Filename [FILIPAK.OMNI]THEORY.LST

THE THEORY BEHIND OMNI MARCH 7, 1984

A PRIMER ON DIGITAL GRAPHICS FOR BROADCAST TELEVISION

FOREWORD

A dirth of technically competent management is a serious handicap that a technologically based company can ill afford. A prevailing view is that Atari's corporate management is not just ill-equipped to understand the issues inherent in the computer and video graphics industries, but, worse still, that management is actively unconcerned with such details, that is, that they don't want to know such unfathonable things. The fear of technology runs rampant in today's society. Should this fear not naturally find its way into the corporate board room? It would be very surprising if it did not. Technological leadership and innovation go hand in hand. In their roles as pace setters, corporate leaders are obliged to become at least modestly conversant with the technology which they promote. I prepared this document with that end in mind. Though intended as an OMNI promotional piece, it contains much practical knowledge relating to television and digital video graphics. (The document has been organized in such a way that the OMNI material can be deleted by ripping out the pages specifically dealing with it.) My co-workers have commented that if such a document were generally available in the past, the present Atari products would probably be better and cheaper. I have tried to present all subjects in a nontechnical way. The feedback I have gotten is that I have generally succeeded. So, please persevere. If you have any questions or comments, you will find me a good listener.

MARCH, 1984 MARK FILIPAK

A PRIMER ON DIGITAL GRAPHICS FOR BROADCAST TELEVISION

MARCH, 1984 MARK FILIPAK

CONTENTS TOPIC NUMBER

CHAPTER 1 -- RESOLUTION AS IT RELATES TO GRAPHICS 1

Graphics designers and engineers call for ever higher resolution to produce better graphics. Many insist that standard broadcast TVs are not up to the task and, therefore, champion the use of expensive high resolution monitors. Other people are beginning to question this. They ask, "Why can't Atari make a system that looks as good as my home TV?"

I question the requirement for a monitor. I believe that a standard broadcast TV can produce astounding graphics. All I have to do is tune in a standard broadcast channel and look at it. (The running joke is "Wow! Great graphics. How'd they do it?") Considering this obvious capability, I reviewed Atari's present and planned products and those of our competition and rejected them as simply mismatched to the standard broadcast television medium.

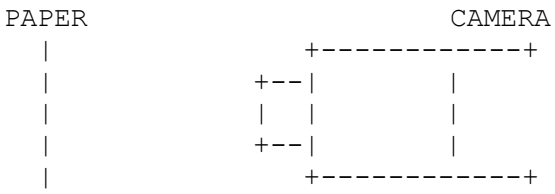
Part 1 -- Pixel Resolution 1.1

A pixel is defined as the smallest unit that can be created in a graphic medium. I will show that the pixels that can be generated on a standard broadcast TV are of such small size that they can be realistically termed "high resolution". Broadcast TV is a medium of rigid limits. Some of these limits are built into the transmitter, some are within the receiver and some are a consequence of human physiology. I will show that if these limits are recognized and properly allowed for, the result can surpass the displays and performance of arcade game machines. If the assumption is that higher resolution means higher frequency, then the system designer is forced by conventional logic to specify a monitor from the outset. The logic of frequency limitations will always justify that decision.

Pixel Resolution -- Luminance Limit 1.1.1

Imagine a camera connected to a TV through a stand- ||||||||||||
ard broadcast modulator. Imagine, further, a large ||||||||||||
sheet of white paper facing the camera upon which a ||||||||||||
pattern of black vertical bars are drawn like this .. ||||||||||||

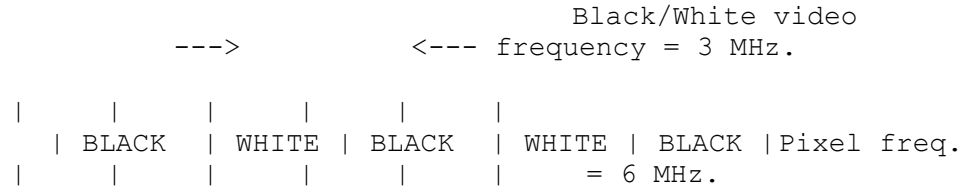
The physical setup looks like this:



What would you see on the television? Assuming there was adequate lighting, you would probably see the bars, but you might see one or more colors, instead. It depends upon the number of bars that were within the camera's field of view. Let me explain. Suppose that you did, indeed, see bars. If you then dollied the camera away from the paper more and more ever thinner bars would be within its field of view and, hence, the TV would show more and more ever thinner bars. But as you dollied back keeping an eye on the TV you might be surprised to discover that there was some kind of threshold position beyond which the TV display magically switches from bars to a shimmering rainbow of colors. Dolly back in across this threshold and the bars reappear, dolly out and, like magic, colors. What's happening here?

The phenomenon is called color artifacting. It is the reason TV performers are prohibited from wearing tightly striped or checked clothing. The problem is not in the camera or in the modulator. It's an inherent propertly of the transmission medium. The TV has a limit of luminance resolution beyond which the alternating black and white bars become so small and tightly packed that they confuse the electronics into interpreting them as color. You would count 324* bars (162 pairs) at the threshold no matter what what size TV you used. These 324 bars (equivalent to a digital frequency of 3 mega-hertz) represent a limit to luminance resolution.

Now that it has been established that 324 black/white alternations is the limit, does it make any sense to produce higher resolution? Yes, it does, as the following chart will illustrate:



```
| B :   | W :   | B :   | W :   | B :   |
| L : G | H : G | L : G | H : G | L : G |   Pixel freq.
| A : R | I : R | A : R | I : R | A : R |   = 12 MHz.
| C : E | T : E | C : E | T : E | C : E |
| K : Y | E : Y | K : Y | E : Y | K : Y |
```

* Comb-filtered TVs and some high quality TVs can display more than 324 black/white bars, but this is a good practical limit.

There is one important fact from this discussion which relates negatively to Atari's present products. Since the pixel clock is fixed at the color sub-carrier frequency of 3.58 MHz, the following pixel sizes, frequencies and pixel counts result:

PIXEL SIZE	FREQUENCY*	PIX/LINE+	COMMENT
half-clock	3.58 MHz	352	Useless - color artifacts
full-clock	1.79	176	Highest useful resolution

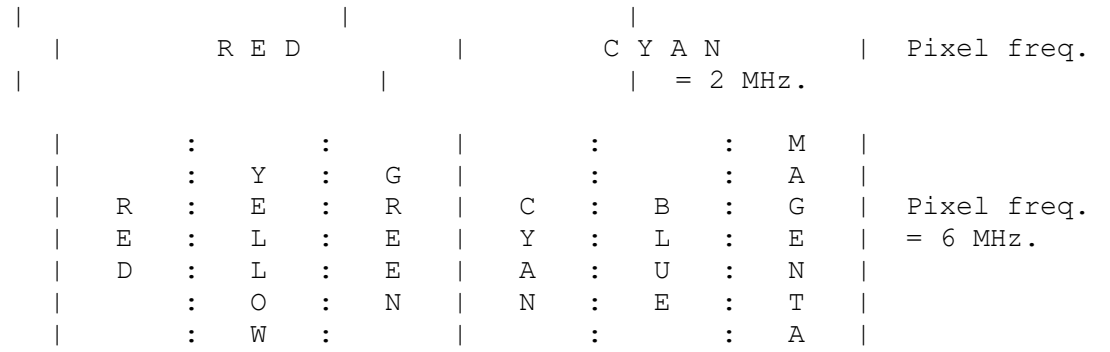
Using 3.58 MHz as the pixel clock was the worst possible choise. The optimum pixel frequency is 3 MHz and integer multiples of 3 MHz. This would have resulted in a highest useful resolution of 294 pixels/line instead of 176 pixels/line -- a 67% increase.

Pixel Resolution -- Chrominance Limit 1.1.2

Suppose you repeated the bar exercise, but instead of alternating black/white bars, you used alternating color/complimentary-color bars (eg, red/cyan, yellow/blue or green/magenta ...) You would find a second threshold at approximately 1/3 the distance from the paper. At a distance less than this chrominance threshold, the TV would display the color bars, but beyond the threshold, gray. You would count 108 bars (54 pairs) at the threshold (equivalent to a digital frequency of 1 MHz)which, once again, would be independent of the TV's size.

Now that you have established that 108 color/complimentary-color alternations is the limit, does producing higher resolution make any sense? Yes, it does, as the following chart will illustrate:

```
          Red/Cyan video
 <------------ frequency = 1 MHz. ------------>
```



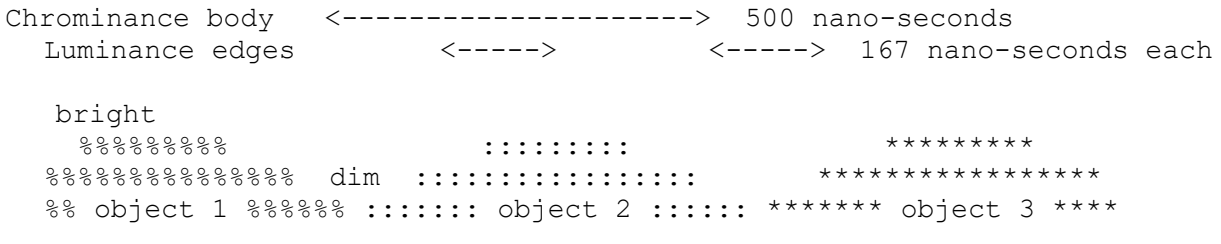
* Highest video frequency of Black/White bars

\+ Line duration used in present products is 49 micro-seconds not 54.2 micro-seconds. There is a 5.2 micro-second underscan.

Luminance variations which have no aggregate alternating characteristic present no luminance bandwidth limitation. However, if aggregate luminance variations produce even one cycle of alternation above 3 mega-hertz, then color artifacting can occur.

Chrominance variations are produced by rotation of a color vector. In the previous example, a sweep from red to cyan represents a 180 degree rotation of the color vector. Likewise, cyan back to red is another 180 degrees so that red to red is full circle. Since a scan line is 54 micro-seconds in duration and 54 such chrominance alternations can be accomodated per line, then one alternation is 1 micro-second long. Since this represents a complete rotation of the color vector, then it is more meaningful to say that the limit of chrominance resolution of broadcast TV is one rotation of the color wheel per micro-second.

