Interactive data inspection and program development for computer vision

Wolfgang Eckstein, Carsten Steger

Forschungsgruppe Bildverstehen (FG BV)
Informatik IX, Technische Universität München
Orleansstr. 34, 81667 München, Germany
E-mail: {eckstein|stegerc}@informatik.tu-muenchen.de

ABSTRACT

In this paper, an integrated program development environment for computer vision tasks is presented. The first component of the system is concerned with the visualization of 2D image data. This is done in an object oriented manner. Programming of the visualization process is achieved by arranging the representations of iconic data in an interactively customizable hierarchy that establishes an intuitive flow of messages between data representations seen as objects. The visualization objects, called displays, are designed for different levels of abstraction, starting from direct iconic representation down to numerical features, depending on the information needed. Two types of messages are passed between these displays (update and result messages) which yield a clear and intuitive semantics.

The second component of the system is an interactive tool for rapid program development. It helps the user in selecting appropriate operators in many ways. For example, the system provides context sensitive selection of possible alternative operators, as well as suitable successors and required predecessors. For the task of choosing appropriate parameters several alternatives exist. For example, the system provides default values, as well as lists of useful values for all parameters of each operator. To achieve this, a knowledge base containing facts about the operators and their parameters is used. Secondly, through the tight coupling of the two system components, paramters can be determined quickly by data exploration within the visualization component.

Keywords: Systems and Applications, Computer Aided Vision Engineering, Graphical User Interfaces for Computer Vision

1 INTRODUCTION

Many computer vision problems can be solved by using a library of image processing algorithms in combination with simple control structures. Examples for this class of problems are chip inspection, counting and measuring objects, and quality assessment. Many industrial vision tasks fall into this category of programs. In this domain it is essential that new applications can be developed in the shortest possible time. Also, for complex applications like aerial image interpretation or active vision rapid prototyping is a very convenient option as a first step towards the solution. Appropriate visualization of the iconic data is very important for the development of image processing applications since it can speed up the process of finding appropriate thresholds and parameter values tremendously. This motivates us to propose an integrated programming environment that fulfills the need for rapid program development. It consists of two major components. The first component is a tool for the visualization of iconic data and is described in sections 2–4. The second component is an interactive tool for program development that is described in sections 5–8. Finally, section 9 concludes the paper.

2 VISUALIZATION OF ICONIC DATA

Through the last years different tools were presented designed to ease and accelerate insight into complex relationships of image based data. Scientists working on problems related with the analysis of 2-dimensional images can be supported by such tools in different aspects of their work, such as:

- interactive or automatic measurement of numerical features, statistics, and iconic features, e.g., enclosing circles;
- developing, programming, and debugging of new algorithms;
- choosing and validating of parameters (e.g., by checking and comparing the results of different segmentations);
- getting an overview of voluminous amounts of data (images or features) to find material of special interest;

Three well known tools to assist such tasks are the Khoros system with its iconic programming language Cantata,[1] the KBVision environment,[2] and the Application Visualization System AVS.[3] They are state-of-the-art in this field, offering an interactive graphical user interface that allows the user to program and run image processing tasks using iconic languages. But they also give the user the ability to manipulate and inspect image data and image-based features by interacting with graphical representations of these features.

With all these tools the user can assemble representations of data and connect them with links. The links are paths for "messages" which arise from user interaction with the original representations or which result from the selection of other derived material for inspection. Therefore, these messages can include mouse events as well as the loading of a new image into the system. For example, marking a range of values within a gray-value statistic can select a number of pixels that could be displayed in an overlaid manner on the original image. Another typical example is the selection of a region of interest, which leads to automated feature extraction and display of the results.

But there are significant differences in how the assembly and definition of relationships between visual objects are handled. Khoros/Cantata and AVS allow visual programming of data flow between operators represented by icons. Calculations and graphical representations of features are interpreted as distinct operators and have to be installed and connected separately. The visualization of feature- and image-data is located in independent windows. This concept makes visualization an integral component of image processing programs.

