On the Power of the Frame Buffer

ALAIN FOURNIER
University of Toronto
and
DONALD FUSSELL
The University of Texas at Austin

Raster graphics displays are almost always refreshed out of a frame buffer in which a digital representation of the currently visible image is kept. The availability of the frame buffer as a two-dimensional memory array representing the displayable area in a screen coordinate system has motivated the development of algorithms that take advantage of this memory for more than just picture storage. The classic example of such an algorithm is the depth buffer algorithm for determining visible surfaces of a three-dimensional scene. This paper constitutes a first attempt at a disciplined analysis of the power of a frame buffer seen as a computational engine for use in graphics algorithms. We show the inherent power of frame buffers to perform a number of graphics algorithms in terms of the number of data fields (registers) required per pixel, the types of operations allowed on these registers, and the input data. In addition to upper bounds given by these algorithms, we prove lower bounds for most of them and show most of these algorithms to be optimal.

One result of this study is the introduction of new frame buffer algorithms for computing realistic shadows and for determining the convex intersection of half spaces, an operation important in computational geometry and in rendering objects defined using planes rather than polygons. Another result is that it shows clearly the relationships between different and important areas of research in computer graphics, such as visible surface determination, compositing, and hardware for smart frame buffers.

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; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*color, shading, shadowing, and texture*; *visible line/surface algorithms*

General Terms: Algorithms, Design, Verification

Additional Key Words and Phrases: Complexity, frame buffer, visibility

1. INTRODUCTION

Although a number of different output devices with distinct characteristics have been in use in computer graphics for some time, raster displays have become predominant in recent years. The reasons for this are quite simple. In the first place, raster displays are the most flexible output devices available because they

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada.

Authors' addresses: A. Fournier, Department of Computer Science, University of Toronto, Toronto, Ontario M5S 1A4, Canada; D. Fussell, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712.

provide the capability of displaying shaded, full-color images, as well as line drawings. Second, hardware advances have driven the cost of raster displays down, making this technology far more affordable and thus more widely available. These hardware advances consist primarily of cheaper and faster memory systems, since the cost of memory has long dominated the cost of a typical raster display system. This is because these displays are organized around a frame buffer memory array, which provides a storage buffer for the digital representation of the raster image. Since typical displays provide on the order of 512 × 512 pixels up to more than 1024 × 1024 pixels, even a black and white display system requires at least 32 kbytes of memory to store an image, while a high-resolution color system can require 3 Mbytes or more.

Regardless of the actual cost of memory in a frame buffer system, the frame buffer itself is a major memory resource, and graphics algorithm designers have understood for some time that this resource can be used for more than merely storing an image description. In previous years a frame buffer system with 18–24 bits per pixel often contained more memory than the host minicomputer to which it was attached. Thus, memory-intensive graphics computations were well advised to take advantage of this resource. The classic example of a frame-buffer-based computation is the well-known depth buffer visible surface algorithm, which is not only the simplest visible surface determination algorithm to implement, but also the most flexible and widely used such algorithm. Now with the advent of inexpensive host computers with very large address spaces that make them capable of directly accessing blocks of memory of the size of the largest frame buffers, the nice properties of this algorithm make it more useful than ever.

Indeed, technological changes have changed main memory from a relatively expensive resource to one that looks ever more cost effective when compared with processor cycles. This has begun to motivate algorithm designers in many areas to look at the trade-off between memory usage and the speed of a computation in a different light. Where before it may have been unreasonable to use a simple and fast algorithm that had huge memory requirements to solve a problem for which a slower algorithm with smaller storage utilization was available, now the former may be the preferred choice. This changing view of the constraints on algorithm design has been reflected in some recent research areas in computer graphics. For instance, the need to create realistic images of high complexity has led to the evolution of sophisticated image composition systems that combine images generated by a variety of rendering algorithms operating on different types of geometric models [13, 27, 29]. These systems present an alternative to monolithic rendering systems that operate on geometric scene representations that are relatively compact. They operate on discretized images that are essentially virtual frame buffers and therefore require a great deal of memory, but they present significant advantages in flexibility and ease of implementation over monolithic rendering systems. Such composition systems not only require the storage of the intensity information of an ordinary frame buffer, but also require additional registers to store properties such as *opacity* and *depth*. They perform a range of pixel-by-pixel operations to be performed on the images being combined, and thus are examples of frame buffer algorithms other than the depth

buffer algorithm. The depth buffer algorithm itself has been extended to allow antialiasing by a number of researchers [7, 15] at a cost of greater memory requirements, providing a further set of examples of the increasing use of frame buffer algorithms.

With the advent of custom VLSI technology, designers of graphics systems have turned their attention to the design of specialized hardware for high-performance graphics. Much of this effort has been devoted to the attempt to make "smart" frame buffers that implement in hardware many of the computations normally performed during the process of scanning out an image into a frame buffer [10, 17–19]. These systems typically perform some variant of the depth buffer computation for visible surface determination and may also do other computations that are analogous to software frame buffer algorithms. Thus, the idea of the designing graphics algorithms that make use of the frame buffer is potentially of increasing importance as the technology of raster graphics systems progresses.

In this paper we examine the power of frame buffer algorithms in a formal framework, with the goal of determining the inherent power of a frame buffer to implement various types of algorithms. The study is intended to resemble the techniques used in the general analysis of the concrete complexity of algorithms within a particular computational model. In this case we define a frame buffer formally as a computation device, with storage capabilities and a set of operations that it is capable of performing. In this way we remove the host computer from the model in order to clarify the power of a frame buffer algorithm, both in terms of the amount of storage required of the frame buffer itself and in terms of the operations required to implement the algorithm. The resulting notion of "frame buffer complexity" proves useful in allowing us to characterize frame buffer algorithms in terms of their relative storage and performance requirements, as well as to determine lower bounds on storage and computation for certain fundamental problems in image generation.

The understanding of frame buffer capabilities is not merely a nice theoretical goal in itself. It also leads to the design of useful new algorithms that might have remained undiscovered in the absence of such a study. We introduce here some new algorithms and include them in our analysis. One is an algorithm for generating realistic shadows cast by opaque objects from any number of light sources. By realistic we mean that the overlapping shadows can vary in color and intensity depending on the light sources obscured, instead of merely having regions that are or are not in shadow. Another algorithm efficiently solves the problem of calculating the projection of a convex volume of intersection of half spaces onto the screen. Also the application of the latter to image generation might not be as obvious as the shadow generation algorithm; it is a problem of general interest in computational geometry and is useful in computer graphics for rendering models conveniently described by planes rather than polygons [16].

As important, in addition to providing a deeper understanding of the nature of frame buffer algorithms, our model is also well suited to identifying some key characteristics of the hardware required for smart frame buffers that are intended to implement a given class of image generation algorithms.

The remainder of the paper is structured as follows: In the next section we formally define our model of a frame buffer and introduce related terminology. This is followed by a classification of several important frame buffer algorithms according to the capability required of a frame buffer that implements them, along with a proof of lower bounds on the frame buffer complexity of several of the most important of these problems. We also introduce our new algorithms at this time. Finally, we discuss the interpretation of these results in terms of their impact on the design of graphics systems, both in hardware and software.

2. THE FRAME BUFFER MODEL

2.1 The Frame Buffer

We define the frame buffer as an abstract machine in a style similar to the definitions of automata theory. We do, however, try to avoid excessive formalism and do not spell out in detail what should be unambiguous. A *frame buffer* is an $m \times n$ array of pixel automata, working independently of each other, but synchronized periodically. The interval between two synchronization signals is called a *frame* or *pass*.

Each automaton consists of the following:

—*A read-only input tape*, from which it reads information in chunks, called *point records*. These records are made of size-bounded rational numbers, integers, and Boolean values. The automaton cannot back up the tape and looks at only one point record at a time.

—*A storage memory*. This memory is finite, of a size variable with the type of frame buffer, but with at least one storage unit, and it stores either size-bounded rational numbers, integers, or Boolean values. It can be write only or read/write and is essentially a register memory. In order to avoid the possibility of using an encoding scheme to allow values of different types to share registers, thus unduly complicating our model, we always assume that each register is of a size sufficient to contain the full range of the values stored and no more.

—*A finite-state control*. In the most general case, this is a mapping from a point record and the register contents to the register contents. Note that we thus define the machine to have only one state to force all information to be stored in the registers. The kinds of frame buffers studied differ both by the number of registers they have and the kind of mapping they allow. In particular, we distinguish automata according to whether they can compute functions of rational numbers to rational numbers (f: $Q^n \rightarrow Q$), predicates on rational numbers, integers, or Boolean values, or neither.[1]

We distinguish between predicates and functions mainly because we want to study some frame buffers that can store in their registers only a copy of a number on the input tape or the contents of another register. They use the predicates to

[1] We do not use real numbers to avoid problems with computability and the size of the numbers on the input tape and in the registers. Turing computable real numbers could be used but would unduly complicate the formalism. As a consequence, we must be careful about closure under the operations applied to the numbers.

effect conditional assignments. In fact, the predicates we use are very easy to compute.

