VDI - A Visual Debugging Interface for Image Interpretation and Other Applications

Gudrun Klinker
Digital Equipment Corporation
Cambridge Research Lab

CRL 91/2 March 7, 1991

Abstract

Higher-level image interpretation systems typically consist of many tightly-coupled modules, each related to a particular aspect of the interpretation task. When such a system produces wrong results, programmers generally spend a lot of time writing special-purpose display routines to facilitate investigating the problem. This paper presents an attempt at unifying such efforts by merging graphics programming with debugging technology. Several generalizations are necessary to provide useful general-purpose display mechanisms: 1) The mechanisms need to be highly interactive such that programmers can select and change display preferences at debugging time. 2) The display routines should not be restricted to traditional, two-dimensional images. 3) The programmer must be able to visualize the data-dependent relationships between modules. VDI is a general-purpose Visual Debugging Interface designed to provide these generalizations. It is highly interactive, operates on arrays of arbitrary dimensionality, and provides mechanisms to describe dependencies between the results generated by various interdependent modules.

Keywords: interactive array visualization, scientific visualization, computer graphics, computer vision, debugging aids.

A shorter version of this paper will be presented at the 2nd Eurographics Workshop on Visualization in Scientific Computing, Delft, Netherlands, 22-24 April, 1991.

Contents

Figures

1 Introduction

For many years, it has been common practice for computer vision labs to develop their own graphics libraries - tailored to their specific hardware environment. With the advent of window system languages and graphics packages [Scheifler and Gettys 86, Gettys et al 90, GKS-3D 87, PHIGS 87], several general-purpose toolkits for viewing images and volumetric data have emerged from computer graphics research [XVision 3.0 89, VoxelView 89, Upson et al. 89]. Some of such visualization systems provide the basic mechanisms for displaying and slicing two- and three-dimensional data blocks and for rendering geometric data. Augmented with a (user-extensible) library of image processing routines and with an interactive visual programming mechanism, they are powerful toolkits for performing a sequence of image processing steps on images. They are designed to be used by (non-programming) scientists who use the toolkit as an aid for enhancing/visualizing images such that they (the humans) can interpret them more easily.

But such toolkits seem to be of limited usefulness to computer vision researchers who are trying to write automatic image interpretation programs. The basic (visual programming) assumption of the toolkits is that image processing operations can be packaged into independent "black-box" modules with easily specifiable input/output behavior and that the image data flows through a network of such modules. This is often the case in low-level image processing tasks. But in image understanding systems, the modules generally don't operate that independently of one another [Hanson and Riseman 78, Klinker et al. 90]. Consider the example of an object recognition algorithm that needs to detect the boundaries of an object in an image. The object may be as complex as a house, consisting of several walls, windows, doors, a roof etc., and it could be viewed from any angle, even be partially hidden behind trees, cars or other houses. Object recognition has to account for such possibilities and exploit partially gathered evidence to build up a consistent interpretation of the entire scene. Such evidence is stored partially in arrays accompanying the original image (such as a segmentation image) and partially symbolically (for example as a region list describing the shape and other properties of the currently segmented regions). Progress in some modules (e.g., for detecting windows) may be tightly coupled with progress in other modules (e.g., for detecting walls). Accordingly, image understanding modules can depend in very complex and global ways on the overall state of the interpretation. Blackboard systems have been suggested as one useful communication scheme between modules [Hanson and Riseman 78, Draper et al. 89, Goto and Stentz 87]. The current visual programming paradigm does not support such tight coupling between modules. It would be inefficient to pass the contents of the entire blackboard between modules. Furthermore, the blackboard paradigm assumes that the various vision modules operate in parallel: every module is an independent agent that sits and waits until some information on the blackboard triggers its interest. This scheme of a static and global arrangement of agents and data stands in contrast to the dynamic data flow concept adopted in visual programming.

But how can image interpretation systems be evaluated, and how can computer vision researchers determine how and why their modules interact in the way they do? I.e., how can researchers debug their systems and their research assumptions? We are back to the situation where individual programmers clutter the module code with special-purpose display routines. This is analogous to adding print-statements to program code in order to debug a program. It would be more efficient if standard debuggers provided the necessary display capabilities such that program-

mers could interrupt the program execution at arbitrary places and visualize the current state of the image interpretation process.

This paper presents an attempt at merging graphics programming with debugging technology. To provide general-purpose interactive graphical display routines for arrays, some common display practices have to be extended. This paper discusses such generalizations. It is essential that the interactive mechanisms match the necessities of the underlying program modules to access many images that depend on each other. That means that if the window detection module depends on (partial) results produced by the wall detection module, then the visual debugging interface must provide mechanisms for viewing the data under such constraints. For this purpose we provide mechanisms to formulate relationships between pixels in different images. The programmer can, for example, request to view all those pixels of the original image which the wall detection module has already labeled as being potential wall candidates.

A vision research workbench, "KBVision", with similar goals appeared several years ago [Stephenson 90]. However, this system was only designed to address the needs of visualizing two-dimensional images. We extend the display capabilities from the traditional two- or three-dimensional image domain to arrays of much higher dimensionality because programming languages provide arrays of higher dimensionality and debuggers should reflect that concept. With this extension, not only the traditional image-oriented applications can benefit from the debugger's display capabilities but any program that operates on arrays. Most current work in computer vision uses two-dimensional images or sets of two-dimensional images. So far, it is most common to display the data as two-dimensional *(x,y)* frames, while providing specialized extensions for representing a third dimension (pseudo-coloring, stereo glasses, movie-loops). But we expect that the displaying needs will change in the future: Some applications exist already now where researchers prefer to view the data along other dimensions, e.g., to display (*x,time*) slices of an image sequence [Bolles and Baker 85, Adelson and Bergen 86]. Other applications need to visualize three-dimensional data volumes [Carlbom et al 91, Klinker et al. 90, Szeliski 90, Szeliski 91]. Furthermore, we expect that research in different fields of computer vision will become more integrated in the future and that image data will then comprise more than three dimensions. Image data could, for example, be a (five-dimensional) time sequence of stereo color images, with different image understanding modules operating on different slices (color, stereo or time) through the data volume. At that point, it will be most useful to think of image data as a high-dimensional array with no preferred axes and let the programmer select viewing and accessing preferences interactively.

According to these observations, we are currently developing a general-purpose Visual Debugging Interface, VDI. VDI is currently a subroutine library of display mechanisms. It can be called with minimal effort from anywhere in an application program, as well as at run-time from a debugger [Saber-C 89]. It operates on multi-dimensional arrays and provides interactive tools to specify criteria for accessing and displaying data, possibly in relationship to data values in other arrays. VDI is written in C, using the X window system and the Xt and XUI toolkits [Scheifler et al. 88, XUI Toolkit Intrinsics 89, XUI Toolkit 89]. The system has been used in applications as diverse as the analysis of highlights in color images and visualizing the enabling and blocking constraints between a set of processes on a MIMD-machine.

2 The Interactive Interface

We have divided interactive input into three classes of commands: commands for 1) specifying data selection criteria, 2) defining data manipulation techniques, and 3) defining the display style. Each of these sets will now be described in detail.

2.1 Data Selection

VDI operates on multi-dimensional arrays for which each dimension is described by a *feature axis*. Figure 2-1 shows an example of a four-dimensional array: a (time) sequence of color images with ***x***, ***y***, the color vector, ***c***, and the time vector, ***t***, defining the feature axes.

High-dimensional arrays generally cannot be displayed in their full dimensionality. VDI provides an interactive interface through which the programmer can slice the data along arbitrary dimensions. The interface also possesses basic cropping and thresholding capabilities, such that the programmer can restrict data access to a subarea and a subrange of values. Finally, the interface provides mechanisms for relating data selection from one array to the values of data elements in other arrays. Such mechanisms allow the programmer to issue a command like "select all pixels from image *i* which belong to region *r* (i.e., whose region label in the related segmentation image *j* is *r*)".

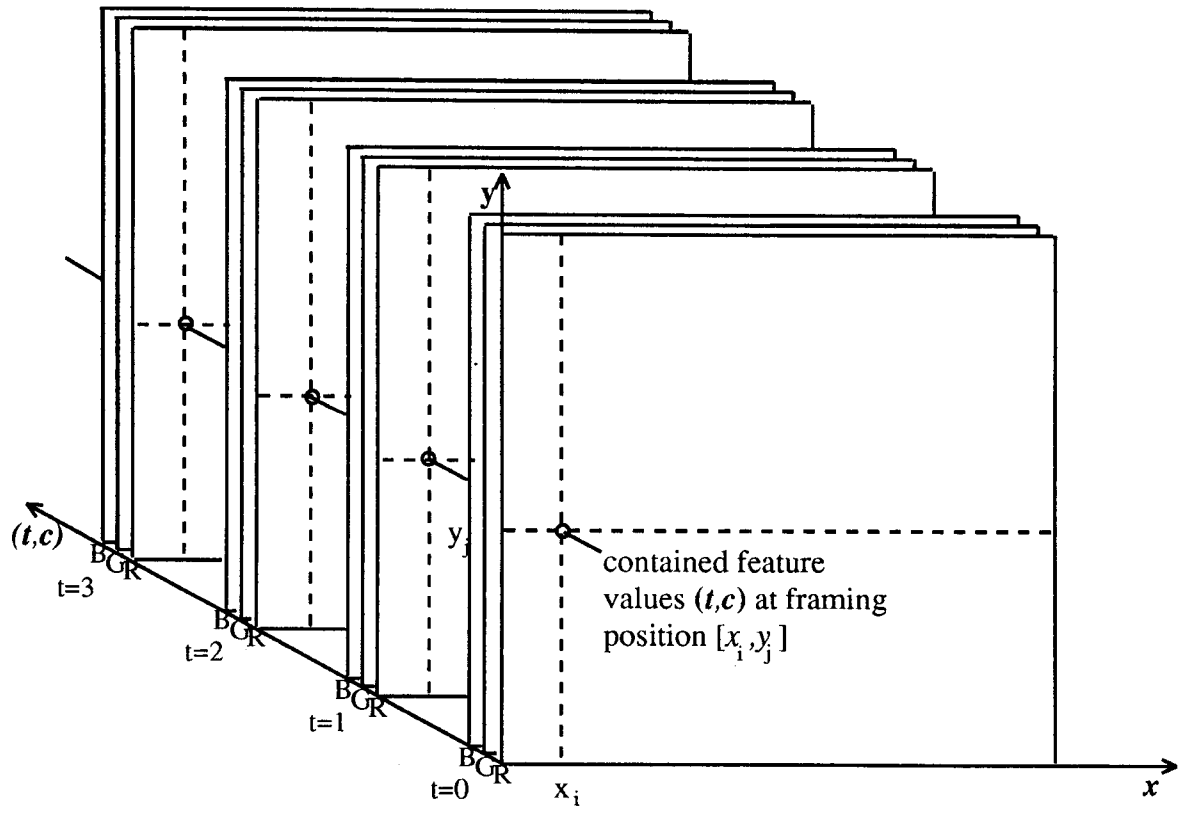

Figure 2-1 A time sequence of color images

2.1.1 Data Slicing, Cropping, Subsampling and Thresholding

VDI's data slicing mechanism distinguishes between two kinds of features: *framing features* and *contained features*. The framing features are those by which the programmer defines a frame for a slice through the data. The contained features are the remaining (i.e., non-framing) features. Figure 2-1 shows a typical arrangement for viewing a time sequence of color images as a sequence of two-dimensional image frames: x and y are the framing features, with each (x,y)-position containing the various pixel values along the color and time dimensions. Any particular image frame represents a slice through the 4D data block, with fixed values for t and c and varying values for x and y.

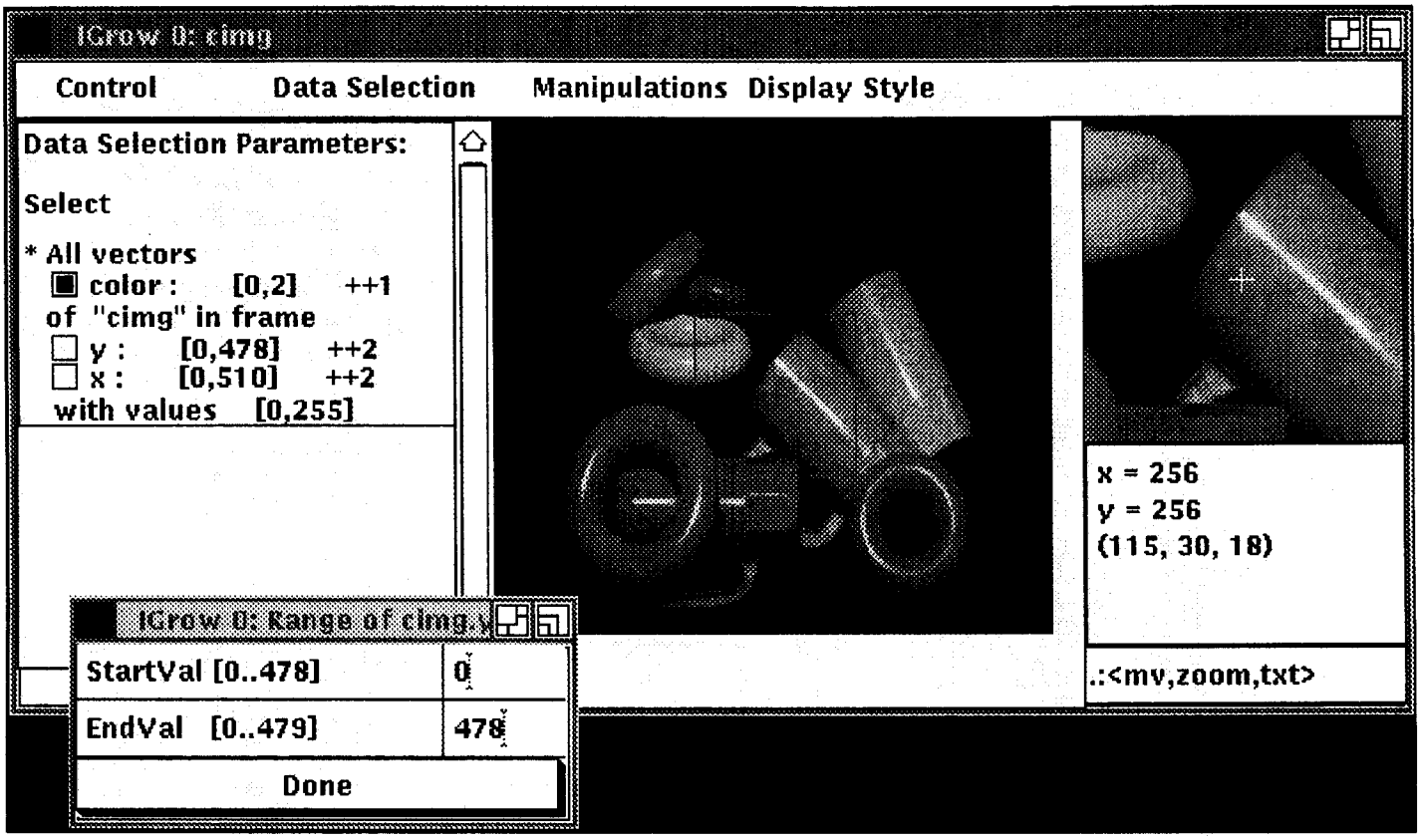

Figure 2-2 A subsampled color image

Figure 2-2 shows a sample interaction with VDI[1]. The center area displays a subsampled color image of a scene of plastic cups and donuts.[2] The area to the left presents the current values of the data selection parameters. Each feature axis is listed with its currently selected range of values (in square brackets) and its subsampling increment (here: ++1 or ++2). VDI provides the interactive tools for slicing, cropping and subsampling arrays. It uses the first index, the last index, and the increment between indices to select data along each feature axis. The programmer

[1] The topmost (shaded) area displays the name of the application program, as well as a window identification number and the variable name of the array in the application program.