In contrast to this, KBVision offers a specialized tool separated from image processing. It consists of an array of display cells packed together in one window. The cells are equal in size and each cell can hold one representation of different images or features. This layout does not satisfy the different needs arising from visualization of numerical features (i.e., textual output or histogram drawing in the case of KBVision) and images. Images are loaded into a cell by selecting them from a menu. Each cell can be configured to hold one kind of representation: text, gray or color images, or statistical diagrams. Especially choosing the type of visualization and the type of feature to visualize is a single user interaction. The contents of displays are automatically updated by messages resulting from interaction within another or the same cell. The flow of information can be customized from one cell to any other, including circular dependencies.

This net of relationships between displaying objects raises questions concerning the semantics and termination of updating caused by interaction, which are not easy to understand for the user. Therefore, we propose how to organize such a dataflow in an intuitive, easy to understand, and efficient way, which will be introduced in section 3. The implementation will be discussed and evaluated in section 4.

3 PRINCIPLES AND CONCEPTS

We have chosen the architecture of separating operator based image processing from displaying and inspection tasks because it allows independent building and reuse of visualization programs. This enables us to use different concepts of visual programming more adequate for this special type of work.

Our idea to ease handling and understanding of message paths between displays is based on the following principles:

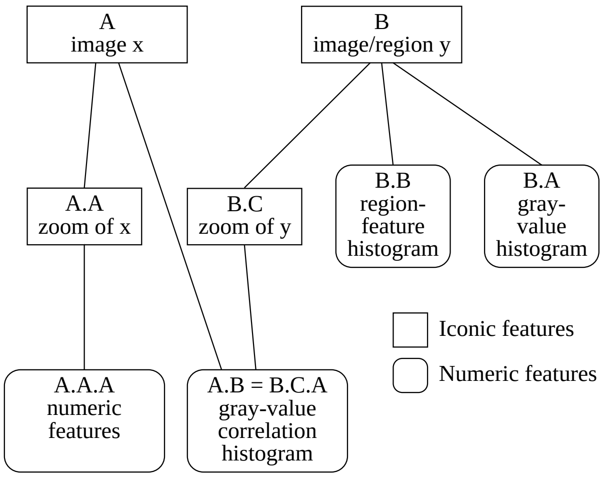


Figure 1: Example of a visual hierarchy

- Computation of features and the visualization of numerical and statistical features are combined into feature-display objects (similar to KBVision). Visualization of iconic data takes place in iconic-display objects. Iconic data are images, regions (arbitrary sets of pixels), polygons, etc., created by image processing programs or user interaction.
- Display objects are arranged in a graph consisting of different levels similar to a tree. Objects on the root level are the entry points for iconic data.
- Messages can flow from higher to lower levels triggering updating of displays and/or calculation of features (*update-messages*). These messages are caused by user interaction with displays or by new information fed into the root display level.
- Special answer messages can flow upwards, but never flow downwards again (*result-messages*). They can lead to additional drawing in their parent displays, but never lead to recalculation of features. They are caused by user interaction with data in displays or image data.

Figure 1 shows an example that illustrates the effects caused by these principles. It displays some typical dataflow relations and message types. An image x is drawn within a display A. Then display A.A on the second level is opened to examine a special region of interest within x. Therefore, it can be seen as a child of A. Selecting a rectangle in display A leads to sending the image to A.A and being visualized there. Drawing a region within image x can be performed in A or A.A, causing the displays to mutually inform each other. After creating a display A.A.A for numerical features of regions (e.g., the enclosing circle) and making it a child display of the zoom-display A.A, the drawn region is sent to the feature-display. The numerical features are computed and shown within this display and the enclosing circle is handed back to A.A and A. There it could be overlaid on the original image and the drawn region. Similarly, display B shows image y and is connected to a gray-value histogram B.A. By marking some part of the statistic for y, a set of pixels can be specified. This leads to sending them back to the parent display B (interpreted as a region), where they can be overlaid. Every change of an image loaded into display A or B causes an update message for all lower level displays, which eases repeating the same inspection-tasks for similar or different images.

