V. Akman, P.J.W. ten Hagen, A.A.M. Kuijk

A vector-like architecture for raster graphics

Computer Science/Department of Interactive Systems Report CS-R8802 January

69K31, 69K33, 69K35

A Vector-like Architecture for Raster Graphics[†]

Varol Akman, Paul ten Hagen, and Fons Kuijk

Department of Interactive Systems

Centre for Mathematics and Computer Science (CWI)

Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

Abstract: Raster graphics, while good at achieving realistic and cost-effective image generation, lacks useful (e.g. high-level) and fast (e.g. almost real-time) interaction facilities. One may try to speed up the entire classical image generation pipeline using much processing power but this would clearly lessen the advantages of raster workstations as popular, relatively inexpensive devices. This paper continues our work in restructuring the functional model (first formulated by Ingrid Carlbom) for high-performance architectures. Central to our approach is a visible concern about the underlying data structures used to represent the geometric objects. This originates from the conviction that only through careful design of appropriate graphics data structures and algorithms one can profitably map software tasks into hardware, specifically VLSI. Here we elaborate on a novel object description scheme called "pattern representation" and its envisioned usage. Our work is decidedly in contrast with several current research efforts in the area of graphics hardware where it is commonplace to simply put several processors into a cooperative effort to share the total burden, with each processor taking responsibility for part of the work.

Categories and Subject Descriptors: I.3.1 **[Computer Graphics]:** Hardware Architecture — *raster display devices;* I.3.3 **[Computer Graphics]:** Picture/Image Generation — *display algorithms;* I.3.5 **[Computer Graphics]:** Computational Geometry and Object Modeling — *curve, surface, solid, and object representations*
General Terms: Algorithms, Design, Performance
Additional Key Words and Phrases: Pattern representation, Domain function, Color function, Wheel of reincarnation, Dynamic feedback, Functional model

1. Introduction

"... By and large, the designs of commercially available products have been motivated primarily by a bottom-up concern with cost-effective hardware technology that meets performance constraints and not by a more general top-down, "software-first" strategy based on user requirements. A proper top-down strategy would not only include cost-effectiveness and performance, but would also take into account programmability and extensibility. The preoccupation with hardware is understandable in the light of the concern with performance, as measured only

[†] This paper was presented at the *Second Eurographics Workshop on Graphics Hardware,* Centre for Mathematics and Computer Science, Amsterdam (Aug. 24-25, 1987). It will appear in **Advances in Computer Graphics Hardware II,** W. Strasser and F. Kuijk (Eds.), Springer-Verlag, Heidelberg (1988).

Report CS-R8802
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

by the number of elements processed per refresh cycle for a flicker-free display." [1]

These words of Foley and van Dam, now about five years old, seem to us an excellent evaluation of the situation computer graphics has been facing since its beginnings. Commercial systems, being products of various hardware design compromises between speed, price, and flexibility, are still mostly designed with short-term market incentives in mind. The outcome of this trend has been the invasion of the marketplace by a large, nonhomogeneous family of devices incorporating (occasionally superficial and even parenthetical) hardware innovations and a state of affairs where software issues are deliberately ignored and delegated to the users. As an example of the latter, we can observe that only recently manufacturers started paying attention to crucial aspects of graphics software such as standardization, e.g. ISO's GKS [2].

An essential requirement for any graphics system is easy and versatile interaction. In fact, nowadays it is *de rigueur* to imply ''interactive computer graphics'' when one uses the words ''computer graphics.'' Yet, only vector graphics has succeeded in making interaction a reality rather than a dream. Raster graphics, as it stands today, is not able to fulfill the interaction requirement fully. The underlying technical reason for this is the discrepancy between objects that are defined by position and shape (e.g. lines and polygons) and their pixel representations. To quote van Harmelen [3] *"... The use of object definitions is at odds with a raster representation, where the objects are defined by a set of pixels with appropriate color values."* In a nutshell, the process of changing from object definitions to raster representation (that is, scan conversion) destroys whatever high-level structure one has in a scene. Since, on the other hand, one generally operates on a semantic level of interaction (a user likes to say "move this polygon from here to there," for example) pixel representation is not flexible enough for manipulation purposes.

Presently, there exist two mainstream approaches in the area of raster graphics to make it a more viable alternative for interaction:

- *Pushing* the hardware limits to the maximum, viz. running many processors per given task in the raster image generation pipeline.
- *Restructuring* the functional model first formalized by Carlbom [1,4] for high-performance architectures, viz. looking at the raster image generation from a fresher perspective. (Whitted and Weimer [5], and Levy [6] also offer useful lessons on raster architectures.)

Obviously, there is nothing wrong with the first approach as long as it stays cost-effective and exhibits extensible (read ''programmable'') behavior. In fact, there are some well-known systems, commercial as well as experimental, which prove the power of this approach: Clark's Geometry Engine (which was initially an experimental system) [7] and Fuchs' Pixel-Planes [8], respectively*. The Geometry Engine is based upon a custom VLSI chip. The chip is the building block of the Engine which consists of a number of chips placed in a pipeline and implements the various steps of matrix, clipping, and perspective operations — all in the classical homogeneous coordinates for handling 3-D projective transformations. The Pixel-Planes, on the other hand, is a frame buffer composed of custom logic enhanced memory chips (also known as a smart frame buffer, or a processor-per-pixel architecture) that perform several pixel-oriented tasks in parallel. With Pixel-Planes, it is most suitable to render images in the specific case that the pixel operations are described by linear arithmetic expressions (nevertheless see the preceding footnote). This is due to the fact that in Pixel-Planes the information sent to the frame buffer is not the triple (x coordinate, y coordinate, RGB value) but the triple (a, b, c); the RGB value at pixel (x, y) is then set to $ax + by + c$.