[2] To simplify the reproduction process, the picture is only shown in black and white.

can click on any of these numbers to change them. In this example, VDI is in the process of accepting a new range for the *y*-dimension. With a similar mechanism, the programmer can threshold the data values. Alternatively to typing in new cropping ranges, a window can also be cropped with the mouse by selecting a rectangular area in the display area (see section 2.2.2).

VDI's interactive data slicing mechanism uses the toggle buttons (squares) to the left of the feature names. Framing features are indicated by hollow squares, contained features are filled squares. The data selection area lists all contained features above all framing features. In Figure 2-2, *x* and *y* are framing features and *color* is the contained feature. Accordingly, each (*x*,*y*)-pixel location contains a color triple <R,G,B>. More data slicing examples are shown in Figures 2-9 through 2-11. The programmer can toggle any framing feature into a contained one and vice versa. It might be useful to augment this mechanism in the future with a tool for interactively switching the sequencing of framing features or contained features, thus providing the means to flip an image along its diagonal.

2.1.2 Links between Several Arrays

As discussed in section 1, image interpretation modules often require information from many arrays holding partial results. At program execution and debugging time, the programmer needs to be able to visualize these relationships between the data. Figure 2-3 shows VDI's menu to link arrays. It provides mechanisms to restrict arrays ("Add restriction" and "Add indirect restriction"), as well as tools to simultaneously show the contents of several arrays ("Add data set"). The following subsections describe these mechanisms.

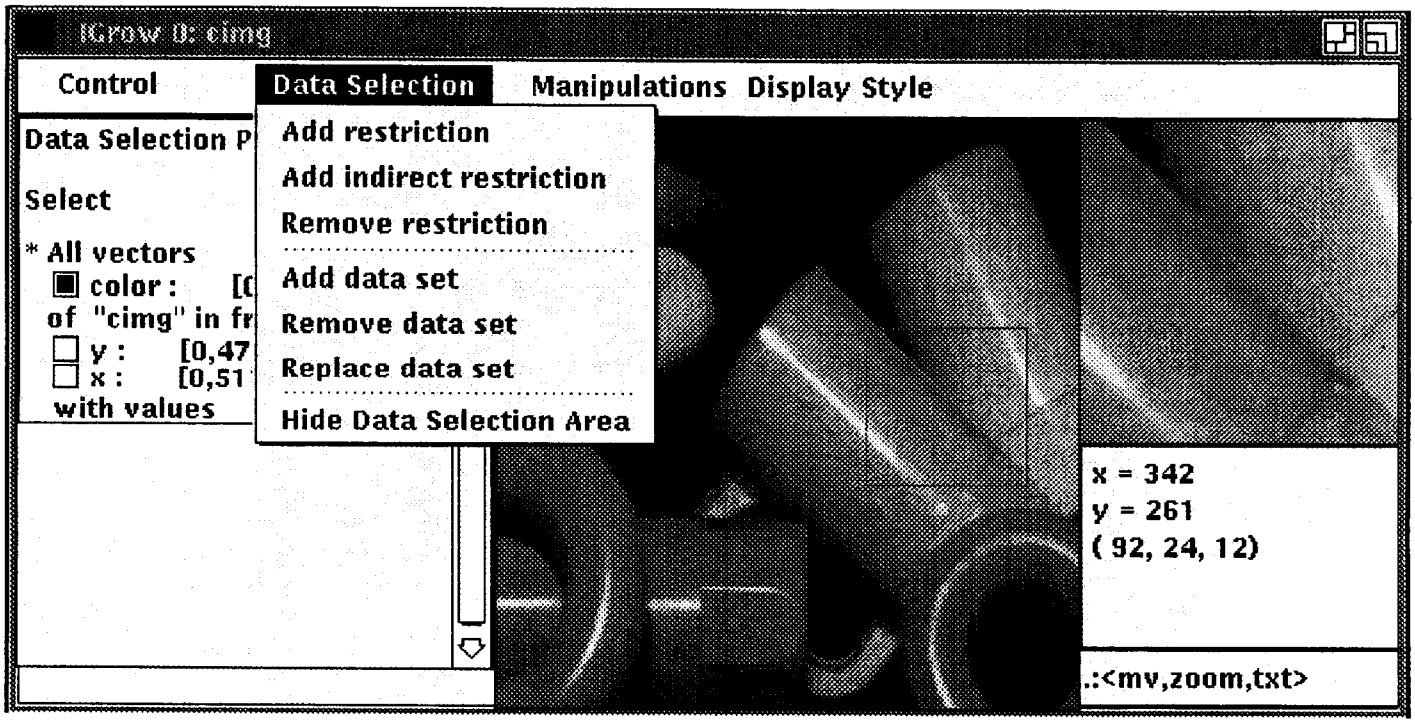

Figure 2-3 VDI's menu for linking arrays

2.1.2.1 Direct Restrictions

VDI provides basic mechanisms for specifying restricting relationships between arrays. When the programmer restricts an array by another array, VDI displays the primary array using the following rule: "display pixel $p[i,j,...]$ of the primary array, if the corresponding pixel $r[i,j,...]$ in the restricting array lies within the user-specified value range. Otherwise, display a user-specified replacement value".

Figure 2-4 shows VDI's restriction mechanism. The upper image in the figure shows a segmentation image which contains masks covering three of the eight objects in the scene. The segmentation pixels in the masks have values 2 and up, whereas the pixels in the unmasked areas have value 1. The lower image of Figure 2-4 shows how such segmentation information can be linked to the original image. In the data selection menu on the left, the programmer has asked VDI to display all those color pixels of the original image for which the corresponding pixel in the segmentation image is 1, i.e., for which no segmentation mask has yet been constructed. All other pixels are displayed using a replacement value (here: a light gray). Effectively, the segmented objects are "masked out of" the original image such that the programmer (as well as the data manipulation modules) can concentrate on the image areas that are not yet interpreted. The masked image shows that parts of the highlights on the objects are not covered by the object masks, and that a small portion of the horizontal (green) cup has been included into the object mask of the diagonal (orange) cup. Similarly, the linking mechanism can be used for analyzing local image properties, such as image gradients or curvature: the original image can be linked to a gradient image, requesting that all pixels with gradients of a specified range be shown.

Currently, VDI only links array elements at the same framing positions in different arrays. If the framing features have different ranges in the arrays, VDI crops the arrays to the common (overlapping) area. A future extension to VDI may provide a transformation mechanism for translating, scaling and rotating coordinates from one array into coordinates in another array. Such mechanisms will be useful for analyzing neighborhood relationships in images (by shifting arrays). Automatic scaling will be essential for analyzing image pyramids, e.g., for visualizing relationships between pixels at different levels of a pyramid.

The programmer can request any number of arrays to be linked using this restriction mechanism. VDI connects the conditions on all restricting arrays via the logical AND-operator. A future extension to VDI may provide mechanisms to specify arbitrary boolean formulas on restricting array variables.

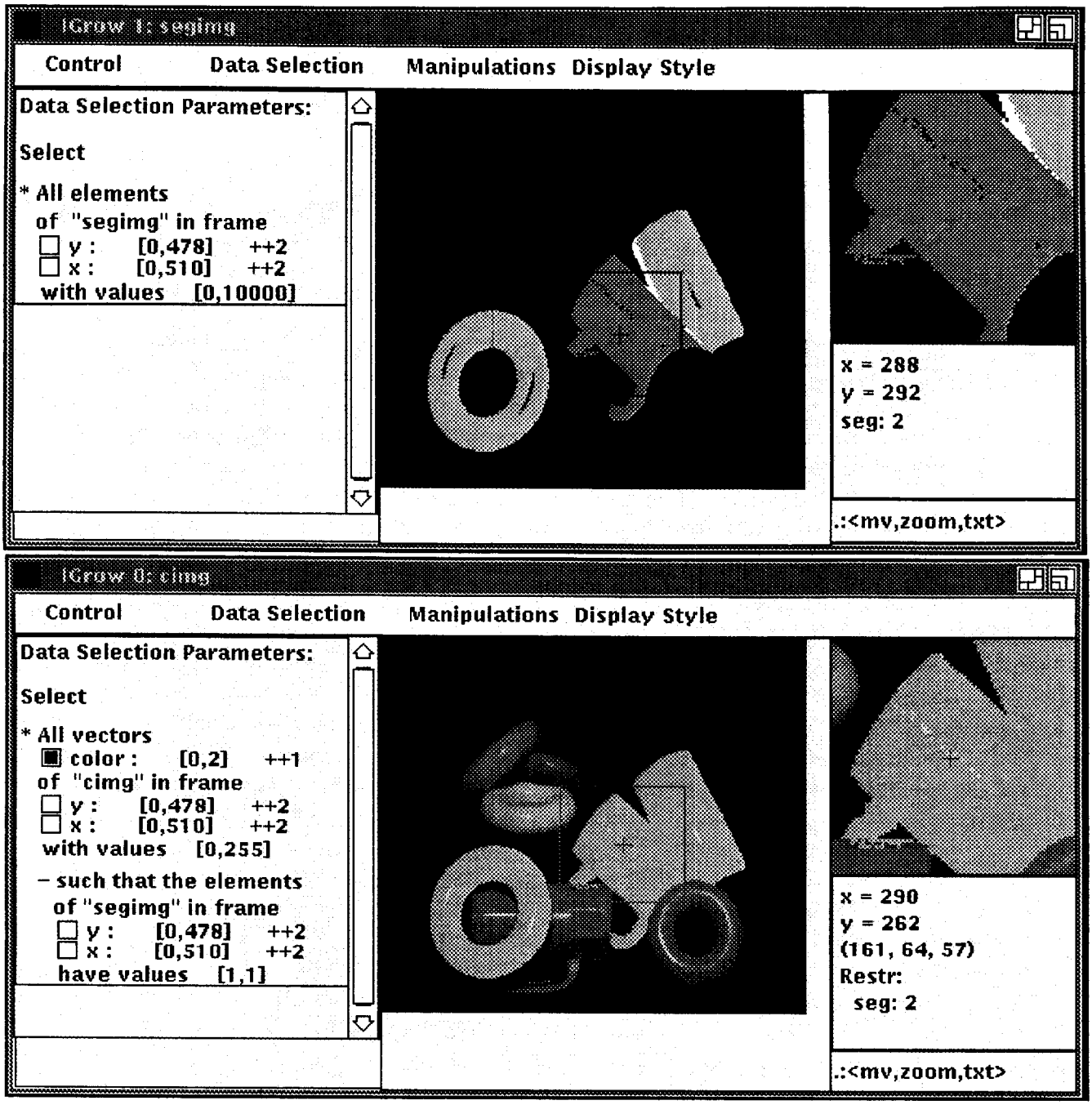

Figure 2-4 Direct restrictions on an array

2.1.2.2 Indirect Restrictions

In addition to direct restrictions on arrays, VDI accepts indirect restrictions. VDI uses the pixel value $p[i,j,...]$ of the primary array as index into indirectly restricting arrays: "display pixel $p[i,j,...]$ of the primary array, if the pixels $r[p[i,j,...]]$ in all indirectly restricting arrays lie in their specified value ranges. Otherwise, display the specified replacement value".

Figure 2-5 demonstrates VDI's indirect restriction mechanism. For this example, we have converted the color image of the plastic scene into a black-and-white image and then generated an intensity histogram - a one-dimensional array of values. The histogram array is shown in the lower half of Figure 2-5.[1] [2] The upper half of the figure displays the intensity image, using the indirect constraint that only pixels be shown whose intensity values occur between 800 and 1500 times in the image.

In general, indirect restrictions can be established between arrays of any dimensions: if the contained feature of the primary array allows for more than one feature value, or if several contained features exist, an index vector, $pvec(1..n)[i,j,...]$, has to be used as a multi-dimensional index into a restricting array of appropriate dimensionality. The index vector contains all elements $(1..n)$ of the cross product of all contained features at the current framing position $[i,j,...]$. VDI supports multi-dimensional indirect restrictions to a limited extent: it follows the multi-dimensional restrictions of the lowest contained feature. If more than one contained feature exist, all but the lowest one are ignored.

Figure 2-6 shows an example for multi-dimensional indirect linking: we have created a three-dimensional color histogram for the color image of the plastic scene, counting how often each color triple <R,G,B> occurs in the image. The lower half of the figure displays the color histogram. To save allocated memory the application program has scaled each color axis down by a factor of 4, such that each color bin now effectively represents 4x4x4 neighboring colors. The application program has informed VDI of this down-scaling operation, and VDI takes it into account in the upper half of Figure 2-6 when it uses the color pixels of the original image as index vectors into the color histogram. In this example, we have requested to visualize all those color pixels in the original image that occur less than 50 times in the histogram. Such pixels are likely to carry a lot of noise, and a vision programmer may be interested in investigating whether there are systematic (i.e., not noisy) patterns. Indeed, the highlights seem to be particularly noise-prone, as well as a small area of inter-reflection between the two diagonal cups. Surprisingly, one of the small donuts also seems to carry more than the average amount of noise. Of course, this restricting relationship between the color histogram and the original image cannot be a substitute for a real analysis of noise in the image. After all, histograms depend on object sizes and shapes. However, an interactive and dynamic capability of linking histogramming information to the original image can provide quick intuitions about aspects of computer vision that may need to be considered in the application program.

[1] The 1D-histogram array is shown as a graph rather than as a row of intensities, because the graph renders the histogramming information (the individual histogram counts) more precisely to the user than the corresponding intensity image. See section 2.2.3 for a discussion of different graphical representations.

[2] To fit the printout onto a page, VDI does not display the data selection area. The dimensionality and size of the histogram array can be seen in the restrictions given in the upper half of Figure 2-5.

