A Device Independent Graphics Imaging Model for Use with Raster Devices

John Warnock and Douglas K. Wyatt

Xerox Palo Alto Research Centers
3333 Coyote Hill Road
Palo Alto, CA 94304

Abstract

In building graphic systems for use with raster devices, it is difficult to develop an intuitive, device independent model of the imaging process, and to preserve that model over a variety of device implementations. This paper describes an imaging model and an associated implementation strategy that:

1. Integrates scanned images, text, and synthetically generated graphics into a uniform device independent metaphor;

2. Isolates the device dependent portions of the implementation to a small set of primitives, thereby minimizing the implementation cost for additional devices;

3. Has been implemented for binary, grey-scale, and full color raster display systems, and for high resolution black and white printers and color raster printers.

Introduction

The work described in this paper is designed for a multi-application programming environment where programmers use raster display devices to provide the visual communication links between users and systems. The displays are used for simple typescript-style text applications as well as for more involved applications requiring drawings, scanned images, and other complex combinations of graphics and text: a music composition system, a general window management package for a programming environment, a high quality document design system, a VLSI design system, and a graphics arts design package.

In an environment that supports such diverse graphic user interfaces on a variety of display devices, it is desirable to maintain a flexible unified graphics imaging model and an associated programming interface, independent of display devices, with which all the application programs can work.

This paper describes an imaging model and its programming interfaces, discusses the advantages in using such a model, and outlines a basic implementation strategy.

Raster Devices

The class of raster devices encompasses a wide range of displays, plotters, and printers. These include full color (24 bit per pixel) displays, grey level displays, simple low resolution binary (1 bit per pixel) displays, electrostatic plotters, high resolution film recorders, and laser printers. Raster devices, because of their potential ability to display a rich set of images, serve as a useful class over which to define a device independent programming abstraction.

Broadening the class of raster devices to include other kinds of graphic devices (vector drawing displays, pen plotters, or storage tube displays) can lead to problems in defining the imaging abstractions. Either the image types become restricted or the imaging metaphors become strained and unnatural. For example, the implementation of solid areas on some vector drawing devices is impractical. Restricting the set of devices to raster devices allows the imaging metaphor to remain simple, consistent, and efficient.

Device Independence

Device independence, in the context of using raster devices, can be defined in several ways depending on the level of abstraction desired.

One definition dictates that the imaging model provides an abstraction of how an image ideally looks on a perfect medium; this model abstracts the *appearance* of the image. The implementation for each specific display must mimic the appearance of an ideal image to the best of its ability. This kind of device independence attempts to maintain global image properties in spite of wide variations in display type. For example, a device that can show grey values might render the appearance of color values by substituting appropriate grey values. A binary device might render colors with stipple patterns that give a visual impression of grey values.

Other kinds of device independent abstractions can be less strict. Image representations may not model image appearance but instead describe some form of information content. In this case, the implementation of a given device is only required to convey certain information content of the application, and may not be constrained in any way to the appearance of the image. For example, a black and white device implementation might choose to display colors with iconic labels rather than intensity levels. With this kind of abstraction, few constraints are put onto particular device implementations. One consequence of this latter kind of abstraction is that the programmer cannot have precise expectations of how a device implementation will attempt to represent images.

Device independence benefits the implementors of a graphics system as well as its clients, since the bulk of the system can be shared across all devices. Only a small portion of the code need be concerned with a specific device type. If the interface to this device-specific code is well designed, implementing a new device type requires minimal programming effort.