A common characteristic of first approach above is then to identify and isolate a simple (subset of) operation(s) and map it, frequently in a conceptually easy manner, to hardware. Since custom VLSI design has become, in the last decade or so, as easy as writing software (certainly only for those with the right kind of facilities), there is a manifest incentive to do this. We, however, observe a major shortcoming with this approach. If there are ultimate limits (as many people believe) to what can be done by brute-force hardware speed-ups, one should be careful not to assign the problems of computer graphics to hardware designers. Software issues should also be painstakingly studied. This consequently brings us to the second approach.

This second approach, the one we subscribe to, is no different from the first one in terms of its goal, i.e. obtaining raster graphics systems with fast response and interaction. Yet, the methodology is quite different. In this case, one tries to develop original data structures and devise new architectural organizations, and then maps expedient tasks to hardware as much as this is justified†. Here, we'll review how we are attempting to do this in our research project at CWI. Accordingly, this paper can also be construed as a position paper.

* However, neither of these systems is extensible. For example, it is planned in Pixel-Planes to have new hardware capability to evaluate quadratic expressions directly and to speed up rendering with the capability to display simultaneously various primitives at different regions of the screen [8].

† An simple analogy here may be useful. Consider the task of searching a set of items for a specified item. One obvious way is to store one item per processor and then broadcast the item we are looking for to the processors. In constant time we know if the search is successful. This is fast but costly in that a linear number of processors would be necessary. Another way to address the problem is to sort the items and then use binary search to achieve a logarithmic time bound. In doing so, we used only one processor and ended up with a reasonably fast

1.1. Hardware for Graphics: Some Systems

To our best knowledge, our way of restructuring the raster image generation pipeline has not been considered before in the literature. The aim of this section is to look at some other efforts of the first sort mentioned in §1. A longer (but not comprehensive) overview can be found in [8]. For brevity, several works are not mentioned here; this shouldn't be taken as a sign of their insignificance. We also do not dwell on well-publicized systems such as the Geometry Engine, the Pixel-Planes, and the 8 by 8 Display [9].

Guttag, van Aken, and Asal [10] discuss issues that must be taken into account in the design of a VLSI 32-bit microprocessor specialized for graphics applications. They particularly stress the requirement that such a processor should be general enough to perform any graphics operation, in contrast to existing graphics controllers whose command sets are frozen in hardware.

England [11] offers a graphics system architecture for interactive, application-specific display functions. A modular architecture with a single common bus structure, a general-purpose 32-bit micro, a large image memory, and a flexible display controller are among the key features of his proposal. England's ideas were implemented first as a research tool, and then as the Ikonas Raster Display System (now available as Adage 3000).

A special issue of *IEEE Computer Graphics and Applications* gives a good picture of some other commercial research efforts [12]. Asal et al. [13] describe the design of the Texas Instruments 34010 Graphics System Processor which is based on a combination of reduced- and complex instruction set architectures. This system is a perfect example of the wheel of reincarnation philosophy to be mentioned in §1.2; the chip is programmed in C to perform image generation *faster* than a general purpose 32-bit micro. Carinalli and Blair [14] describe National's Advanced Graphics Chip Set, and Shires [15] surveys the Intel 82786, a new VLSI graphics processor that supports windowing in hardware. In the latter, to control multiple windows, the central processor supplies the display processor with a description of the parts of various windows that are to be shown on the screen. The display processor then fetches the implied pixels.

Research by Gharachorloo and Pottle [16] is concentrated on a real time graphics system called Super Buffer. Super Buffer is built around a systolic raster graphics engine which in turn is equipped with an array of identical specialized pixel processors. The underlying philosophy of Super Buffer can most characteristically be summarized as a marriage of the classic Watkins scanline algorithm and the processor-per-polygon way of thinking due to

time bound. This was possible because we "structured" our data, albeit in a very simple way for this trivial example.

Kilgour [17]. The system is currently under construction at IBM Thomas J. Watson Research Center.

Demetrescu's [18] SLAM (Scan Line Access Memory) consists of a semiconductor memory augmented with highly parallel but simple processors dedicated to fast rasterization. A frame buffer consisting of SLAM chips can paint an arbitrary horizontal segment of pixels in one memory access. SLAM has been fabricated and tested.

1.2. The Wheel of Reincarnation

The ''wheel of reincarnation,'' first formulated by Ivan Sutherland in the '70's, is a philosophical tendency in graphics hardware design. Consider a frame buffer holding an image generated from geometric data. If the frame buffer is connected to a host processor that paints the picture, the host hardware is somewhat underutilized. It is more advantageous to free the host by giving the frame buffer some independence. Sutherland noticed that there is an inclination among designers to ''off-load'' graphical computation from the host to a graphics processor, and then repeat this process (i.e. off-load the computation again from the graphics processor to a graphics subprocessor, and so on). The process is repetitive since, when designers add more and more registers and functions to a display processor, the processor becomes a full fledged CPU again. At that stage though, the processor has become slow and inefficient so that need arises for a small, fast display processor, starting the cycle anew. It is possible to observe this cycle today in several graphics hardware systems. We think that the wheel of reincarnation may explain our design decisions, too. For example, the separation module and the search-and-administration module described in [19] are good examples of this philosophy.