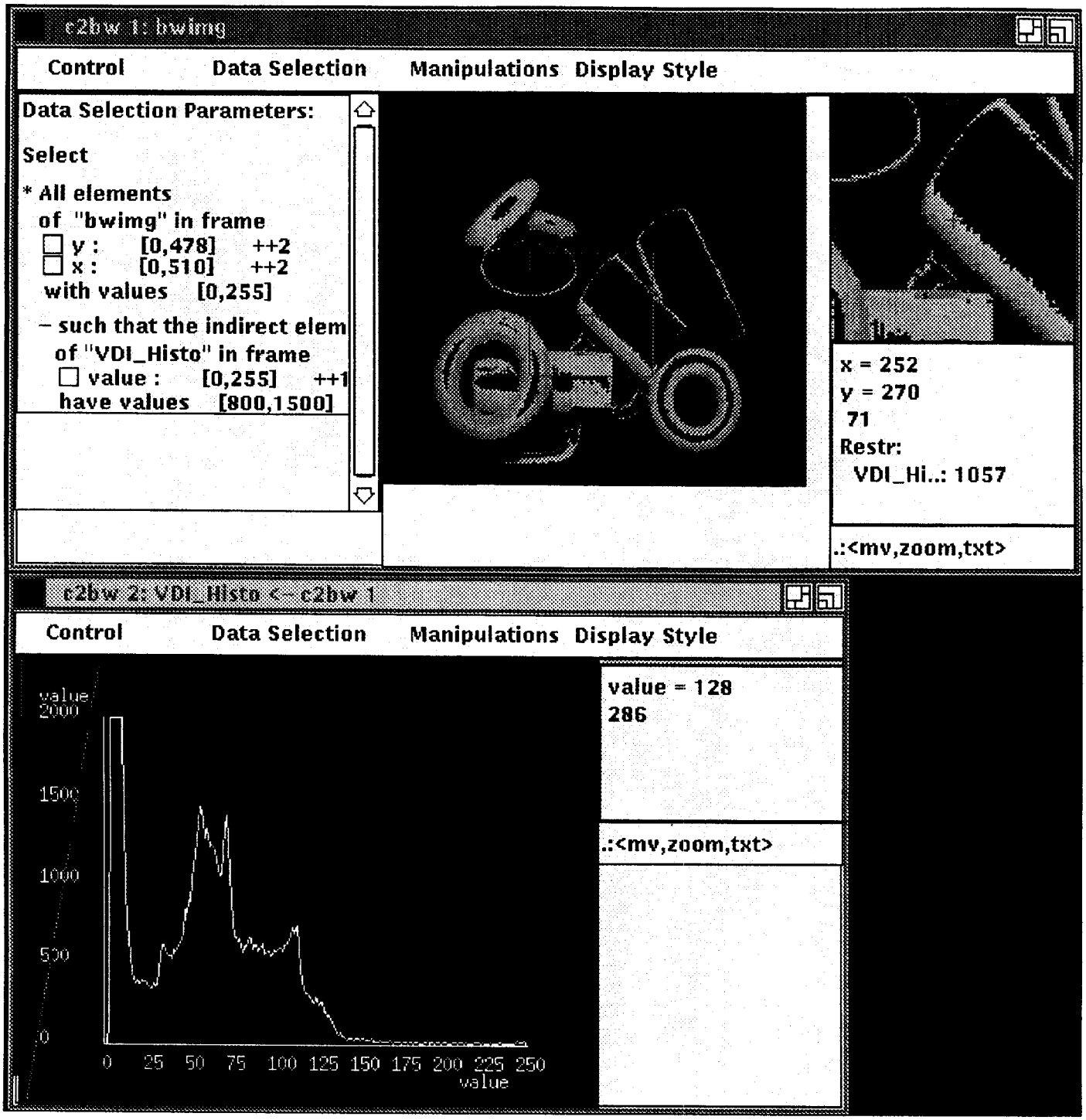

Figure 2-5 Indirect 1-dimensional restrictions on an array

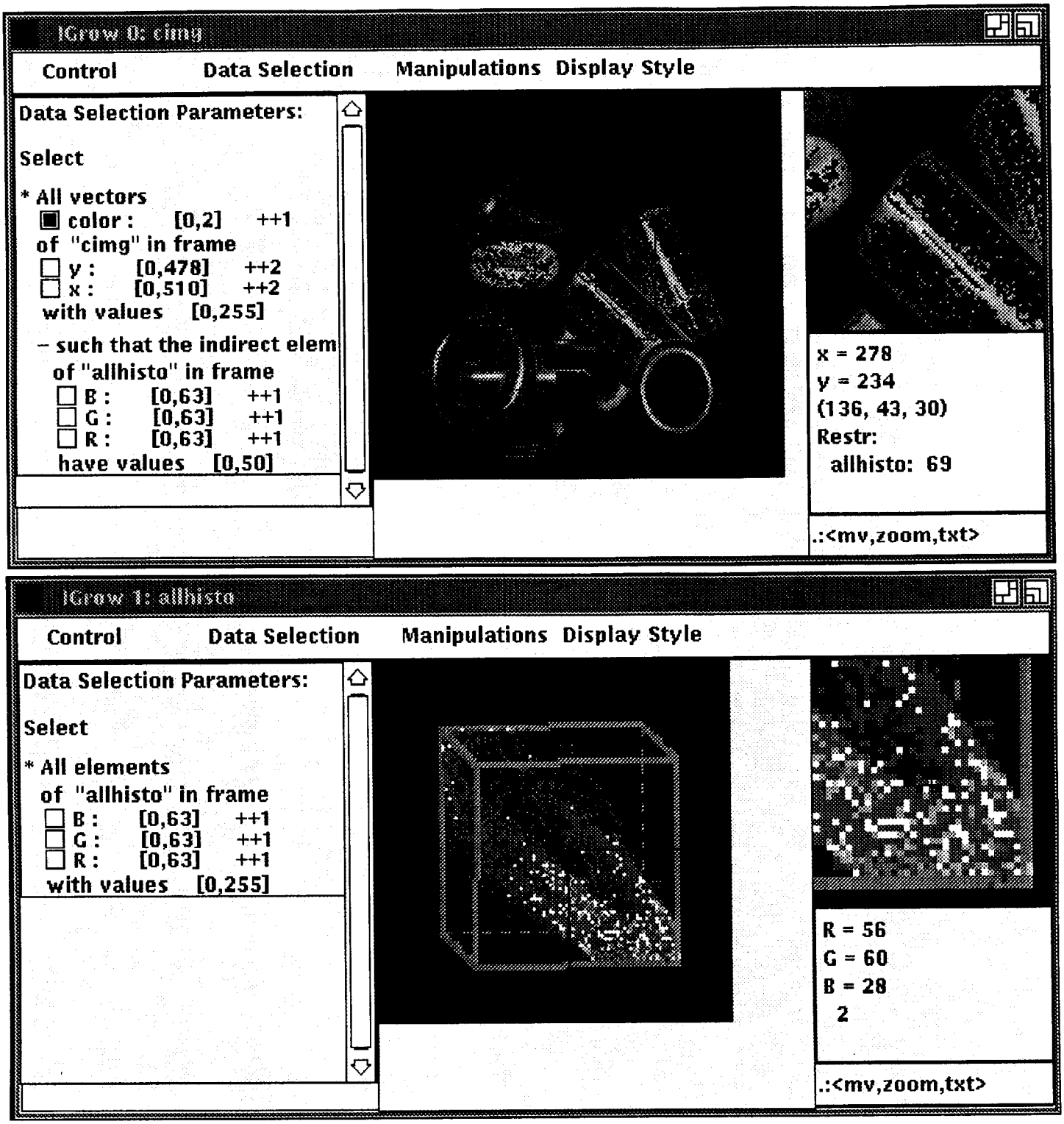

Figure 2-6 Indirect multi-dimensional restrictions on an array

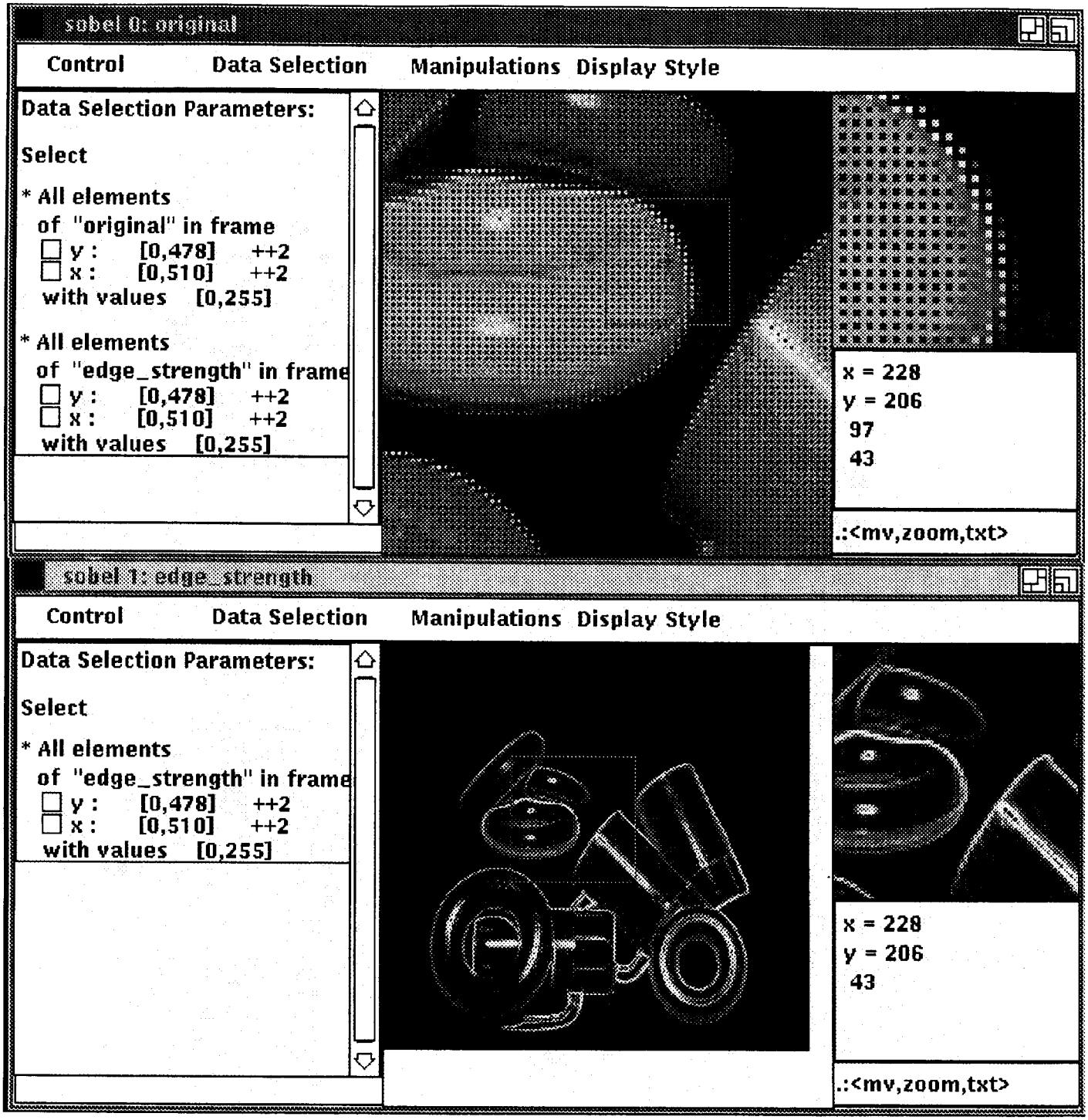

Figure 2-7 Links between arrays

2.1.2.3 Displaying several linked arrays together

Besides imposing restrictions on the display criteria of a primary array, programmers also need to be able to view several images together. They may either want to see them side by side, with cursors on all arrays moving jointly, or they may prefer seeing all arrays overlaid in a single frame. Preferences will depend on the array contents, the preferred graphical representation, and the amount of detail under consideration. For example, vision researchers have in the past overlaid image flow fields as vector fields on original images, or printed data values (pixels, flow vectors, etc.) on top of (largely zoomed) images.

VDI's data selection area provides the tools to specify such overlay requests interactively. Figure 2-7 shows an example of simultaneously displaying the plastic scene and an edge image generated by the Sobel edge operator. The lower half of the figure shows the edge image. In the upper half, VDI displays all pixels of the original image, as well as all pixels of the edge image. In the selected display style, both images are shown as intensity images. They are overlaid by using a 2x2 pixel area per array element. Within this area, the pixel from the second (edge) image is allocated in the upper right bin. The remaining three bins show the pixel value of the first (original) image.[1] Section 2.2 discusses in more detail which different overlay mechanisms could be used under various circumstances, and which ones are realized within VDI.

2.2 Data Display

VDI provides several mechanisms for displaying arrays. Our central interactive tool is a VDI-window. The programmer can interactively open and close a large number of VDI-windows to view any of the arrays of the application program.

2.2.1 Arrangement of Information in a VDI-Window

Figures 2-2 through 2-7 have shown typical VDI-windows: The center area displays the array. If the array is larger than the display area, the programmer can scroll the display area with the left mouse button across the array. As described in section 2.1, the area to the left maintains the current slicing, cropping, subsampling and array linking specifications. A small area in the upper right contains a zoomed view of a subregion of the array. The programmer can interactively scroll the zoomed subregion with the middle mouse button across the array. The subregion is indicated in the center display area by a red square. The data element at the center of the zoomed region (indicated by a cross) is also presented in printed form in the area underneath the zoomed subregion. The printout shows the position of the element on the framing feature axes (e.g., its x,y position), as well as its values along the first contained feature axis (e.g., its red, green and blue color values). Numbers in the following lines represent the related data values in linked or restricting arrays. The programmer can arbitrarily modify this printout in the application program by supplying a specialized printing routine to display any arrangement of pixel-related information in this window.

[1] To avoid aliasing, we have zoomed the display by a factor of 2. Accordingly, every array element now covers a 4x4 pixel area, with the upper right 2x2 pixel showing the edge data. The overlaying characteristics are even more pronounced in the small display area to the right, which is zoomed by another factor of 2.

2.2.2 Interpretations of Mouse Buttons

Depending on the context, VDI interprets the mouse buttons in various ways. The user can interactively select a context by pressing "Ctrl" and one of the keys described in the table below. VDI displays the current context in the lower right corner of each VDI-window.

<Ctrl>x	<mv,zoom,txt>	standard
<Ctrl>t	<mouse,txt,i>	homogeneous rotations of 3D data
<Ctrl>s	<sel,stop,i>	select subarea of the image (crop)
<Ctrl>u	<...,...,...>	user-defined procedure

Section 2.2.1 has described the standard interpretation of mouse buttons, with the left button triggering a scrolling operation on the large display area, the center button controlling the positioning of data in the small, zoomed area, and the right button causing a data display window with printed pixel values to pop up (see section 2.2.3.3). VDI-windows initially operate under this mouse interpretation until the programmer changes it interactively. The default can be reinstalled by pressing "<Ctrl>x".

The programmer can rotate three-dimensional displays by selecting the "<Ctrl>t" mouse interpretation. The display then rotates while the programmer presses the left mouse button and drags it. If the center button is pressed, a text window pops up, asking the user to specify a rotation axis and a rotation angle. The right button resets the display to the initial position.

Window cropping under mouse control can be activated via the "<Ctrl>s" mouse interpretation. When the left mouse button is pressed, one corner of the rectangle is positioned. The opposite corner then follows the dragging mouse, receiving its final value when the button is released. If the center button is pressed in addition to the left button, the cropping procedure is aborted. The right button resets the cropping parameters to show the entire image.

