storage dependent on character size on screen, efficient for small and medium-sized characters, but inefficient for very large characters
- Only for 2-d characters.
- Antialiasing through transparency mask less costly than method no.1.

**3. Bit-blitting in GPs from transparency mask bit pattern representation**

- Rendering time depending on character size, inefficient for large characters.
- Character font storage dependent on character size on screen, efficient for small characters, but inefficient for large characters, and always less efficient than with method no. 2
- Only for 2-d characters.
- Antialiasing through transparency mask less costly than method no.1.
- Requires transmission of large amounts of pixel data from GPs to Renderers.

---

**Table B.1** Summary of 3 possible methods for character drawing on a Pixel-Planes 5 based system.

## 5. Acknowledgements

This research has been partially supported by USAF Project No. 0100, contract number F33615-89-C-1848, "An Architecture for Advanced Avionics Displays", issued by the Cockpit Integration Directorate of Air Force Wright Aeronautical Laboratories. Additional funding was provided by the Defense Advanced Research Projects Agency, DARPA ISTO Order No. 6090, the National Science Foundation, Grant No. DCI-8601152, and the Office of Naval Research, Contract No. N0014-86-K-0680.

I would like to thank Henry Fuchs, Principal Investigator on the USAF project. Thanks also go to John Eyles and to my colleagues on the Pixel-Planes Team, especially Greg Turk, who provided and implemented the algorithm of method no. 2. Both Greg Turk and Brice Tebbs reviewed and commented on a draft version of this document.

## 6. References

**[Brusq89]**: Brusq, Roger, "High-Performance Computer Graphics Architectures," *Technical Report TR89-026*, Department of Computer Science, University of North Carolina at Chapel Hill, July 1989.

**[Eyles89]**: Eyles, John, "Renderer Overview," Chapter III.4 in the *Pixel-Planes 5 System Documentation,* Department of Computer Science, University of North Carolina at Chapel Hill, 1989.

**[Fuchs89]**: Fuchs, Henry, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *ACM Computer Graphics*, Volume 23, Number 3, July 1989.

**[Fuchs89a]**: Fuchs, Henry, *Personal Communications,* September-December 1989.

**[Good89]**: Good, Howard, *Personal Communication,* November 1989.

**[Tebbs89]**: Tebbs, Brice, *Personal Communication,* November 1989.

**[Turk89]**: Turk, Gregory, *Personal Communications*, November 1989.