2. Pattern Representation

"... The choice of representation has a significant impact on the overall design of the [raster graphics] system. A system to handle lines and dots is likely to be similar to a line-drawing graphics system and indeed can offer an identical function set and can use much of the same software. A solid-area graphics system is rather different; internally it must incorporate scan-conversion software, and it must provide functions that permit the programmer to define each solid object's mask, shading, and priority. A complete contrast is provided by systems using raster representation, for here the programmer is concerned with operations on arrays of intensities; the operations include moving and copying arrays, applying half-

toning or enhancing algorithms, and so on." [20]

Patterns are area-oriented picture elements. The representation for patterns is such that efficient mapping to the frame buffer of a raster display is possible. Ideally, this mapping should be comparable in speed to the vector generators of conventional vector displays. A strong point of patterns is their independence from the type and resolution of raster hardware. The following description is quite condensed; cf. the original paper [21] for details of the representation and several examples of the set-theoretic manipulation of pattern domains.

2.1. Definitions and Semantics

Definition A *pattern primitive* (*pattern* in short) is a pair (*Dom*, *Col*) where *Dom* is a *domain function* and *Col* is a *color function*. *Dom* specifies a bounded planar section of a 2-dimensional manifold and conceptually can have a curved boundary although the present discussion assumes a polygonal boundary. *Col* assigns a color to each point of *Dom*. □

This division of an area element into two parts means the following. When a pattern is visualized, *Dom* will have to be mapped to CO^{dev} (see §3.1 for abbreviations). During this mapping, each raster element bounded by *Dom* will be painted with the e.g. RGB value specified by *Col*.

Normally, a system will have a number of standard domain functions like *TRIANGLE*, *SQUARE*, and *POLYGON* and a number of elementary color functions like *RANDOM*, *SOLID*, and *INTERPOLATED*. Since we haven't said anything to the contrary, domains may self-intersect and even have disconnected parts. Domains are subject to pattern combining operations which are all set-theoretic. Assume that we have two patterns $Pat_1 = (Dom_1, Col_1)$ and $Pat_2 = (Dom_2, Col_2)$. If we now take $Pat_1 \cup Pat_2$ then the result will consist, in the general case, of three patterns. (*N.B.* For brevity, here we assume that Pat_1 and Pat_2 do not interpenetrate although this wouldn't cause any difficulty. In fact, in §2.2 this case will be accounted for.) The first pattern is $Dom_1 - (Dom_1 \cap Dom_2)$ with associated color function Col_1. The second pattern is $Dom_2 - (Dom_1 \cap Dom_2)$ with color function Col_2. Finally, we have $Dom_1 \cap Dom_2$ with a color function that is up to user's specifications. (If both domains are solid then this pattern would have one of the domains' colors; in case of transparency the situation would be different.) The following definitions are thus useful.

Definition A *mix-attribute* specifies how the color functions of two domains should be handled (mixed) when they are subjected to set-theoretic operations such as $\cup$ and $\cap$. □

Definition Color functions may be either *generators* or *pseudo-rasters*. A generator is a procedure (usually a one-line statement or formula) yielding a color for each point of a domain. A pseudo-raster is a 2-D array of color cells covering a domain. Combining patterns may require the rewriting of generators or pseudo-rasters. □

A geometric transformation of a pattern can affect both domain and color functions. A color transformation obviously affects only the color function. The use of this simple property is quite powerful: interactive support of blinking, highlighting, etc. can be done by a simple update of the color function of a domain. After a point-location (hit-detection) to find the pattern that is pointed out, an update of the color function is done on-the-fly (using, if necessary, a simple special purpose function box).

When representing domains a useful structure to have in order to facilitate comparisons between domains is the bounding-box of the domain. All subsequent operations (manipulations) with the domain maintain this box. Furthermore, a *characteristic function* is used to indicate whether an element is a member of the pattern. This is a raster-independent function and when one has to finally rasterize the pattern, the characteristic function should only be evaluated for those raster points. The representation of domains is reminiscent of run-length encoding. The coding here is raster-independent. The scanlines in the encoding can be at arbitrary y-values; as a result, the number of scanlines used by the representation will be minimum.

2.2. Visibility for Patterns: A Simple View

This section studies the visibility problem when patterns are the underlying representations. The reader is cautioned that this is only an abstract view in that it does not exactly correspond to our concrete proposal given in [19]. Nevertheless, like Fiume and Fournier [22] we believe in the importance of such abstract analyses. Note also that we want an incremental visible surface algorithm which supports addition/deletion of objects to/from the scene. A simple scenario goes as follows. Imagine a screen holding a picture with hidden surfaces removed. One then points with a mouse to particular region of the screen and picks it. There are several alternatives to what happens next:

- The picked region is highlighted.
- The boundary of the object facet which gave rise to this region is highlighted.
- Besides this region, all other visible regions which are parts of the facet which gave rise to this region are also highlighted.

It is our understanding that an ideal system would give the user the last feedback. Similarly, when an object is deleted by deleting its visible regions from the picture, previously invisible object parts may become visible. On the other hand, inserting a new object may cause previously visible regions to become invisible. The picture must be updated, in (almost) real-time, to reflect these changes. A lazy evaluation is always possible. After updating the picture itself, the necessary structural changes that should be incorporated in the medium and high level descriptions can be done in the background (cf. §3.2).