Here are the brightness profiles for the smallest objects a broadcast TV can easily produce:

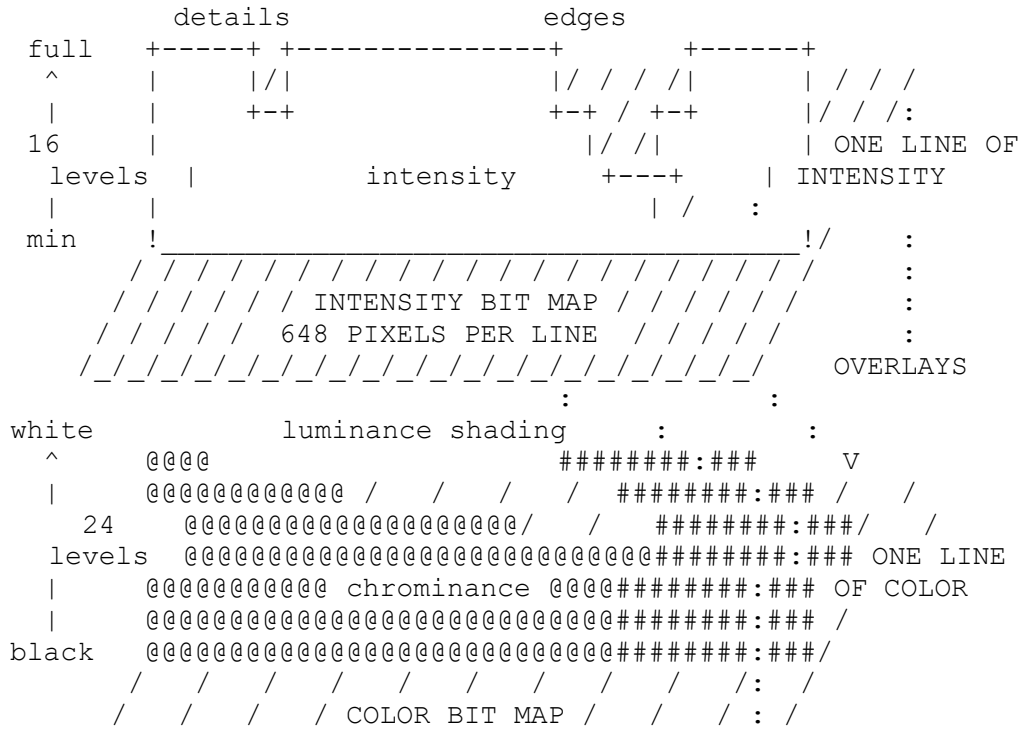


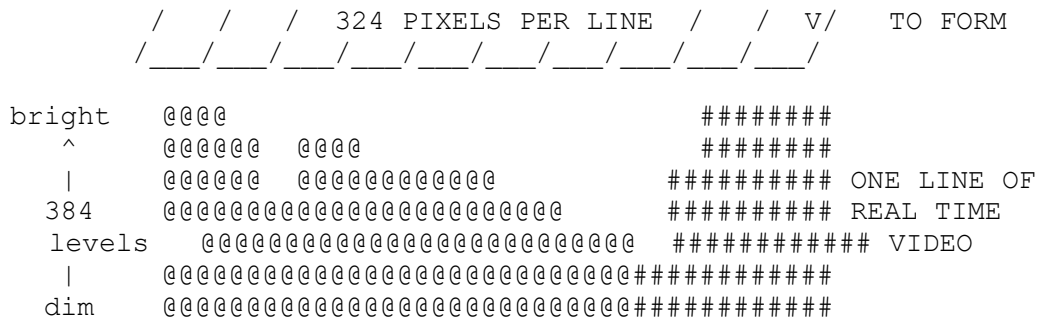
Objects 1, 2 and 3 cannot possibly produce excessive chrominance changes and their edges cannot possibly produce color artifact-

ing. A standard TV can display 108 such objects without violating either chrominance or luminance limits. At this point the reader must decide whether displaying 108 colored objects per line, with 1/10 inch diameters and 4/100 inch shaded edges each (on a 25 inch TV), can be called "high resolution". If it can, we must conclude that high resolution can be acheived on a standard broadcast TV. With judicious choices for adjacent colors, even higher resolution can be achieved.

Pixel Resolution -- OMNI 1.1.4

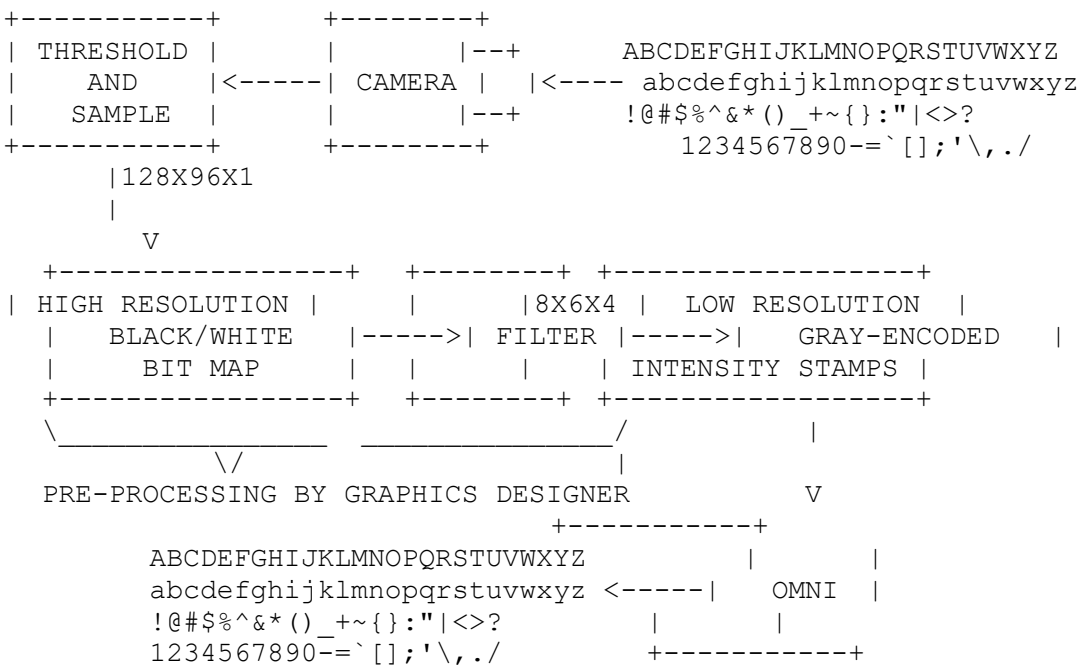
Imagine two bit maps. One has 480 lines of 324 color-pixels per line to portray the chrominances and luminances in a scene; each color-pixel can be any one of almost 2800 colors. The other has 480 lines of 648 intensity-pixels per line to define edges or to do shading, texturing, etc.; each intensity-pixel can be any intensity from full (as determined by color) to black in 16 steps. Further, imagine that these two bit maps are overlaid, intensity upon color, in the output to produce the TV image. That is OMNI.





OMNI overlays intensity upon color to create the output.

The 16 intensity levels per intensity-pixel can also be used for a a dynamically redefinable gray encoded character font. Instead of displaying hard-edged white characters on a black background, OMNI can produce soft-edged gray-level characters and overlay them upon a colored background to produce greater character recognition.



Creation of a gray-encoded font from a camera input.

With gray-encoded characters stored in the intensity-pixel stamps maps it may be possible to produce readable 80 character text on a standard broadcast TV. This intriguing possibility and some complications will be explored further in the next section.

Part 2 -- Physical Dot Resolution 1.2

The previous discussion accurately presented the electrical resoluiton limits of NTSC transmission and reception. One would think that these should be the only limits upon resolution of a graphics image but, unfortunately, there can be a mechanical limit as well. Along the inner surface of the TV screen there lies a metal plate, called the shadow mask, which is perforated by thousands of holes (or slots). Electrons fired from the color guns must pass through these gaps to reach the screen. The mask's function is to isolate the colored dots (or stripes) from one another so that stray electrons intended to excite one dot are prevented from striking its neighbors. This has the effect of sharpening what would otherwise be a somewhat fuzzy image. In analog TV the shadow mask does not degrade the intended image, but in digital TV the shadow mask can present a problem that I call the "shooting gallery".

Physical Dot Resolution -- The Shooting Gallery 1.2.1

Imagine that you are at a shooting gallery which has several hundred uniformly spaced targets continuously parading from right to left at a uniform rate. You have two guns available, a low resolution shotgun and a high resolution rifle. There is one catch. The guns cannot be turned. They only point straight ahead.

You select the shotgun and fire. It's easy. Four targets are knocked down. No challenge here. You can fire whenever you want and you hit four targets every time. No great resolution, though. Obviously if you want to hit just one target you have to switch to the rifle; now comes the challenge. Not being able to aim to left or right, you have to time your shots carefully. You find that the task requires you to establish a rhythm to your firing. Too fast or too slow a rate results in hit-hit-miss-hit-hit-miss... or some such pattern. After spending ten dollars you establish the right

cadence and mow down the entire row. The barker hands you a three dollar stuffed animal and asks you to move on.

So you go to the next gallery. You spot a passing friend and say, "Hey! Watch this." Using the same rhythm you established at the last booth you time you first shot just right and continue firing. Hit-hit-miss ! Your friend walks away chuckling. They tricked you of course. This booth requires a unique cadence. The targets are not spaced the same as they were at the last booth, that is, their pitch is different. Had you used the shotgun at both, chances are you wouldn't have noticed this discrepancy, but with the rifle it is obvious.

In TV, the targets are the holes in the shadow mask through which the electrons must pass. The rate at which the string of targets moves is a constant, one full line in 54.2 micro-seconds, but the spacing (pitch) between targets and the total number of targets is different from one size of TV to another and from one manufacturer to another for equally sized TVs. For low resolution systems (the shotgun), timing is not as critical as for high resolution systems (the rifle).

Physical Dot Resolution -- Conclusion 1.2.2

If a high resolution digital graphics system uses one, fixed pixel frequency, the picture that looks fine on one TV may have distinct moire interference patterns on another. These moire patterns form as a result of multiple hits on the same target. An example might help here. Below is a chart of some common RCA video tubes.

SIZE	PART NUMBER	ELEMENT TYPE	PITCH	ELEMENTS/LINE
25 in.	25VCZP22	DOT	.66 mm.	769
21	21VBEP22	DOT	.66	646
19	19VEDP22	DOT	.61	633
19	19VEJP22	STRIPE	.826	467
17	17VAYTC02	STRIPE	.826	418
15	15VAETC02	STRIPE	.826	369

Suppose that a high resolution graphics system produces 640 pixels per line (ie, the trigger is pulled on the hypothetical rifle 640 times) on a 19 inch dot matrix TV tube (with 633 targets). It is obvious that there will necessarily be seven double hits per line. Each double hit will be flanked by a number of semi-double hits to either side. In fact, there will be seven groups of double hits flanked by semi-double hits in each line that forms the 480 lines of the screen. Taken en masse across all 480 lines they will tend to form a visually displeasing pattern. This is the moire pattern

referred to. It will appear as more or less vertical bands on the screen. This moire pattern manifests itself as an indestinctness of edge where the moire and an object coincide. It is most noticable when attempting to display text as an apparent subpixel shift in certain character positions. This moire effect will be present even when a monitor is used (as the owners of Apple 80 column text cards have discovered). Using a tunable pixel clock to match the number of shots to the number of targets will eliminate moire for most televisions.