We occasionally assume that each automaton knows its address (ij), $1 \leq i \leq m$, $1 \leq j \leq n$, but in no case does an automaton have the ability to communicate with other automata. In essence, this model of a pixel automaton is similar to the model of the RAM [2], with the crucial difference that the number of registers is bounded. Lower and upper bounds for similar machines have been obtained for selection and sorting algorithms [12, 23].

A major goal of this work is a study of both time and space complexity for algorithms in this model. The space complexity is simply the number of registers required of our frame buffer in order to perform a particular algorithm. Once it is recognized that in an on-line algorithm all the input tape has to be read at least once, it is clear that the time complexity is the product of the number of passes and the number of steps necessary to process each record. The latter is always a small constant in the algorithms given. Thus, the number of passes is the crucial factor in the time complexity.

2.2 Notation and Semantics

The result of a computation for each automaton is defined to be the contents of a distinguished register (the automata have at least one register each) when it exhausts its input tape. In more familiar terms, this register represents the *pixel value*. There can be more than one such register for color pictures. The set of all distinguished registers for the array of automata is called the *picture*. This register has to be readable from the outside, to be able to display the picture. We use this fact in some proofs. $\mathbf{SP}_{ij}$ (for stored pixel) is used to denote this register for automaton (ij). Most of the time, the ij subscript is dropped since we consider only one automaton at a time. Other registers we might name, to help remember their semantics, are **SZ** (stored Z value) and **CNT** (a counter). On the input tape, the most frequent elements of a point record are rational numbers X, Y, Z, and C. The X, Y, and Z are to be understood as coordinates, and C as a color. The triple (X, Y, Z) is called **P**. Global values can be carried on the input tape, but have to appear on each record that needs them. They are, in effect, global *constants*. A typical use is to define a background color.

In practice, the data on the input tape are the output from a scan conversion process sent (possibly in parallel) to the automata. We do not specify here the algorithm for the conversion. We show that the automata themselves can compute it.

To help in the nomenclature of frame buffers, we use a shorthand notation. FB_{rbc} is a frame buffer, where r is the number of rational-number registers, b is the number of Boolean-valued registers (i.e., flags), and c is a symbol for the capabilities of the automaton; where 0 means that the only function allowed is an identity mapping from one of the point records to one of the registers, p that the automaton can only compute predicates of the input and/or register contents, and f that it can compute both functions and predicates. It is occasionally useful to distinguish functions of the input alone from those of both the input and the register contents. Figure 1 contains a pictorial representation of an abstract frame buffer of type FB_{11c}, with the **SP** register and a single Boolean flag.

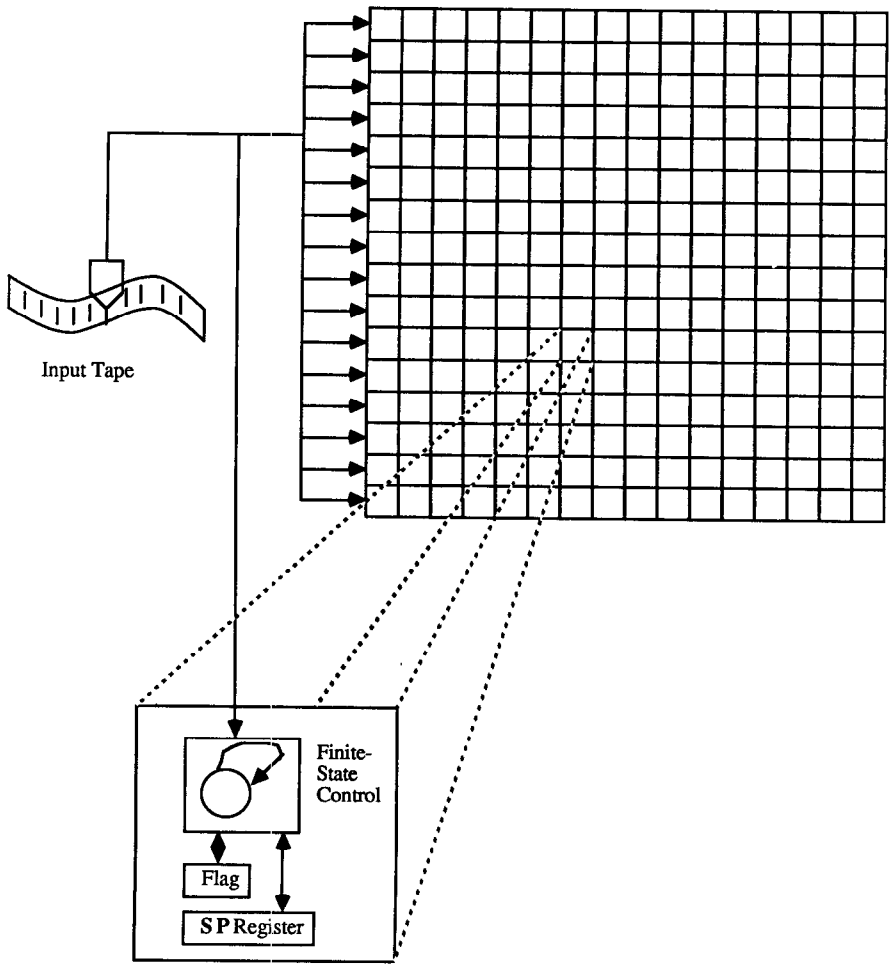

Fig. 1. Pictorial representation of FB_{11c}.

2.3 Problems to Solve

The most general form of the problem we ask our machine to solve could be described as "produce a picture." In more precise terms, we want the contents of the $\mathbf{SP}_{ij}$ when all the input tapes are exhausted to be such that they satisfy some relationship based on the content of the input tapes.

2.3.1 *The Visible Surface Problem.* As an example, let us look at the *Visible Surface Problem* (*VSP*). There have been several attempts at formalizing the VSP from various points of view [20, 25]. In our context, we can define it as follows.

First, we define the Hide (H) relation in $\mathbf{Q}^3 \times \mathbf{Q}^3$. Given $\mathbf{P}$ and $\mathbf{R}$ triples of rationals (X_p, Y_p, Z_p) and (X_r, Y_r, Z_r), $\mathbf{P}H\mathbf{R}$ or $(\mathbf{P}, \mathbf{R}) \in H$ iff $X_p = X_r$, $Y_p = Y_r$, and $Z_p < Z_r$. It is easy to see that H is irreflexive, nonsymmetric, and transitive. Therefore, the relation H', defined as H but with the inequality replaced by

$Z_p \leq Z_r$, is reflexive, antisymmetric, and transitive, and defines a partial order. It is often more convenient to use H, since it corresponds to the usual notion of a point hiding another. Note that this is geometrically exact if we consider points in screen coordinates after the perspective transformation. A point **P** is *visible* if there is no **R** such that **R***H***P**. The VSP is then: *given a set of points, find the set of visible points.* Since the property of visibility depends on the coordinate system, we use the notation $V_E(\mathbf{P})$ to indicate that **P** is visible in the coordinate system E, unless the coordinate system is unambiguous from the context.

Generally, the point sets of interest in computer graphics are infinite and are not given explicitly, but rather as continuous geometric figures. They are typically straight-line segments defined by endpoints, and polygons defined by vertices. It is convenient to assume that the input tape for pixel atomaton (i, j) contains only point records for which $\lfloor X_p \rfloor = i$ and $\lfloor Y_p \rfloor = j$. In other words, the pixel represents an area from lower left corner (i, j) to upper right corner $(i + 1, j + 1)$, and the data have been scan converted. By this definition, the VSP for one automaton is as difficult as the general problem. Since in our model the only allowed storage for the automaton is of bounded size and the answer to the VSP can be of arbitrary length, it is not possible to give an exact answer to the VSP even for this one pixel. However, we could be satisfied with the answer to the *Exact Area Sampling Problem* (*EASP*). The EASP is defined as follows: Given A_p, the area of a pixel, and n sets of points $S_1, S_2, \ldots, S_n$, with set S_i defining visible area[2] A_i and having color C_i, find the *pixel color* C_p such that

$$C_p = \frac{1}{A_p} \sum_{k=1}^{n} A_k C_k.$$

We leave it vague, in general, as to how a point set defines a visible area, but it is easy to specialize this definition to polygons, parametric patches, etc. In practice, areas are convolved with some filter to get a better estimate of each contribution to the pixel. We can look at the A_i as being the results of such a convolution, and at A_p as being a normalization factor.