Let us start with a definition of the visibility problem. We have a scene consisting of a set of patterns, $\{P_1, P_2, \cdots, P_n\}$. These patterns normally come from the facets of the 3-D objects in the scene. Facets originate from a boundary representation (*Brep*) and are coded as filled areas. While most visible surface algorithms use some form of sorting, we'll take a different approach in the following description. Each pattern P_i is a domain - color function pair (D_i, C_i). Assume, without loss of generality, that the viewpoint is at $+\infty$ in the z direction and we are going to produce a visible surface picture in the xy-plane.

An important issue mentioned in §2.1 is that of *interpenetration.* Explaining what is a "visible" part of an object would be easier if objects did not interpenetrate. However, in reality, visible surface algorithms are required to deal with interpenetration. There are two solutions: either a special treatment or a clear, more semantic understanding of interpenetration. Since majority of objects do not interpenetrate in a normal scene, time spent in handling special cases may be justified. We however choose the second alternative and base our visibility notions on top of interpenetration.

Given two patterns P_i and P_j, we define $D_i \; \omega \; D_j$ (the overlap operation) as follows (we'll sometimes denote D_i by a and D_j by b):

$$a \;\omega\; b = \{\hat{a}, \hat{b}, ab, ba\}$$

where

$$\hat{a} = a - b \text{ and } \hat{b} = b - a$$

$$ab = a \cap b \text{ restricted to } a \text{ before } b$$

$$ba = a \cap b \text{ restricted to } b \text{ before } a$$

The property here is that $\hat{a}, \hat{b}, ab, ba$ are all disjoint. If D_i and D_j coincide partially or completely then it is undefined at the coincident region which facet is visible. The predicate "before" has the obvious meaning in this setting; all operations refer to the projections in the xy-plane and "before" makes them meaningful in 3-D.

Now, for visibility, it is seen that

$$vis(D_i, D_j) = \{\hat{a} \cup ab, \hat{b} \cup ba\}$$

This is basically the visible picture of these two domains. It is equivalent to $\{a - ba, b - ab\}$. It is noted that if $\{b_i\}_{(1,n)}$ denotes a sequence of n disjunct facets then $vis(a, \{b_i\}_{(1,n)}) = \{vis(a_0, b_i)\}_{(1,n)}$. Here we see that, for i running from 0 to $n - 1$,

$$a_{i+1} = a_i - b_{i+1}a_i \text{ with } a_0 = a$$

As a result, $a_n = a - \biguplus_{1}^{n} b_i a_{i-1}$ where all $b_i a_{i-1}$ are mutually disjunct.

Merging two lists $A = \{a_i\}_{(1,n)}$ and $B = \{b_i\}_{(1,m)}$ can be done using parallel hardware. If this can be performed fast, then the visible picture for $\{P_1, P_2, \cdots, P_n\}$ can also be computed and maintained (in the presence of additions and deletions of objects) fast.

3. Advanced Display Architecture

"... In an ideal world of zero-cost logic and memory, which tasks would we choose to implement with special-purpose hardware and which with software running on general-purpose hardware? Because interactive graphics is so performance-sensitive, we would choose to carry out both the output and the input transformations (from model to image and from user action to model modification) in hardware, provided we could exercise some choices about the structure of the model, the interaction dialogue, etc. In essence, we would want to implement the standard algorithms of the entire output and input pipelines in customizable hardware for maximum performance. Thus we would be able to combine flicker-free display with dynamic updating of an application model and its views. More specifically, we would combine the best features of a flicker-free storage display with dynamic, selective updating." [1]

Raster systems contain, in general, a frame buffer whose main use is to decouple the refresh and scan conversion processes. Clearly, the information in a frame buffer is unstructured. Besides, the resolution of a frame is a function of the resolution of the display. Furthermore, unless a double buffer is used, a change cannot become visible at once. On the other hand, a nice property of frame buffers is that one can use one of the bit planes for interaction response.

A structured display file on the other hand is compact, reducing the bandwidth requirements of the memory. Information is structured and this facilitates the manipulation of the image. Local changes can be performed immediately and correlations with accompanying objects can be made direct.

3.1. Carlbom's Model

Let us start with making the usual distinctions about the coordinate spaces. Here also, we'll try to isolate important material in definitions; the reader is warned that the definitions shouldn't be taken as formalisms.

Definition *World coordinates* (CO^{world}) are used by the application models. *Normalized device coordinates* (CO^{norm}) are used internally by the graphics packages to provide device independence. Finally, *device coordinates* (CO^{dev}) are used to display an image, in other words, address the raster screen. □

Carlbom, in her dissertation titled *System Architecture for High-Performance Vector Graphics,* provided a common framework for studying the architecture of graphics systems [1]. Her work was originally oriented towards performance modeling for vector graphics hardware. It is nonetheless easily translatable to the raster realm.

The model, in its simplest version, is a pipeline which takes application commands from the user and produces images on a screen (Figure 1).

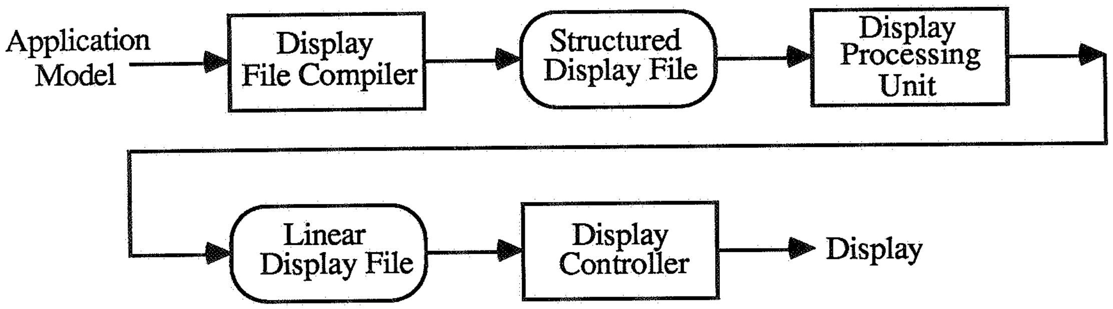

Figure 1. Carlbom's classical architecture

Definition An *application data structure* contains both graphical and nongraphical data. The geometric objects are in CO^{world}. The *structured display file* contains only graphical data and these are in CO^{norm}. The *transformed segmented display file* contains primitives that can be directly mapped to raster; hence its contents are in CO^{dev}. □

The elements of the pipeline are transformed by intermediate "agents."