OMNI uses a software controllable pixel clock which will minimize the "shooting gallery" moire. The viewer will simply 'focus' the screen through an interactive software routine. The alternative, as I have previously stated, is to force the consumer to provide a high resolution monitor. OMNI is the only system that I have seen which would allow a variable pixel rate.

Part 3 -- Color Resolution 1.3

Color resolution refers, not to size, but to the degree to which colors differring only slightly in hue and brightness are created by the graphics hardware and detected by the human eye.

Color Resolution -- Selection of the Palette 1.3.1

The phenominon of vision depends upon two types of eye cells: rods and cones. The rods, which are sensitive to variations in brightness, are most discriminating if the brightness variations happen to be colored green. Thus, TV uses green to carry the majority of luminance information (59%). The cones detect variations in hue. It so happens that their hue descrimination is keenest in orange. Therefore, a video system should produce greens, reds and yellows with the highest resolution possible.

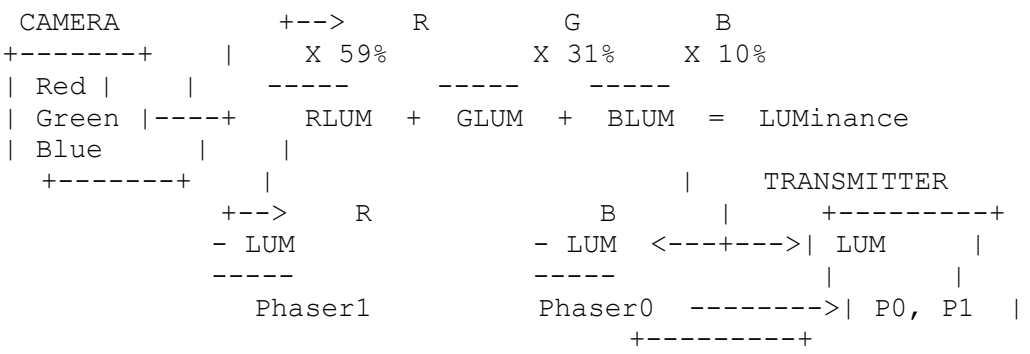
What about blues? As it turns out, the spectrum from cyan through blue to magenta is the lowest resolution region for both luminance and chrominance. So any good video system should not transmit the blues with the highest spectral resolution. In other words, there should be many more hues of reds, yellows and greens available to the graphic artist than cyans, blues and magentas. Unfortunately, Atari's present products give equal weight to all colors. Hence, its palette is dominated by almost indistinguishable hues of blue.

Color Resolution -- Shading & Color Tracking 1.3.2

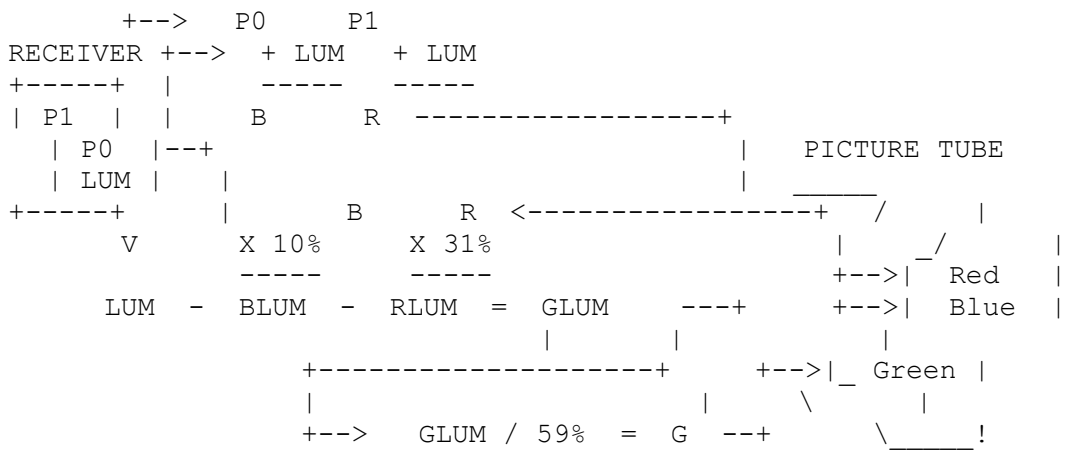
Color shading on a monitor is an easy task. Equal amounts of red, blue and green can be added to the basic color to produce a shade of that color, but adding luminance to a broadcast TV will create a pastel which tends toward blue or has a blueish semi-tone added. To counter this tendency, the chrominance signal must be increased proportionately. Technically speaking, this compensation ensures

a constant percentage of color saturation throughout the range of shades. Atari's present products don't make this compensation, so oranges go to pink, reds go to pale magenta, etcetera.

A television station transmits three signals: LUMinance and two chrominance Phasers, calculated like this:



The TV decodes the primaries from these three quantities by reversing the mathematics like this:



As an example, to create an orange let R,G,B = {1,.5,0}. Then using the transmitter calcuation proceedures above:

LUM = .605 P1 = .395 P0 = -.605

And applying the receiver equations we see:

R,G,B = {1,.5,0} = orange.

Now to create an orange with twice the luminance

The correct way is to double LUM, P1 & P0	The present Atari way is to double LUM but not P1 & P0
LUM' = 2LUM = 1.21	LUM' = 2LUM = 1.21
P1' = 2P1 = .79	P1' = P1 = .395
P0' = 2P0 = -1.21	P0' = P0 = -.605

The TV decodes and displays:

R',G',B' = {2,1,0} = orange R',G',B' = {1.6,1.1,.6} = pink !!

Color Resolution -- OMNI 1.3.3

OMNI will generate and display more colors with fewer parts than any other graphics system. It generates colors mathematically to simulate the output of a 203 stage delay line, but without a delay line or baseband modulator. The color palette contains the mathematical description of one cycle of each of almost 2800 colors for use by color-pixels (324 per line) and OMNI will generate constant saturation throughtout all shades produced while overlaying color-pixels with intensity-pixels (648 per line) as they perform their edge smoothing and texturing jobs. OMNI will mix in video disk or other electronic sources and sychronize to their signals. OMNI is made for the broadcast television medium.

Atari's present products allow 16 colors to be simultaneously displayed. The market demands more. There are proposed systems that would display up to 256 colors. Is that enough? To answer that question, one must calculate how many hues and shades 256 colors really represent. To illustrate, here is a rundown of the colors OMNI can produce contrasted with the colors which can be produced by the traditional approach:

OMNI APPROACH			TRADITIONAL APPROACH		
5 bits of LUMinance			3 bits of red		
4 bits of Phaser0			3 bits of green		
5 bits of Phaser1			2 bits of blue		
HUE	NUMBER OF HUES	NUMBER OF COLORS (%)	HUE	NUMBER OF HUES	NUMBER OF COLORS (%)
gray	1	24 (0.9)	gray	1	4 (1.6)
red	42	689 (24.7)	red	17	47 (18.4)
yellow	38	524 (18.7)	yellow	17	49 (19.1)
green	47	595 (21.3)	green	17	47 (18.4)
cyan	26	390 (13.9)	cyan	14	38 (14.8)

blue	26	323 (11.6)		12	34 (13.3)
magenta	23	249 (8.9)		13	37 (14.4)
	---	----------	--	----------	
	203	2794 (100.0)		91	256 (100.0)

X 16 intensities/color

44,704 color & intensity combinations

13.76 shades per hue — 65% in red-yellow-green — output is composite video

2.81 shades per hue — 56% in red-yellow-green — output is RGB

OMNI's composite video output means that the signal is already of the proper type for application to a base-band monitor or standard broadcast TV while to the traditional approach one must add either a 91 tap delay line with decoder or a quadrature modulator. If an RGB output is mandatory, it can be created from the OMNI composite video by adding a simple resistor matrix and one amplifier. OMNI scales the chrominance & luminance proportionately when intensity is applied to create shading or an edge. As can be seen from the conclusion, this results in accurate color tracking.

CHAPTER 2 -- THROUGHPUT AS IT RELATES TO GAMING 2

Throughput is often a somewhat nebulus term. In gaming, it can be thought of as the upper bound on the playability of a game system; the amount of action and interaction in the most complex game that a system can support. It includes the creation and positioning of graphics, the evaluation of user responses, the interpretation of game rules or constructs, the creation of sounds, etc. If a game can be thought of as an artificial reality with which and within which one or more human players can interact, then throughput is a measure of the complexity of that reality and the richness of experience available to those players.

The meterstick with which the throughput of competing game systems can be measured and compared is the complexity of games which each will support. If one game system can 'play' a certain game that another cannot, the former system can be proclaimed as superior to the later. However, such a benchmark game cannot exist during the development of the competing systems (according to game designers, such a benchmark game will never exist), so the solution to the paradox of how to judge the puddings before they have been baked is obviously not in the tasting, it lies in a comparison of their recipes. Here are two important ingredients:

Maximizing CPU Timespace -- the percentage of CPU time available for game play and
Minimizing Graphics Overhead -- the percentage of CPU time spent tending to graphics tasks.

Part 1 -- Maximizing CPU Timespace 2.1

The most powerful graphics hardware will not appreciably increase throughput if it obtrusively halts the CPU for significant amounts of time. In fact, obtrusive graphics hardware can so drastically

decrease game play that is a liability. Three tasks which relate to graphics which can impact the CPU timespace are:

Loading graphics,
Running graphics and
Refreshing graphics memory.

Maximizing CPU Timespace -- Loading Graphics 2.1.1

A graphic consists of a parameter block: object position, format code, etc., and a data block: graphic format, colors, intensities, etc. A block can either be passed through CPU registers or DMAed (direct memory accessed).

Passing graphic blocks through CPU registers is the most invasive method of loading graphics. The CPU must get the block from whatever mass storage is used and then load it into the GPU (Graphics Processing Unit). Of course, this must be done one word at a time with the CPU controlling the source and destination addresses and associated byte or word counter. This scheme has one advantage. Since the CPU is intimately involved in the loading procedure, it can alter a block on the fly as it is loaded. However, since this capability does not lend itself to structured programming, ie, the creation of programs constructed from functionally differentiated routines, it is of dubious utility. Register passing has an overriding disadvantage, though. It takes approximately twice as long to accomplish the load as DMA does.