Regions that are imported into display B from image processing can be overlaid on image y. Additionally, this triggers calculation in the region-feature histogram display B.B of a numerical feature (e.g., size) for all regions imported into B. The values are displayed as a distribution. We want to stress here that the selected region from the gray-histogram B.A that is displayed in B is not automatically forwarded to B.B. This prevents the number of messages from exploding and non terminating in certain constellations, as well as unwanted side-effects, which can be hard for the user to estimate in advance in complex visualization programs.

Finally, a display can have more than one parent and messages can skip levels (here our system differs from a tree). In our example this would happen if the user opens another zoom display B.C and wants to compute a gray-value correlation histogram (2-dimensional) between the images in B.C and A (assuming they have identical size). A special situation arises (not in our example) if a node has two children feeding a common display, that sends result-messages back to them. Then two duplicates of each result message reach their parent. We do not consider this a problem, because it can be handled by assigning a unique message identification.

We see two basic advantages in this way of organizing and interpreting image inspection as a hierarchical layered network:

- The dataflow itself can be visualized clearly laid out in an iconic diagram (e.g., in a manner similar to figure 1).
- Such a graphical representation can be used as an editor to create image inspection programs (as discussed in section 4). No programming in the classical sense is required.

4 IMPLEMENTATION AND EXTENSIONS

4.1 Integration of the concepts into an image inspection application

As outlined in section 3, we distinguish between operator based image processing and interactive inspection tasks. Therefore, we decided to implement two independent applications that are tightly coupled via sockets or files and act as one image analysis tool. These are the *HORUSDevelop* programming and debugging system, described in section 8 and the *HORUSInspector* application, which we want to focus on in this section. They are based on the HORUS image processing environment[4] and the OSF/Motif standard for graphical user interaction. The *HORUSInspector* is directly designed based on the ideas developed so far in this paper.

4.2 Data types and message types

We intentionally delayed the discussion of image data types and message types exchanged by the described displays until now, because they depend in some way on the underlying image data structures. The HORUS library offers 2-dimensional image processing and supports regions, polygons, and images with a variable number of gray-channels. Therefore, we restricted our system to the visualization of 2-D material and the calculation of features generated from 2-dimensional data.

From a 2-D image we consider the following elements suitable for being objects of interest: gray-channels, coordinates (with the resolution of the image or to subpixel accuracy), polygons, splines, regions of interest (interactively defined areas), arbitrary regions, and sets of these objects. All of them are potential message objects, which can be the result of image processing or user interaction with iconic representations. Besides the ability to determine a zoom by specifying a region of interest, they can be used to derive a vast amount of different feature types that can be classified into three (not always distinct) classes:

- simple numerical features: gray-value, color, length, extent, area, standard deviation, mean-value, etc.;
- iconic features: coordinates, center of gravity, enclosing rectangle, etc.;
- complex statistical features derived from objects or sets of objects: region-feature statistics, gray-value statistics, gray-value correlation statistics, etc.

The current implementation, which will be described now, limits the user selectable types on 2-D material to pixels, regions and multi-channel images. This was done because they are best supported by the HORUS image processing library, not restricting the principles of dataflow in any way.