Since the answer is now a single number, one could hope to be able to get it from our automata, but it can be shown that in our model the answer to the EASP cannot be obtained. The proof uses the fact that we have a limited number of registers, and that the FB performs an on-line algorithm and therefore should have the right answer at any time.

THEOREM 1. *An automaton with a finite number of registers cannot compute the answer to the EASP.*

PROOF. Assume an automaton with k registers can solve the EASP. Further assume it reads as input a tape containing two kinds of records: The first is a series of $n (n > k)$ sets of records, such that for each pair of sets S, T, $A_s = A_t = A$, $C_s \neq C_t$, and none of the visible areas overlap. After this first series, the **SP** register of this automaton now contains the current answer to

[2] Computing areas is not a closed operation with rationals, but no algorithm in this paper performs such an operation.

the EASP:

$$C_p = \frac{A}{A_p} \sum_{i=1}^{n} C_i.$$

On the input tape there is now a single record defining a set of points such that the visible area they define hides all but one of previous visible areas, and their color is 0. Note that for the purpose of the proof the color could be any known value, but 0 makes it simpler. This particular coverage is easy to achieve, since none of the previous areas overlap. It also can be done in constant time (consider, for example, each area to be described by four corner points, forming narrow strips vertically across the pixel), even though this is not crucial for the proof. After this last set, the correct answer for the EASP is $C_j \times A/A_p$, if S_j is the set of points defining the only currently visible area. From this the color of that area can be computed easily. But that means that from k registers we can retrieve any of n values ($n > k$), which is impossible in our model. Therefore, the automaton cannot solve the EASP for its pixel. This (regretfully) eliminates the possibility of getting exact antialiasing in a frame buffer. □

It has been shown [14a, 15] that sorting is reducible to the EASP, which obviously means our model is not powerful enough to sort.

A more restricted problem is obtained when we discretize it. All the point records on a particular input tape are assumed to have the same X_p and Y_p ($\lfloor X_p \rfloor$ and $\lfloor Y_p \rfloor$, respectively), which means these values do not have to appear explicitly on the tape. Then the VSP for each automaton is simply "find a visible point on the input tape," that is, one **P** such that $Z_p \leq Z_q$ for all **Q** on the input tape. This can clearly be solved with a bounded number of registers, as is shown later (this is the depth buffer algorithm). We call this version of the VSP the *Discretized Visible Surface Problem* (*DVSP*).

2.3.2 *The Shadow Problem.* Another classic problem in raster graphics is to compute shadows. Let L be a coordinate system in which the light source for the scene is at the origin. We define the shadow predicate for a point, $\mathrm{Sh}_E(\mathbf{P})$, to be true for any coordinate system E if $V_L(\mathbf{P})$ is false. This is the same as saying that any point that is invisible from the point of view of the light source is in shadow. Since the shadow predicate depends only on the light coordinate system, we can henceforth drop the subscript referring to the viewer's coordinate system. If the scene contains several light sources, say, n of them, then we can extend the definition so that $\mathrm{Sh}_{L_i}(\mathbf{P})$ is true if $V_{L_i}(\mathbf{P})$ is false in the light coordinate system L_i of light source i, $0 \leq i < n$.

In the Shadow Problem, we need to find all the visible points in E and for each of them evaluate all predicates Sh_{L_i}. In the following we assume that we can go from the coordinates of **P** in E to the coordinates of **P** in L_i and back in constant time. We do not discuss the problems associated with the perspective transform (see [31]), but they do not affect any conclusion drawn here. It is clear that the Shadow Problem is as difficult as the VSP, since the former includes the latter, but not more difficult in the asymptotic sense, since it can be solved by solving the VSP twice. A good review of the problem and several solutions to it can be found in [11].

Other problems are described as they arise in the next section.

3. HIERARCHY OF THE FRAME BUFFER AND COMPLEXITY RESULTS

We now examine different frame buffers in increasing order of power.

3.1 Frame Buffers with No Predicate or Function (FB_{rb0})

3.1.1 *One-Register Frame Buffers* (FB_{100}). This is the so-called "dumb" frame buffer. The absence of functions or predicates means it can only take one value from the input stream for each record and write it into the register. If the value is the color, it effectively implements the "painter's algorithm" in which the value displayed is the last one received.

We can define a *nondeterministic buffer* (*NFB*) in a way similar to other nondeterministic automata. That means NFB_{100} can choose whether to write the current color in its register or not. It can then solve the discretized version of the VSP. It is, of course, hard to build in practice.

3.1.2 *Frame Buffers with More Than One Register* (FB_{rb0}). The other frame buffers in this group all have the same power in their deterministic version. Since they cannot make decisions or compare values based on the input stream, the additional registers do not help. The nondeterministic versions have the same power as the one-register NFB_{100}.

3.2 One-Register Frame Buffers with Operations

3.2.1 *Frame Buffers Computing Predicates* (FB_{10p}). We now consider a frame buffer that can compute predicates based on the current input record values, register value, and constants, and decide whether to write to its register. Perhaps surprisingly, several graphics problems can be solved with this modest computing power.

Thresholding

/* *black* and *white* are constants. */
if ($C >$ *threshold_value*) **then SP** $\leftarrow$ *white*;
  else SP $\leftarrow$ *black*;

Viewporting

/* *left*, *right*, *bottom*, *top*, and the color make the point record. */
if ($i \geq$ *left* & $i \leq$ *right* & $j \geq$ *bottom* & $j \leq$ *top*) **then SP** $\leftarrow C$;

These algorithms are based only on input values. The next one is based on the stored value:

Chromakeying

/* *key-color* is a constant. */
if (**SP** = *key_color*) **then SP** $\leftarrow C$;

This can be considered as a two-pass algorithm. On the first pass, a picture is written. On the second pass, another picture is written only on the pixels of a given color.[3]

The semantics of these three algorithms is straightforward. It is easy to show that this frame buffer cannot compute the DVSP, since either the color or the Z value can be stored, but not both.

[3] In multipass algorithms we can attach a value to each input record that indicates which pass it is. Equivalently, we can think of the frame buffer as having as many states as there are passes.

3.2.2 *Frame Buffers Computing Functions* (FB_{10f}). With the addition of functions, a one-register frame buffer can compute an image with total transparency.

Total transparency. Along with the usual values, a point record contains a rational value T (the transmission coefficient). **SP** is initialized to the background color. The algorithm is as follows:

SP $\leftarrow T \times$ **SP**;

The assumption is that transparency is commutative.[4] This can be considered a two-pass algorithm if an original background picture is needed.

The weakness of these frame buffers is that with only one register they cannot "remember" more than the color of the points. Much can be gained from one additional register, even a Boolean one.

3.3 One-Register, One-Flag Frame Buffers with Operations

3.3.1 *Frame Buffers Computing Predicates* (FB_{11p}). The presence of an additional Boolean register (a flag) allows the depth buffer algorithm to be solved in two passes.

Depth Buffer Algorithm

/* *color* is a Boolean register initialized to *false*. **SP** is initialized to a background Z value (the largest value the register can hold). */

Pass 1
if ($Z <$ **SP**) **then SP** $\leftarrow Z$;

Pass 2
if ($Z =$ **SP** & *not color*) **then** {
  color $\leftarrow$ *true*;
  SP $\leftarrow C$;
}

THEOREM 2. *This algorithm yields a correct answer to the DVSP with* FB_{11p}.

PROOF. First consider that at the end of the first pass **SP** holds the value of the minimum Z. In the second pass, only the color of the point with the same Z gets written. The color flag indicates whether a Z or a color is currently written and is necessary, since we assume they are both numbers with the same range. □

Note that in this algorithm, if $Z_p = Z_q$ and they are visible, then the point found first is the one whose color is written. Since the definition of the DVSP implies that there could be more than one correct answer, an arbitrary decision of that nature is unavoidable.

3.3.2 *Frame Buffers Computing Functions* (FB_{11f}). We have seen that FB_{10f} can solve the total transparency problem and that FB_{11p} can solve the DVSP. As a result, one might naturally expect that FB_{11f}, which represents the union of the resources of these two frame buffers, would be able to solve the union of these two problems, that is, the DVSP for scene descriptions containing both transparent and opaque objects. Perhaps surprisingly, this is not the case.

[4] Transparency is commutative when convolving transmittance spectra and light spectra if the filters are separated by a common medium, but not necessarily so, as computed in existing display systems.