There are two types of DMA: transparent and cycle stealing. With transparent DMA, the CPU is unaware of DMA activity. The transfer occurs during cycles in which the CPU is busy performing internal operations such as register-register transfers, calculations and the like. Of course, there must be a way for the CPU to tell the DMA controller when it does not need the memory and the length of time the memory will be free. Many new, powerful microprocessors operate essentially asynchonously to memory so that this criteria cannot be easily met. Also transparent DMA has two inherent drawbacks which require some careful thought to understand.

First, since the memory transfers are transparent, the CPU cannot automatically know when DMA is done. So there is a possibility it may try to alter the loaded data before it is actually there, that is, before the load has finished. The common way to guard against this is to have the CPU poll the DMA controller to ascertain when it has finished. While polling, though, the CPU is not doing any constructive work, thus negating the logic of transparent DMA.

The second drawback is very esoteric but of paramount importance. As viewed from the CPU's perspective, transparent DMA, is an asynchronous operation, therefore, it adds unpredictability that real

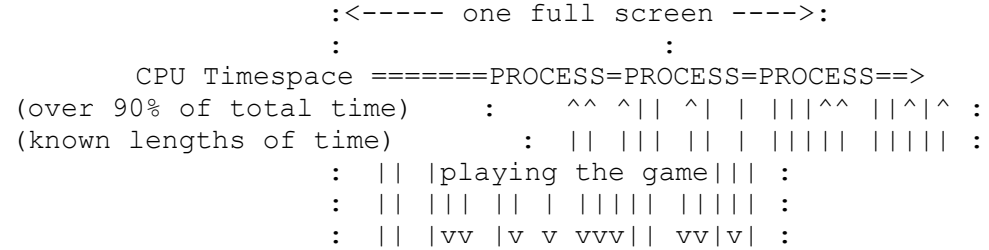
time programming, ie, games, cannot easily cope with. The reliability of a game must be a major consideration for an unreliable game is a bad product. Game designers prefer a DMA that requires the allocation of a known period of enforced CPU inactivity. Register passing ensured this but was too slow. A good compromise is cycle-stealing DMA.

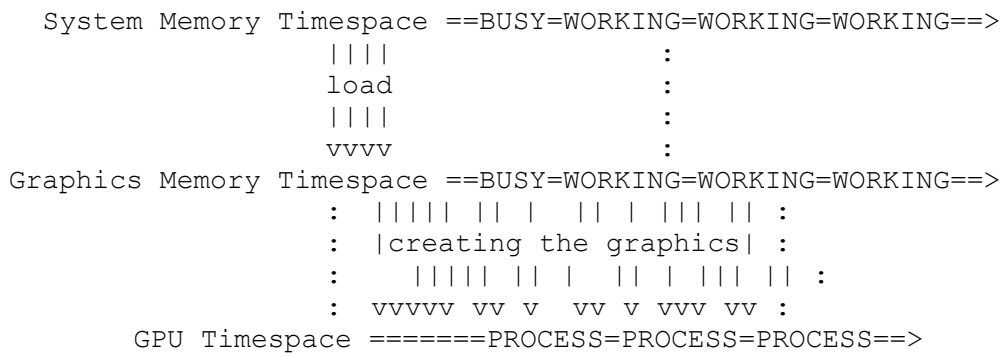
In cycle-stealing DMA, the DMA controller halts the CPU (steals a cycle) for each word that is transferred. While a block is being transferred en masse the CPU is halted for the duration. But the CPU knows how long that is. It is "blocklength" number of cycles. Cycle-stealing DMA has the speed of direct memory access with the predictability of register passing.

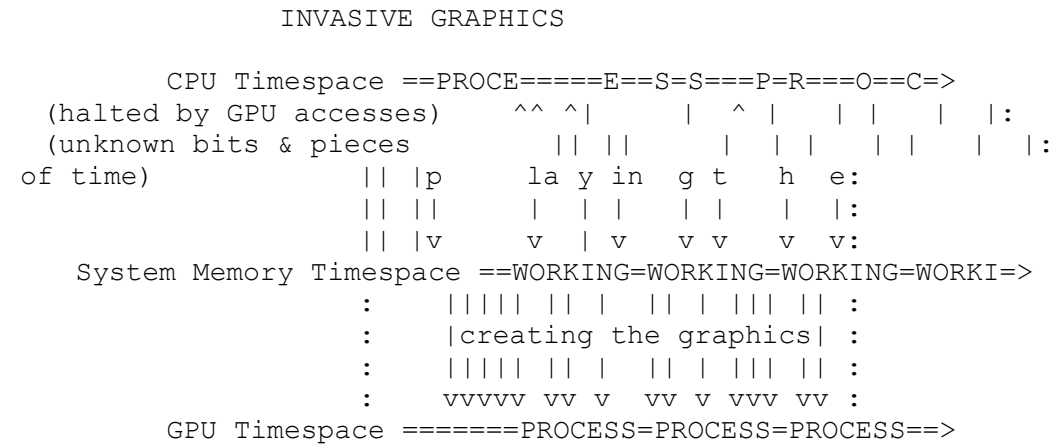
While it's desirable to halt the CPU while loading graphics, it's disruptive to do it every time the GPU access those graphics, as in Atari's present products, and it's disasterous to halt the CPU during the entire active video screen time, as has been proposed. I shall explain. Even though the CPU controls the flow of information to the graphics hardware, it does not know how complex the resulting images will be or how much time it will take the GPU to create them or how many memory accesses the GPU will have to make. In other words, the actual creation of the graphics is an asynchronous operation with which the CPU need not be concerned. It is for this reason that special purpose graphics hardware exists.

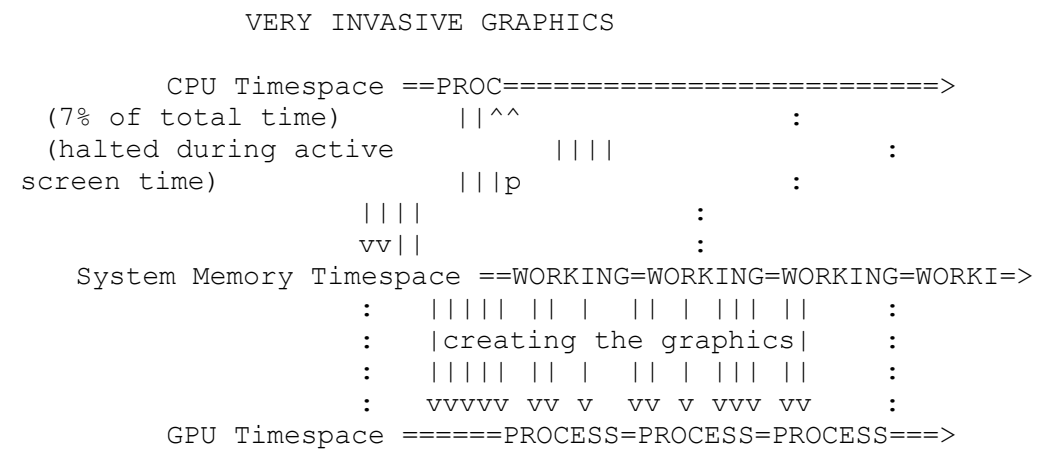
To be truely worth the expense, the generation of graphics should be a parallel process that the CPU can set and forget. This set & forget capability absolutely dictates that the graphics memory be separate from main system memory so that the GPU will not halt the CPU for unknown periods of time. The following time lines graphically illustrate this non-invasive set & forget graphics, an invasive architecture in which memory is shared and a very invasive scheme in which the CPU is denied the use of memory during active screen time.

NON-INVASIVE SET & FORGET GRAPHICS









Maximizing CPU Timespace -- Refreshing Graphics Memory 2.1.3

Dynamic memory is cheaper than static or pseudo-static memories. For that reason, it is preferred. But dynamic memory requires a

refresh cycle every 4 milli-seconds to maintain its data. This refresh accounts for approximately 0.4% of the CPU timespace, a small overhead. Since system memory usually also includes some dynamic memory, for the same reason, then it would make sense to refresh both simultaneously. Failing that, the graphics memory should be transparently refreshed by the graphics hardware or by the memory controller.

Maximizing CPU Timespace -- Conclusion 2.1.4

To maximize CPU timespace, then, one should minimize the numbers of asynchronous events that it has to deal with. To that end, a graphics system should

1, have graphics memory separate from the system memory which is either transparently refreshed or is refreshed simultaneously with system memory refresh and,
2, use cycle-stealing DMA to load graphics.

OMNI has been designed to utilize dynamic RAM for graphics that is separate from main system RAM. This graphics RAM is loaded, upon CPU initiation, by means of cycle-stealing DMA. It is refreshed simultaneously with system RAM refresh. In this way, the CPU is running and working all the time except during those predictable lengths of time when DMA occurs. Since the programmer will usually initiate DMA during vertical blanking, he or she should have the entire screen time (93% of total time) in which to do game play. Furthermore, the programmer has access to the graphics RAM all of the time - even during the active screen time - and can, therefore, read or write graphics whenever desired. Since the graphics hardware must have unrestricted access rights during screen time, CPU requests for reads from or writes to graphics RAM will be on a lower priority catch as catch can basis. If this is unacceptable, the programmer should retain copies of critical parameters in main system RAM for reference and leave graphics RAM undisturbed during the active screen at his or her option.

Part 2 -- Minimizing Graphics Overhead 2.2

Splendid graphics can be produced using a bit mapped screen and no specialized graphics hardware. The process of forming an image of many objects at once on a simple bit mapped screen is slow, software intensive and very tedious. Independent motion objects generated by specialized graphics hardware make the creation of fast moving objects a manageable proposition. Atari's present products utilize motion object generators and there is every reason to continue that development. Assuming that this trend is to continue, it is of further advantage to relieve the CPU of as many graphics tasks as possible relating to the creation and use of independent motion objects. These tasks typically include:

Positioning graphics horizontally (in X),
Positioning graphics vertically (in Y),
Prioritizing graphics visually by depth (in Z),
Scaling & zooming graphics,
Rotating, inverting & reflecting graphics,
Clipping partially off-screen graphics,
Stenciling one graphics object by another,
Detecting collisions,
Creating animation sequences,
Reusing graphics object generators,
Changing the color palette and
Creating special effects.

Minimizing Graphics Overhead -- Positioning Graphics 2.2.1

In a three-dimemsional graphics system, each object displayed must be positioned horizontally, vertically and by screen depth, ie, in X, Y & Z. There are two alternative schemes. In one, the initial position is loaded when the object is created. The object is then programmatically moved about the screen by means of an associated

pointing vector which can describe either its velocity and direction or its acceleration and direction. The hardware automatically updates its position at the beginning of each screen based upon its position during the previous screen and the magnitude and the direction of its pointing vector. In the second, simpler method, the CPU maintains the pointing vector in software and directly updates the object's position at the beginning of each screen.