Definition The *display file compiler* inputs the application data structure and outputs the structured display file. The *display processing unit* inputs the structured display file and outputs the segmented display file. The usual *scan conversion* and *refreshing* steps are the last two steps of the pipeline. □

3.2. Our Model

We propose three levels of representation for 3-D objects: high, medium, and low (Figure 2).

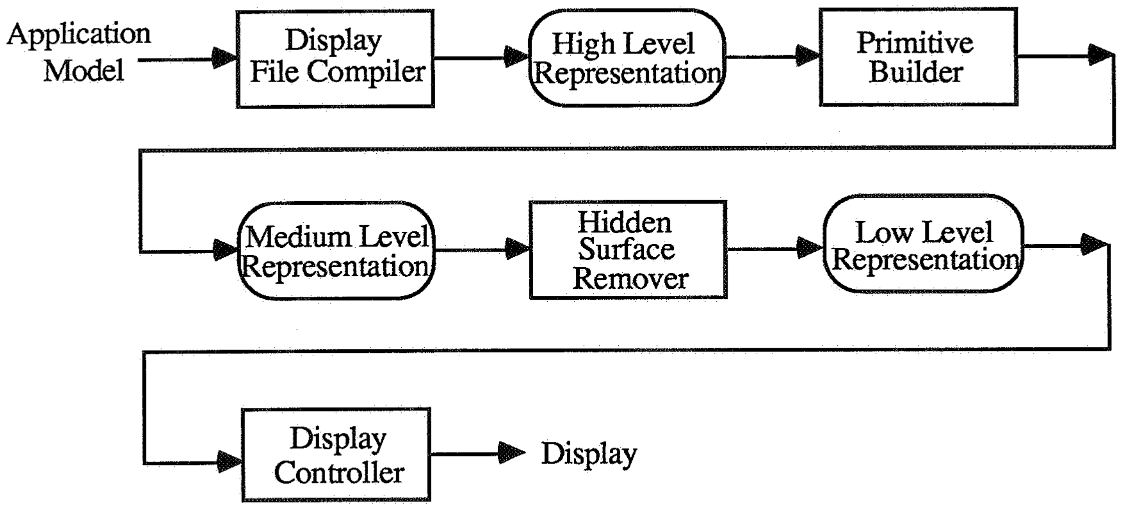

Figure 2. The proposed modified architecture

<u>Definition</u> The *high level* contains an object representation directly resulting from the user's definition, and in his coordinate system. This may be thought of as an equivalent of the GKS segment representation. At this level there is no limit on the type of the primitives that can be present. Simple primitives include filled areas, lines, characters, markers, etc. while complex primitives include things like B-splines, etc. □

<u>Definition</u> The *medium level* consists of elements representing a domain with one color function. (At the high level a domain can have several color functions.) Disjunct domains are divided and stored as separate elements. At this level, the areas are still overlapping; that is, a visible surface algorithm is necessary before they are mapped to the low level. Each filled area is one connected region. If x is a primitive from the high level which gave rise to y at the medium level then, considering the mapping f between these elements, the relation $f^{-1}: y \rightarrow x$ is a function whereas $f: x \rightarrow y$ needn't be a function. □

<u>Definition</u> The *low level* representation is the resulting list of visible areas of the original objects. In the functional model this would be equivalent to the linear display file. It consists of the visible areas of the original objects. It contains disjunct areas whose color functions have been modified to reflect the effects of transparency, shading, etc. □

It is required that we maintain consistency among the three levels of representations. For example, in 3-D, to recover hidden elements after a change, going back to higher levels is necessitated. This means that, when pointing to an element of the low level, there should be a possibility to find the corresponding element in the medium level representation.

Similarly, an element in the medium level should know the corresponding element in the high level representation. Considering the representations as objects in the sense of object-oriented languages [23] may clarify the situation. Each object (at any given level) has a set of associated ''methods'' that it responds to. When an action is performed on an object, the object itself may initiate a series of actions to keep the data structures intact. Some actions require permanent changes in the objects at the high level (e.g. the action of deleting an object). Some changes are just illusory (e.g. making an object transparent to see what is behind). These may be effectuated by providing each object with the appropriate methods. The user need not (and should not) know how these administrative tasks are carried out. The situation is similar to sending a file to the spooler for printing. One is not concerned with what kind of intermediate level representations are created to print a file nor is he concerned with the font files to be called, bit maps to be operated upon, etc. Similarly, while interacting with a graphics system, all a user wants is to see the immediate changes that he wants to carry out and is to be guaranteed that all through the interactions the geometry is kept correct.

The methods and their division according to the level they should be carried out are given below:

- Picking, highlighting, and blinking belong to the low level.
- Visibility (priority), transparency, shading and reflection changes belong to the medium level. This level also includes the geometric transformations scale, translate, rotate, and clip.
- Grouping and viewing control belong to the high level. Depending on the choice, object insertion and deletion may be at this or at the medium level.