THEOREM 3. *FB_{11f} cannot solve a combination of the total transparency problem and the DVSP.*

PROOF. Consider an input sequence of point records $S = \mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_n$, where $\mathbf{P}_n = (Z_n, T)$ represents a point on a transparent object, and the previous point records represent a point on opaque objects. Let $\mathbf{P}_i = (Z_i, C)$ be the record containing the visible (i.e., nearest) opaque point in the sequence. Then when $\mathbf{P}_n$ is read by the pixel automaton, it must compute a function $f(C, T)$ to obtain a new color in the event that $Z_i > Z_n$. Otherwise, the old color is retained. Thus, it is necessary to know the values of Z_i, C, Z_n, and T at the same time, which requires at least two rational number registers no matter how many passes are used. □

Convex intersection of half planes. One interesting new problem that this frame buffer (FB) can solve is the *convex intersection of half planes*: The point records XYZ are taken as the coefficients in the equation of an oriented line: $x \times X + y \times Y + Z = 0$. The algorithm is as follows:

/* *inside* is a Boolean register initialized to true. */
if $(i \times X + j \times Y + Z < 0)$ **then** *inside* = *false*;

At the end of the input tape, only the automata (i, j), such that a point of coordinate (i, j) is inside all lines, have *inside* = *true*. This, of course, scan converts a convex polygon given only the line equation of its edges. This is essentially the algorithm used in Pixel-Planes [17]. A second type of record can give the plane equation, which would be used to compute the depth at that pixel (see Section 3.5.2).

Inclusion in a simple polygon. The preceding algorithm computes the inclusion of the pixel in a convex polygon given by its edge equations. This approach does not work for simple polygons (polygons whose boundary edges do not cross) that are not convex. A different approach is required to test for inclusion in any simple polygon.

The point records now contain pairs of points (X_1, Y_1), (X_2, Y_2), which are the vertices of each edge. No particular order is assumed for the edges or for the vertex pairs.

/* Note that the top vertex is considered out and that horizontal edges are skipped. */
/* *inside* is a Boolean register initialized to *false*. */
if $((Y_1 \leq Y_2)$
/* Vertex 2 is the highest */
$\&\ (Y_1 \leq j)\ \&\ (j < Y_2)$
$\&\ (((i - X_1) \times (Y_2 - Y_1) - (j - Y_1) \times (X_2 - X_1)) \geq 0))$ **then** *inside* $\leftarrow \neg$ *inside*;
/* Vertex 1 is the highest. */
else if $((Y_2 \leq j)\ \&\ (j < Y_1)$
$\&\ (((i - X_2) \times (Y_1 - Y_2) - (j - Y_2) \times (X_1 - X_2)) \geq 0))$ **then** *inside* $\leftarrow \neg$ *inside*;
}

The algorithm works by computing the parity of the number of intersections between the edges and a semi-infinite horizontal line to the left of the point (i, j) tested for inclusion. The Boolean *inside* is *false* for an even parity and *true*

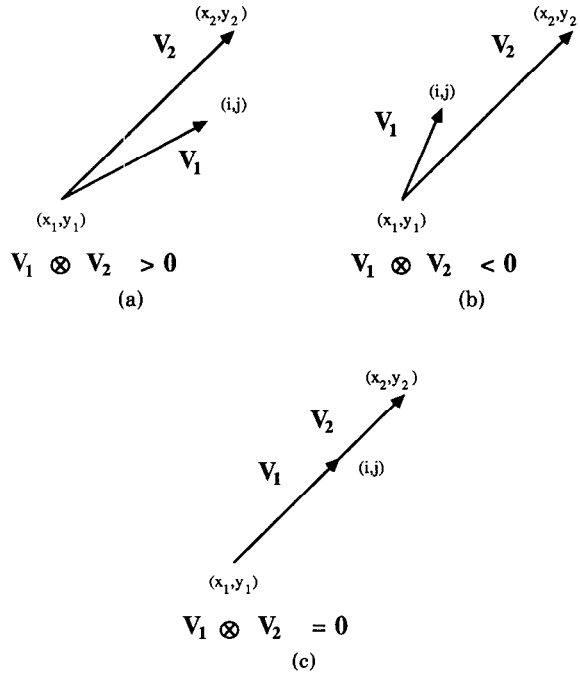

Fig. 2. Testing if a point is to the right of an edge. (a) Test point is to the right of V_2. (b) Test point is to the left of V_2. (c) Test point is on V_2.

for an odd parity. The quantity $((i - X_1) \times (Y_2 - Y_1) - (j - Y_1) \times (X_2 - X_1))$ is the cross product $V_1 \otimes V_2$ (see Figure 2), and is positive if the test point is to the right of the line supporting the edge. So *inside* is negated if the semi-infinite line intersects the edge. Note again that only the semi-open edge segment is considered, so that an intersection at a vertex is only counted once if it goes through the boundary. In the case of horizontal edges no intersection is counted, even if the test point has the same Y coordinate.

The drawback of this algorithm is that points on the boundary can be counted inside or outside depending on their position (left, right, or on a horizontal edge). A consistent determination can be made at the cost of an additional Boolean register that is set *true* when the test point is found to be on the boundary.

The reader should verify that the answer is valid for all simple polygons (i.e., whose edges do not intersect) and also for polygons with holes. This algorithm can be adapted to compute inclusion for self-intersecting polygons, but first the *inside* predicate has to be carefully defined. This would take us too far afield, but the interested reader should consult [26].

It is interesting to note that both the above algorithms are well known in their standard versions [30], and that both have been used in the form of operations on Boolean coverage masks, the first one in [15] and the second in [7].

3.4 Two-Register Frame Buffers with Operations

3.4.1 *Frame Buffers Computing Predicates* (FB_{20p}). This frame buffer can solve the DVSP using the classic depth buffer algorithm. This algorithm was probably first published in [8].

/* **SZ** is initialized to background Z. **SP** is initialized to a background color. */
if ($Z <$ **SZ**) **then** {**SZ** $\leftarrow Z$; **SP** $\leftarrow C$;}

Note that the inequality means that in case of ties the color of the first record of minimum Z is written. A $\leq$ relation changes this to the last.

3.4.2 *Frame Buffers Computing Functions* (FB_{20f}). With this frame buffer, we can now solve the DVSP with transparencies, but only in two passes. The records are as before with the addition of a transmission factor T. $T = 0$ for an opaque polygon.

Pass 1
if ($Z <$ **SZ** & $T = 0$) **then** {**SZ** $\leftarrow Z$; **SP** $\leftarrow C$;}

Pass 2
if ($Z <$ **SZ** & $T > 0$) **then** {**SP** $\leftarrow T \times$ **SP**;}

Two registers are necessary, since both the color and the depth of the closest opaque point have to be known simultaneously for a correct answer.

THEOREM 4. *The DVSP with transparencies cannot be solved in one pass in this model.*

It is important to note that we assume the transparent objects are freely mixed with the opaque ones.

PROOF. Consider Table I, where π is a permutation of (1, 2, 3, ..., n). After record n, the correct value for **SP** is **SP** $= C_0 \times T_1 \times T_2 \times \cdots \times T_n$. After record $n + k$, the correct value should be **SP** $= C_0 \times \Pi\ T_j(j \mid \pi(j) \leq n - k)$ (see Figure 3).

Therefore, the automaton can correctly determine the rank of any of the first $n + 1$ records. But it can be shown that any sorting algorithm using p passes requires $\Theta(N/p)$ storage locations [23]. Thus, no frame buffer with a constant number of registers can solve the transparency problem in one pass. □

3.5 Three-Register Frame Buffers with Operations

3.5.1 *The Convex Intersection of Half Spaces Problem.* The three-dimensional version of the convex intersection of half planes is the *convex intersection of half spaces* (*CIHS*). The problem occurs when objects are defined by planar boundaries, but the edges or vertices of the polyhedrons are not explicitly given. Such situations are encountered in the simulation of crystal growth or the hierarchical definition of solid objects.

The general problem has a lower and upper bound of $\Theta(n \log n)$ [6]. FB_{30f} can compute a discretized display of the convex polyhedron obtained with the following algorithm.

The point records contain a, b, c, and d, the coefficients of the equation for each plane: $aX + bY + cZ + d = 0$. They also contain C, the color of the plane.

Table I. Input Sequence of Point Records

Record	Z	C	T
0	$n+1$	C_0	0
1	$\pi(1)$	—	T_1
2	$\pi(2)$	—	T_2
3	$\pi(3)$	—	T_3
...	...	...	...
n	$\pi(n)$	—	T_n
$n+1$	$n-\epsilon$	C_0	0
$n+2$	$n-1-\epsilon$	C_0	0
...	...	...	...
$2n$	$1-\epsilon$	C_0	0

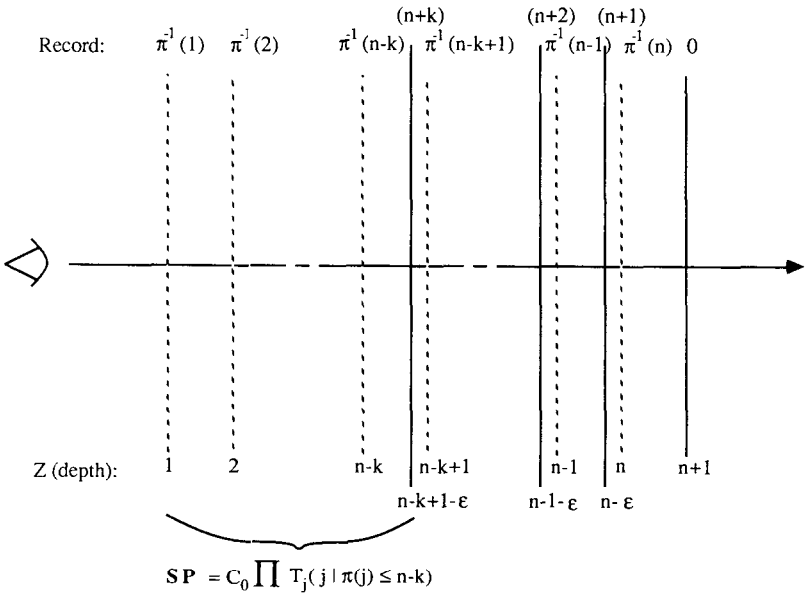

Fig. 3. Visibility after record $n + k$.