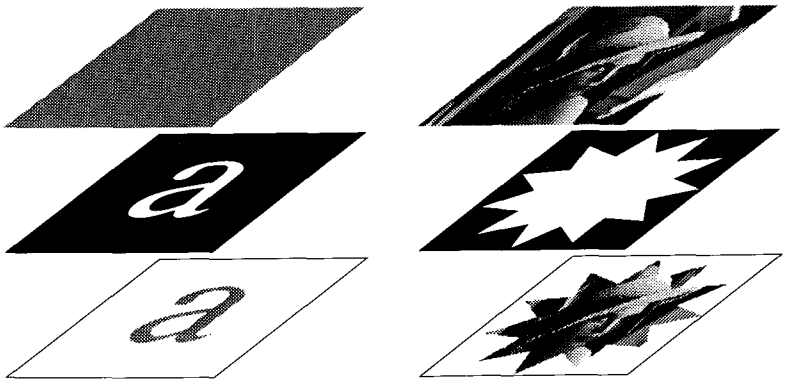

Figure 1: The imaging model.

The Imaging Model

The imaging model described here is designed for applications related to the typesetting and graphics arts industry, where image appearance is vital. For this reason the model abstracts the geometric and color properties of an image. In taking this approach, the imaging model makes two value judgements: first, that global image fidelity is important; second, that it is valuable for the application to be able to rely on a device implementation to render images as accurately as possible. It should be noted at the outset that for a number of applications this choice may be inappropriate.

The imaging model specifies how geometric shapes and colors are combined. It follows a metaphor that loosely corresponds to the procedure used by a silk-screen printer: pushing colored ink through a stencil onto paper. The left side of Figure 1 illustrates this operation. The ink, above, is solid gray; the screen, in the middle, has an a-shaped opening; on the paper, below, the result is an a-shaped patch of gray ink. The artist can build a complex image by repeating this basic operation with different combinations of screens and inks. Ink laid down later may obscure ink laid down earlier.

The programming interface presents a similar model. The programmer calls a series of procedures to define a *stencil*, and other procedures to define a *source*. Each primitive display procedure produces on the display the effect of pushing a given source through a given stencil. The programmer can build a complex image by calling a sequence of display primitives with different combinations of stencils and sources.

Stencils may be represented in two forms: *shapes* and *masks*. A shape consists of a collection of closed piecewise analytic curves (straight lines and parametric cubics); these curves represent the outlines of holes in the stencil. A mask consists of a binary two dimensional array; "ones" in the array represent holes in the stencil. There is no special representation for text. Characters are just letter-shaped stencils, which can be represented either as closed analytic outlines or as masks.

Sources (inks) may be represented either as single colors or as multi-colored two dimensional sampled images. When the source is multi-colored, the imaging model is much more powerful than any analogous silk-screening operation. Picture a slide projector shining a general colored image through a stencil onto the paper. The right side of Figure 1 shows the result of pushing a multi-colored source through a stencil.

Other properties of the imaging model lack good analogies in the silk-screening metaphor. The first of these is an additional level of stencil called a *clipping region*. The purpose of the clipping region is to restrict the area where ink is displayed regardless of what other shapes or masks are used. When a clipping region is specified, then only ink falling inside that region is displayed. Figure 2 illustrates the effect of a clipping region.

Also unlike anything in the silkscreening process are the model's general *mapping* facilities. Under control of the application, stencils and sources may be mapped independently through any linear transformation prior to display. Imagine rotating the stencil, or stretching a rubber stencil to expand or skew it; at the same time, imagine rotating the slide projector, or pulling it back to enlarge the image. The mapping facility gives the application program a great deal of flexibility in the composition of images.

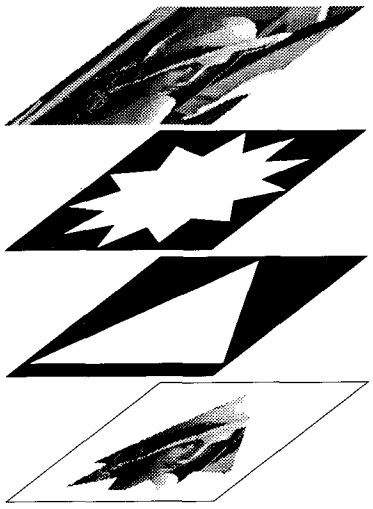

Figure 2: A clipping region.