It should be emphasized that it is crucial to maintain consistency in the intermediate representations we're using. Assume that, in case of 3-D, we want to recover the previously hidden elements after the deletion of an object. Going back to higher levels is necessary. Although several trade-offs suggest themselves, the least overhead is probably incurred when one maintains ''pointers'' to the corresponding elements at each level. (This is one example of trading memory with computation.) Thus, when pointing to an element from the low level, the corresponding element in the medium level can be found. Analogously, elements in the medium level have corresponding high level elements. Concrete data structures for doing this in the case of hidden surface removal are given in [19].

4. Principles of Interaction: Dynamic Feedback

"... In raster graphics machines, the screen image is generated from the raster, which is a persistent data structure, and which does not automatically change with deletions from the transformed segmented display file. Even if deletions from the segmented display file do automatically trigger the scan conversion process to perform deletions from the raster, a problem remains: deletions from the raster are performed by drawing the object to be deleted in the background colour of the screen image. If the deleted object formerly obscured a remaining object, then after deletion the remaining object will have background coloured holes in it where it was previously covered by the deleted object." [3]

4.1. Soft Cursors

A cursor moving across a primitive temporarily adds an icon on top of the primitive. When the cursor moves, the original image of the primitive need to be restored. A tradeoff suggests itself using, in both cases, a separate raster (other than the one holding the image itself). In case of a ''small'' raster, the position is updated under the control of a a location register that is updated by an interrupt routine. This rather inflexible method is able to display only small cursor patterns. In the case of a ''large'' raster, one can implement a dragged object, or a rubber object [3].

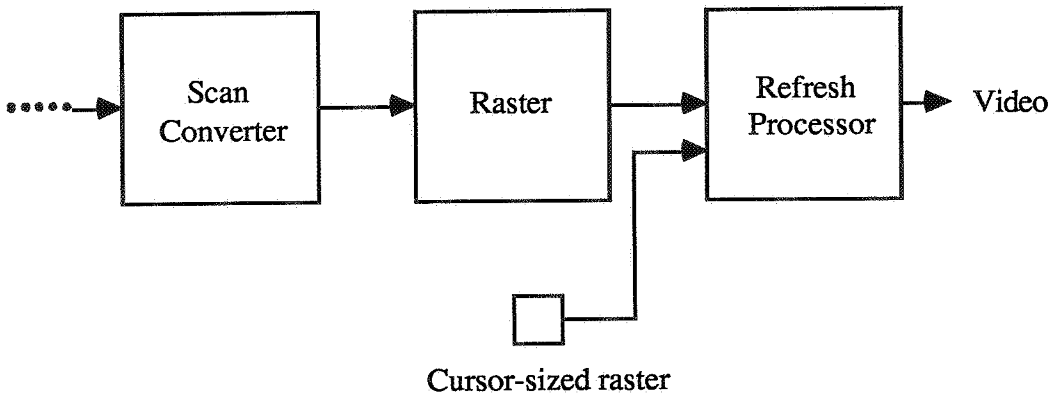

Figure 3. (Adapted from van Harmelen) A small raster for cursor feedback

In other words, until now, there have been two methods for providing dynamic feedback. Both methods provide a separate raster which is solely used for feedback. This separate raster can be modified without modifying the contents of the raster for the current image. The first method (Figure 3) provides a small raster containing cursor information. The position of the overlay is held in a position register. The position register is changed must change according to the changes in the position of the graphics device, say mouse. In the second method (Figure 4) the feedback raster is as big as the screen raster and overlaid in the same location. The reader is referred to [3] for details.

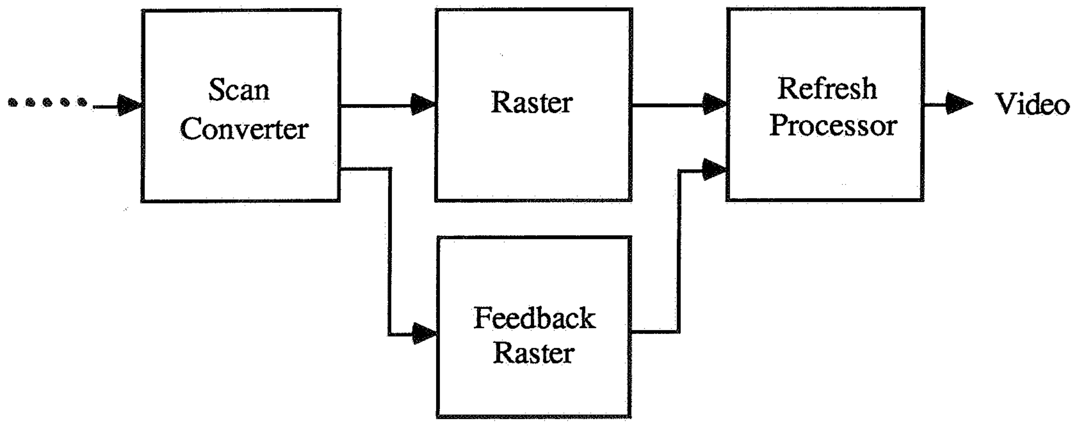

Figure 4. (Adapted from van Harmelen) A screen-sized raster for dynamic feedback

4.2. Camera Control

This refers to slight changes in the viewpoint. Shifting a camera could be implemented as a small rotation plus a scale plus a small translation. Equivalently, one takes the original picture, removes all objects and puts them back at a slightly different location. In any case, it should be possible to make use of the coherence. Given a pattern representation for a filled area, it is clear that a slightly moved version of the pattern would have almost the same pattern representation which can be evaluated fast (at least approximately) and shown on the screen. Since users are interested in seeing some kind of action taking place in return for their commands interactively, the picture need not be perfect but is registered to be so in the upcoming cycles. This should be compared with the ideas of garbage collection in programming languages with dynamic data structures. In general, things would go fast except for those rare moments when the system may simply go to sleep to take care of the effectuated changes — in our case, extending the effects to the higher levels to register them permanently.