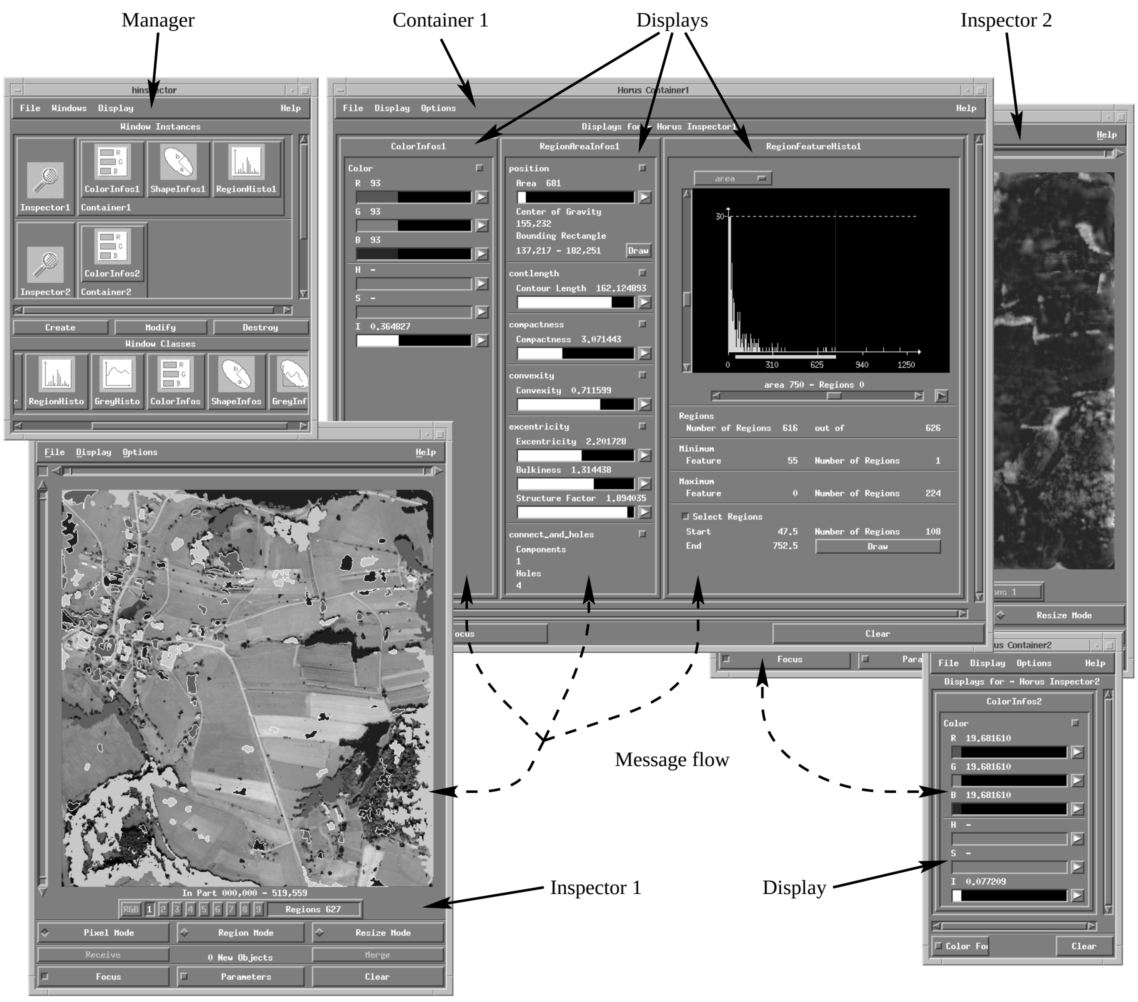


Figure 2: Main components of the HORUSInspector system

4.3 Classes of visualization objects

The distinct feature classes we introduced imply different ways of representation and interaction. Simple numerical features are best displayed in textual or simple graphical form and are not very useful for interaction. They reside in feature-displays. Iconic features are suited for overlaying them on the basic image. But we decided to place a second textual representation of them in feature-displays, where their values can be determined. Complex statistical features should be displayed in their own windows and can be used for selecting interactively sets of pixels or sets of regions. A second useful way to classify features in our context is to distinguish whether they are based on the object itself (e.g., lengths and coordinates) or on the object combined with the underlying image channels (e.g., color, gray-value, and gray-value statistics).

Our last criterion for classification is the type of source object to be examined.

From these criteria we designed a system of basic visualization and feature object classes (figure 2 shows an example of the appearance of the system):

- Objects for direct interaction with iconic data (images and regions). An instance of this class is called *inspector*. This display allows integrated measuring of simple pixel based features like coordinates and distance. Furthermore, it is possible to zoom into parts of the image and select different display modes, e.g., color look-up tables or $2\frac{1}{2}$-D plots. This display is very powerful in itself and suffices for many applications.