Although the vector method sounds attractive, it is too automatic. The game program can easily lose track of objects. Though a means of reading an object's position could be provided, this would be a classic case of diminishing returns. It has been established that it is easier to compute and load object positions in software than to track and control self-motivated objects. This also results in more compact and efficient hardware. Of course, it should be possible to change one position coordinate, say X, without having to reload unchanged coordinates, Y & Z in this case.

It should also be possible to group objects together in formation and then position then en masse instead of individually. This can be easily done by subdividing an object's coordinates into the sum of positions and offsets. Thus, COORDINATE(X,Y,Z) = {X,Y,Z} where

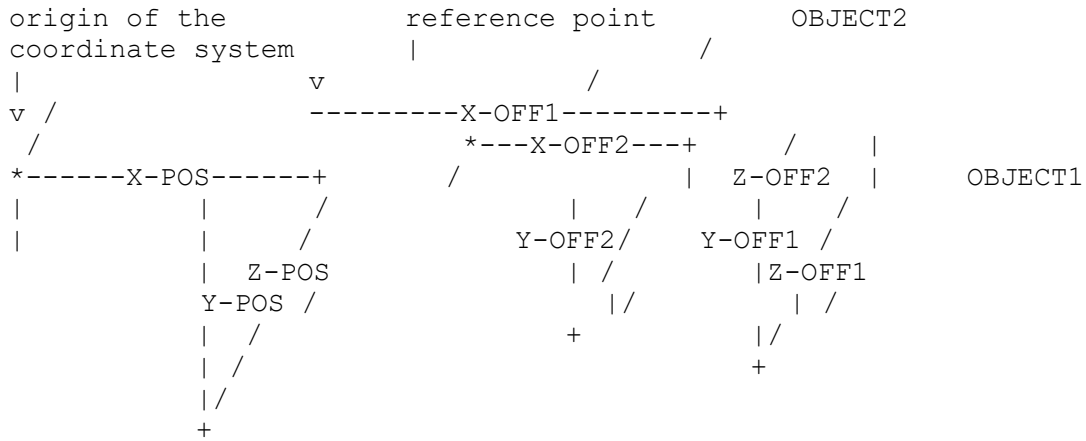
|X| = X-POSition + X-OFFset,
|Y| = Y-POS + Y-OFF and
|Z| = Z-POS + Z-OFF are the magnitudes of X, Y & Z,

is equivalent to COORDINATE'(POS,OFF) = POS + OFF where

POS(X,Y,Z) = {X-POS,Y-POS,Z-POS} and
OFF(X,Y,Z) = {X-OFF,Y-OFF,Z-OFF} are vector quantities.

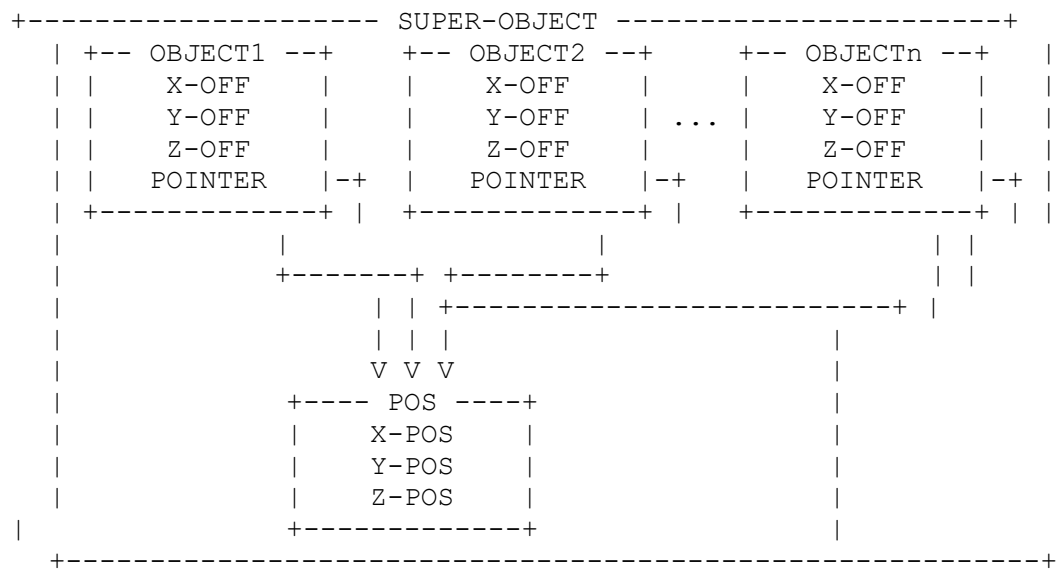
POSition is then the coordinates of the formation's common reference point and OFFset is the coordinates of a particular member of the formation relative to POSition.

Schematically a position relative offset system works like this:



Objects 1 & 2 can be moved relative to each other by manipulating OFFsets and can be moved together, in formation, by changing POSition. This way a super-object consisting of 20 sub-objects can be moved, by POSition, with only one to three stores instead of the 20 to 60 stores that would otherwise be required for that many objects.

Programmatically, 'n' objects point to a common parameter area containing their POSition, thus:



It should be possible to create any number of these super-objects.

Perspective is the intentional distortion of the display space to achieve an enhanced illusion of depth. It is acheived by adding a differential offset to the coordinates which would otherwise produce a non-perspective view. The problems encountered are formidable. Besides the obvious difficulties, there is a subtle aspect. If the TV is to act as a window into the display space, then the player's field of view should be taken into account when figuring perspective. The field of view is determined by two measurements: the distance from the player to the screen and the screen's size. The player could enter these parameters at the start of the game. They can then be used to scale the apparent depth of the display

space to give it a natural aspect. This is important as it should be done to avoid tunnel-vision, that is, the illusion that the TV screen is a 'lens' with a different perspective than reality. To appreciate this effect, play any first-person space game on a six foot projection TV while sitting six feet away, a 55 degree field of view, and then play the same game while sitting three feet away from a 21 inch TV, a 30 degree field of view. Admittedly, this is a minor correction, but ascendancy in future markets may require such attention to detail. Perspective positioning in hardware is probably be too expensive and too inflexable; it is done best by the game designer in software.

Finally, the coordinate system should be right-orthagonal, ie, it should have mutually perpendicular ordinates (the usual X,Y,Z type with which most people are familiar) rather than spherical or cylindrical ordinates (neither of which are really suited to rectangular TVs), which conforms to the right-hand convention for normal cartisian coordinate spaces (in which tightening a Z-axis aligned screw from above produces a sweep from Y to X) so that rotation in X, Y & Z will produce proper, not reversed, motion.

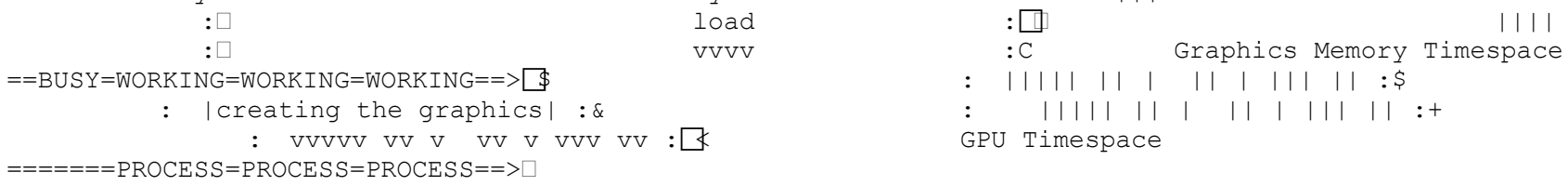
If two or more objects are coincident in X and Y but overlap in Z, then it is logical that the most frontal of them would obscure the rest and be the only one displayed. Deciding which of them is in front is called display prioritization.

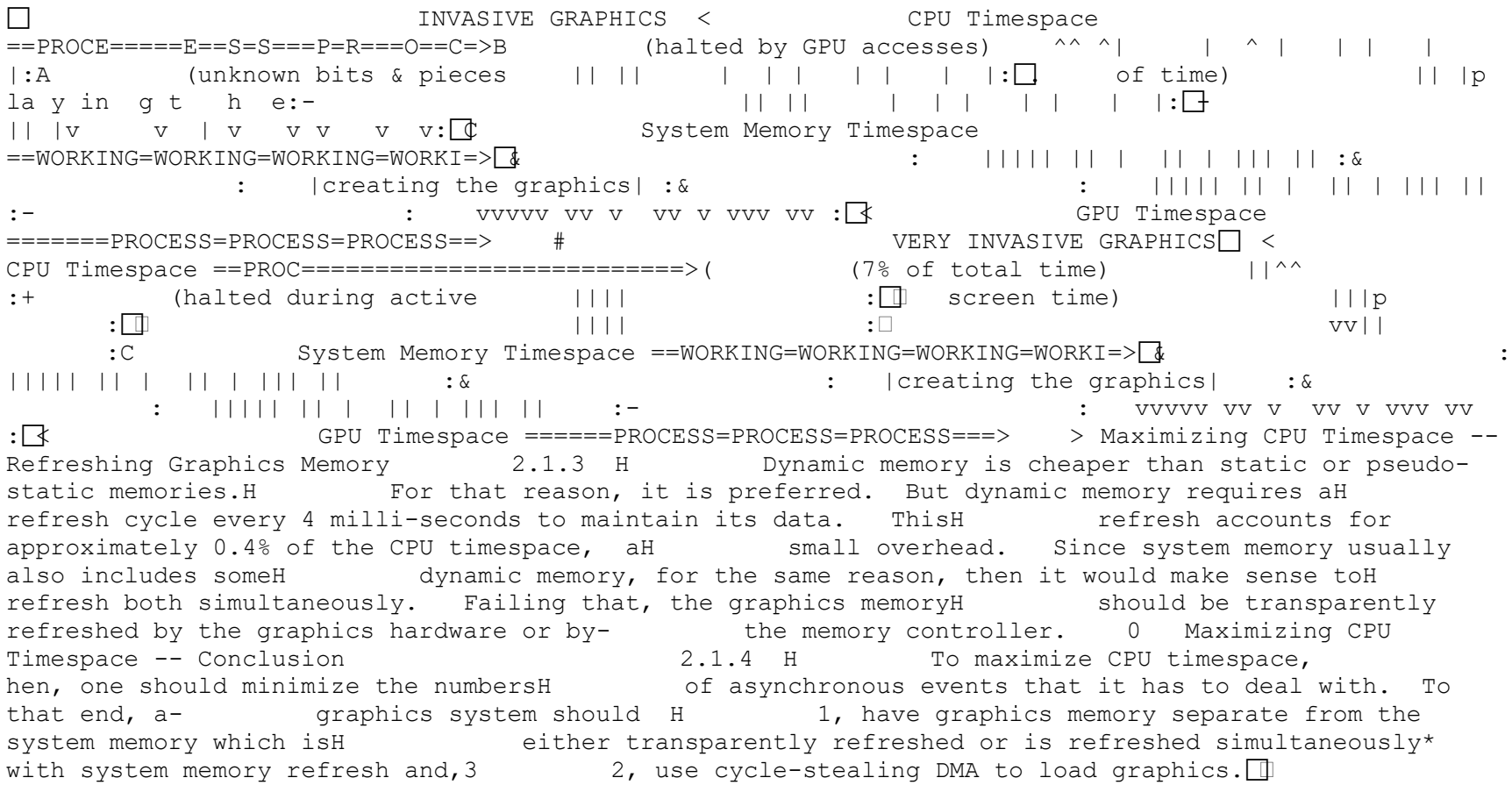
Priority can be determined on an object basis or on an individual pixel basis. If it is done on an object basis, then all pixels in a particular object either have priority or don't have priority en masse. This is undesirable since as soon as one object overlapped another, even by one pixel, the anterior object would suddenly and completely disappear. To overcome this drawback, the obsured part of the anterior object could be lopped off in the data. But this would have no advantage, whatsoever, over a simple bit map. So to preserve the obvious advantage of independent motion objects, they must be capable of prioritizing themselves pixel by pixel.