An automaton (i, j) corresponds to a line $X_p = i$, $Y_p = j$. The intersection between a plane and this line is the point (X_p, Y_p, Z_p) such that

$$Z_p = -\frac{a}{c} X_p - \frac{b}{c} Y_p - \frac{d}{c}.$$

By convention, a point is inside the half space defined by a plane if $aX + bY + cZ + d \geq 0$. The planes are oriented, and *front facing* if $c > 0$ and *back facing* if $c < 0$. Thus, if a plane is front facing $(c > 0)$, $Z \geq -(a/c)X - (b/c)Y - (d/c)$, and if $(c < 0)$, a back-facing plane, $Z \leq -(a/c)X - (b/c)Y - (d/c)$. If $c = 0$, then the plane is parallel to the viewing direction (Z axis) and may be considered front facing for any point contained within the half space it defines, and back facing otherwise.

A point belongs to the convex intersection iff it is inside all the half spaces. A point on the line $(X = X_p, Y = Y_p)$ contained within the half spaces defined by

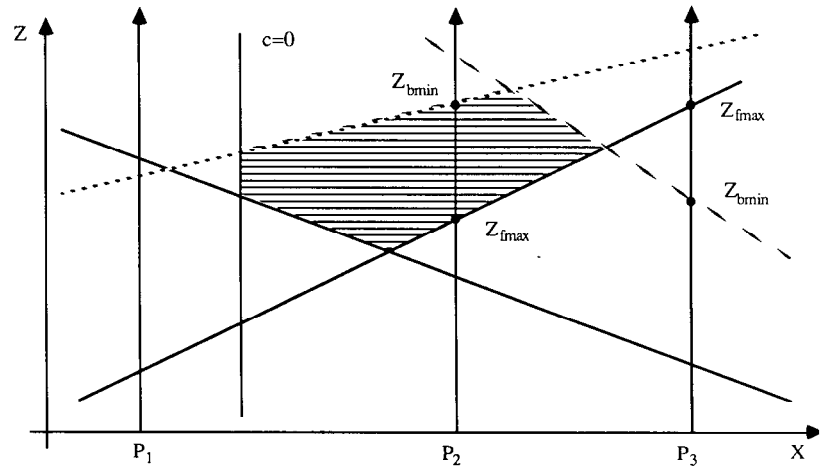

Fig. 4. An example CIHS.

all planes for which $c = 0$, then, belongs to the convex intersection iff its Z is smaller than or equal to the Z of the intersection with the line and all the back-facing planes, and larger than or equal to the Z of the intersection with the line and all the front-facing planes. If we call $Z_{f\max}$ the intersection between the line and the front-facing planes with the largest Z, and $Z_{b\min}$ the intersection between the line and the back-facing planes with the smallest Z, then there is a point of the line within the convex intersection only if $Z_{b\min} \geq Z_{f\max}$, and the visible point of this intersection is at $Z_{f\max}$. Figure 4 shows a two-dimensional cross section of a convex volume bounded by two front-facing planes (solid lines), two back-facing planes (dashed lines), and a single plane for which $c = 0$. In the figure, P_1 is outside the volume because it is outside the half space defined by the $c = 0$ plane. P_2 is within the projection of the volume because the farthest front-facing plane at that x, y position is closer than the nearest back-facing plane. P_3 is outside the volume because the farthest front-facing plane is farther than the nearest back-facing plane at that position.

It is assumed that clipping has taken place, so that we need not check whether the eye is actually inside the convex intersection. Clipping can be made part of the computation by adding the front clipping plane (the *hither* plane) as a front-facing plane, and the back clipping plane (the *yon* plane) as a back-facing plane.

The registers are $\mathbf{SZ}_f$, initialized to 0; $\mathbf{SZ}_b$ initialized to the background Z, $Z_{\text{background}}$, which is assumed to be greater than 0; and **SP**, initialized to the background color $C_{\text{background}}$.

```
if (SZ_f ≤ SZ_b) then {
  if (c = 0 & aX_p + bY_p < −d) then {SZ_f ← Z_background; SP ← C_background}
  else {
    Z_p = −(a/c)X_p − (b/c)Y_p − (d/c);
    if (back facing & (Z_p ≤ SZ_b)) then SZ_b ← Z_p;
    else if (front facing & (Z_p > SZ_f)) then {SZ_f ← Z_p; SP ← C;}
  }
  if (SZ_f > SZ_b) then SP ← C_background;
}
```

THEOREM 5. *At the end of the execution of the CIHS algorithm,* **SP** *contains the background color if the convex intersection is not visible at that pixel, and the color of the farthest front-facing plane if it is.*

PROOF. Initially $\mathbf{SZ}_f < \mathbf{SZ}_b$. During the execution of the algorithm, $\mathbf{SZ}_f$ can only increase, since it is assigned either $Z_{\text{background}}$ or a Z_p greater than itself. On the other hand, $\mathbf{SZ}_b$ cannot increase. Therefore, if at any point in the algorithm $\mathbf{SZ}_f > \mathbf{SZ}_b$, this will remain true until the end, and either at least one back-facing plane is in front of one front-facing plane at this pixel, or the pixel is outside of a plane for which $c = 0$. In both cases the convex intersection is not visible at this pixel, and the last assignment ensures that **SP** gets the background color. If this never happens, **SP** contains the color of the front-facing plane with the largest Z because of the previous conditional assignments. □

THEOREM 6. *At least three registers are necessary to compute the convex intersection of half spaces.*

PROOF. Obviously, one register is required to store the color. Now assume an input sequence for which just before the last plane in the input is examined $\mathbf{SZ}_f < \mathbf{SZ}_b$, and let the final plane be front facing. On consideration of this final plane, there are three distinct possibilities for the final color: the background color, the color of the current visible plane, or the color of the new one. Thus, at least two comparisons must be made in processing this plane to decide the result. These comparisons require two operands other than the input, and thus with the color a total of three registers are required. □

Note that one of the registers involved in the comparisons can be Boolean, and so FB_{21f} can compute the answer to this problem, using one extra pass and a trick similar to the one used with FB_{11p} for the depth buffer algorithm. We leave the derivation of the algorithm to the reader.

3.5.2 *Frame Buffer Shadow Algorithm.* In [11] three approaches to the rendering of polygonal scenes with shadows are discussed. Of these approaches, the solution in object space is not applicable to our model, and the second, which consists of solving the VSP twice, cannot be used since we forbid communications between automata (pixels). An algorithm like that given in [32], using two frame buffers and a mapping between them, also does not fit in our framework. Crow's third technique, using shadow volumes, is applicable in general. To give an explicit algorithm, we must make some assumptions about the type of information available on the input tapes. The reader should keep in mind that this is only one of the possibilities.

We specialize the objects to surfaces described as polygons. If a polygon is given as a list of vertices $(\mathbf{P}_0, \mathbf{P}_1, \ldots, \mathbf{P}_{n-1})$, a point is in the shadow of this polygon only if it is inside the volume determined by this polygon and the "shadow polygons" formed by each edge $(\mathbf{P}_i, \mathbf{P}_{i+1})$ and the semi-infinite lines defined by the points $(\mathbf{L}, \mathbf{P}_i)$ and $(\mathbf{L}, \mathbf{P}_{i+1})$, and that do not contain **L** (**L** is the position of the light). The planes of support of these shadow polygons can be oriented so that the inside is consistently defined. It can be seen that, once the shadow polygons have been computed, the shadow predicate can be computed in any coordinate system that preserves the ordering of points along a line.