Finally, programmers can specify their own mouse interpretations. This mechanism is mainly used in data manipulation routines provided by an application program. For example, the object masks in Figure 2-4 were generated by a region growing module in a color computer vision program which analyzes color variation of shaded objects with highlights. The module requests that the mouse be positioned in an area of homogeneous object color. When such an area is selected, the module gathers color statistics from a small square area around the mouse position to determine typical color changes. If the left button is clicked, the module collects the statistics and presents them to the programmer, without starting to grow a region (the programmer can "peak" at a candidate starting position). When the middle button is clicked, the module collects the statistics and then goes on to request that another position be picked, this one on a highlight. When this second position has been set with the middle button, the module gathers statistics on color variation in the highlight area and finally starts the region growing process. The right button aborts the module. When the user module starts, the mouse interpretation is automatically changed to the one used in the module. The programmer can interactively switch to other mouse interpretations, e.g. in order to scroll or crop the window. The mouse interpretation supplied by the application program can be reinstalled by pressing "<Ctrl>u".

2.2.3 Graphical Representations of Arrays

Depending on the application and on the current debugging purpose, programmers need to view their arrays in different graphical representations. To gain a global impression of the data, they may want to look at an intensity-based rendering of the data. Under other circumstances, it may be important to get a closer look at the data in specific areas of interest (see Figure 2-5). Such detail may be more easily conveyed through graphs. In some cases, programmers will even want to see the actual numbers of the data values in a selected region.

In VDI, the programmer can interactively request to view array data as intensities, graphs (terrain maps) or printed numbers. Other representations, such as needle maps, have also proven to be useful. Furthermore, interesting research on using audio output and specially designed icons with variable properties (e.g., variable lengths and angles of line segments) is under way[Grinstein et al. 89, Smith and Williams 89]. VDI is a framework through which such display mechanisms can easily be made available to vision programmers. We plan to integrate more representations in the future. We may also provide a mechanism for programmers to register their own, application-specific display routines with their arrays in the application program such that they can interactively switch into such specialized display modes.

One important issue in displaying arrays in the various graphical representations is the question of dimensionality. How many framing features and how many contained features can be represented adequately as intensities, graphs or printed numbers? The graphical representations are limited in how many framing features they can display at the same time. VDI generally supports one-, two- and three-dimensional display facilities, allowing the programmer to interactively rotate three-dimensional renderings. If the programmer specifies higher-dimensional framing criteria, VDI defaults to the *n* lowest lowest framing features, that can be presented. Other considerations apply to the contained features. The graphical representation, the display hardware and the contained feature type determine how many contained array elements can be shown per framing position. The semantics of VDI's data selection menu request in most general terms, that the cross product of all contained feature vectors be shown simultaneously for all linked arrays, with the programmer being responsible for cropping the volume of displayable information to a reasonable size. Such needs can be satisfied to different degrees in the different graphical representations. If VDI cannot show all data, it defaults to the lowest *n* contained features that it can display.

The following subsections discuss the exact limitations on framing and contained features for intensities, graphs and printed numbers.

2.2.3.1 Intensity-based displays

Section 2.1 has already shown many examples of intensity-based displays of two- and three-dimensional arrays. In general, VDI is able to generate intensity displays as vectors, images and volumes for up to three framing features. To support the display of volumes, we provide a very simple translucency concept: currently all voxels with value 0 are considered transparent, all others are considered opaque. This mechanism is good enough to visualize sparse arrays of data, such as the color histogram in Figure 2-6. A more sophisticated concept will be needed in the future.

Intensity-based capabilities for displaying more than one contained feature value are rather restricted. A few specialized techniques for displaying particular kinds of contained feature vectors have emerged in past years, such as color images (with the contained feature being a color vector of three elements), movie loops (with the contained feature being a time vector or a z-axis) [VoxelView 89], and blink mechanisms (for contained feature vectors of two elements) for which the system alternates between displaying a front and a back buffer [Carlbom et al 91]. Some systems also have a special mechanism to display stereo images. Many of these techniques are by now supported by special hardware. VDI exploits existing color display capabilities, analyzing the descriptions of feature vectors supplied by the programmer (or extracted from the file name) when the array is created. Accordingly, VDI switches from a black-and-white color map to a colored one, if the contained feature vector under consideration is a color vector, and if the vector has not been cropped to show only one or two color bands. VDI does not yet support stereo display capabilities and automatic blinking. However, as an aid to view a sequence of frames, VDI does provide an interactive mechanism similar to a movie loop to step along a contained feature axis. The mechanism provides a slider with which the programmer can select frames in arbitrary sequence. These capabilities will be extended as necessary.

In addition to providing specialized display mechanisms for specific kinds of contained feature vectors, we need some general mechanisms for viewing arbitrary contained feature vectors. Several windows with linked cursors could be positioned side by side, each showing one particular slice through the data volume. Alternatively, several contained values at a particular framing position can be interlaced such that each framing position displays all contained values in a small, local area (see section 2.1.2.3 and Figure 2-7 for an example). Currently, VDI provides an interlacing mechanism to overlay several linked images. For every single array, however, it displays only one contained value (or color triple) per framing position (taken from the lowest index of the lowest contained feature). VDI does not yet support cursor linking to display several linked images side by side. The programmer can, of course, display several arrays independently in different VDI-windows, as shown in Figures 2-4 through 2-7. But the cursors have to be positioned individually. We plan to provide a cursor linking capability in the future, along with an interactive mechanism to switch between interlacing and cursor linking. In the most general terms, VDI will need to provide interactive tools for the programmer to specify sets of contained features or linked arrays that need to be interlaced, with each set being shown in a separate window. The programmer should also be able to specify interactively, if particular array links or feature vectors should be attached to special-purpose display mechanisms, such as movie loops, blinking mechanisms or color displays. For example, Figure 2-7 might benefit from showing the original image in red and the overlaid edge image in green.

2.2.3.2 Graphs (terrain maps)

As an alternative to viewing arrays as intensity-based displays, programmers may sometimes want to visualize their data as graphs or terrain maps. This graphical representation is more suitable to conveying detailed information from a small area of the array, whereas the intensity display provides a more global overview.

Since graphs need an extra dimension to display the values of the array elements as heights, graphs can only display up to two framing feature vectors. A single framing feature is shown as a graph, with the horizontal axis representing the feature values, and the vertical axis representing the element values. Two framing features are displayed as a three-dimensional drawing of a

surface (a terrain map). In this case the two horizontal axes represent the framing features, and the vertical axis represents the element values. The programmer can interactively move and rotate terrain maps.

Contained features underlie less restrictions in terrain maps than they did in intensity-based displays: terrain maps of many contained feature values can be shown simultaneously in a single display. VDI supports the display of several terrain maps in a VDI-window. So far, VDI generates terrain maps for all feature values of the lowest contained feature, from all linked arrays. We plan to extend this facility towards displaying several contained features in the future.

Figure 2-8 shows a small area of the color image as a set of terrain maps (center window) and as graphs (lowest window). In this graphical representation, each color band is shown as a separate terrain map or graph. On the screen, the maps are usually shown in different colors, red, green and blue. The maps show a small area on a green cup in the image. In the segmentation image in Figure 2-4, this area had falsely been assigned to the diagonal (orange) cup above. The terrain maps help us understand the problem: The low plateau in the front part of the terrain maps corresponds to the (very dark) planar facet on the green cup that was falsely segmented into a region with the orange cup. Due to the green object color, the green terrain map has the highest pixel values, while the red and blue color bands should have lower pixel values. The blue terrain map (the lowest one) follows the overall shape of the green map, at lowered intensities. However, the red map (the middle one) behaves rather differently. Instead of dropping into a lower plateau, it actually shows higher values that slowly decrease along the x-axis. The optical phenomenon we are encountering here is inter-reflection of light from the orange cup onto the green cup. This orange light is reflected specularly off the green planar facet into the camera, raising the red values disproportionately. Thus, the pixels from this part of the green cup have a much more reddish color than the rest of the cup. The graphs in the lower window show the same phenomenon for a single line along the x-dimension. This is an example of how a programmer can use VDI's interactive debugging capabilities to investigate unexpected results ("problems") when they occur.

So far, we have not yet implemented any cursor-based query mechanism for terrain maps. Accordingly, the small zoomed display area in the upper right corner does not yet show an enlarged version of the terrain map but rather an intensity-based rendering of the entire subarray that the terrain map depicts. Neither does VDI update the printed information underneath the small zoomed display area. We plan to develop a mechanism similar to our technique for printing out data values in a intensity-based volumetric display: if the cursor is positioned directly on a point representing a data value in a terrain map (i.e.. on an intersection point in the net of lines forming a terrain map), the corresponding array element (its framing position, as well as its contained feature values) are printed out. As a visualization aid, we may draw a vertical line through the data point, connecting corresponding contained values in all terrain maps and showing the framing position on the "floor" of the three-dimensional display. When this is implemented, we will also consider providing linked cursors in side by side displays of terrain maps.

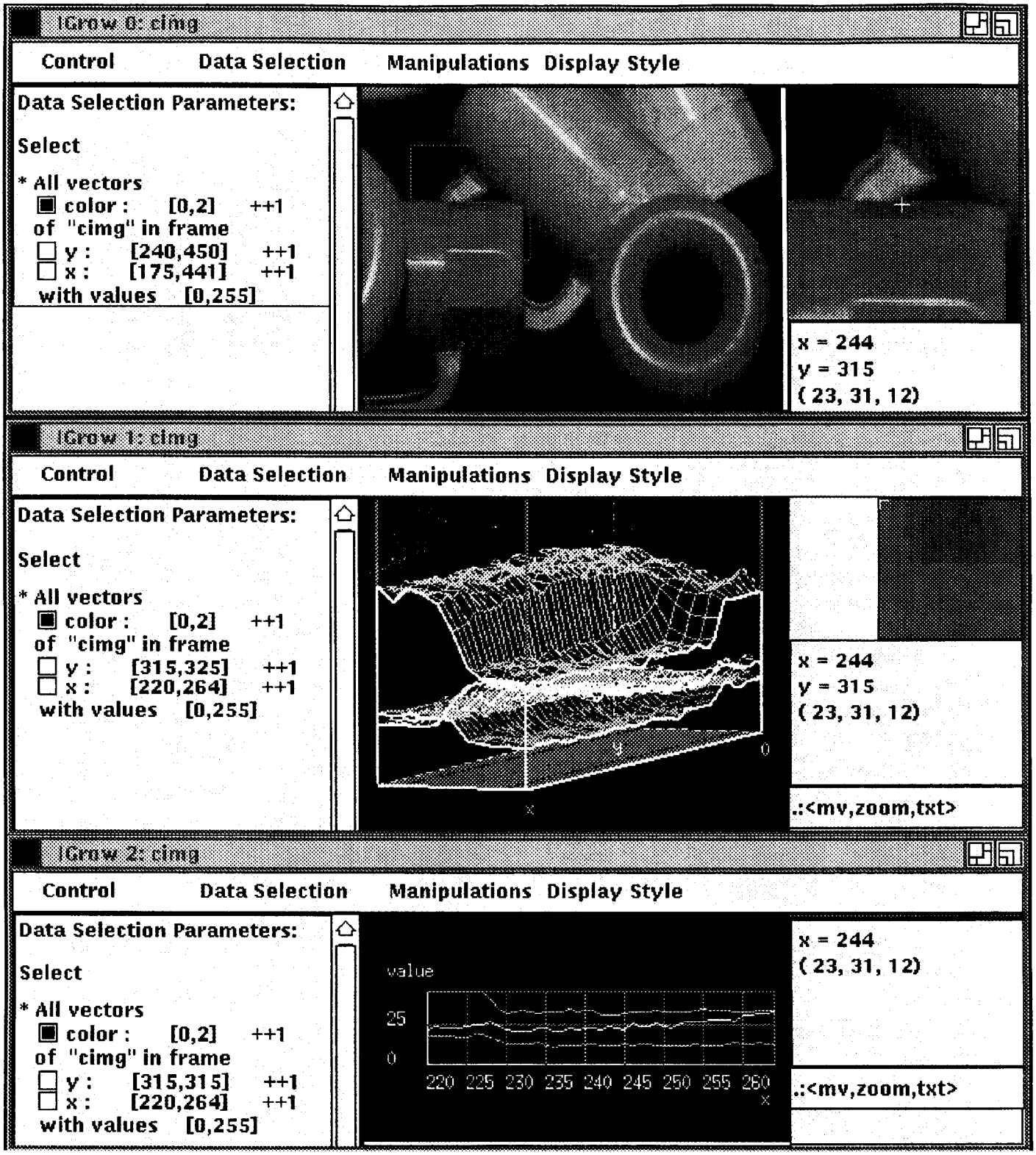

Figure 2-8 Viewing an array as a terrain map or a graph

2.2.3.3 Printed numbers

Layouts of printed numbers on a screen or on a piece of paper are limited to two dimensions. Thus, no more than two framing features can be shown concisely in this graphical representation. However, if the programmer is willing to scroll back and forth between several related positions in a printout, more than two framing features can be shown as separate blocks of data that are arranged one above the other or next to one another.

Contained feature vectors can be handled the same way: if more than one contained feature value exist, each feature value can be shown in a separate table. This layout style is similar to the cursor-linking mechanism discussed above. Alternatively, contained feature values (as well as any framing feature values in excess of two) can be interlaced. Each table entry then becomes an n-tuple of information, with n being the cross product of all interlaced feature vectors. Clearly, this solution underlies the same layout restrictions as "cursor-linking": more than two dimensions of contained features will be difficult to show within a well-structured layout. Interlacing seems to be most useful, when restricted to one dimension, or when it is used to interlace corresponding array elements from several linked arrays.

VDI's current layout strategy exploits the observation that people tend to be interested in only a small data area around a critical point of interest and that entire printouts of large arrays by far exceed the display capabilities of screens and paper. Accordingly, VDI restricts its printout to a small square subarea of the array, centered around the current cursor position. An interactive parameter, "data window size" defines a cropping criterion for all framing features. We do not crop the contained features since the programmer cannot position the cursor with respect to these feature vectors. The programmer can select an area of interest with the cursor by pressing the right mouse button. When an area is selected, VDI pops up a new data display window and prints the values of all array elements in the subarea. The programmer can then study the printout while keeping an eye on the "overall picture" in the intensity-based display shown separately. If requested, VDI copies the printed numbers to a file such that the programmer can obtain a hardcopy. This practice differs from VDI's mechanism of interactively switching between an intensity-based display and a terrain map. We expect that programmers will generally be interested in checking the array data in several different locations with this mechanism, scrolling the data window across the array by keeping the right mouse button pressed. It would be inconvenient for them to switch graphical representations each time. We may consider popping up a separate window for terrain maps, too, if that proves to be a much-needed shortcut for finding the appropriate small subarray for a terrain map from an intensity-based overview.