Pixel display prioritization can be accomplished in several ways. The best way is to have each object generator use its Z-coordinate as a weapon in a fight for display survival. The winner gains the right to display its pixel. On the very next pixel, the battle is repeated with more or fewer object generators joining in, depending upon which of them have a pixel to output on that clock. And so on, pixel after pixel, for every pixel in the visible screen. This Z-combative method has one great advantage: the battle rages in parallel. It matters not whether there are two object generators in the frey or two hundred. The length of time it takes for a winner to emerge is essentially constant. Also, the Z-combative method is self-maintaining, that is, when an object is assigned a new value of Z which moves it closer to the screen, it has simultaneously been given a bigger weapon with which to do battle.

A less desirable alternative assigns each object generator a fixed priority based upon its position in a serial hardware chain. This

idea is passive, not combative. If the generator with the highest priority has no pixel to output, then it passes the display rights to the generator which has the next highest priority, and so on. As with all such serial chains, this one suffers from propagation delays that are additive. These delays limit the number of





Refreshing Graphics Memory 2.1.3 Dynamic memory is cheaper than static or pseudo-static memories. For that reason, it is preferred. But dynamic memory requires a refresh cycle every 4 milli-seconds to maintain its data. This refresh accounts for approximately 0.4% of the CPU timespace, a small overhead. Since system memory usually also includes some dynamic memory, for the same reason, then it would make sense to refresh both simultaneously. Failing that, the graphics memory should be transparently refreshed by the graphics hardware or by- the memory controller. 0 Maximizing CPU Timespace -- Conclusion 2.1.4 To maximize CPU timespace, hen, one should minimize the numbers of asynchronous events that it has to deal with. To that end, a- graphics system should 1, have graphics memory separate from the system memory which is either transparently refreshed or is refreshed simultaneously* with system memory refresh and,3 2, use cycle-stealing DMA to load graphics.

□ * Maximizing CPU Timespace -- OMNI 2.1.5 J OMNI has been designed to utilize dynamic RAM for graphics that isJ separate from main system RAM. This graphics RAM is loaded, uponJ CPU initiation, by means of cycle-stealing DMA. It is refreshedJ simultaneously with system RAM refresh. In this way, the CPU isJ running and working all the time except during those predictableJ lengths of time when DMA occurs. Since the programmer will usuallyJ initiate DMA during vertical blanking, he or she should have theJ entire screen time (93% of total time) in which to do game play.J Furthermore, the programmer has access to the graphics RAM all ofJ the time - even during the active screen time - and can, there-J fore, read or write graphics whenever desired. Since the graphicsJ hardware must have unrestricted access rights during screen time,J CPU requests for reads from or writes to graphics RAM will be on aJ lower priority catch as catch can basis. If this is unacceptable,J the programmer should retain copies of critical parameters in mainJ system RAM for reference and leave graphics RAM undisturbed during/ the active screen at his or her option.□

Splendid graphics can be produced using a bit mapped screen and no specialized graphics hardware. The process of forming an image of many objects at once on a simple bit mapped screen is slow, software intensive and very tedious. Independent motion objects generated by specialized graphics hardware make the creation of fast moving objects a manageable proposition. Atari's present products utilize motion object generators and there is every reason to continue that development. Assuming that this trend is to continue, it is of further advantage to relieve the CPU of as many graphics tasks as possible relating to the creation and use of independent motion objects. These tasks typically include:

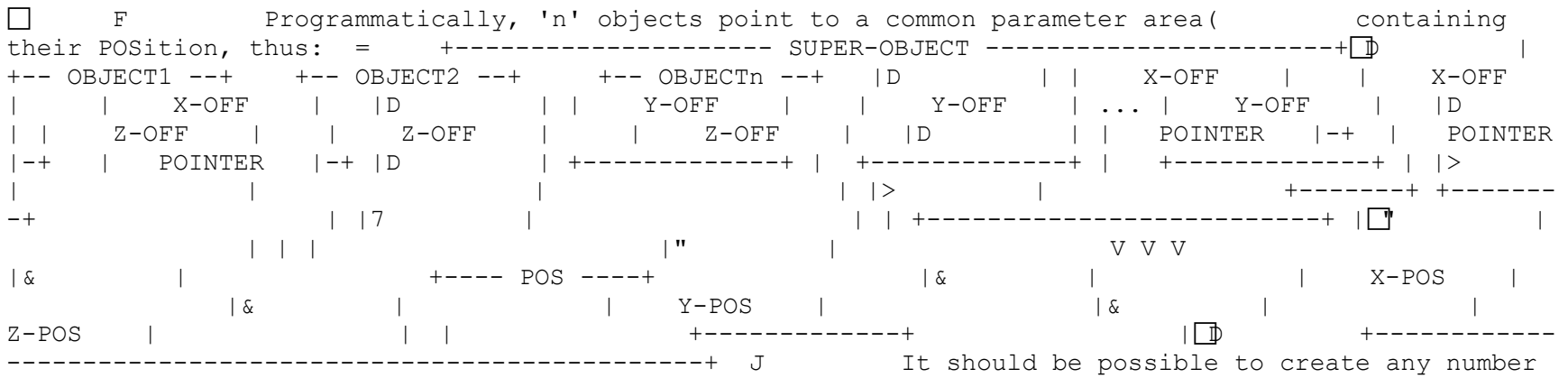
- Positioning graphics horizontally (in X),
- Positioning graphics vertically (in Y),
- Prioritizing graphics visually by depth (in Z),
- Scaling & zooming graphics,
- Rotating, inverting & reflecting graphics,
- Clipping partially off-screen graphics,
- Stenciling one graphics object by another,
- Detecting collisions,
- Creating animation sequences,
- Reusing graphics object generators,
- Changing the color palette and
- Creating special effects.

Minimizing Graphics Overhead -- Positioning Graphics 2.2.1

In a three-dimemsional graphics system, each object displayed must be positioned horizontally, vertically and by screen depth, ie, in X, Y & Z. There are two alternative schemes. In one, the initial position is loaded when the object is created. The object is then programmatically moved about the screen by means of an associated pointing vector which can describe either its velocity and direction or its acceleration and direction. The hardware automatically updates its position at the beginning of each screen based upon its position during the previous screen and the magnitude and the direction of its pointing vector. In the second, simpler method, the CPU maintains the pointing vector in software and directly updates the object's position at the beginning of each screen. Although the vector method sounds attractive, it is too automatic. The game program can easily lose track of objects. Though a means of reading an object's position could be provided, this would be a classic case of diminishing returns. It has been established that it is easier to compute and load object positions in software than to track and control self-motivated objects. This also results in more compact and efficient hardware. Of course, it should be possible to change one position coordinate, say X, without having to reload unchanged coordinates, Y & Z in this case.

□ J It should also be possible to group objects together in formationJ and then position then en masse instead of individually. This canJ be easily done by subdividing an object's coordinates into the sumJ of positions and offsets. Thus, COORDINATE(X,Y,Z) = {X,Y,Z} where % |X| = X-POSition + X-OFFset,□ |Y| = Y-POS + Y-OFF and< |Z| = Z-POS + Z-OFF are the magnitudes of X, Y & Z, ? is equivalent to COORDINATE'(POS,OFF) = POS + OFF where□ - POS(X,Y,Z) = {X-POS,Y-POS,Z-POS} and□@ OFF(X,Y,Z) = {X-OFF,Y-OFF,Z-OFF} are vector quantities. J POSition is then the coordinates of the formation's common refer-J ence point and OFFset is the coordinates of a particular member of+ the formation relative to POSition.□ H Schematically a position relative offset system works like this: 9 origin of the reference point OBJECT2□* coordinate system | /□ | v /□* v / ---------X-OFF1---------+< / *---X-OFF2---+ / |H *------X-POS------+ / | Z-OFF2 | OBJECT14 | | / | / | /3 | | / Y-OFF2/ Y-OFF1 /□4 | Z-POS | / |Z-OFF17 Y-POS / |/ | /□/ | / + |/□. | / +

|/ +□ J Objects 1 & 2 can be moved relative to each other by manipulatingJ OFFsets and can be moved together, in formation, by changing POS-J ition. This way a super-object consisting of 20 sub-objects canJ be moved, by POSition, with only one to three stores instead ofJ the 20 to 60 stores that would otherwise be required for that many□ objects.□