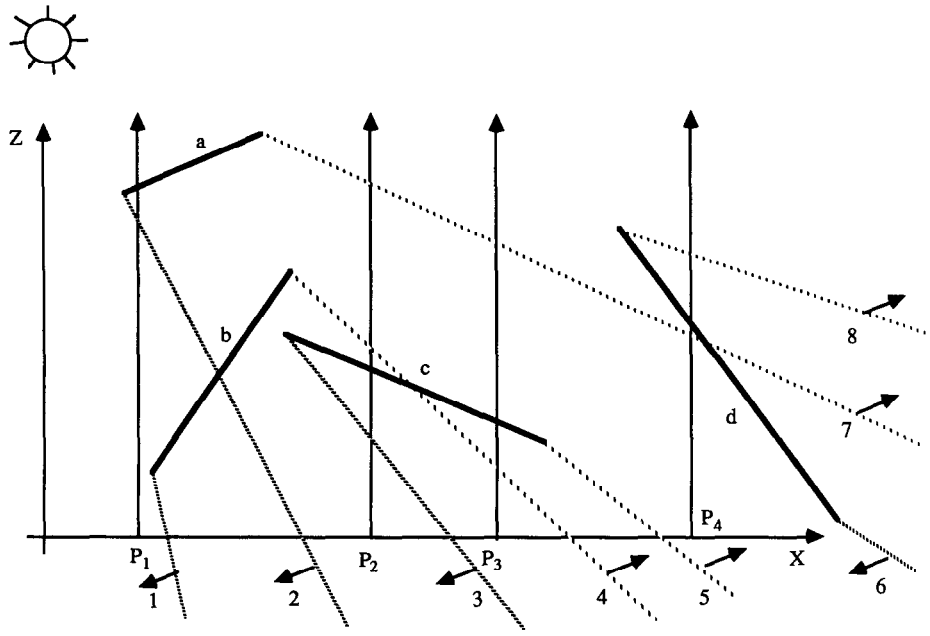

Fig. 5. Shadow computation using shadow polygons.

The shadow polygons are transformed then to the E (eye) coordinate system and tagged as front facing if the origin E is outside their plane of support, or back facing if E is inside their plane of support. Next they are scanned out, and among the point records for each automaton are the point records defining the shadow polygons, identified as such by a Boolean and tagged as front facing or back facing. In order to solve the problem, it must be discretized for the same reasons we had to discretize the VSP. If we follow the line $X_p = i$, $Y_p = j$ from E, we enter a shadow every time we cross a front-facing shadow plane and leave a shadow every time we cross a back-facing shadow plane. If a visible point is at Z_p, the count of front-facing shadow polygons minus the count of back-facing shadow polygons in front of it gives the number of polygons shadowing it. Figure 5 gives a two-dimensional example of a set of polygons (solid lines) and the shadow polygons (dashed lines) they would generate from the given light source. The orientations of the shadow polygons are indicated by their normals, so 1, 2, 3, and 6 are front facing, and 4, 5, 7, and 8 are back facing. P_1 has a shadow depth of 1, since a is the visible polygon at that position and it shadows itself. Only shadow polygon 2 is in front of a at P_1, so the count at P_1 is 1. P_2 is at shadow depth 3, because the visible polygon there is c and it is shadowed by a, b, and c itself. The count at P_2 is also 3, since only shadow polygons 1, 2, and 3 are in front of polygon c at P_2. P_3 is at shadow depth 2, since the visible polygon is c and it is shadowed by a and c. The count at P_3 is 2, since it is the count at P_2 minus 1 for the back-facing shadow polygon 4 that is not in front of c at P_2. Finally, P_4 has shadow depth 0, since the polygon d visible there has nothing between the light source and itself, and the count is also 0 since front-facing

shadow polygons 1, 2, and 3, and back-facing shadow polygons 4, 5, and 7 are all in front of polygon d at P_1.

It is obvious that only shadow polygons matter for this count, since by definition the visible point does not have any other polygon in front of it. It is less obvious that the order in which the shadow points appear in front of the visible point is immaterial.

THEOREM 7. *The shadow depth is computed correctly independently of the order in which the shadow polygons are processed.*

PROOF. We represent the front-facing shadow points as opening parentheses, and back-facing shadow points as closing parentheses. The shadow depth is the depth of nesting of a point in the parentheses. For example, **P** below is at depth 3:

$$0(1(2(3)2(3(4)3\mathbf{P}(4)3)2)1)0.$$

We can exchange any adjacent pair of parentheses in front of **P** without affecting the nesting depth. If the two parentheses are of the same type, nothing is changed. If they are of the opposite type, the situation is as follows:

Depth before switch

$$k(k+1)k \quad \text{or} \quad k)k-1(k.$$

Depth after switch

$$k)k-1(k \quad \text{or} \quad k(k+1)k.$$

In any case, the depth behind the switched pair is the same. If any two adjacent pairs can be switched, then any permutation of the parentheses in front of **P** can be achieved without affecting the depth of **P**. Therefore, we can process the shadow points in front of **P** in any order. □

We now can give the two-pass algorithm used with FB_{30f} to compute the answer to the discretized Shadow Problem. The point records consist of Z, *Boolean shadow*, *Boolean front*, and C. **SP** is initialized to background color, **SZ** to background Z, and **CNT** to 0.

Pass 1
/* Compute visibility. */
if (*not shadow* & ($Z <$ **SZ**)) **then** {**SP** ← C; **SZ** ← Z;}

Pass 2
/* Compute shadows. */
if (*shadow* & ($Z <$ **SZ**)) **then if** (*front*) **then CNT** ← **CNT** + 1;
  else CNT ← **CNT** − 1;

The previous discussion presents the main facts necessary to establish the correctness of the algorithm. A third pass is necessary in practice to modify the color of the points in shadow. The color can be made a function of the shadow depth.

THEOREM 8. *The answer to the Shadow Problem cannot be computed in one pass in our model.*

PROOF. Assume a one-pass algorithm is used. The automaton receives an input tape consisting of $2n$ records, describing n pairs of front-/back-facing shadow points, none of the pairs overlapping in Z, and followed by a record describing an ordinary visible point. To determine whether this last point is in shadow cannot be done with less than $O(n)$ storage, since it means determining inclusion in one of n intervals. Therefore a one-pass algorithm would require more than constant storage and hence is impossible. □

THEOREM 9. *A Boolean register is not sufficient to compute the shadow predicate in our model in a constant number of passes over the input tape.*

PROOF. Assume an input tape with one ordinary point record, followed by n back-facing shadow points, and n front-facing shadow points, each one of decreasing depth (i.e., in front of each other). For any constant number of passes p, at least one pass processes at least $O(n)$ back-facing points in the order given, and at least one pass processes at least $O(n)$ front-facing points in the order given. So there is a stage in the processing where $O(n)$ back-facing (alternatively, front-facing) points have been processed, and $O(n)$ front-facing (back-facing) points remain to be processed. From this state the automaton has to be able to "count to n," that is, differentiate between the $O(n)$ possible states, just to establish whether or not the ordinary point is in shadow. □

Note that this means that an automaton with bounded size registers can fail, but since the size limitation is $O(\log n)$, not $O(n)$, this is not too serious in practice. If the polygons casting shadows are all convex, the shadow volume is the convex intersection of the half spaces defined by the shadow planes. If, in addition, it is assumed that the shadow planes from each polygon are grouped and separated from each other, and we only compute a global shadow predicate, not the shadow depth, then an algorithm similar to the CIHS algorithm given above is applicable. This is what is described in [18] for pixel-planes. Closer to the algorithm presented here is the one described in [5]. There, an actual count of the shadow depth is kept. The main differences are that the algorithm is designed for several light sources, and the objects are assumed to be convex. In their discussion the authors describe possible simplifications that bring their algorithm closer to the one in [18].

3.6 Frame Buffers with More Than Three Registers

The power of the frame buffer clearly continues to increase as the number of registers increases, and it tends toward the power of the RAM, at which point it escapes both the specificity and the practicality of computer graphics. We mention only two other interesting possibilities, still well within practical limits.

3.6.1 *Visible Surface Determination of Convex Intersections of Half Spaces.* We saw in Section 3.5.1 that FB_{30f} can compute the visible surfaces of convex polyhedrons defined by half spaces. FB_{50f} can compute the visible surfaces in a scene made of several such objects if all the defining planes for each polyhedron are grouped on the input tape. When the list of polyhedron planes is over and the farthest front-facing plane has been determined, then its color and Z value

are kept iff it is the closest such plane so far. This is simply a combination of the depth buffer algorithm and the CIHS algorithm.