Additional operators *generate* shapes that correspond to drawn lines and curves. These operators take trajectories (piecewise analytic segments, open or closed) and brush information (just another shape) and generate closed shapes that correspond to lines and curves drawn with the given brush. The resulting lines and curves then act like any other stencils, and may have inks pushed through them.

This model does not address issues concerning the display of projected three dimensional objects, or issues dealing with complex conformal mappings. It is assumed that these kinds of objects, if desired, are transformed into appropriate two dimensional imaging constructs prior to display.

Interfaces and Implementation

The following description centers on the *interfaces* which define the boundaries between program components. A carefully designed interface effectively decouples its *clients* from the internal details of its *implementors* by defining a set of available *operations*. Each client retains a pointer to the *state information* needed by the implementor, but uses only the interface-defined operations to manipulate that state. The implementation outlined here relies heavily on this notion of an interface to achieve device independence.

The programmer who wants to display or print pictures is a client of the *application* interface. The operations defined in this interface allow the application programmer to construct images by combining various sources and stencils. This is the interface that presents the imaging model described above. Equally important, however, are the *internal* interfaces which separate the device-independent components of the implementation, from the device-dependent components.

These interfaces isolate most of the implementation from the peculiar characteristics of different display devices and image sources.

Coordinate Systems

One of the key ideas in making applications independent of devices is defining coordinate systems and isolating them from each other. Because pixel addressing conventions vary across different raster devices, it is particularly important to isolate the *device coordinate system* (DCS) from the application program view of the system. To achieve this isolation, the imaging model defines an intermediate coordinate system called the *virtual coordinate system* (VCS). This common coordinate system serves as the meeting ground for device implementations and user applications. The system implementor writing the device dependent code for a particular display is concerned only with the mapping between the VCS and the DCS. The application programmer using the imaging operators is concerned only with building images relative to the virtual coordinate system.

Another aspect of coordinate systems often overlooked in building raster display sytems is the isolation of the *source coordinate system* (SCS) in which sampled image sources are defined. Sampled images are, in some sense, dual to display devices. They provide a raster input form for images in the same way that devices provide a raster output form for images. The application program should be no more concerned with scanning properties or coordinate system particulars of source images than with scanning properties or coordinate sytems of display devices; it should deal with images in the virtual coordinate system.

To accomplish this, the imaging model adopts conventions for source independence analogous to its conventions for device independence. Knowing these conventions, the application can predict how images or masks will be mapped into the virtual coordinate system. Therefore, the application can manipulate and transform the image geometrically without regard to the scanning or resolution properties of the image or mask.

Treating source images in this way gives a pleasing symmetry to the implementation as well. The interface to a raster input source provides the source's boundary in the source coordinate system, and provides a mapping from the source coordinate system to the virtual coordinate system. The interface to a raster output device provides a mapping from the virtual coordinate system to the device coordinate system, and provides the device's boundary in the device coordinate system. Given these interfaces, the implementation has all the information it needs to map coordinates directly from source to destination, and to compute the intersection of their boundaries.

The following description of an implementation of the imaging model will illustrate how all the above concepts hang together.

Application Interface

Each application using the imaging model may invoke a collection of imaging operators through the application interface. These operators define mappings, clipping regions, shapes, colors, masks and image sources, and cause shapes to be displayed. Because of the many differences in languages and operating systems, only an indication of the framework of operators is given here. Additions are needed to fill in the details for a specific programming environment.

The state information associated with an application is called the *display context*. As an application uses the display, the imaging operators use and modify the information in the display context. This information includes:

1. An interface to a device.
2. The current position (*cpx,cpy*) in the device coordinate system.
3. The transformation matrix *T* that maps application defined shapes into the device coordinate system.
4. The clipping region (*CR*).

The notation used to indicate the procedure interface is of the form:

ProcedureName: PROCEDURE [*p1*: PType1, *p2*: PType2, ...]
RETURNS [*r1*: RType1, *r2*: RType2, ...];

Here it is assumed that each procedure takes input parameters (named *p1*, *p2*, ...) of various types, and returns results (named *r1*, *r2*, ...) of various types. Some of the type names should be obvious (*e.g.*, Real for floating point numbers); others (*e.g.*, Trajectory) are left undefined. For these undefined types, it is assumed that the implementation will define an appropriate data structure. The particulars of the data structures chosen are not important to this discussion, and need not be known by the application using the interface.