4.3. Color Functions

After a change of the linear display file a pre-evaluation of color functions can be done. This pre-evaluation can do such things as calculating the effect of the angle of incidence of the light and the plane normal of the domain on the color function. This can be a simple effect if just smooth coloring is wanted, but it might even be combined with texture. For example, think of a texture pattern imitating a brick wall. If the light source is moved, different sides of the bricks can light up. These pre-evaluations may be complex, but they only have to be done after a change. This is as opposed to the pre-evaluation to be performed for each scanline in the classical case.

5. Summary and Future Work

"... Computer graphics has always been an expensive proposition and its users a demanding, unsatisfied lot — the picture never got onto the screen fast enough, or later, never moved fast enough once it got to the screen, and then, the picture was never sharp enough, never realistic enough." [8]

Computer graphics is now commonplace thanks to continuing reductions in hardware prices and enhancements in processing capabilities. Especially raster graphics profited from these improvements. Combining flexibility with ever-increasing realism, raster is establishing itself as the true ''future'' of graphics.

We find today's raster graphics devices quite impressive for the amount of processing power and memory they have and for their innovative techniques in video generation, bitmap graphics, geometric transformations, clipping, multiple window support, etc. Little however has taken place in regards to the graphics display architecture. Especially, recurrent high-level interactions such as those involving camera control, color changes, shading, reflections, transparency, highlighting, visibility, object insertion/deletion, picking, etc. are not supported in their entirety by current systems.

In this paper, we gave an overview of our novel object description scheme called ''pattern representation.'' In this scheme, only one type of primitive — a pattern consisting of a domain function and a color function — is permitted. Patterns have an associated set of operations. This is reminiscent of abstract data types (or more properly, the ideas of object-orientation [23]) and provides the much needed conceptual simplicity in several interactive graphics tasks.

A natural outcome of using pattern representation has been a re-examination of the classical image generation model [1,4]. Briefly, in our revised model, an application as defined by the user contains possibly nongraphical data. It is in continuous coordinates and there is no limit to the variety of primitives that can be present (the ''high'' level). At the ''medium'' level, only patterns are allowed. In a sense, patterns are preprocessed representations for filled areas and resemble scanline representations. At the medium level patterns are still overlapping. At the ''low'' level are the resulting visible regions. Up to this point everything can still be kept in the object space. The display controller maps them to the image space. Figure 5 summarizes this architecture.

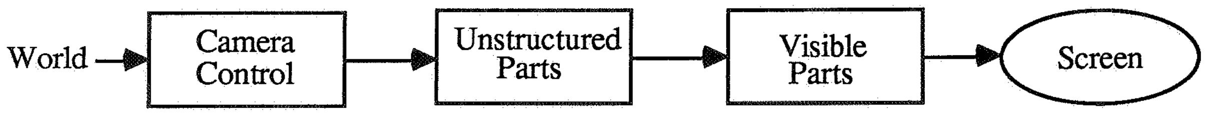

Figure 5. A simplified view of three-level representation

A lot of work remains to be done. As we are currently doing with the visibility [19] problem, our algorithms must be coded and tested to identify bottlenecks and to determine their suitability for realization in silicon. Our study so far gives us hope that the methodology summarized in this paper will lead to productive results, cf. [21,24] for details.

We want to close this paper with a pronouncement on the methodology of graphics hardware design. Today, we detect a tendency among designers for mapping software tasks to hardware without really thinking about any optimization in terms of data structures and algorithms. Probably the best examples of this course are the ever-growing family of ''ray tracing'' systems which basically consist of a large number of distributed processors [25], and constructive solid geometry systems which map CSG trees to hardware as such [26]. Computational complexity community has long ago come to know that the laws of parallel computation are qualitatively different from that of the sequential computation. (Think of all that research in parallel sorting algorithms.) In short, algorithms do not always smoothly translate from uni- to multiprocessor architectures. We believe that without clarifying the algorithmic improvements, such brute-force mappings into hardware will introduce only temporary speed-ups[‡] and these improvements will be nullified, in time, by growing user demands. The real solutions to the hard problems of graphics will come, in our view, from a direction which worries about the intrinsic difficulty of problems from a computational standpoint, cf. Fiume and Fournier [22] for an exemplary study of this sort.

[‡] For example, one cannot obtain fast (that is, polynomial time) algorithms for NP-complete problems by running a bounded (in the problem size) number of processors in parallel.

Acknowledgments

This paper is a condensed version of [24] and [21], the original articles on the subject matter. Several ideas in this paper were influenced by the following: Clark [27], Kilgour [17], and van Harmelen [3]. We also benefited from a rereading of Chapter 10 of Foley and van Dam's [1] and Part 4 of Newman and Sproull's [20] textbooks. The interpretations presented here are nevertheless our own and shouldn't be taken as reflecting the views of the above authors.