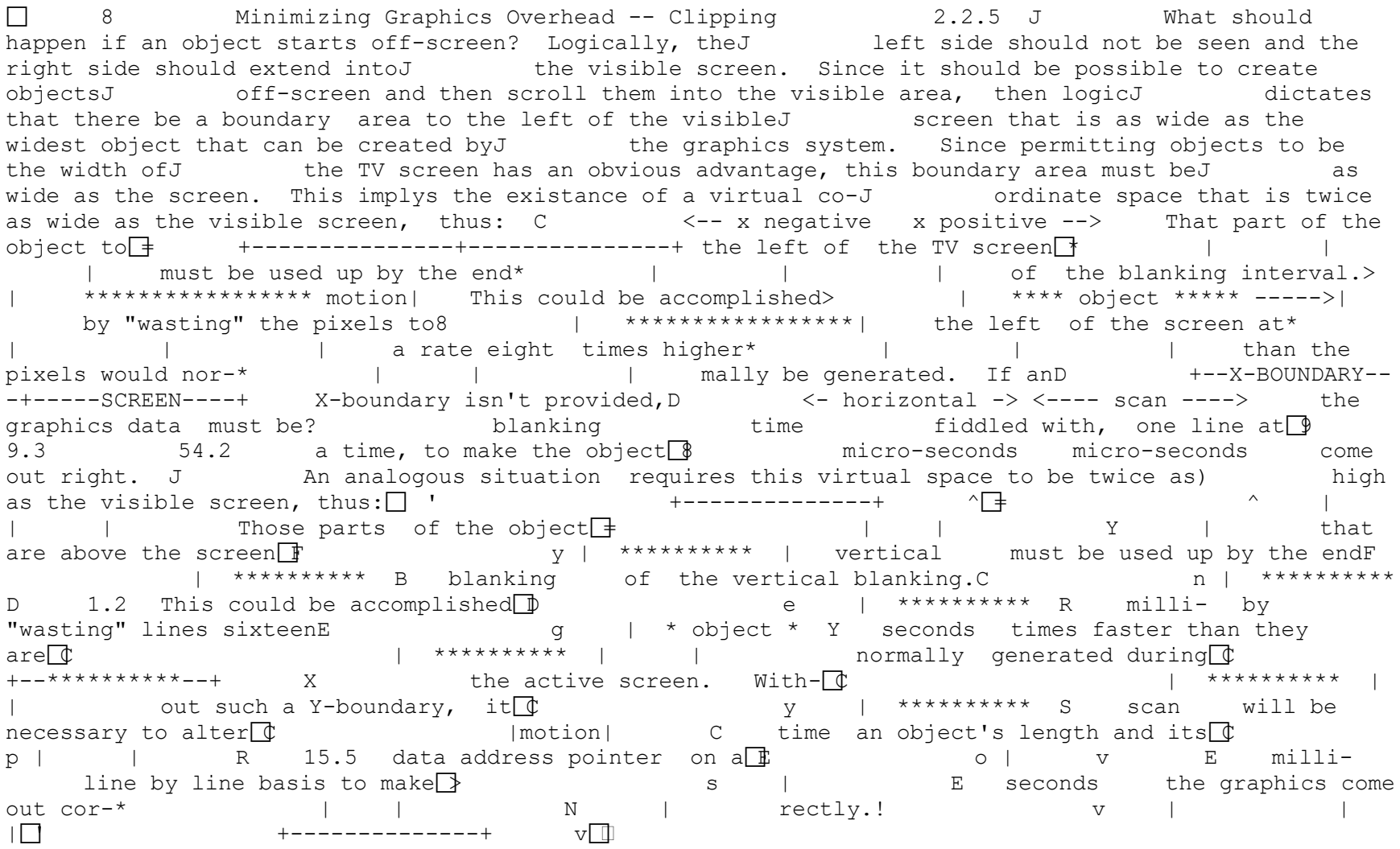
□ E Minimizing Graphics Overhead -- Display Prioritization 2.2.2□ J If two or more objects are coincident in X and Y but overlap in Z,J then it is logical that the most frontal of them would obscure theJ rest and be the only one displayed. Deciding which of them is in/ front is called display prioritization.□ J Priority can be determined on an object basis or on an individualJ pixel basis. If it is done on an object basis, then all pixels inJ a particular object either have priority or don't have priority enJ masse. This is undesirable since as soon as one object overlappedJ another, even by one pixel, the anterior object would suddenly andJ completely disappear. To overcome this drawback, the obsured partJ of the anterior object could be lopped off in the data. But thisJ would have no advantage, whatsoever, over a simple bit map. So toJ preserve the obvious advantage of independent motion objects, theyB must be capable of prioritizing themselves pixel by pixel. J Pixel display prioritization can be accomplished in several ways.J The best way is to have each object generator use its Z-coordinateJ as a weapon in a fight for display survival. The winner gains theJ right to display its pixel. On the very next pixel, the battle isJ repeated with more or fewer object generators joining in, depend-J ing upon which of them have a pixel to output on that clock. AndJ so on, pixel after pixel, for every pixel in the visible screen.J This Z-combative method has one great advantage: the battle ragesJ in parallel. It matters not whether there are two object genera-J tors in the frey or two hundred. The length of time it takes forJ a winner to emerge is essentially constant. Also, the Z-combativeJ method is self-maintaining, that is, when an object is assigned aJ new value of Z which moves it closer to the screen, it has simul-E taneously been given a bigger weapon with which to do battle.□ J A less desirable alternative assigns each object generator a fixedJ priority based upon its position in a serial hardware chain. ThisJ idea is passive, not combative. If the generator with the highestJ priority has no pixel to output, then it passes the display rightsJ to the generator which has the next highest priority, and so on.J As with all such serial chains, this one suffers from propagationJ delays that are additive. These delays limit the number of objectJ generators which can participate. Also, whenever an object passesJ another in Z, so that it becomes closer to the screen, the entireJ parameter package of the passing object and the passed object mustJ be interchanged. This is a serious software overhead. This hand-J icap can be relieved somewhat by link-listing parameters, that is,J by providing an indirect register within each object which pointsJ to its parameters. Then, instead of interchanging the parameters,J the pointers are interchanged. This overhead was not needed withJ Z-combative display prioritization. It is plain, then, that fixedJ chain prioritization is inferior to Z-combative prioritization inJ speed, capacity and CPU overhead. I will return to the subject ofJ fixed-chain -vs- Z-combative prioritization in the section dealing, with the reuse of object generators.□

Another alternative method to assign pixel priority is through the color palette. The idea here is simple: one color is assigned the highest priority, another color is assigned next highest priority, and so on. This color palette method is fast and straight forward but it obviously limits the way objects can use color. It has two interesting advantages, though: colors can be mixed directly with- in the palette and special effects can be created using the color, or the lack of color, of overlapping objects. I have more to say about this in a later section.

Minimizing Graphics Overhead -- Scaling & Zooming

2.2.3 An object's Z-coordinate can be used for automatic scaling, thereby enhancing the three-dimensional effect. This scaling function is truely useful only if fractional values are allowed. To illustrate, imagine that an object is at scale factor one. Each pixel of data that makes up that object is reproduced on the screen full sized, ie, with one-to-one pixel correspondance. If only integer scale factors are allowed, then as the object moves toward the screen, it would reach a point where scale factor would go to two. At that point, the object would double in size. Each pixel of data would produce four pixels on the screen causing the object to literally leap out at the viewer. At a scale factor of three, it would appear to jump out once again as its size increased by half again. Not as big a jump as before, but still quite abrupt. At scale factor four, the object would jump 25% in size; at scale factor five, 20%; and so on. This jumpping effect would not be tolerable until the scale factor reached twenty (producing a 5% increase in size). But by then each pixel of data would be replicated 400 times (20 times horizontally & 20 times vertically); very low resolution, indeed. Fractional scale factor would relieve the problem; the object could go from scale factor 12 to scale factors 1.1, 1.2, 1.3, and so on. But even with fractional scale factors there would still be severe problems. First, imagine that the scale factor goes from 1 to 1.1 requiring an eight pixel object to generate a ninth pixel. Which pixel should be repeated? That depends upon whether the object is mostly color or mostly details and on where the color/detail transitions occur in the string of eight pixels. Second, the decision as to which pixel to repeat is dependent upon its size. An object which is small because it is far away carries most of its recognition by its color. When the very same object is closer and larger its detail will probably be its most important property. If the wrong pixel is repeated, the object may very well take on a completely different appearance. Only the graphic designer can make that decision for it is highly object dependent.

□ J And third, since the object should double in size as the distanceJ between it and the viewer is halved, the scale factor must be non-J linear. An approximation to this non-linearity is to either shiftJ the scale factor (equivalent to division by two) or look it up inJ a Z-coordinate to scale factor software conversion table. You mayJ have noted that this scale factor adjustment is, in part, determ-J ined by the distance between the object "and the viewer." It is aJ situation very similar to that outlined in perspective positioningJ where the player's field of view was found to affect the apparentJ depth of the display space. The differential offset applicable toJ the display space for perspective postitioning and the scale fact-J or mentioned here are, essentially, the same things, so that theJ player's field of view should be taken into account when figuring zoom scale factor, also. J So scaling is a function of object size, the distribution of colorJ and intensity across the object, its Z-coordinate, the player toJ TV distance and the size of the screen. It is doubtful whether aJ single factor can, or should, be devised which takes all this intoI account in hardware. In short, zooming is best done in software.□ N Minimizing Graphics Overhead -- Rotating, Inverting & Reflecting 2.2.4 J Plane polygons can be represented parametrically or topologically.J With such descriptions as a starting point, a generalized rotationJ algorithm can be a straight forward proposition, but there is noJ single algorithm which will successfully rotate the complex, non-J polygonal objects that will be in general use. Rotation is a fun-J ction of object size and the distribution of color and intensityJ across the object. The problem is similar to that encountered inJ zooming, outlined in the previous section. Since there exists noJ single algorithm can successfully rotate all objects, it should be□ handled in software. J Reflection and inversion of graphics is another matter. GraphicsJ hardware should be capable of flipping an object end-over-end orJ side-to-side. This will greatly simplify the rotational task. X-J Y co-planer rotations could be generated for rotation angles fromJ zero to 89 degrees: the first quadrant. The remaining three quad-J rants of rotation could then be accomplished with combinations of6 hardware implimented reflection and inversion.□



□ J Likewise, the virtual space should be twice as deep as the maximumJ depth displayed to allow objects created at infinity to move for-J ward into visible space and to prevent objects moving in and out: from piling up at the front or back of the screen. J So the virtual display space should be eight times larger than theJ region which is displayed. It should be noted that objects whichJ spill off to the right or the bottom do not require boundaries.J They will naturally be terminated by the end of the scan line orJ the end of the screen; however, they should not be allowed to wrapJ around to the next line or the next screen. This should not pre-J clude the possibility of intentional wrap around in the data; this. situation which will be covered later. : Minimizing Graphics Overhead -- Stenciling 2.2.6 J Any high performance graphics system worth its salt should be cap-J able of stenciling one object by another. Stenciling is analogousJ to masking, but more powerful and efficient. It is best explainedJ by example. Suppose that in the course of a first person advent-J ure game, the player finds himself or herself in a dark cave. TheJ player must find a pot of gold and simultaneously avoid the attackJ of a vampire bat. The player can see only by flashlight. The potJ of gold and the bat are motion objects while the sides of the cave- are described by a scrolling bit map.□ J The circle of light thrown out by the flashlight is a third motionJ object which stencils the cave bit map, the pot of gold & the bat.J As the circle is moved about, only that portion of cave wall with-J in its circumference is visible. Likewise, the circle defines theJ visibility of each pixel of bat and pot. All of the objects areJ "there" all the time; the game programmer need not manipulate themJ in any other than moving the bat around. But the circle of lightJ stencils them thereby eliminating them from the output, pixel byJ pixel, for each pixel that is not coincident with it. TechnicallyJ speaking, stenciling produces the topological intersection of theJ stenciling object and the stenciled objects. In this way a planetJ of blue, to represent water, could stencil a bit map representingJ its continents and cities. As the bit map is scrolled, the planetJ would appear to revolve ... only the portion of bit map coincidentC with the planet's outline would show and be overlaid on it.□ B Minimizing Graphics Overhead -- Collision Detection 2.2.7 J Hardware collision detection is a controversial subject. On oneJ side of the controversy are hardware engineers who, generally, areJ strong advocates; it's neet. On the other side are game designersJ who have had experience with hardware collision detection; it isF utterly counterproductive and has no socially redeeming value.□