The Device and Image types hold state information for particular display devices and image sources. The interfaces to devices and images will be described below. Note, however, that the application can treat them entirely as "black boxes"; it need not know even their interface definitions.

NewXXXDevice: PROCEDURE [<optional parameters>]
RETURNS [Device];

A procedure of this form is provided by each device implementation.

NewXXXImage: PROCEDURE [<optional parameters>]
RETURNS [Image];

A procedure of this form is provided by each scanned image type.

NewDisplayContext: PROCEDURE [*device*: Device]
RETURNS [*dc*: DisplayContext];

Initializes a display context. The transformation *T* is initialized to the VCS-to-DCS mapping provided by the device. The clipping region *CR* is initialized to be the boundary of the device. The current position (*cpx,cpy*) is set to 0,0.

GetCurrentPosition: PROCEDURE [*dc*: DisplayContext] RETURNS [*x*,*y*: Real];

Returns *x*,*y* such that $(x,y)T = (cpx,cpy)$.

SetCurrentPosition: PROCEDURE [*dc*: DisplayContext, *x*,*y*: Real];

Sets (cpx,cpy) to $(x,y)T$.

NewTrajectory: PROCEDURE [*x*,*y*: Real] RETURNS [*t*: Trajectory];

Returns a new trajectory. Every trajectory has a *first position* (FP) and a *last position* (LP); for a new trajectory, both FP and LP are set to (*x*,*y*).

LineTo: PROCEDURE [*t*: Trajectory, *x*,*y*: Real] RETURNS [*u*: Trajectory];

Returns a trajectory *u* that includes, in addition to *t*, the line segment from *t*'s LP to (*x*,*y*). The LP of *u* is (*x*,*y*).

CurveTo: PROCEDURE [*t*: Trajectory, *x1*,*y1*,*x2*,*y2*,*x3*,*y3*: Real] RETURNS [*u*: Trajectory];

Returns a trajectory *u* that includes, in addition to *t*, a cubic curve segment from *t*'s LP to (*x3*,*y3*). The curve segment is defined by its four Bezier control points: *t*'s LP, (*x1*,*y1*), (*x2*,*y2*), and (*x3*,*y3*). The LP of *u* is (*x3*,*y3*).

Close: PROCEDURE [*t*: Trajectory] RETURNS [*u*: Trajectory];

Returns a trajectory *u* that includes, in addition to *t*, the line segment from *t*'s LP to *t*'s FP. The FP and LP of *u* are equal.

Rectangle: PROCEDURE [*xl*,*yl*,*xu*,*yu*: Real] RETURNS [*t*: Trajectory];

A convenience function, equivalent to:

```
t ← NewTrajectory[xl,yl];
t ← LineTo[t,xu,yl];
t ← LineTo[t,xu,yu];
t ← LineTo[t,xl,yu];
t ← Close[t];
```

NewShape: PROCEDURE RETURNS[*s*: Shape];

Returns an empty shape list.

AddToShape: PROCEDURE [*s*: Shape, *t*: Trajectory] RETURNS [*r*: Shape];

Returns a Shape *r* that contains, in addition to the trajectories of *s*, the trajectory *t*.

MakeLineShape: PROCEDURE [*brush*: Shape, *t*: Trajectory] RETURNS [*s*: Shape];

The locus of each point interior to the brush is computed as the origin of the brush shape is moved along the trajectory. The union of all these loci form a set of solid areas. The boundaries of these areas make up a shape. It is this shape that is returned by MakeLineShape. Note: the above definition is just that, and does not describe how line shapes might really be computed.

MakeColorSource: PROCEDURE [*hue*,*sat*,*brightness*: Real] RETURNS [*s*: Source];

Supplies a Source data structure representing solid ink of the specified color.