VDI's layout routine accepts an appropriately cropped n-dimensional block of data. It prints it using the following guidelines: Considering only non-trivial feature vectors that are not cropped to size 1, it orders the feature vectors, starting with the lowest framing feature and ending with the highest contained feature. The lowest feature is printed horizontally. The second lowest feature is laid out vertically. These two features define the basic arrangement of values in tables (blocks). If the next feature is small enough, VDI arranges it horizontally, forming a sequence of blocks (see the arrangement of color tables in Figure 2-9). All further features are presented vertically in consecutive tables, with appropriate labels for the feature values of the higher dimensions. In each table, elements that fit the current cursor position are marked by stars. We plan to provide an interlaced layout style as alternative to the generating separate tables for

higher dimensions. Programmers may also want to be able to provide VDI with their own, specialized layout routines. Finally, the current layout does not yet support data linking.

Figures 2-9 through 2-12 show examples of VDI's data printing technique. Figure 2-9 shows the typical arrangement of a color image, with *x* and *y* as framing features and *color* as the contained features. For a data window of size 3, the *x* and *y* vector are cropped to three elements each, yielding a 3x3-sized table. Different tables next to one another correspond to different color values, 0 (red), 1 (green) and 2 (blue).

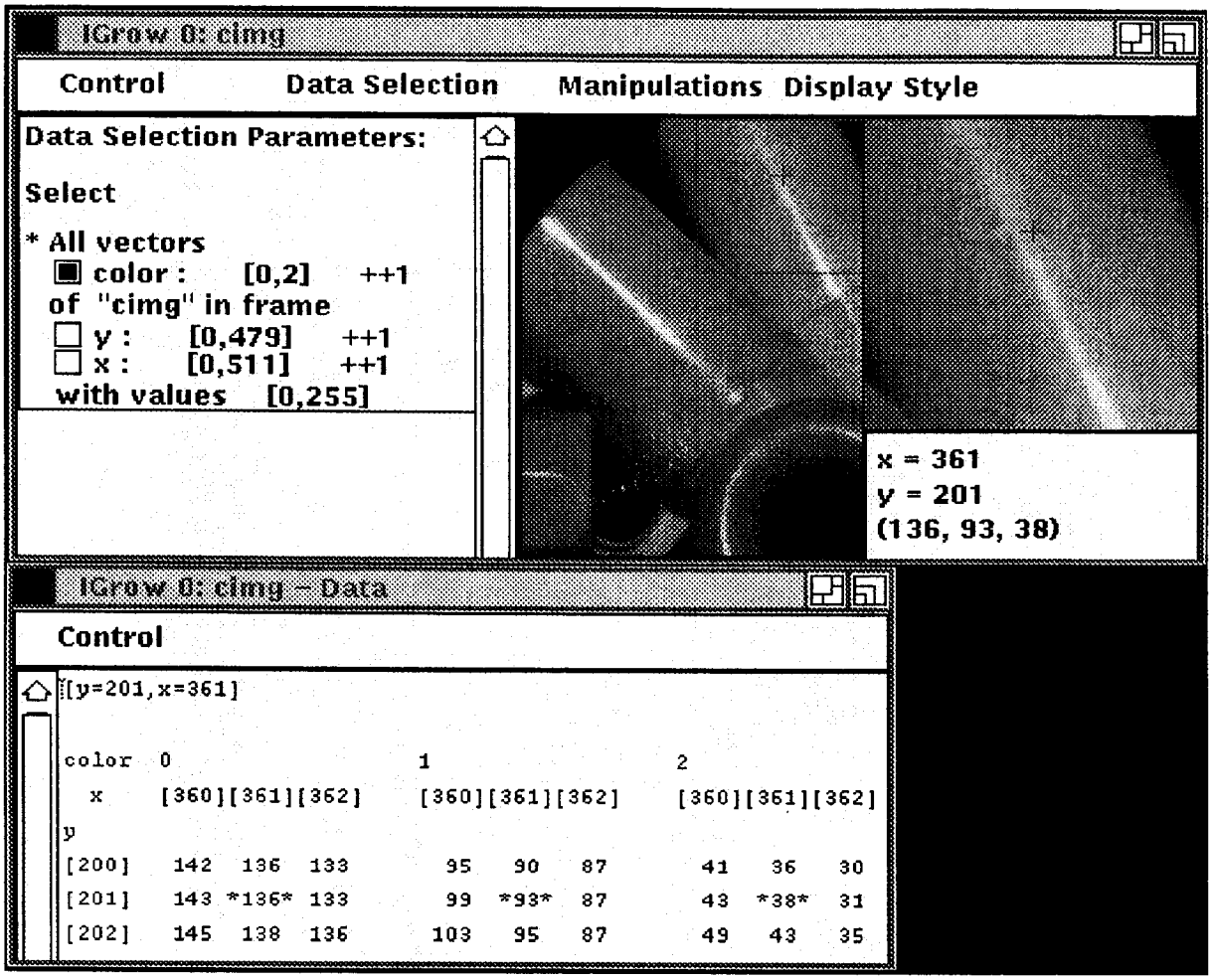

Figure 2-9 An x-y-based layout of printed numbers from a color image

In Figure 2-10, only the *y*-dimension has been selected as a framing feature. Accordingly, the intensity display shows a one-dimensional array (which has been zoomed by a factor of 4). The data printout shows *y* laid out horizontally, cropped to a data window of size 3. The first contained feature, *x*, is printed vertically. It has not been cropped beyond the cropping criteria specified in the data selection menu. The second contained feature, *color*, is small enough to fit as three separate horizontal tables onto the screen.

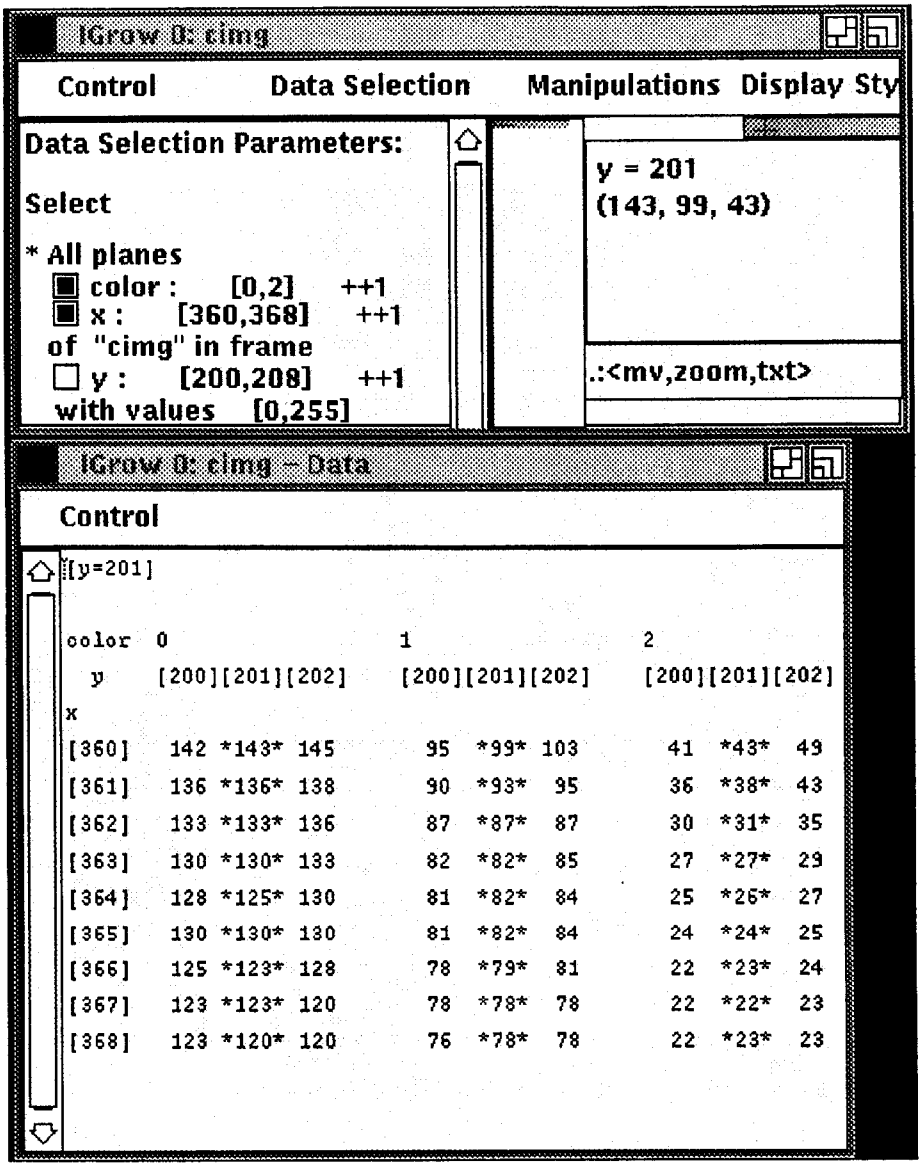

Figure 2-10 A y-based layout of printed numbers from a color image

Figure 2-11 shows the color image, with none of the features being selected as framing features. In this case, the frame is reduced to a single element, containing a three-dimensional space of information. Accordingly, the intensity-based display shows a single square which is related to the color triple at the first index of *x* and *y*. The printed data shows the entire three-dimensional space of contained information. The first contained feature, *x*, is laid out horizontally, the second one, *y*, vertically. Since the *x*-vector contains quite a few (9) elements, three of such tables don't fit horizontally onto the screen/page and the *color* vector is thus laid out vertically.

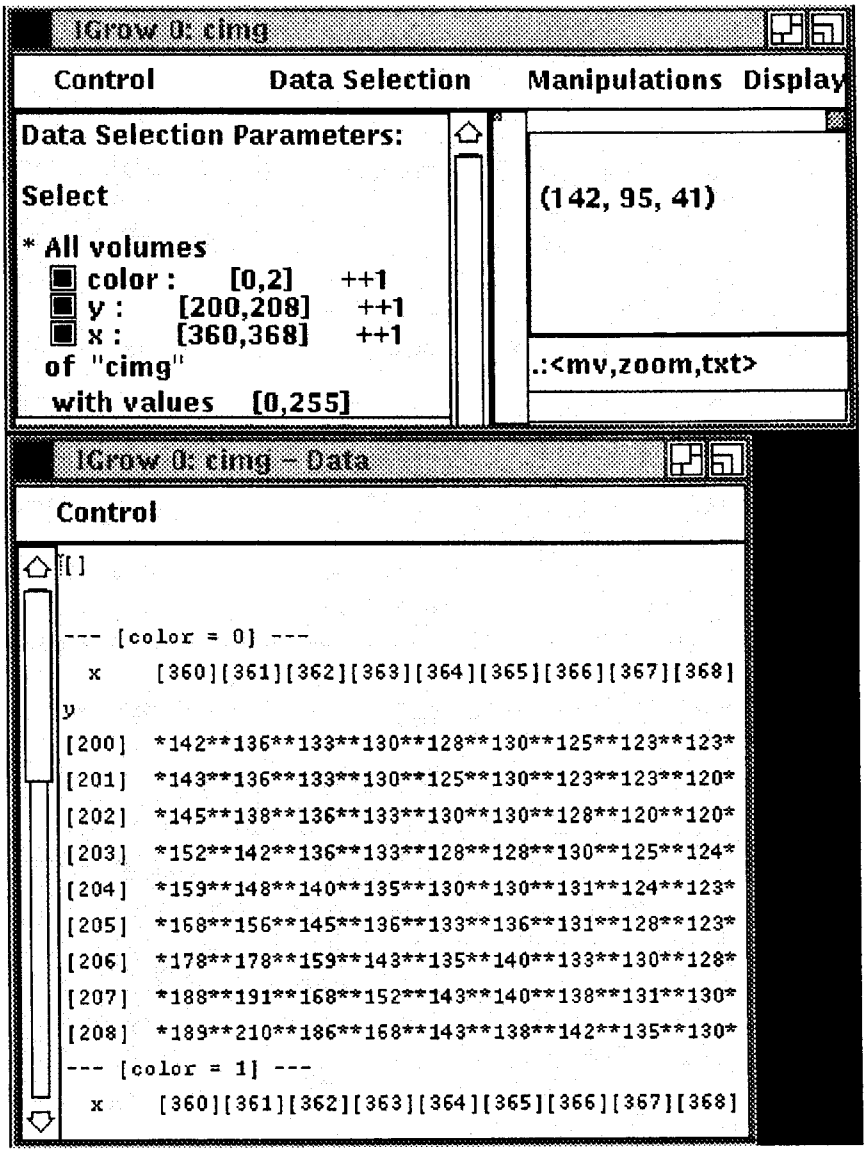

Figure 2-11 A null-based layout of printed numbers from a color image

Figure 2-12 picks up on the discussion in section 2.2.3.2 concerning inter-reflection from the orange cup onto the green cup. Using a data window size of 20, we now have printed all pixel values (and more) that were shown in terrain maps and graphs in Figure 2-8. The figure presents part of the red color band, exhibiting an increase along the x-dimension in the appropriate data area (for $y >= 315$).

IGrow 0: cimg - Data

Control

[y=315,x=240]

--- [color = 0] ---

x / y	[222]	[224]	[226]	[228]	[230]	[232]	[234]	[236]	[238]	[240]	[242]	[244]	[246]	[248]	[250]	[252]	[254]	[256]	[258]	[260]
[297]	6	7	6	8	6	7	9	41	75	84	85	51	49	49	44	38	40	41	46	48
[299]	6	7	6	6	8	12	43	79	73	110	125	82	57	52	49	41	38	38	42	46
[301]	6	6	7	10	24	75	81	61	96	125	125	117	96	54	54	48	38	34	35	41
[303]	7	7	6	19	66	66	55	84	120	130	135	130	125	92	58	50	43	28	30	37
[305]	6	6	11	51	68	55	97	117	119	125	135	131	138	130	102	68	55	29	27	30
[307]	7	8	30	68	62	107	142	123	125	128	135	130	135	110	89	58	35	24	24	27
[309]	6	13	54	62	96	136	156	130	125	128	125	99	85	59	39	26	16	15	21	25
[311]	7	25	60	71	110	135	138	133	125	106	84	57	42	26	14	6	5	5	11	21
[313]	13	15	19	46	92	78	90	102	84	62	43	29	16	9	7	8	7	8	6	12
[315]	23	23	25	27	22	21	23	21	23	*23*	22	23	26	24	24	26	27	28	28	31
[317]	16	18	18	23	19	19	21	21	21	22	23	23	24	26	26	26	29	29	29	32

Figure 2-12 Printed numbers for an inter-reflection area in the color image

2.2.3.4 Combinations of several graphical representations

Beyond switching between several graphical representation in a VDI-window, we expect programmers to be interested in overlaying several linked arrays in different graphical representations. As a typical example, programmers have in the past overlaid optical flow fields as vector fields on suitably enlarged intensity displays of the original image. Even printed numbers of several linked arrays (such as original intensity data, image gradients or curvature, and flow estimates, shown as a table of records) have been superimposed on an enlarged intensity rendering of the original image. Other combinations of several graphical representations can be easily imagined. So far, VDI does not yet provide the means for such mechanisms. But the concepts of VDI can easily be extended to accommodate such requests.