□ J All hardware collision detect schemes with which I am familiar areJ basically embodiments of the old 'video coincidence' idea. ThatJ is, at the time of video output, all the object generators 'look'J to see if the pixel they are currently sending out is coincidentJ in time with the pixel being generated by any other object gener-J ator. If it is, they raise their collision detection flags. ThisJ 'video coincidence' scheme can be nicely integrated into displayJ prioritization, hence, it is a very attractive idea and keeps get-J ting reinvented and incorporated into new video system designs byJ enthusiastic and well-intentioned engineers. From a purely hard-J dware viewpoint, automatic collision detect makes sense. But from= a gaming point of view it is useless for two reasons.□ J First, for all practical purposes, hardware collision detect mustJ be two screens behind the actual game action. This produces gamesJ with distinctly slow reaction times. The first screen lag is, ofJ course, the screen in which the actual collision occurred (call itJ the collision screen). The second lag results from the fact thatJ during the collision screen, the game formats the following screenJ (call it the post-collision screen). At the end of the collisionJ screen the post-collision screen is sent to the graphic hardware.J Only then can the game check for and discover the collision. ThenJ it can use the fact of the collision in formatting the post-post-J collision screen, but by then it is too late. What is needed is aJ collision look-ahead that can be used to appropriately format theJ collision screen itself. This look-ahead is successfully done inJ software. Before a screen is even created, collisions are checkedH and the outcome is then passed to the screen formatting routine. J This software collision detection also successfully overcomes theJ second drawback of hardware collision detection -- unreliability.C For "video coincidence" to work, the colliding objects must be co-□J incident during the collision screen. This will not, usually, beJ the case. Fast moving objects can often pass through one anotherJ between screens -- apparently just interchanging their positions.J To ensure that a collision will not be missed, each object gener-8 ator must have the following a priori knowledge: D 1, its trajectory in the form of its velocity and direction,? 2, the trajectories of all other objects on the screen,□J 3, its cross-sectional profile taken perpendicularly to the lineJ segment drawn from its geometric center to the geometric center, of each of the other objects and,J 4, the cross-sectional profiles of all other objects taken perpen-1 dicularly to those same line segments.□

□ J For 20 objects, each object generator would have to "know" 20 setsJ of three velocity vectors each and would require a means of rep-J resenting and storing 20 cross-sectional profiles. These must beJ updated for each screen. Each would hen compare the 'other' pro-J files against its 'self' profile in a mannar analogous to videoJ coincidence. Further, each object generator must be able to trackJ these cross-sectional profiles between screens. Each generatorJ would then be much larger than 20 non-collision-detecting object1 generators. This is clearly impractable.□ J Various schemes for getting around this problem have been proposedJ involving 'padding' each object with a 'collision-space' which isJ larger than the objects themselves and then detecting collisionsJ between 'collision-spaces'. Without detailling the required dis-J tortions to the 'collision-spaces' to account for relative motionsJ (everything from 'collision-planes' to egg shapes have been tried)J let it suffice that all approaches tried have resulted in a worseJ situation than missed collisions. They have produced erroneous< collisions during what should have been near misses. J The verdict on hardware collision detection, as passed down by allJ experienced game designers of my aquaintance, is, "I'd rather doJ it myself." With Atari's present products, hardware collision de-J tection is either not used (the choise in fast moving games) or itJ is used merely as a flag to indicate that soft collision detectionJ 'may' be needed and should be checked. "If it may ever need to beJ checked", game designers say, "for consistent and reliable program= timing, it should always be checked ... in software."□ G Minimizing Graphics Overhead -- Reusing Object Generators 2.2.8□ ? Minimizing Graphics Overhead -- Special Effects 2.2.9□ F Minimizing Graphics Overhead -- General Considerations 2.2.10 J There are various ways of formatting graphic parameters; the meansJ used can have a significant impact upon the overhead the hardwareJ introduces into the CPU's life. Formats with consistent, simple,J straight forward structures and which have liberal rules regardingJ their usage will yield programs that are easier to write, that areJ more nearly bug free and that run faster. Graphic parameters can! be classified as follows:□ ' 1, Object position and offset,□" 2, Object attributes and,(3, Object processing functions. J Positions and offsets are used to place the object(s) referencedJ in the display space. To maximize the flexability in which theyJ can be used and to minimize the CPU time spent manipulating them,J positions and offsets should be usable either individually or inJ combination. Taken individually, each should be fully capable ofJ spanning the entire display space. To put it the other way, takenJ together, position plus offset should be capable of spanning twiceJ the display space. For example, if the X-dimension of the displayJ space is 11 bits in extent, that is, the object can be placed atJ any one of 2048 X-coordinates, then both X-position and X-offsetJ should be 11 bits long. You may ask why this apparent redundancyJ is important, or even desired. The reason is that if both are 11J bits long, the game designer has several

options in their usage.J The position can be zeroed out and the offset used alone, the off-J set can be zeroed out and the position used alone or the positionJ and offset can both be non-zero so that they are used in combina-J tion. If both are the full 11 bits in extent, then the three op-J tions are equally capable of placing the object at any X-coordin-! ate point in the display.□ J Suppose that the 11 bit X-position is a signed quantity, that is,J the 2048 values of X-coordinate range from -1024 to +1023. TheJ most significant bit is its sign bit. If this 11 bit value is toJ be stored in a 16 bit memory word, then the sign bit should be the* most significant bit of that word: (

like this not like thisI

J The left-hand method of storing X-position gives a testable signJ bit that will immediately indicate whether the object is currentlyJ residing in the positive or negative halves of object space. This% will streamline the software.□ J Finally, all positions and offsets should be binary numbers. TheJ fact that engineers love to use polycode numbers because this res-J ults in smaller hardware should not mandate saddling the softwareJ with bulky, time consuming binary-to-polycode & polycode-to-binaryJ conversion routines. It is better to impliment these routines inJ hardware and leave the software unencumbered. After all, hardware) is but once; software is forever.□ C Object attributes include such things as height, width, map□ dimensions H Object parameters are usually directly addressed. The following+ is an example of direct addressing:□ , OBJECT_NUMBER: OBJECT_PARAMETER J In this example, graphics memory location 'OBJECT_NUMBER' containsA the 'OBJECT_PARAMETER' related to that object. But, this□D parameter is known only to object number 'OBJECT_NUMBER' andH cannot be shared with another object. Address pointers are usedJ to indicate the locations of object parameters which are shared byG several objects at once. An example of this is the X-POSition,□J sited previously, which defined the X-coordinate of a formation ofE objects and, therefore, was shared by all the objects in that□ formation.* Pointers which can span the entireD range of the graphic memory are superior to offset, paged orI segmented memory usage. For example, suppose that graphic memory□D is 16K words in extent. This requires the use of 14 bits ofE address. From a purely software point of view, it would seem□H logical to have an associated address parameter which is 14 bits□ in length like this: + OBJECT_NUMBER: ADDRESS_POINTER□

:- ADDRESS_POINTER: OBJECT_PARAMETER□ J In this example, graphics memory location 'OBJECT_NUMBER' contains@ a 14 bit quantity, 'ADDRESS_POINTER', which the hardwareI interprets as a second address, also in graphics memory. At this□A second address (called the indirect address) is found 'OB□ 8 But harware engineers feel compelled to minimize@ memory usage by compacting such address pointers withoutJ considering the

impact such compaction has upon the software. ForA example, the 14 bit address could be compacted like this:□ "OBJECT_NUMBER: OFFSET
:2 OBJECT_NUMBER+OFFSET: OBJECT_PARAMETER ' OBJECT_NUMBER: PAGE_NUMBER□
:4 PAGE_NUMBER+OBJECT_NUMBER: OBJECT_PARAMETER - OBJECT_NUMBER: ADDRESS_POINTER/4□
:- ADDRESS_POINTER: OBJECT_PARAMETER□ I Adopting the conventions outlined above will go a long way toward□D minimizing the numbers of routines that a game designer mustG write, debug, and integrate, thereby reducing the complexity of□ the software and the length of time writing it, improving its□ reliability, minimizing CPU overhead and the amount of memory□ required and, throughout the life of product, saving money. TheJ hardware costs are minisule compared with the savings that will beI made in each and every piece of software to run on that hardware.□ □

Minimizing Graphics Overhead -- Conclusion

2.2.11 The execution of the graphics functions outlined in the preceeding sections combined the application of hardware and software. Some functions are most efficiently performed by the hardware while others can be done best by the software, thereby giving the programmer the control he or she needs. Here is a rundown of them:

HARDWARE FUNCTIONS

SOFTWARE FUNCTIONS;

Horizontal Positioning

Perspective

Positioning2

Vertical Positioning

Zooming4

Display Prioritization

Rotation3

Reflection

Collision Detection

Inversion"

Clipping off-screen graphics

Stenciling

SHARED FUNCTIONS

Reuse of Graphics

Special Effects

Part 1 of this chapter established that the use of graphics RAM that is separate from the main system RAM will allow the CPU to operate at full speed, even during the active screen time. Now we know what the CPU will do with that time, asside from playing the game.

Some of the important features of the graphics system are now in

focus: J 1, A self-maintaining three-dimensional display space based upon a. right orthogonal coordinate system,J 2, A virtual positioning space eight times larger than the displayC space in which the display space is the positive octant, 3, Automatic generation of independent motion objects within the virtual space, 4, Automatic grouping and maintenance of super-objects made up of numbers of the motion objects in formation,J 5, Automatic display prioritization of objects based upon their Z-5 coordinates on a pixel by pixel basis and, 6, Built in aids to graphic object management for the reuse of object generators.