MakeImageSource: PROCEDURE [*image*: Image] RETURNS [*s*: Source];

Supplies a Source data structure representing the specified sampled image.

DrawShape: PROCEDURE [*dc*: DisplayContext, *shape*: Shape, *source*: Source];

Maps the shape and source through the transformation *T*, clips the shape against the *CR*, and displays the shape with the given source as the ink.

DrawMask: PROCEDURE [*dc*: DisplayContext, *mask*: Image, *source*: Source];

Maps the mask, its boundary and the source through the transformation *T*, clips the boundary against the *CR*, and displays the resulting clipped mask with the given source as the ink.

SetClipShape: PROCEDURE [*dc*: DisplayContext, *shape*: Shape];

Maps the shape through the transformation *T*, clips the shape against the *CR*, and installs the shape as the new clipping region.

Translate: PROCEDURE [*dc*: DisplayContext, *x*,*y*: Real];

Builds a transformation matrix *M* that will translate (0,0) onto (*x*,*y*), and sets *T*, in the display context, to *MT*.

Rotate: PROCEDURE [*dc*: DisplayContext, *angle*: Real];

Builds a rotation matrix *M* that will rotate (1,0) onto (cos(*angle*),sin(*angle*)), and sets *T*, in the display context, to *MT*.

Scale: PROCEDURE [*dc*: DisplayContext, *sx*,*sy*: Real];

Builds a transformation matrix *M* that will scale (1,1) onto (*sx*,*sy*), and sets *T*, in the display context, to *MT*.

Concatenate: PROCEDURE [*dc*: DisplayContext, *m*: Matrix];

Sets *T*, in the display context, to *mT*.

GetMatrix: PROCEDURE [*dc*: DisplayContext] RETURNS [*t*: Matrix];

Returns the matrix *t* such that if *M* is the matrix that transforms VCS to DCS, then $t = TM^{-1}$.

Because characters of fonts are treated like any other shapes in this imaging model, routines to display text do not properly belong in the above set of primitive operators. However, since applications often use text and characters extensively, convenience routines can be provided to make the display of text simple. A typical set would include:

MakeFont: PROCEDURE [<font name>] RETURNS [*f*: FontId];

DisplayChar: PROCEDURE [*c*: Character, *f*: FontId];

DisplayText: PROCEDURE [*s*: String, *f*: FontId];

GetCharMetrics: PROCEDURE [*c*: Character, *f*: FontId] RETURNS [<whatever metrics a font provides>];

Device Independent Procedures

The routines that implement the imaging model depend on a large number of ideas and algorithms. It is not practical to describe these in detail. Instead the few critical observations and algorithms necessary to get the central ideas of the implementation are discussed.

To best understand how these ideas work together, one needs to understand several fundamental operations found in the DrawShape, DrawMask and SetClipShape procedures.

Shape Mapping

Given a shape (set of closed trajectories) and a transformation matrix M, we will say that the shape is *mapped* using M when all points and cubics that comprise each of the shape's trajectories are transformed using M.

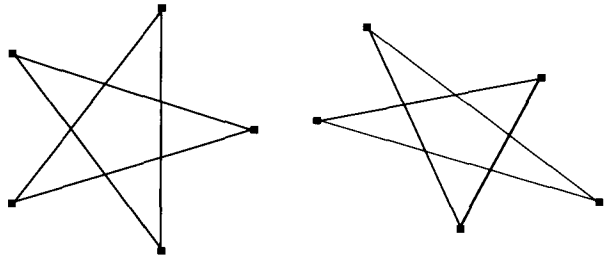

Figure 3: Shape mapping.

Shape Approximation

Given a shape (set of closed trajectories) in the DCS we will say that the shape is *approximated* when all cubics, in each of the shape's trajectories, are replaced by piecewise linear approximations. Since the shape is in the DCS, it is possible to make piecewise linear approximations as a function of device resolution.

An approximated shape exists as a collection of polygons, which may be mutually intersecting, self intersecting and concave.