2.2.4 Overlays of symbolic data

In addition to seeing array data, programmers may also want to view symbolic information, overlaid on the images. For example, an image interpretation program might compute a bounding box or the center point of a segmented image area, and the programmer might want to display this information on the original image or draw the boundaries of a segmented region. VDI provides basic programming tools for a programmer to write drawing routines and to connect them with VDI. The programmer specifies the drawing commands with respect to the array coordinate system. VDI scales, scrolls, rotates and projects the drawings at run-time, according to the interactively selected subsampling, cropping and zooming preferences of the user.

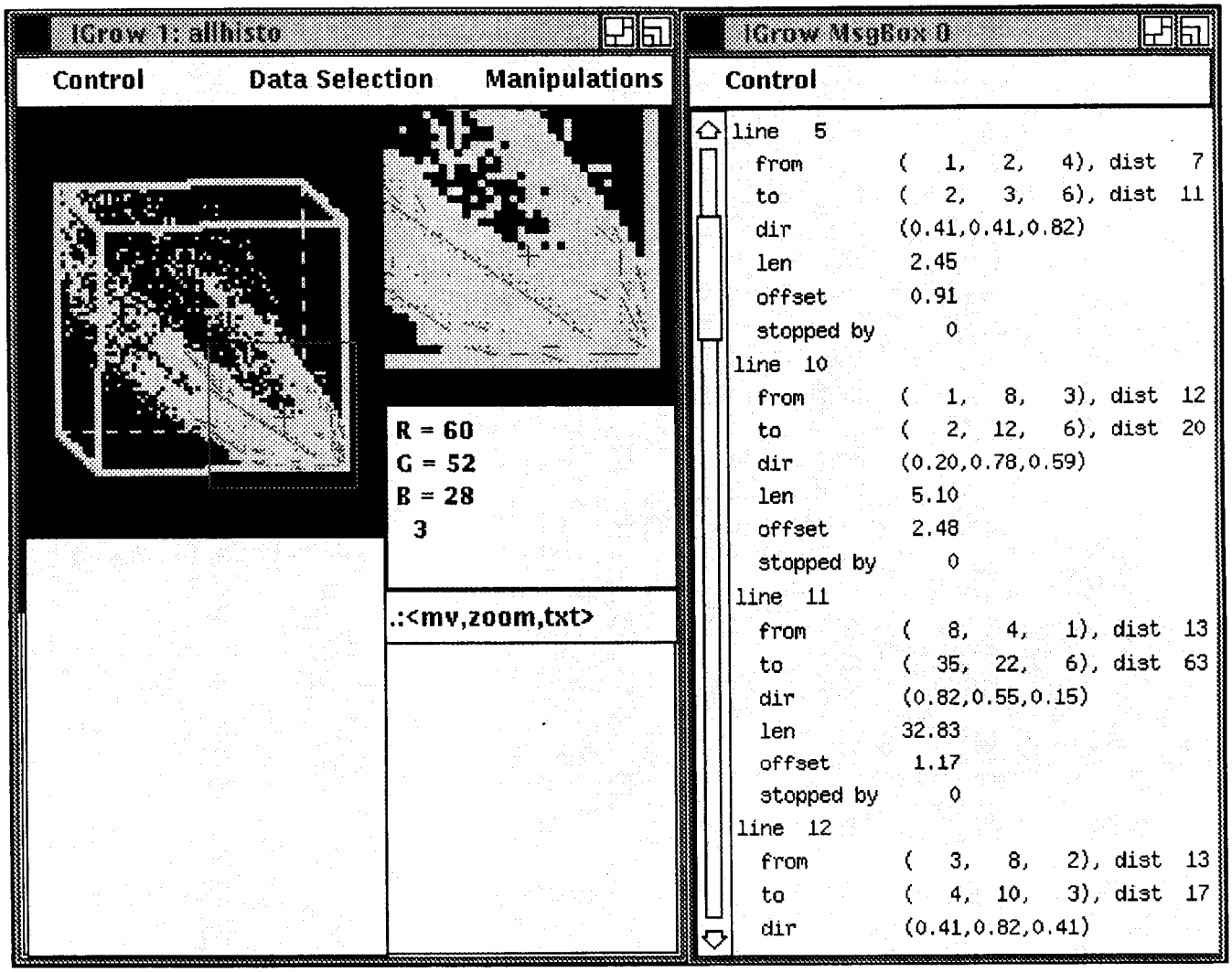

Figure 2-13 Graphical overlay of symbolic data on an array

Figure 2-13 shows an example of the use of overlays: A module in the color computer vision program has tried to determine major colors and their typical color variations in the image of the plastic scene (Figure 2-2) by searching the color histogram (Figure 2-6) for large linear clusters. Overlaid on the color histogram in Figure 2-13 are lines and points, indicating the spines of linear color clusters, as detected by this module. The text window to the right provides

more information about the line segments. For example, line 11 is the long diagonal line running through most of the middle cluster, a yellow color cluster. It contains the pixels of the rightmost diagonal cup in Figure 2-2.

2.2.5 Display Styles

In addition to choosing between several graphical representations, programmers can manipulate the display style in several ways. Figure 2-14 shows VDI's menu for setting and changing various parameters which influence the display style.

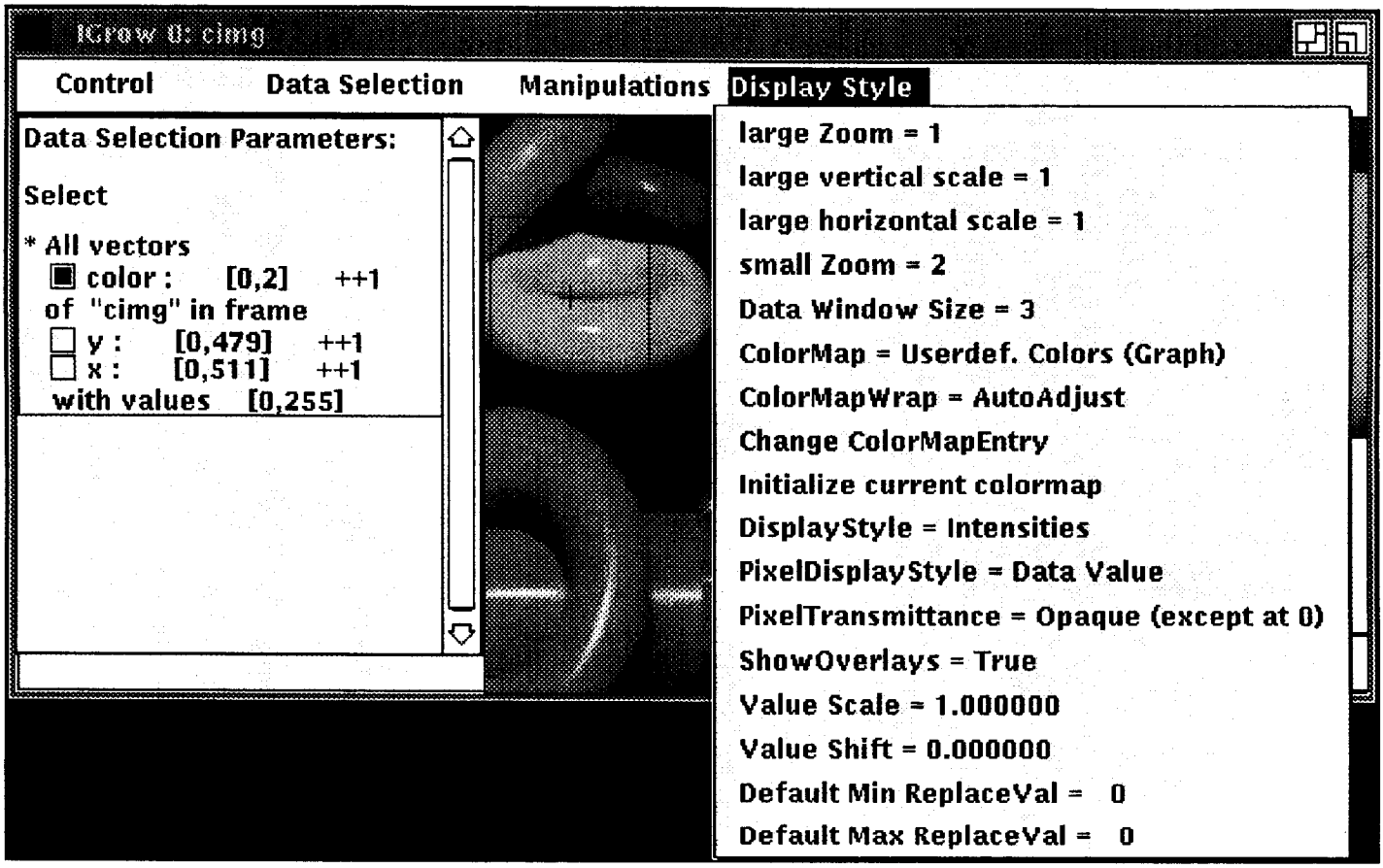

Figure 2-14 VDI's menu for selecting display styles

The first four entries handle different aspects of zooming the large and small array display areas. The large zoom and the large vertical and horizontal scale factors influence the scaling of the large display area. The final vertical scale factor is the product of the zoom factor and the vertical scale. The final horizontal scale factor is computed similarly. Scaling of the small display area is the product of the small zoom factor and the respective large zoom and scale factors. VDI currently zooms by simple pixel replication.

The "Data Window Size" parameter specifies the size of the region for which the data elements are printed, as shown in Figures 2-9 through 2-12. The parameter is used to define a square clipping box centered at the current cursor position.

The next four entries are related to different aspects of setting and changing color maps. VDI maintains a large collection of color maps, such as grayscale and color displays, false (pseudo) coloring and random coloring, as well as two maps that can be changed interactively, either via a table of color names or via a graphical interface in which the relationship between pixel values and displayed values is defined by a red, a green and a blue graph. Each graph is a piecewise linear function. The programmer can interactively create new knots to break the graphs into smaller linear segments or move or delete existing knots (see Figure 2-15).

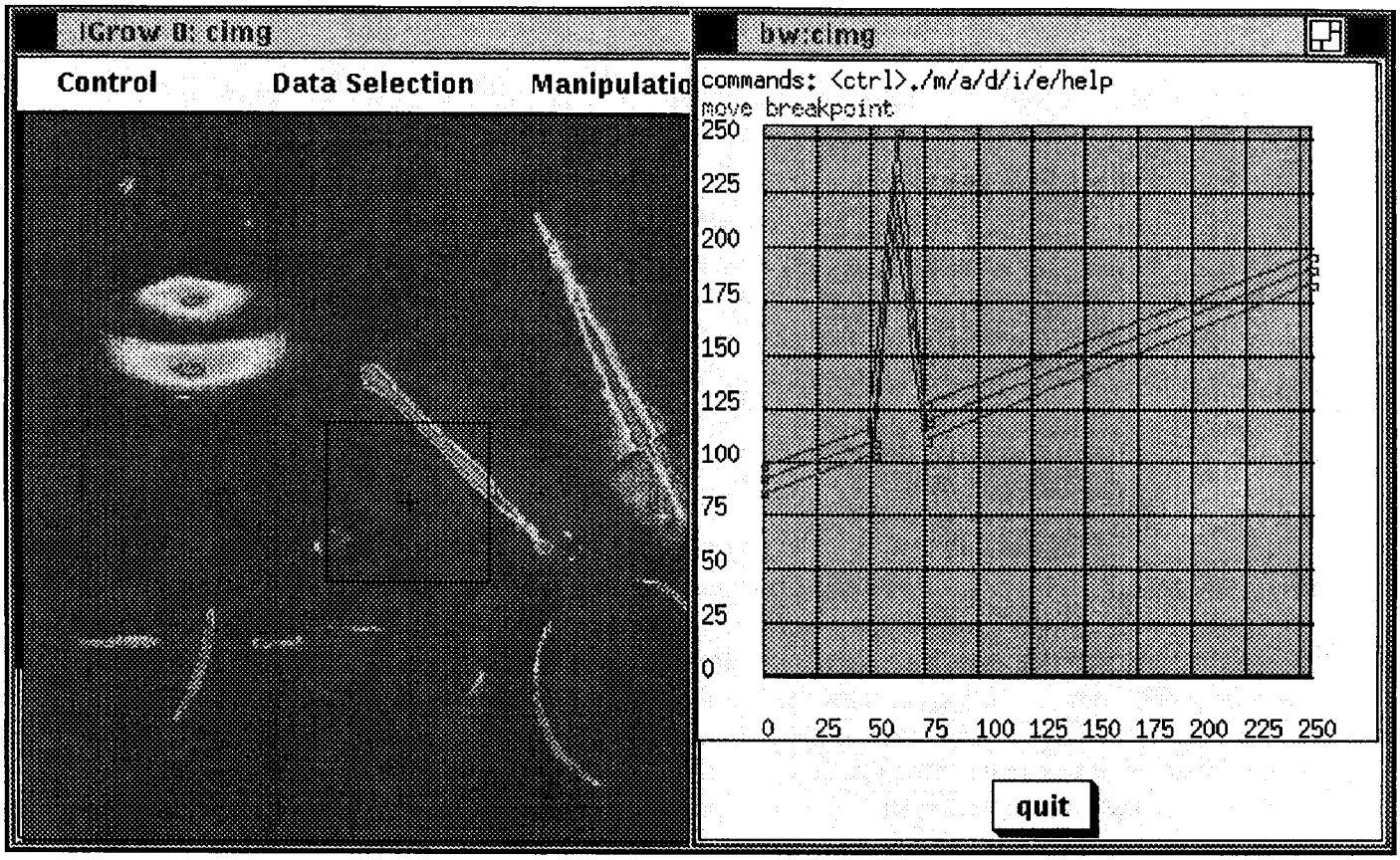

Figure 2-15 Graphical interface to change color maps

The tenth entry, "DisplayStyle", lets the programmer switch between an intensity-based graphical representation and a terrain map.

The parameter "PixelDisplayStyle" specifies whether the displayed colors or intensities are to encode positional information (i.e., the framing position of a data element) or content information (i.e., the values along the contained feature axes). The option of encoding positional information can be useful for viewing high-dimensional sparse arrays (e.g., clusters in high-dimensional feature spaces, such as color histograms) where it is more interesting to see which frame positions are occupied than to know the exact value.

"Pixel Transmittance" is used for displaying three-dimensional array information. So far, VDI accepts only *opaque* and *opaque (except at 0)* as transmittance parameters. The latter option is useful for visualizing sparse arrays, such as the color histogram in Figure 2-6. We plan to include a translucency concept in the future, including interactive capabilities for specifying

translucency as a function of position (depth along the viewing axis) and/or as a function of the array element values.

VDI's mechanism for graphically overlaying symbolic information on arrays can be turned on and off with the "Show Overlays" button.

The next two entries, "Value Scale" and "Value Shift" specify whether and how to modify the array values before they are displayed, using the formula,

$$\textit{Displayed Value} = \textit{Array Value} * \textit{Value Scale} + \textit{Value Shift}.$$

This transformation from array values to displayed values is useful for viewing integer or floating point arrays which need to be mapped into bytes before being displayed.