- *Color-info displays* serve for displaying attributes of pixels, e.g., gray-, RGB-, and HSI-values.
- *Region-feature displays* provide textual and simple graphical representations of region based numerical features like area, compactness, number of holes, or enclosing rectangle. Each display can be configured with respect to the range of values and the scaling (e.g., logarithmic scale).
- *Gray-region-feature displays* are responsible for gray-component dependent numerical region features, e.g., medium intensity, standard deviation, or entropy. They are represented textually and graphically.
- *Gray-value histograms* calculate and display the statistical distribution of pixel-values within a region or a region of interest.
- *Region-histogram displays* show the distribution of numerical features of a set of regions. Typical features are area, excentricity, circularity, or convexity.

For all of these basic classes we implemented one subclass, that can be instantiated by user at runtime. The instances appear to the user as windows that serve to hold the visualization the user can interact with. Beside this they contain control elements allowing to change parameters for graphical representation, for feature calculation, and in some cases the feature to be selected itself (e.g., in the region-histogram display).

4.4 Interaction with iconic and numerical data

We now want to give a more detailed insight into the techniques we offer in the *inspector* class implementation.

The user can load images, regions, or polygons into an inspector at root level by dragging a textual or iconic representation, e.g., from the $\mathcal{HORUS}$*Develop* application, into the inspector window. Executing a send command referring to the target inspector within an image processing program has the same effect. The material can replace or be merged into the existing data, either under user control or automatically. Images, regions, and polygons can be displayed overlaid or separately with customizable color lookup tables and drawing colors respectively. Single gray-channels can additionally be shown as $2\frac{1}{2}$-D plots or contour lines. All displays that are connected to a particular inspector are kept in a *container* window. This prevents the desktop from being cluttered with too many windows.

Different operating modes are offered for the creation of message objects. In pixel mode, coordinates and sets of coordinates can be selected. In resize mode a region of interest (rectangular) can be drawn or chosen with special resize sliders. It is displayed directly within the window or sent to child displays and inspectors, where it is interpreted as a region or a resize message respectively. In region mode the user can draw or select a single region by directly pointing at it with the mouse. If regions overlap, this is handled with the help of an additional menu that allows to select them from an ordered list. Figure 3 shows an example inspector window.

Special effort was made to give the user graphical feedback for each action he performs. For instance, when pressing a mouse button within a region, it is drawn in a special color. If more than one region was selected (regions can overlap), pre-selection message-objects are sent to the child displays. Region-based feature-displays will then compute the relevant features and display them. Based on this visual feedback, the user can select the desired region or abort the selection. In the first case they fix the new state. In the second case this causes all children to redraw their windows to the state before the aborted attempt. Also, single displays can be excluded from this snooping if feature computation is too time consuming. Displays can be completely excluded from updating if they are to be compared with other displays showing recently inspected feature data.

The second interactive display types are the histogram based ones. As already indicated above, one can mark a range within the histogram with the mouse. For this range, a selection can be started which determines all pixels or regions whose gray-value or feature lies within the range. These result-objects are automatically sent to the parent inspectors which display them.