Shape Reduction

An approximated shape is *reduced* when the polygons that make up the shape are converted into a collection of disjoint, convex polygons which tile the *interior* of the shape. The locus of the shape's interior is determined by applying a wrap number convention. There are a number of known tiling algorithms [1,2,3].

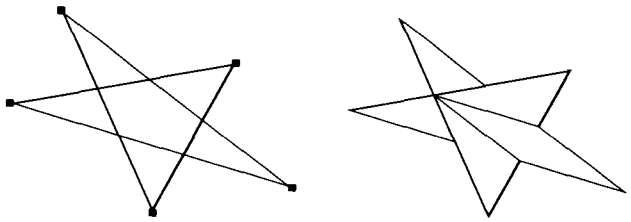

Figure 4: Shape reduction.

Shape Clipping

An approximated reduced shape will be called *clipped* if each of its convex polygons has been clipped against the set of the convex polygons that make up a clipping region.

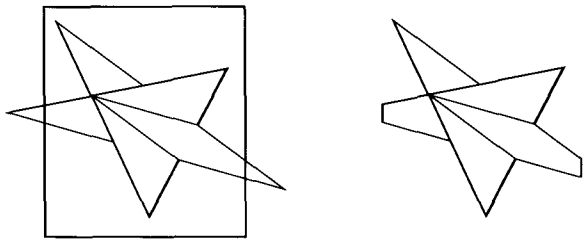

Figure 5: Shape clipping.

The above concepts are used freely in the process of displaying a shape. Of equal importance is the description of the basic operation that is carried out by the device dependent part of the system.

Internal Interfaces

Just as the application program is a client of the application interface, the device-independent portion of the system is a client of the *device* and *source* interfaces.

Information available at the *device interface* includes:

1. The transformation matrix that maps the virtual coordinate system to the device coordinate system.
2. The shape (in the device coordinate system) that bounds the display area.
3. A vector of procedures that implement the scan conversion primitives. These are *device-specific* procedures; their calling sequences do not vary from device type to device type, but the way they perform their function may vary dramatically across device types.

The *source interface* is used both for scanned image sources and for masks. Information available at the source interface includes:

1. The transformation matrix that maps the image or mask coordinate system to the virtual coordinate system.
2. The shape (in the source coordinate system) that bounds the image or mask area.
3. A vector of procedures that implement the source accessing primitives. These are *source-specific* procedures.

The operations defined by these interfaces are called as required by the device independent portions of the system. Some examples of their use are illustrated below.

Device Dependent Procedures

Scan Conversion

Strictly speaking, a given device implementation need supply only one procedure, **DisplayConvexPolygon**. This procedure implements the most general case of scan conversion that the device must handle: pushing a general mapped scanned image, as a source, through a mapped mask. The arguments taken by the procedure are:

1. A convex polygon. This polygon represents either part of a shape (if no mask is given), or the boundary of a mask (if a mask is given).
2. A source, which is either a constant value, or:
   a. A mapping S from the source's SCS to the DCS, and
   b. A pointer to the source sample array.
3. An optional mask which includes:
   a. A mapping M from the mask's SCS to the DCS, and
   b. A pointer to the mask sample array.

The operation carried out by this procedure is:

For each pixel position (x,y) in the interior of the convex polygon in DCS, compute $(x_s,y_s) = (x,y)S^{-1}$, and $(x_m,y_m) = (x,y)M^{-1}$. If the value in the mask array at $(x_m,y_m) = 1$, then interpret the source value at (x_s,y_s) for the device type and display an appropriate value at (x,y). Note: in practice, instead of mapping each point in DCS through two inverse mappings (computationally expensive), incremental mapping techniques are used; in this way each mapping is replaced with two add operations. This is discussed later in the section on optimizations.

In most implementations of a given device, special cases that are expected to occur frequently can be provided as subcases of the above. Two common special cases are listed below.