Finally, the "Default Min Replace Value" and "Default Max Replace Value" parameters are related to the array restriction mechanism (see section 2.1.2). They specify which value will be displayed, if a pixel does not meet all conditions specified in the data selection area. If the pixel is below the acceptable value range, it is replaced by the minimum replacement value. If it is above the range, it is replaced by the maximum replacement value. We plan to extend this mechanism from specifying two parameters to defining a lookup table such that a separate replacement value for every display value can be set. Such capability can be used for dimming all restricted values or for selectively showing subsets of restricted values in different colors.

2.3 Data Manipulation

VDI provides a few very basic data-manipulation routines, such as file input and output, copying, histogramming and projecting arrays. However, we expect that in most higher-level data interpretation programs, data manipulation is tightly controlled by the application program and that it depends in complex ways on the contents of many related arrays and on the overall state of the data interpretation. For most purposes, we thus expect programmers to write their own manipulation routines and control structures. The programmer can register data manipulation routines with VDI so that the routines can be executed interactively. This provides the mechanism for supplying libraries of standard, general-purpose data interpretation routines for interactive use. It also opens the door for tentative execution of data manipulation routines, such that the programmer can ask questions such as "what would happen, if I applied the following operation to my data in the current program state?"

This approach has already proven to be very useful for analyzing highlights in color images. In this application, the programmer has provided application-specific image analysis modules that determine areas with shaded objects and highlights in images and then remove the highlights from the objects [Klinker et al. 90]. Figure 2-16 shows the array manipulation menu that is provided for this application. The first 8 menu entries (above the first separating line) are general-purpose tools that VDI always provides. All remaining entries represent modules that were provided by the application program. Using this menu, the modules can be called at run-time to operate on the image data, exploiting all data selection constraints that have been specified interactively. Modules can thus be called to operate on the entire image or on a small, subsampled and cropped subpart, or on masked image data (see Figures. 2-4 through 2-6). Some of the modules also request that a starting pixel be specified interactively (see section 2.2.2).

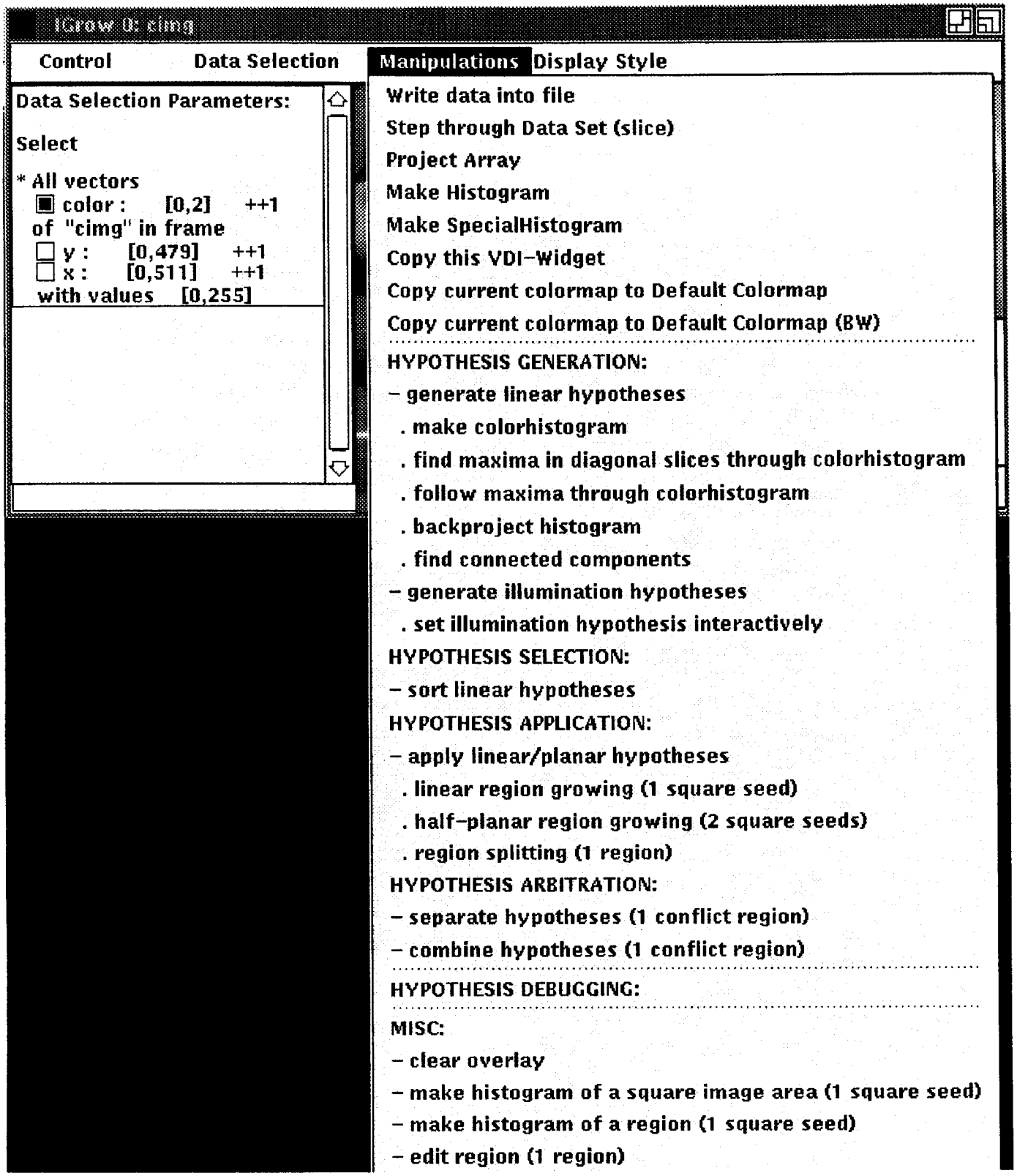

Figure 2-16 VDI's menu of data manipulation operations

3 The Program Interface

The intent in building VDI was to provide programmers with a library of graphical debugging routines that can be included with minimal effort into an application program. It is the goal to keep the program interface as simple and small as possible. Ideally, the VDI-package should be an integral component of a standard debugger or of an integrated programming environment such that it can be called at runtime, without any changes to the application code.

VDI's program interface uses a special data structure, a VDI_Array[1], to communicate with application programs. This data structure contains a pointer to the data, as well as header information that describes the dimensionality of the array and the ranges and names of the feature axes. VDI provides procedures and macros for initializing, reading in, writing out, copying and accessing VDI_Arrays.

VDI internally maintains a list of all arrays that an application program registers with VDI. An application program can announce arrays to VDI by either explicitly registering them or by requesting that they be displayed. In the first case, VDI only enters the array into its list. In the second case, VDI enlists it and also pops up a new VDI-window that displays the array. In either case, the array is thereafter available for interactive viewing, i.e.. the programmer can interactively request that it be displayed in a new, pop-up VDI-window or that it be linked to other arrays in already existing windows.

In order for VDI to be a useful debugging tool, it is essential that VDI can be called from any place in the application code and that the application program can continue its computation when the programmer exits from VDI. The most common programming style in X-based windowing systems does not support such needs: usually, window-based programs enter an infinite loop, waiting for and dispatching events on an event queue, as initiated through programmer input. Unlike such typical X-based systems, VDI can be called from an application program as a subroutine, returning control to the application program when the programmer chooses to terminate the interactive session with VDI or when VDI runs out of work (its event queue is empty). The application program can specify which of these two exit modes to use when it calls VDI. We also provide an interrupt mechanism via which an application program can test whether the programmer has pressed a special interrupt button to halt the application.

Figure 3-1 shows an example of how programs operate on VDI_Arrays and how they can interact with VDI. The example reads in a two-dimensional image and displays the image in a VDI-window. It registers an array-inversion routine, '`my_InvertImage2d`', with VDI and then calls VDI to start an interactive debugging session. When the user selects the array-inversion routine from the manipulation menu, the module inverts the image, according to the user-specified subsampling, cropping and masking criteria. It periodically (in ProgressIncrements of 10 rows) reports progress to the user by displaying the current contents of the inverted image. The user can interrupt the inversion process to inspect the progress.

[1] The same data structure is also used in another, related project in the Visualization Group of the Cambridge Research Lab which provides parallel programming support for homogeneous data manipulation (SIMD).

```
#include "VDI.h"
VDI_Array Img,InvImg;

main(argc,argv)
int  argc;
char *argv[];
{
     void my_InvertImage2d();
     if (argc < 2) {
          printf ("Usage: InvertImage2d <filename>\n");
          exit(1);
     }
     VDI_Init("InvertImage2d",&argc,argv);
     VDI_OpenArray(&Img,argv[1],"Img");
     VDI_DisplayArray(&Img,NULL);
     VDI_RegisterProcedure("InvertImage2d",my_InvertImage2d,0,0,0,VDI_LOCAL);
     VDI_Run(VDI_UNTIL_USER_EXIT,VDI_REMOVE_WINDOWS);
}
void my_InvertImage2d(n,ProcType,ControlTag)
int n,ProcType,ControlTag;
{
     VDI_Array     this_img,restricting_mask;
     VDI_RestrList restrictions;
     int           x,xs,xe,xinc,y,ys,ye,yinc,maxval,
                   ProgressIncrement=10,InitVal=UNKNOWN;

     VDI_GetWindowHeader (n,&this_img,&restrictions,"");
     if (VDI_Rank(this_img) != 2) {
          printf("my_InvertImage2d: two-dimensional array expected\n");
          return();
     }
     VDI_CompileRestrictions
       (&this_img,&restricting_mask,&restrictions,VDI_AND_NFS);
     xs   = VDI_FeatStart(this_img,1);  ys   = VDI_FeatStart(this_img,0);
     xe   = VDI_FeatEnd(this_img,1);    ye   = VDI_FeatEnd(this_img,0);
     xinc = VDI_FeatStride(this_img,1); yinc = VDI_FeatStride(this_img,0);
     maxval = VDI_ElemEnd(this_img);
     VDI_CopyArrayHeader(&this_img,&InvImg,"InvImg");
     VDI_AllocArrayData(&InvImg,InitVal);
     VDI_EnableInterrupt("Interrupt Img Inversion");
     for (y=ys; y<=ye; y+=yinc) {
          for (x=xs; x<=xe; x+=xinc)
               if (VDI_ucElement2D(restricting_mask,x,y))
                    VDI_ucPutElement2D
                        (InvImg,x,y,maxval-VDI_ucElement2D(this_img,x,y);
          VDI_ShowProgress(InvImg,y,ProgressIncrement,NULL);
          if (VDI_Interrupt()) VDI_Run(VDI_UNTIL_USER_EXIT,VDI_KEEP_WINDOWS);
     }
     VDI_DisableInterrupt();
     return();
}
```

Figure 3-1 Example: Program to invert a two-dimensional image

4 Other Applications for VDI

VDI is a very general toolkit that can be useful in many application areas other than image interpretation. Any application using multi-dimensional arrays can benefit immediately from using VDI, simply by calling the VDI library during an interactive session with a debugger, such as Saber-C, or by using VDI from inside the program or at the shell-level by using a simple program, "show_array". VDI may be especially useful for analyzing and debugging programs that run on massively parallel (SIMD) machines. Beyond applications that are already using multi-dimensional arrays, many applications exist that do not traditionally present their results in array form, although the data can easily be converted to arrays. For example, many software development tools exist which gather statistical data on the performance of programs, indicating where programs are spending/wasting their time. To demonstrate VDI's general usefulness, we have converted such trace data from two software tools into array form.

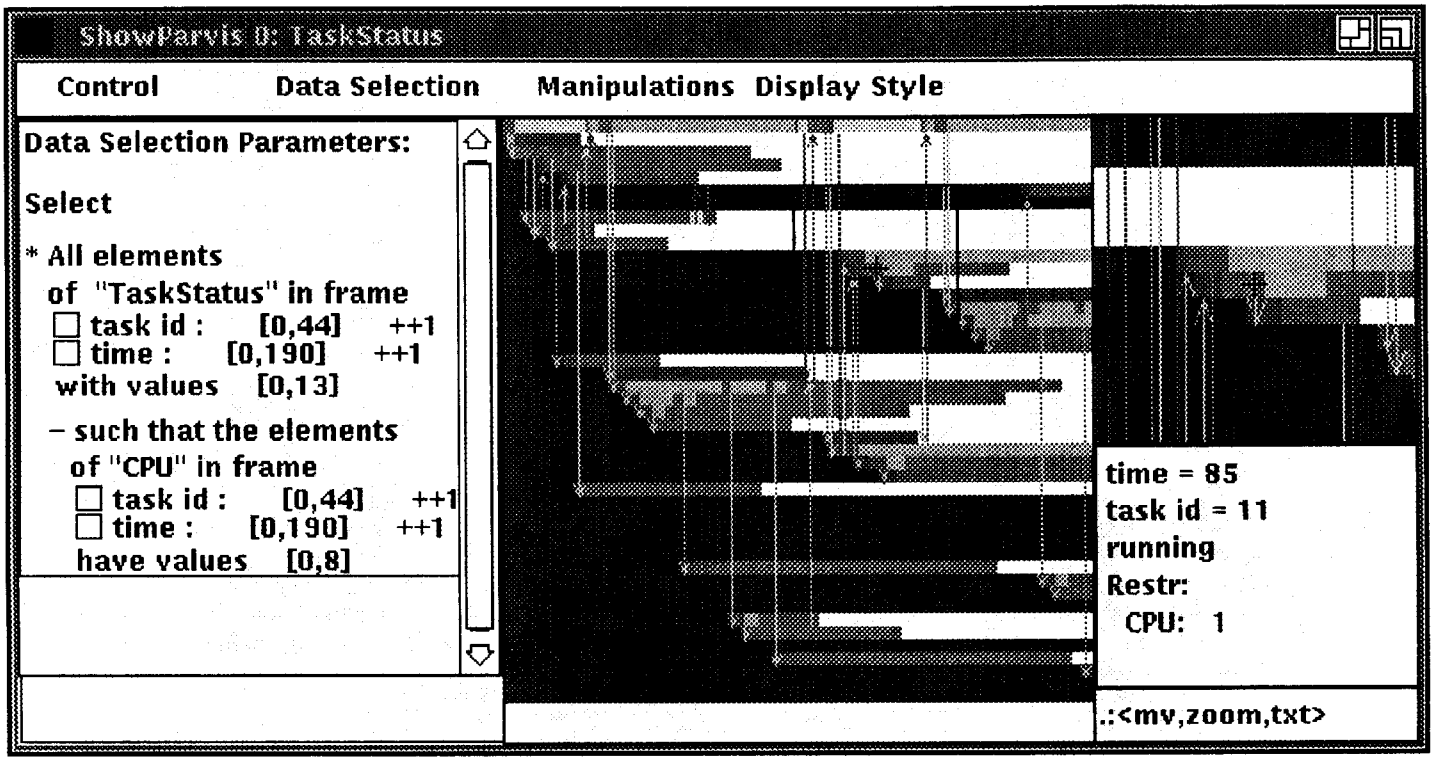

Figure 4-1 VDI as a tool for visualizing constraints between parallel processes on a MIMD-machine

Figure 4-1 shows a program trace of a program executing as several parallel processes (tasks) on a MIMD-machine [Halstead and Kranz 90], displayed in a VDI-window. The course of time is laid out horizontally, while the vertical axis shows different tasks. The array itself, called "TaskStatus", shows when the tasks are created, run, blocked and terminated during program execution. The overlaid vertical lines illustrate several kinds of dependencies between the tasks.[1] A second array, "CPU" of the same dimensions states which task runs on which processor at any given point in time. This array can be used to restrict the TaskStatus array such that

[1] On the display screen, the different task states and the dependencies between tasks are shown in different colors.

the programmer can focus on scheduling patterns observed on subsets of processors. The small, zoomed display area and the text area underneath indicate that task 11 is running on processor 1 at time 85, creating process 12.