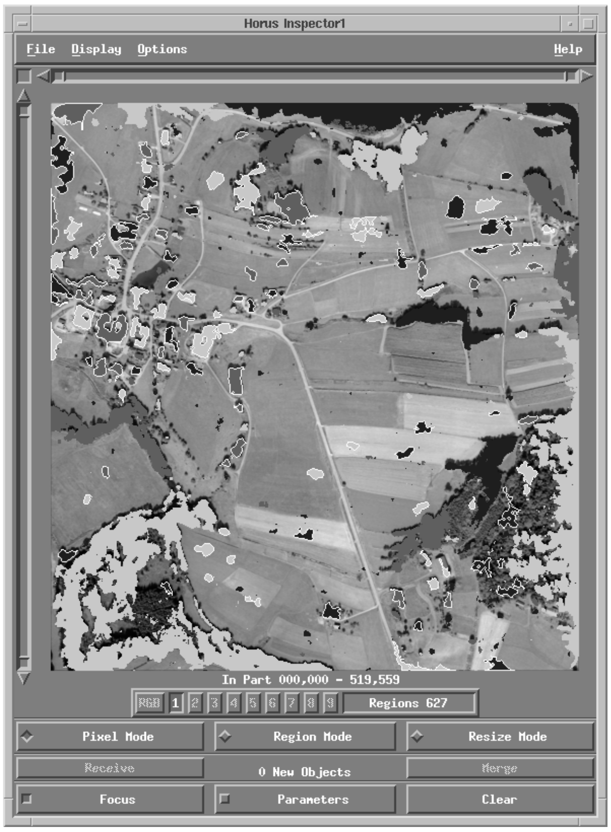


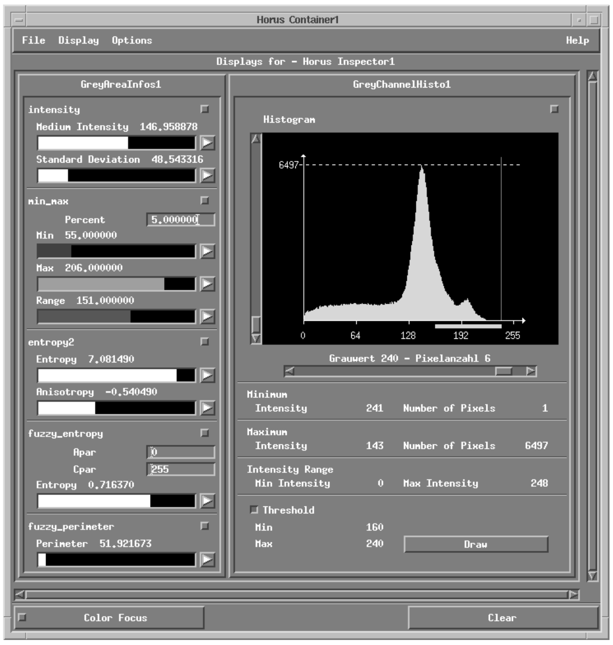
Figure 3: An inspector showing an image overlaid with regions resulting from a segmentation operation

The displays for simple numerical features show them textually or in the way of a gauge (a speedometer would be another good, but more space consuming, possibility). In contrast to the statistical feature displays, a set of features shown within one display, because we consider the automatic and intuitive layout of one single big window (e.g., the middle display in figure 2) easier than that of many small ones. Similarly, the display parameters first follow a default configuration that offers typical ranges and scales and can be changed at runtime. We also integrated the generation of numerical based, but iconic interpreted, features into these displays. An example is the enclosing rectangle supported by our region-feature display, but its representation there is only textual. The calculation of an iconic representation can be induced by a special export button and sent as a result-object to the parent display. An example of three numerical displays is given in the container window in figure 2. Figure 4 shows a further example of a display set-up.

4.5 Constructing visual hierarchies

We want to close our system description with some remarks about the visual programming tool we designed to create hierarchical networks, named *manager*.

To create a new instance of a display, the user can choose an icon from a toolbar of icons representing the different display classes. The manager window contains the visualization of the message flow graph. The nodes in this graph are represented by icons of the respective classes. By dropping a class-icon upon an element of a hierarchy, an instance of this class is created and connected as a child display or child-inspector of the target object. Instantiated objects can also be used as prototypes in the same way. Because in this case all the visualization and feature relevant parameters are passed to the new incarnation, this eases constructing hierarchies for many similar tasks. In the current implementation, we reduced the height of these networks to the root and one child level and chose graphical representation based on a line of icons. The root of