1. **DisplayRectangularMask**. This procedure is a special case of the above where: the polygon is rectangular in the DCS, the source is a constant color, and the mapping from the SCS to DCS contains only a translation component. This procedure is typically used for the display of most characters.
2. **DisplaySimplePolygon**. This procedure is a special case of DisplayConvexPolygon where the source is constant, and there is no mask. Most line drawings and simple filled shapes use this procedure.

It can be seen how the above device independent and device dependent notions are brought together by examining the processing steps associated with the DrawShape, DrawMask and SetClipShape procedures.

DrawShape (source is a constant color)

1. Map the input shape into the DCS using T (the transformation from the display context).
2. Approximate the resulting shape, reduce it, and clip it to *CR* (the clipping region from the display context).
3. Pass each resulting convex polygon to the DisplayConvexPolygon procedure (found in the device interface in the display context), along with the constant color value.

DrawShape (source is a scanned image)

1. Concatenate the SCS-to-VCS transformation Q (from the scanned image interface) with the VCS-to-DCS transformation T (from the display context) to form the SCS-to-DCS transformation S ($S = QT$).
2. Map the shape that bounds the scanned image (from the scanned image interface) into the DCS, using S.
3. Approximate the resulting shape, reduce it, and clip it to *CR*.
4. Use the resulting set of convex polygons to define a new clipping region, *CR'*.
5. Map the input shape (the argument to DrawShape) into the DCS, using T.
6. Approximate the resulting shape, reduce it, and clip it to *CR'*. At this point, the resulting convex polygons represent the intersection of the mapped boundary of the image and the mapped input shape.
7. Pass each convex polygon to the DisplayConvexPolygon routine, along with transformation S and a pointer to the image samples.

DrawMask (source is a constant color)

1. Concatenate the SCS-to-VCS transformation R (from the mask interface) with the VCS-to-DCS transformation T to form the SCS-to-DCS transformation M ($M = RT$).
2. Map the bounding shape of the mask into the DCS, using M.
3. Approximate the resulting shape, reduce it, and clip it to *CR*.
4. Pass each resulting convex polygon to the device's DisplayConvexPolygon procedure, along with the constant color value, transformation M, and a pointer to the mask samples.

DrawMask (source is a scanned image)

1. Concatenate the SCS-to-VCS transformation Q (from the scanned image interface) with the VCS-to-DCS transformation T to form the SCS-to-DCS transformation S ($S = QT$). This mapping transforms coordinates from the scanned image to the device.
2. Map the bounding shape of the image into the DCS, using S.
3. Approximate the resulting shape, reduce it, and clip it to *CR*.
4. Use the resulting set of convex polygons to define a new clipping region, *CR'*.
5. Concatenate the SCS-to-VCS transformation R (from the mask interface) with the VCS-to-DCS transformation T to form the SCS-to-DCS transformation M ($M = RT$) This mapping transforms coordinates from the mask to the device.
6. Map the bounding shape of the mask into the DCS, using M.
7. Approximate the resulting shape, reduce it, and clip it to *CR'*.
8. Pass each resulting convex polygon to the device's DisplayConvexPolygon procedure, along with transformations S and M and pointers to the image and mask samples.

SetClipShape

1. Map the input shape into the DCS, using T.
2. Approximate the resulting shape, reduce it, and clip it to *CR*.
3. Install the resulting set of convex, non-intersecting polygons as the new *CR* in the display context.

Note that the DisplayConvexPolygon routine never needs to do any boundary checking. When a scanned image or mask is present, the shape to be scan converted always represents an interior portion of the image or mask. Also, the clipping region in the display context always lies inside the device boundary.

Optimizations

Although the above algorithms may involve extensive computation, certain simple expected cases can be made to short circuit most of the above machinery without losing any of the generality. Three of these short cuts will now be described.