The point records are as in Section 3.5.1, except for an additional Boolean value *lastplane*, *true* if it is the last plane equation for a polyhedron. The two new registers are $\mathbf{SZ}_c$, the Z value of the current closest visible plane, and $\mathbf{C}_f$, the color of the current farthest front-facing plane for the current polyhedron. These are initialized to the background Z and background color, respectively. Other variables are initialized just as in the CIHS algorithm.

```
if (lastplane) then {
  if (SZ_f < SZ_c) then {SP ← C_f; SZ_c ← SZ_f;}
  SZ_f ← 0; SZ_b ← Z_background; C_f ← C_background;
}
else if (SZ_f ≤ SZ_b) then {
  if (c = 0 & aX_p + bY_p < −d) then SZ_f ← Z_background;
  else {
    Z_p = −(a/c)X_p − (b/c)Y_p − (d/c);
    if (back facing & (Z_p ≤ SZ_b)) then SZ_b ← Z_p;
    else if (front facing & (Z_p > SZ_f)) then {SZ_f ← Z_p; SP ← C;}
  }
  if (SZ_f > SZ_b) then C_f ← background_color;
}
```

FB_{41f} can be used to do this computation in two passes by combining the CIHS algorithm with the two-pass depth buffer algorithm described with FB_{11p}. The development of this algorithm is left as an exercise for the reader. Note that we have not proved five registers are necessary for this algorithm.

3.6.2 *Scan Conversion with Visible Surfaces, Shadows, and Transparencies.* We have seen how to compute visible surfaces with shadows and with transparencies separately. To be able to compute all of them together, we need to assume that all the line equations for a polygon are grouped together with the plane equation to compute the depth at the pixel.

The records contain XYZ, the coefficients of the oriented line equations of the polygon boundaries, the color C, the transparency T, and the Booleans *opaque*, *transparent*, *shadow*, *front*, and *last*. A variant record, indicated by *last* = *true*, contains the plane equation coefficients a, b, c, and d, instead of XYZ (note that the plane equation can be computed from the edge equations, but we include it here for clarity). The registers are **SP**, **SZ**, **CNT**, and Boolean *inside*. *inside* is initialized to *true*, **SP** to the background color, **SZ** to the background depth, and **CNT** to 0. The frame buffer is then FB_{31f}.

```
Pass 1
/* Depth buffer algorithm for the opaque polygons. */
if (opaque) then {
  if (last) then {
    Z_p ← −(a/c)X_p − (b/c)Y_p − (d/c);
    if (inside & (Z_p < SZ)) then {SZ ← Z_p; SP ← C;}
    inside ← true;
  }
  else if (X_pX + Y_pY + Z < 0) then inside ← false;
}
```

```
Pass 2
/* Process shadow polygons. */
if (shadow) then {
  if (last) then {
    Z_p ← −(a/c)X_p − (b/c)Y_p − (d/c);
    if (inside & (Z_p < SZ)) then
      if (front) then CNT ← CNT + 1;
        else CNT ← CNT − 1;
    inside ← true;
  }
  else if (X_pX + Y_pY + Z < 0) then inside ← false;
}
/* Process transparent polygons. */
if (transparent) then {
  if (last) then {
    Z_p ← −(a/c)X_p − (b/c)Y_p − (d/c);
    if (inside & (Z_p < SZ)) then {SP ← T × SP;}
    inside ← true;
  }
  else if (X_pX + Y_pY + Z < 0) then inside ← false;
}
```

It is immediate that two passes are required since the separate computation of visible surfaces and either shadows or translucent polygons requires two passes. Note that we assume that there is no shadow on the transparent polygons, and that the transparent polygons either cast no shadow or cast a shadow similar to the shadow of the opaque polygons. The system described in [5] implements similar capabilities with more complex algorithms.

4. PRACTICAL IMPLICATIONS OF THE RESULTS

As we have noted above, our abstract model of a frame buffer was not chosen merely because it allows a simple complexity model to be developed. Indeed, if the model did not appropriately reflect the constraints of real implementations of frame buffer algorithms, it would be a rather futile exercise to do this analysis. We have chosen a complexity model in which we consider the number of registers required per pixel. This measure of storage complexity has been chosen precisely because it reflects the complexity of the hardware required to realize a given algorithm. Normally, frame buffers are measured in terms of both performance and cost, according to the number of bits per pixel that they contain. Very often, frame buffers are modularly expandable by the addition of more memory planes, allowing an increased number of bits per pixel. The measure of storage complexity as the number of registers required for a pixel directly reflects this real measure of the cost of a frame buffer. Currently, the most common reasons to design a frame buffer with a large number of bits per pixel is to allow double buffering, to provide an *opacity* channel, or to provide a depth buffer. Our work indicates to owners of raster systems that there is a great deal more that they can do with the extra memory, and it further motivates increasing the number of bits per pixel in systems to provide the power to implement more powerful frame buffer algorithms. As our results show, in many cases there is a space–time trade-off

for these algorithms. The more bits per pixel in a system, the fewer the passes required to implement many algorithms.

The operations of compositing as formalized in [13] and [29] can be directly related to some of the algorithms discussed here. Their properties, in particular, *transitivity*, *commutativity*, and *associativity*, are needed to establish theorems about the correctness of the algorithms. For example, the question of whether the **comp** operation in [13] is correct regardless of order is almost the same as deciding whether FB_{11f} can compute transparency and the DVSP. These observations lead to interesting conclusions: One is that some of the multipass algorithms described here can be couched in terms of multipass compositing operations. The other is that FB_{rbc} can be seen as a stored image with pixels made of $r + b$ fields. This suggests that there is a need to extend even further the number of pixel fields commonly stored. In [24] this is done by defining a large number of *appearance elements* potentially attached to each pixel and by using a modular approach to build graphs to implement the operations on these elements. It is also important not to let ourselves be restricted to unary or binary compositing operations just because the current tools make them easier.

The justification for separating Boolean-valued registers from numeric registers should also be quite clear in the context of actual implementations. A Boolean register requires only a single bit to implement, whereas a numeric register would typically require at least 16 bits. Thus, it would be far less realistic to equate the two types of register than to make the distinction that we presently do. Not only is the distinction a functional one, it is also clearly one of cost. Of course, it might be argued that one should, by the same token, not equate all numeric registers, since in fact it may be appropriate to allocate different numbers of bits to registers used for different functions. Although this is quite likely to be the case, it is by no means clear that there is an inherent number of bits appropriate to a given type of register, whereas the distinction between 1-bit Boolean registers and others is quite a fundamental one.

The reader might feel that there is some arbitrariness in not treating the various initialization steps as additional passes. Another related question is the need for some process to access all the **SP** registers to actually display the picture. In current systems these actions can be performed in hardware and are fairly distinct from the scan conversion operations. The same kind of investigation, however, can be directed at these operations. A recent paper by Booth, Forsey, and Paeth [3] does exactly that, and shows that extra registers (mostly Boolean) can help make these operations concurrent with the ones discussed here. The authors' aim in this paper is to promote hardware solutions, but the methods we use here can be applied to investigate the complexity of solutions like theirs.

From the point of view of system architecture, each of the algorithms described has to be reexamined to decide on the amount of locality of the operations and the possibilities of parallelism. In our model we do not specify where the finite-state control for each automaton resides. In a traditional system, it is a single graphics processor that computes for each pixel in turn. Therefore, there is only one copy for the resources needed to implement the algorithm. At the other extreme, each automaton can have all the hardware necessary for the algorithm. But then the only real difference between the register part of the frame buffer

storage and the registers used during the computations is that the latter are not assumed to retain their values from record to record. In most cases a distinction should be made between the constants and the quantities independent of the pixel position on one hand, and the quantities that depend on the pixel position on the other. In the former case these quantities can be stored and/or computed globally, to be broadcast in parallel to each pixel; in the latter case a compromise has to be found between more local hardware or less parallelism. This has been approached in a very ingenious way in the Pixel-Planes system [17, 18]. This system incorporates the processing power necessary to compute linear functions of two variables simultaneously for all addresses in the frame buffer, so the necessary functions are neither global nor local, but distributed through a tree of simple bit operations. This system in its current form is capable of computing visible surfaces, determining the region contained within a polygon's boundary incrementally using the convex intersection of half planes algorithm we have described, calculating shadows using the CIHS algorithm combined with the shadow algorithm, and rendering spheres. Thus our model and this research promise to be as appropriate for investigating the characteristics of custom hardware most useful in the design of the graphics systems of the future as we think it is for guiding users of existing frame buffer systems in extending the use of their hardware.

5. CONCLUSION

In this paper we take an abstract look at the power of a frame buffer for performing graphics algorithms, and at the complexity of some of these algorithms in the frame buffer model of computation. We argue that the frame buffer as a model of computation for graphics algorithms makes sense in view of the insights it provides into the design of hardware for existing and future raster display systems. In addition, we show lower bounds for many of the most important problems we have considered in the model, in addition to designing algorithms for these problems whose complexity meets the lower bounds. As a result, we show that many straightforward algorithms for solving a number of problems in computer graphics are the best algorithms one can hope to design in the sense of asymptotic time–space complexity. Moreover, for some of these problems, we show a simple time–space trade-off for their solution. These results provide new motivation for the use of frame buffers with the capability of doing more than simply storing a digital representation of a scene to be displayed on a raster monitor. In many cases, having an extra number of bits per pixel can provide a system with a natural facility for performing a number of useful image space algorithms besides the well-known depth buffer algorithm. The future design of high-performance systems based on custom VLSI hardware can also benefit from the results of this study, since we have seen that a smart frame buffer can potentially be used for more than just fast scan out, shading, and visible surface calculations.