Displays for - Horus Inspector1

GreyAreaInfos1

intensity
Medium Intensity 146.958878
Standard Deviation 48.543316

min_max
Percent 5.000000
Min 55.000000
Max 206.000000
Range 151.000000

entropy2
Entropy 7.081490
Anisotropy -0.540490

fuzzy_entropy
Apar 0
Cpar 255
Entropy 0.716370

fuzzy_perimeter
Perimeter 51.921673

GreyChannelHisto1

6497

Grauwert 240 - Pixelanzahl 6

Minimum
Intensity 241 Number of Pixels 1

Maximum
Intensity 143 Number of Pixels 6497

Intensity Range
Min Intensity 0 Max Intensity 248

Threshold
Min 160
Max 240

Figure 4: A container window consisting of displays for gray-region features and gray-value histograms

the tree is displayed on the left, while all displays connected to it are arranged in a linear fashion to the right. This does not allow to deviate from a tree and will be changed into a layout that resembles figure 1. In the upper left part of figure 2 the manager window is shown that was used to create the various displays in that figure.

The manager of the next generation will also allow to load and save complete hierarchies with all parameters of the objects. It will support distributing inspectors and displays across several X-displays, to get rid of the problem of limited desktop space in the context of complex visualization networks.

5 EVALUATION OF PROGRAMMING PARADIGMS

The remaining sections of this paper will focus on program development during the realisation of computer vision applications.

In recent years several programming paradigms have found widespread use for creating computer vision applications. In this section these paradigms are examined to determine which one will facilitate rapid program development to the highest degree.

At the moment, two major graphical programming paradigms for creating computer vision algorithms based on an underlying image processing library can be identified. The first paradigm is "graph-oriented." Software systems like Khoros/Cantata,[1] AVS,[3] or KBVision[2] allow the user to compose a graph with the nodes representing calls to the library and the arcs symbolizing data flow. The second paradigm is to create a textual description of the algorithm. This can either be done by creating the program with an ASCII editor, which normally has no knowledge about the library being used but may give partial syntactic support, or by using a graphical editor that has full knowledge about the underlying image processing library. An example of the ASCII category is the Vista system.[5] The Visilog system is an example of text-based tool with graphical support. The Image Understanding Environment[6] will eventually allow both paradigms to be used.

The advantages and disadvantages of each paradigm can be evaluated using various criteria. First of all, readability and understandability of the programs have to be considered. For small programs with less than ten operations graph-based programs are easier to understand because the whole program fits onto the screen and therefore the program structure and data flow is readily apparent. However, the situation changes if large scale programs have to be developed. Here, one of two problems occurs. Either the program is displayed in a scaled down manner to make it fit onto the screen or the user only sees a certain part of the program. In the first case the structure may be hard to understand because usually the arcs connecting the operators will cross each other or run very close to each other. So it is very difficult to tell which arc connects to which operators. Therefore, the structure and data flow of large programs are not readily apparent. In the second case, the user is limited to looking at the program through a small peephole and consequently has to scroll through the whole graph when trying to understand a particular program. While the user can identify the data flow between operators that have been laid out closely to each other, the overall structure of the program is equally hard to understand as in the first case. Furthermore, if the underlying image processing library provides high level data structures, e.g., sets of points, lines, or regions, a program will in most cases be a linear sequence of operators. Hence, the graph structure will be of no advantage. Therefore, if larger image processing algorithms have to be developed, the textual description of the algorithm has a clear advantage compared to the graph-based description.

The second criterion that can be used to evaluate each paradigm is the time it takes to learn to program with the system. Here, it seems that graph-based systems can be learned more easily because they can be operated more intuitively and because the user does not have to deal with variables and other distractions. Text-based systems, on the other hand, require more time to learn since the user has to make the data flow explicit by using variables and has to learn the syntax of the operators that he wants to use.

Finally, the visualization of the program has to be considered. While many algorithms can be visualized adequately using the graph-based approach, many commonly used programming constructions cannot be represented as graphs in a natural manner, e.g., loops and arithmetic expressions, and some cannot be visualized as a graph at all, e.g., data type definitions. Most graph-based systems that allow the user to use variables and expressions resort to textual input of these types of data. This is a clear break of the graph-based programming paradigm, since the user now has to learn the syntax of the expressions that he wants the system to evaluate. Therefore, one of the advantages of the graph-based approach is lost. Here, the text-based systems have the big advantage that every programming construct can be described within the same paradigm and syntax, namely the textual description of the program. No switching between paradigms is necessary.