The display of characters

In situations where high performance is required, character fonts are designed to work with a given display type, *i.e.*, a character is defined to be a mask whose resolution and scanning characteristics are the same as the device's. Also the bounding shape of a character mask is a rectangle. Under these circumstances, the mapping taking the mask in SCS to VCS, when concatenated with the device's mapping from VCS to DCS, will be the identity mapping (with, perhaps, an application-introduced translation component). Since this is the case, the bounding rectangle of the mask will map into a rectangle in the DCS. If in addition the source color is solid black and the bounding rectangle is not clipped, the scan conversion process reduces to the one-to-one transfer of pixels in one rectangle to pixels in another. To avoid checking for the identity transformation for each character, the combined transformation from SCS to VCS to DCS can be noted as an identity in the display context. If no operations that change transformations (other than translation) take place between the display of succesive characters, then no transformation checking need be done.

If any of the above conditions is not true, then the optimization fails and the more general machinery will display the character.

Transforming sources and masks

The rotation, scaling and sampling of sources and masks can be computationally expensive. To reduce this computation the following steps are taken (the discussion centers on the handling of sources, but the same discussion applies to masks).

Scan conversion, in the device coordinate system (DCS), is carried out pixel-by-pixel along successive scan lines. If the unit vector (1,0) representing the delta vector between successive pixels in a scan line is mapped through S^{-1} (the mapping from the device coordinate system to the source coordinate system), then the delta vector *ds* between required sample positions in the source is obtained. Successive sample positions in the source that correspond to the successive pixels along the scan line can be incrementally computed by mapping the beginning of the scan line in the DCS through S^{-1}, and incrementally adding *ds* to obtain the position of the next sample to be used in the scan line. This optimization replaces a general point transformation with two additions.

Bounding boxes

An additional artifact, a bounding rectangle associated with a shape, can be introduced after mapping into the DCS. Bounding rectangles can be used to short cut the amount of computing done by the shape reduction and clipping machinery; *i.e.*, if a bounding rectangle is exterior to the set of clipping regions, then the associated shape need not be processed further. On the other hand, if a bounding rectangle is totally within the clipping region, then the shape need not be clipped since it is totally within the region.

Conclusion

A consistent, simple, device independent imaging model, and an implementation of the model, provide powerful tools to the application programmer. The model presented in this paper, presents a general interface that has been useful over a wide range of applications and devices. Users have commented positively on the simplicity of the model and its ease of use.

The aspects of this model that have been most successful are the following.

Treating text characters like other graphic objects, and not as low-level, device dependent primitives has been a big win. High level font dependent abstractions are kept in the application programs and not in the low-level device driving programs. Furthermore, because of optimization of special cases, no significant performance loss results from this generality.

The generalized clipping facility is valuable. There are simple interactive metaphors and complex images that are difficult to build without the clipping facilities. In addition to the application advantages, the clipping facilities are used extensively in the implementation to avoid bounds testing in the low-level routines.

With this model an application does not know the particulars of the device, or even whether the device is shared. High level display management facilities can allocate a portion of the screen, set the clipping region to that portion, and pass the display context, representing the entire "device," to a subapplication. This aspect of device independence simplifies systems integration activities.

Acknowledgements

Many people have contributed ideas relating to the imaging model and its implementation. In particular, we would like to thank Martin Newell, Warren Teitelman, and Bob Sproull for their valuable contributions and suggestions.

Bibliography

[1] Garey, M. R., D. S. Johnson, F. P. Preparata, and R. E. Tarjan. "Triangulating a Simple Polygon." *Information Processing Letters* 7, 4 (1978): 175-179.

[2] Lee, D. T., and F. P. Preparata. "Location of a Point in a Planar Subdivision and its Applications." *SIAM J. Comput.* 6, 3 (1977): 594-606.

[3] Newell, M. E., and C. H. Sequin. "The Inside Story on Self-intersecting Polygons." *Lambda* 1, 2 (1980): 20-24.

[4] Newman, W. M., and R. F. Sproull. *Principles of Interactive Computer Graphics.* Second Edition. New York: McGraw-Hill, 1979.

[5] Sutherland, I. E., and G. W. Hodgman. "Reentrant Polygon Clipping." *CACM*, 17(1):32, January 1974.