Two important aspects of real frame buffer systems have been omitted from our abstract model. We have assumed that one register in each pixel of the frame buffer directly represents the intensity of a dot on the display screen, without

considering the possibility that this value may simply lie in the domain of a function that calculates the intensity of the dot. This capability is implemented in real systems in the form of video lookup tables (VLTs), in which the function is implemented directly as a table lookup. VLTs provide a powerful additional capability for frame buffers, which can be exploited to achieve some nontrivial and nonobvious functions such as animation, chromakey, and thresholding. Since the latter functions are also capabilities that can be implemented in our model of a frame buffer without VLTs, it is clear that the capabilities provided by a VLT overlap those of a frame buffer that lacks this output function. Just as the use of frame buffers has previously been the province of graphics folklore, in which a number of useful algorithmic techniques were well known and used extensively in the computer graphics community without having been formally studied, so it is with VLTs now. We feel that a careful look at the capabilities of VLTs in the spirit of the present work will be of value in understanding how to fully utilize them.

The other aspect we do not model, being specifically excluded by the formal model, is that some operations, in software as well as in hardware, require communications between pixels. Among them are most bit-map manipulations [21], some algorithms for shadow [32], transparency and refraction [22], rotation in the frame buffer [4, 9], and many image processing operations. The two most popular questions at computer graphics conferences are, "What is the resolution?" and "What about antialiasing?" Indeed, they are real concerns and are the main problems in raster graphics. We discuss within-pixel antialiasing in Section 2.3.1, and it is obvious that the algorithms discussed all apply to any supersampling scheme. We also mention the connection of some algorithms with the coverage-masks schemes [7, 15]. Filters that cover more than one pixel can also be accommodated by these types of algorithms, but details on this will have to await further work.

Integrating all these raster-level operations into the conceptual framework of computer graphics is a considerable challenge. Work has been done in that direction for bit-map operations [21, 28], rasters and VLTs [1], and most rendering [14], but it is still a new and fertile area of research.

ACKNOWLEDGMENTS

Computer graphics researchers have never been accused of being too rigorous or formal. In this attempt to break the tradition, we benefited from advice and pointers by A. Borodin, E. Fiume, and J. Amanatides. We are also indebted to Dan Bergeron and the anonymous referees for their diligence, which has led to significant improvements in this paper. Xerox PARC provided support and shelter to the first author during the latter part of this work, and he is grateful.

REFERENCES

1. ACQUAH, J., FOLEY, R., SIBERT, J., AND WENNER, P. A conceptual model of raster graphics systems. *Comput. Graph. 16*, 3 (July 1982), 321–328.
2. AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1976.

3. BOOTH, K. S., FORSEY, D. R., AND PAETH, A. W. Hardware assistance for *Z*-buffer visible surface algorithm. In *Proceedings of Graphics Interface '86* (Vancouver, B.C., Canada, May 26–30). 1986, pp. 194–201.
4. BRACCINI, C., AND MARINO, G. Fast geometrical manipulations of digital images. *Comput. Graph. Image Process. 13* (1980), 127–141.
5. BROTMAN, L. S., AND BADLER, N. I. Generating soft shadows with a depth buffer algorithm. *IEEE Comput. Graph. Appl. 5*, 12 (Oct. 1984), 5–12.
6. BROWN, K. Q. Geometric transforms for fast geometric algorithms. Ph.D. dissertation, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., 1979.
7. CARPENTER, L. The *A*-buffer: An antialiased hidden surface method. *Comput. Graph. 18*, 3 (July 1984), 103–109.
8. CATMULL, E. A subdivision algorithm for computer display of curved surfaces. Ph.D. dissertation, Dept. of Computer Science, Univ. of Utah, Salt Lake City, 1974.
9. CATMULL, E., AND SMITH, A. R. 3-D transformation of images in scanline order. *Comput. Graph. 14*, 3 (July 1980), 279–285.
10. CLARK, J. H., AND HANNAH, M. R. Distributed processing in a high performance smart image memory. *Lambda 1*, 3 (1980), 40–45.
11. CROW, F. C. Shadow algorithms for computer graphics. *Comput. Graph. 11*, 2 (Aug. 1977), 242–248.
12. DOBKIN, D. P., AND MUNRO, J. I. A minimal space selection algorithm that runs in linear time. Res. Rep. 106, Dept. of Computer Science, Yale Univ., New Haven, Conn., 1978.
13. DUFF, T. Compositing 3-D rendered images. *Comput. Graph. 19*, 3 (July 1985), 41–44.
14. FIUME, E. Mathematical theory and semantics of computer graphics. Ph.D. dissertation, Dept. of Computer Science, Univ. of Toronto, Toronto, Ontario, May 1986.

14a. FIUME, E., AND FOURNIER, A. The visible surface problem under abstract graphic models. *Theoretical Foundations of Computer Graphics and CAD*, R. A. Earnshaw, Ed. Nato ASI Series, Series F., vol. 40. Springer-Verlag, Berlin-Heidelberg, 1988.

15. FIUME, E., FOURNIER, A., AND RUDOLPH, L. A parallel scan conversion algorithm with anti-aliasing for a general-purpose ultracomputer. *Comput. Graph. 17*, 3 (July 1983), 141–150.
16. FOURNIER, A., AND GRINDAL, D. A. The stochastic modelling of trees. In *Proceedings of Graphics Interface '86* (Vancouver, B.C., Canada, May 26–30). 1986, pp. 164–172.
17. FUCHS, H., POULTON, J., PAETH, A., AND BELL, A. Developing pixel-planes, a smart memory-based raster graphics system. In *Proceedings, Conference on Advanced Research in VLSI* (Cambridge, Mass., Jan. 25–27). MIT, Cambridge, Mass., 1982, pp. 137–146.
18. FUCHS, H., GOLDFEATHER, J., HULTQUIST, J. P., SPACH, S., AUSTIN, J. D., BROOKS, F. B., EYLES, J. G., AND POULTON, J. Fast spheres, shadows, textures, transparencies, and image enhancements in pixel-planes. *Comput. Graph. 19*, 3 (July 1985), 111–120.
19. FUSSELL, D. S., AND RATHI, D. P. A VLSI-oriented architecture for real-time raster display of shaded polygons. In *Proceedings, Graphics Interface '82* (Toronto, Ont., Canada, May 17–21). 1982, pp. 373–380.
20. GILOI, W. K. *Interactive Computer Graphics*. Prentice-Hall, Englewood Cliffs, N.J., 1978.
21. GUIBAS, L. J., AND STOLFI, J. A language for bitmap manipulation. *ACM Trans. Graph. 1*, 3 (July 1982), 191–214.
22. KAY, D., AND GREENBERG, D. Transparency for computer synthesized images. *Comput. Graph. 13*, 2 (Aug. 1979), 158–164.
23. MUNRO, J. I., AND PATERSON, M. S. Selection and sorting with limited storage. Comput. Rep. 24, Dept. of Computer Science, Univ. of Warwick, Warwick, Coventry, England, 1980.
24. NADAS, T. The computation of appearance elements. M.S. thesis, Dept. of Electrical Engineering, Univ. of Toronto, Toronto, Ont., Canada, 1986.
25. NAYLOR, B. F. A priori based techniques for determining visibility priority for 3-D scenes. Ph.D. dissertation, Program in Mathematical Sciences, Univ. of Texas, Dallas, 1981.
26. NEWELL, M. E., AND SEQUIN, C. H. The inside story on self-intersecting polygons. *Lambda 1*, 2 (1980), 20–24.
27. PAETH, A. W., AND BOOTH, K. S. Design and experience with a generalized raster toolkit. In *Proceedings of Graphics Interface '86* (Vancouver, B.C., Canada, May 26–30). 1986, pp. 91–97.
28. PIKE, R. Graphics in overlapping bitmap layers. *Comput. Graph. 17*, 3 (July 1983), 331–356.

29. PORTER, T., AND DUFF, T. Compositing digital images. *Comput. Graph. 18*, 3 (July 1984), 253–259.
30. PREPARATA, F. P., AND SHAMOS, M. I. *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985.
31. RIESENFELD, R. F. Homogeneous coordinates and projective planes in computer graphics. *IEEE Comput. Graph. Appl. 1*, 1 (Jan. 1981), 50–55.
32. WILLIAMS, L. Casting curved shadows on curved surfaces. *Comput. Graph. 12*, 3 (Aug. 1978), 270–274.

Received September 1984; revised January and August 1987; accepted October 1987