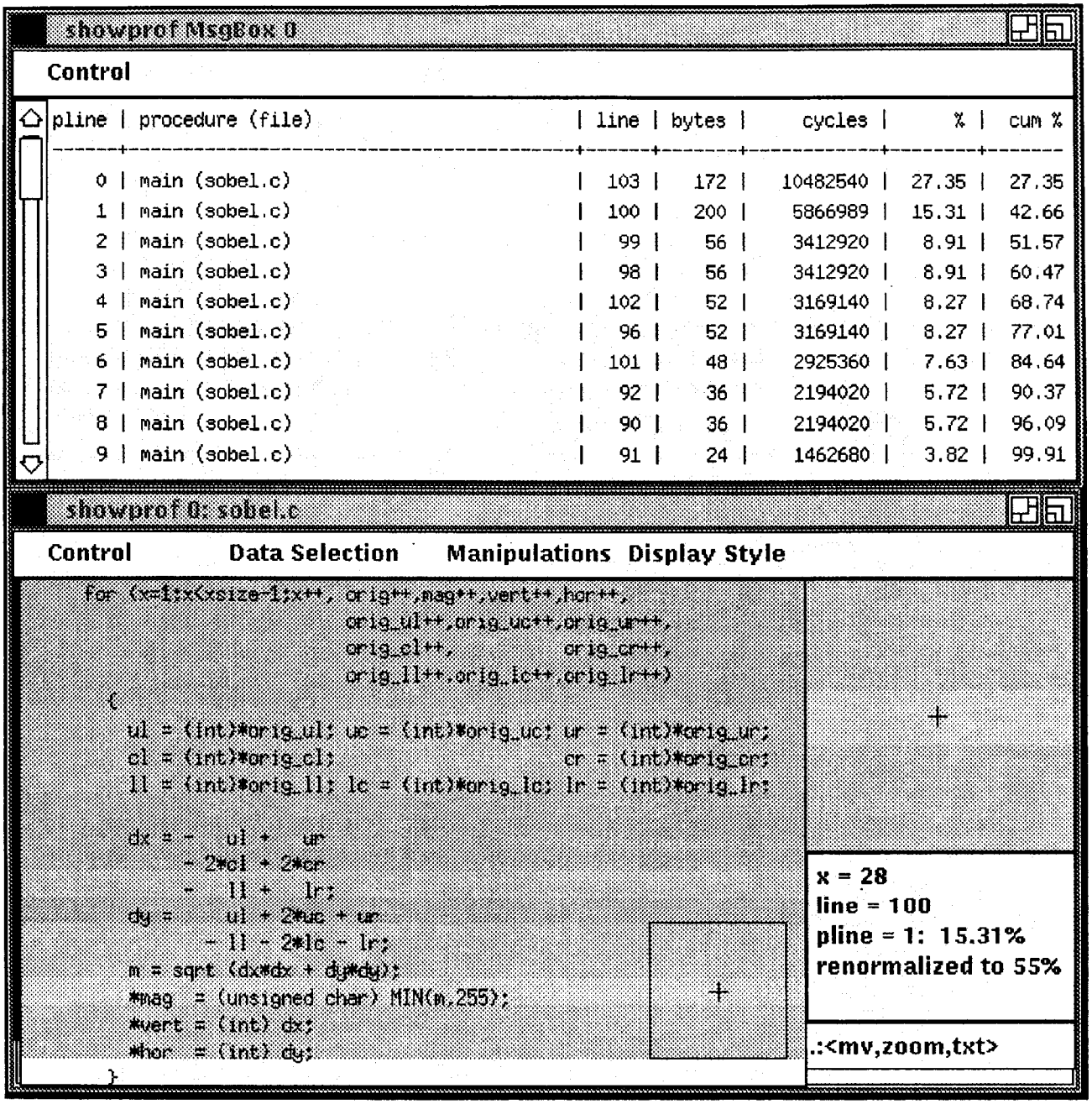

Figure 4-2 demonstrates how VDI can be used to visualize profiling information, gathered by a software tool during program execution. The figure shows such data for the Sobel program that was used to generate Figure 2-7. Typically, such profiling information is presented to a programmer as a sorted list, indicating the program lines in which the highest percentage of time is spent (see the upper window in Figure 4-2). The programmers then have to associate this information with the program text, using other software tools to actually see the busy lines. And they still don't know then how much time is spent in the neighboring lines. It is thus tedious to ana-

lyze an entire program segment. We have devised a simple program, "showprof" which compiles performance statistics into array form. In principle, the information can be stored as a one-dimensional vector, with line numbers as index. However, to ease overlaying the program text on the profile data, we are using a two-dimensional array. The vertical axis shows increasing line numbers. The horizontal axis presents no added information; the original performance data just is replicated a number of times. The lower window in Figure 2-7 shows the array after appropriate zooming. Different percentages are shown as different shades of gray.[1] The text window to the right indicates that the program spent about 15% of its time in line 100. This is the second highest percentage, covering about 55% of time when compared to the busiest line.

5 Summary

This paper points out that higher-level image understanding systems are lacking adequate debugging tools that pay attention to the needs for visualizing many highly inter-related multi-dimensional arrays of data. We suggest to merge graphics programming with debugging technology. To provide general-purpose graphical display routines for multi-dimensional arrays, some common display practices have to be extended. This paper discusses such generalizations and presents VDI as an example implementation. We describe operations on multi-dimensional arrays as composed of three steps, each of which imposes different kinds of requirements on an interactive data visualization tool:

- **Data Selection**
  When inspecting arrays, a programmer needs to be able to concentrate on subarrays. This requires interactive cropping, subsampling, and thresholding mechanisms, as well as methods to select displayable slices through high-dimensional arrays. The programmer must also be able to inspect arrays in relationship to data in other arrays, if operations on one array depend on data values in the other arrays.
- **Data Display**
  Many representations for displaying arrays of data have been used in various application areas, such as graphs, intensity images and arrays of printed numbers. It depends on the current focus of the data inspection process which representation is the most suitable. A general-purpose display system needs to provide several such representations and offer tools to the programmer to interactively switch between them.
- **Data Manipulation**
  Besides inspecting the current array contents, a programmer may also wish to interactively test the effect of certain modules of his own on the current data interpretation state. Visual programming is a powerful paradigm for interactively exploring such questions. Within a debugging framework, the programmer should have the option to either keep such investigation tentative or to assign the results to the array variables in the program, thus having permanent effect on the further execution of the application program.

[1] On the display screen, the performance data is shown as a rainbow map (also known as heat map), with high percentages being shown in red and low percentages being shown in blue.

So far VDI is only a first, crude step in the direction of providing a visual debugging environment to vision programmers. More needs to be done: VDI is not yet an integral part of a debugger, it can only be called as a display library from a debugger. We also plan to add more data manipulation capabilities to VDI: using a visual programming style, the programmer should be able to execute programmer- and VDI-provided procedures on arrays, with temporary or permanent effect on the array variables. Further extensions to the current framework include adding interactively specifiable transformations between positions in different arrays, providing more extensive programmer customization capabilities, and having VDI run in parallel to the application in a multi-threaded environment or on a separate processor across a network.

The concept of extending debuggers or programming environments with a graphical component for displaying multi-dimensional arrays promises to be a valuable visualization tool for many applications. General-purpose programming languages have seen the use of multi-dimensional arrays over a large range of applications. Among such applications are, of course, the typical image-oriented areas, such as computer vision, computer graphics and image processing. But applications in other areas also often operate on multi-dimensional arrays, such as operations research, scientific visualization, pattern recognition and, in general, numerical or statistical data analysis packages. Furthermore, results in other areas, such as simulation, may be restructured into array form. For example, research on the efficiency of parallel programming mechanisms (or systems monitoring computation load on a computer network) could visualize simulation results in array form, with one dimension representing time, another one the different processors, a third one different strategies for spreading the work across the processors, with the array content showing the load level at the processors at a given instant in time under a given strategy. Instead of writing many, special-purpose visualization routines for every such application area, the debugger or programming environment offers the chance to unify efforts across applications.

Acknowledgments

The design of the Visual Debugging Interface has benefited from many discussions with the members of the Visualization Group at the Cambridge Research Lab, Richard Szeliski and Ingrid Carlbom. Thanks to Bert Halstead for many long discussions on the similarities and differences of visualizing MIMD-processor allocation versus visualizing image interpretation modules. VDI's graphical interface for changing color maps is strongly influenced by discussions with Jim Gettys concerning toolkits for displaying astronomic images. Richard Szeliski has provided valuable comments on an earlier draft of this report.

References

[Adelson and Bergen 86]
E.H. Adelson and J.R. Bergen.
The extraction of Spatio-temporal Energy in Human and Machine Vision.
In *Workshop on Motion: Representation and Analysis*, pages 151-155.
IEEE Computer Society Press, May, 1986.

[Bolles and Baker 85]
R.C. Bolles and H.H. Baker.
Epipolar-Plane Image Analysis: A Technique for Analyzing Motion Sequences.
In *IEEE Proc. of the third Workshop on Computer Vision: Representation and Control*, pages 168-178. IEEE, Bellaire Michigan, October, 1985.

[Carlbom et al 91] I. Carlbom, D. Terzopoulos, and K.M. Harris.
Reconstructing and Visualizing Models of Neuronal Dendrites.
In *Proc. CG International'91: Visualization of Physical Phenomena*, Boston, MA, June 24-28, 1991, Springer-Verlag, Tokyo, Japan.
Also available as Technical Report CRL 90/14, Digital Equipment Corporation, Cambridge Research Lab, Cambridge, MA, December 1990.

[Draper et al. 89] B.A. Draper, R.T. Collins, J. Brolio, A.R. Hanson, and E.M.Riseman.
The Schema System.
International Journal of Computer Vision 2(3):209-250, January, 1989.

[Gettys et al. 90] J. Gettys, P.L. Karlton, and S. McGregor.
The X Window System Version 11.
Technical Report CRL 90/8, Digital Equipment Corporation, Cambridge Research Lab, Cambridge, MA, and Silicon Graphics Computer Systems, December 1990.

[GKS-3D 87] International Standards Organization.
Graphical Kernel System for Three Dimensions (GKS-3D).
Standard ISO/DIS 8805S, International Standards Organization, Geneva, April, 1987.

[Goto and Stentz 87]
Y. Goto and A. Stentz.
MobileRobot Navigation: The CMU System.
IEEE Expert 2(4):44-54, Winter, 1987.

[Grinstein et al. 89]
G. Grinstein, R.M. Pickett, M.G. Williams.
EXVIS: An Explanatory Visualization Environment.
In *Graphics Interface*, pages 254-261. London, Ontario, Canada, June, 1989.

[Halstead and Kranz 90]
R.H. Halstead and D.A. Kranz.
A Replay Mechanism for Mostly Functional Parallel Programs.

Technical Report CRL 90/6, Digital Equipment Corporation, Cambridge Research Lab, Cambridge, MA, November 13, 1990

[Hanson and Riseman 78]
A.R. Hanson and E.M. Riseman.
VISIONS: A Computer System for Interpreting Scenes.
In A.R. Hanson and E.M. Riseman (editor), *Computer Vision Systems*, pages 303-333. Academic Press, New York, 1978.

[Klinker et al. 90] G.J. Klinker, S.A. Shafer, and T. Kanade.
A Physical Approach to Color Image Understanding.
International Journal on Computer Vision 4(1), 1990.

[PHIGS 87] International Standards Organization.
Programmer's Hierarchical Interactive Graphics System (PHIGS).
Draft Standard ISO dp9592-1, International Standards Organization, Geneva, October, 1987.

[Rost et al. 89] R.J. Rost, J.D. Friedberg, and P.L. Nishimoto.
PEX: A Network-Transparent 3D Graphics System.
IEEE Computer Graphics and Applications 9(4):14-26, 1989.

[Saber-C 89] Saber Software Inc.
Saber-C User's Guide Introduction.
Technical Report Version 2.1, Saber Software Inc., Cambridge, MA, 1989.

[Scheifler and Gettys 86]
R.W. Scheifler and J. Gettys.
The X Window System.
ACM Trans. Graphics 5(2):79-109, April, 1986.

[Smith and Williams 89]
S. Smith and M.G. Williams.
The Use of Sound in an Exploratory Visualization Environment.
Technical Report R-89-002, Department of Computer Science, University of Lowell, Lowell, MA 01854, May, 1989.

[Stephenson 90] T. Stephenson.
Computer Graphics and Computer Vision.
Advanced Imaging, May 1990.

[Szeliski 90] R. Szeliski.
Real-Time Octree Generation from Rotating Objects.
Technical Report CRL 90/12, Digital Equipment Corporation, Cambridge, Research Lab, Cambridge, MA, December 1990.

[Szeliski 91] R. Szeliski.
Shape from Rotation.
In *Proc.of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'91)*, Maui, Hawaii, June 3-6, 1991.

Also available as Technical Report CRL 90/13, Digital Equipment Corporation, Cambridge Research Lab, Cambridge, MA, December 1990.

[Upson et al. 89]
C. Upson, T. Faulhaber,Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam.
The Application Visualization System: A Computational Environment for Scientific Visualization.
IEEE Computer Graphics and Applications 9(4):30-42, 1989.

[VoxelView 89] Vital Images, Inc.
VoxelView/PLUS, The Interactive Volume Rendering System.
Technical Report, Version 1.3 Release Notes, Document Number VVP-RN-0190, Vital Images, Inc., Fairfield, Iowa, 1989.

[XUI Toolkit 89] ULTRIX Documentation Group.
Guide to the XUI Toolkit: C Language Binding.
Technical Report AA-MA95A-TE, Digital Equipment Corporation, Nashua, NH,1989.

[XUI Toolkit Intrinsics 89]
ULTRIX Documentation Group.
Guide to the XUI Toolkit Intrinsics: C Language Binding.
Technical Report AA-MA96A-TE, Digital Equipment Corporation, Nashua, NH, 1989.

[XVision 3.0 89] University of New Mexico, Department of Computer and Electrical Engineering.
XVision 3.0.
1989. Also known as the Visualization Workbench from Paragon Imaging.