References

1. J. Foley and A. van Dam, *Fundamentals of Interactive Computer Graphics,* Addison-Wesley, Reading, Mass. (1982).
2. F.R.A. Hopgood, D.A. Bruce, J.R. Gallop, and D.C. Sutcliffe, *Introduction to the Graphical Kernel System (GKS),* Academic Press (1983).
3. M. van Harmelen, "Process-data models for raster graphics and image analysis," in *Theoretical Foundations of Computer Graphics and CAD*, ed. R.A. Earnshaw, NATO ASI Series, Springer-Verlag, Heidelberg (1988, to appear).
4. I. Carlbom and J. Michener, "Quantitative analysis of vector graphics system performance," *ACM Transactions on Graphics* **2**(1), pp. 57-88 (1983).
5. T. Whitted and D.M. Weimer, "A software testbed for the development of 3D raster graphics systems," *ACM Transactions on Graphics* **1**(1), pp. 43-58 (1982).
6. H.M. Levy, "VAXStation: A general-purpose raster graphics architecture," *ACM Transactions on Graphics* **3**(1), pp. 70-83 (1984).
7. J.H. Clark, "A VLSI geometry processor for graphics," *Computer Graphics (Proceedings of SIGGRAPH'82)* **16**(3), pp. 127-133 (July 1982).
8. H. Fuchs, "An introduction to Pixel-Planes and other VLSI-intensive graphics systems," in *Theoretical Foundations of Computer Graphics and CAD*, ed. R.A. Earnshaw, NATO ASI Series, Springer-Verlag, Heidelberg (1988, to appear).
9. R.F. Sproull, I.E. Sutherland, A. Thompson, and C. Minter, "The 8 by 8 Display," *ACM Transactions on Graphics* **2**(1), pp. 32-56 (Jan. 1983).
10. K. Guttag, T. Van Aken, and M. Asal, "Requirements for a VLSI graphics processor," *IEEE Computer Graphics and Applications* **6**(1), pp. 32-47 (1986).
11. N. England, "A graphics system architecture for interactive application-specific display functions," *IEEE Computer Graphics and Applications* **6**(1), pp. 60-70 (1986).
12. J. Staudhammer, "Reaping the benefits of the hardware revolution," *IEEE Computer Graphics and Applications* **6**(1), pp. 14-15 (1986).
13. M. Asal, G. Short, T. Preston, R. Simpson, D. Roskell, and K. Guttag, "The Texas Instruments 34010 Graphics System Processor," *IEEE Computer Graphics and Applications* **6**(10), pp. 24-39 (1986).
14. C. Carinalli and J. Blair, "National's Advanced Graphics Chip Set for high-performance graphics," *IEEE Computer Graphics and Applications* **6**(10), pp. 40-48 (1986).
15. G. Shires, "A new VLSI graphics coprocessor — The Intel 82786," *IEEE Computer Graphics and Applications* **6**(10), pp. 49-55 (1986).
16. N. Gharachorloo and C. Pottle, "Super Buffer: A systolic VLSI graphics engine for real time raster image generation," pp. 285-305 in *1985 Chapel Hill Conference on VLSI*, ed. H. Fuchs, Computer Science Press, Rockville, Maryland (1985).
17. A.C. Kilgour, "Parallel architectures for high performance graphics systems," pp. 695-703 in

Fundamental Algorithms for Computer Graphics, ed. R.A. Earnshaw, NATO ASI Series, Vol. F17, Springer-Verlag, Heidelberg (1985).

18. S. Demetrescu, "High speed image rasterization using scan line access memories," pp. 221-243 in *1985 Chapel Hill Conference on VLSI*, ed. H. Fuchs, Computer Science Press, Rockville, Maryland (1985).
19. F. Kuijk, P. ten Hagen, and V. Akman, "An exact incremental hidden surface algorithm," *these proceedings.*
20. W.M. Newman and R.F. Sproull, *Principles of Interactive Computer Graphics,* McGraw-Hill, New York (1979).
21. P. ten Hagen and T. Trienekens, "Pattern representation," Report CS-R8602, Center for Mathematics and Computer Science, Amsterdam (Jan. 1986).
22. E. Fiume and A. Fournier, "The visible surface problem under abstract graphic models," in *Theoretical Foundations of Computer Graphics and CAD*, ed. R.A. Earnshaw, NATO ASI Series, Springer-Verlag, Heidelberg (1988, to appear).
23. M. Stefik and D.G. Bobrow, "Object-oriented programming: Themes and variations," *AI Magazine* **6**(4), pp. 40-62 (1986).
24. P. ten Hagen, F. Kuijk, and T. Trienekens, "Display architecture for VLSI-based graphics workstations," Report CS-R8637, Center for Mathematics and Computer Science, Amsterdam (Nov. 1986).
25. N.S. Williams, B.F. Buxton, and H. Buxton, "Distributed ray tracing using an SIMD processor array," in *Theoretical Foundations of Computer Graphics and CAD*, ed. R.A. Earnshaw, NATO ASI Series, Springer-Verlag, Heidelberg (1988, to appear).
26. S.C. Marsh, "Fine grain parallel architectures and the creation of high-quality images," in *Theoretical Foundations of Computer Graphics and CAD*, ed. R.A. Earnshaw, NATO ASI Series, Springer-Verlag, Heidelberg (1988, to appear).
27. J.H. Clark, "Graphics software standards and their evolution with hardware algorithms," pp. 619-629 in *Fundamental Algorithms for Computer Graphics*, ed. R.A. Earnshaw, NATO ASI Series, Vol. F17, Springer-Verlag, Heidelberg (1985).