6 REQUIREMENTS FOR A PROGRAMMING ENVIRONMENT FOR IMAGE PROCESSING

There are a number of requirements that a programming environment has to fulfill in order to help the user to develop computer vision applications quickly. We will present them in the order of their importance.

- *Rapid prototyping.* This means that the user should be able to solve a given computer vision problem quickly and be able to develop a prototype of his application rapidly. Every program step should immediately move him closer to the solution.
- *Reduction in programming time.* This covers the task of editing. The system has to minimize the effort for the actual composition of the program.
- *Support in all aspects of programming.* The system should help the user in the selection of useful parameters for operators or even in the selection of the most appropriate operator itself.
- *Minimization of programming errors.* Syntactic and semantic errors should be avoided wherever possible and be detected as early as possible by the system.
- *Tools for debugging.* The system must help the user to localize the errors that the system cannot detect automatically.
- *Development of complex applications.* Control structures, powerful data types, and variables have to be an integral part of the language.
- *Integrability into other tools.* Since no tool can provide the user with any need he might have, the code that is developed must be able to be integrated into other tools the user uses.

- *Maintenance and adaptation of programs*. Programs that have been developed should be easy to adapt to new requirements.
- *Documentation of programs.* The system should support the user in documenting the program he has developed.
- *Portability of programs*. Adaption of programs to other hardware platforms must be feasible in minimal time.

As a conclusion of the discussion in section 5 and from these requirements it can be seen that the most viable programming paradigm in this case is a textual description of the program in an interpreted language that is edited within a graphical user interface.

7 CONCEPTS FOR FACILITATING RAPID PROGRAM DEVELOPMENT

In this section the concepts that are necessary to fulfill the requirements from section 6 will be presented.

7.1 Rapid prototyping and reduction of programming time

In order to reduce the amount of time that is spent to compose a program, the effort to edit it has to be minimized. There are a number of means by which this can be achieved. First of all, the user must be able to select the operator he wants to use with minimal use of the keyboard. Therefore, the system should have menus which present the operators in a structured manner to the user. This means that operators have to be grouped into meaningful chapters and sections. For example, all the edge detection operators should be grouped into the chapter “Filters/Edge-Detection.” Of course, every user has a different notion of meaningful chapters. Therefore, the system should allow different structuring criteria. Once the user has acquired some knowledge about the system and knows the names of the operators that are most useful to him, it is often more convenient for him to enter the name of the operator rather than to select it from the menu. In order to minimize the number of keystrokes, the system must be able to select the appropriate operator if the user enters a substring of the name. If more than one operator name matches the name that the user provides, he should be given a list of operators that match and should be able to select the appropriate one.

Another concept to reduce the programming time is that the system should provide reasonable names for output variables by default and provide the user with a list of all variable names that have been used so far, so he can quickly enter variable names into a parameter field. Furthermore, the system has to provide useful default values for each operator. Additionally, it must take care of the administration of variables, i.e., their graphic representation and memory management. This relieves the user from low-level details that would require a high percentage of his programming time. Furthermore, the system should automatically lay out the program text in a manner in which the structure of the program is readily apparent.

One last important point is that the system and the language it uses should be self contained and have minimal overhead. If the programming language C is taken as an example, the user first has to include some necessary header files and then edit the main function of the program. All this overhead is unnecessary in a system that is aimed at reducing program development times.

7.2 Support in all aspects of programming

To facilitate rapid program development, the user should receive maximum support and help by the system. This means that the system has to provide useful suggestions for the operators that can be used. It must provide suggestions for possible preceding and succeding operations. For example, if the user wants to apply a dynamic threshold operation to an image, the system should list some smoothing operations, like a mean or a Gaussian mean, as a predecessor. Furthermore, it should suggest some morphological operations, like connected component determination or opening, as a successor. Also, the system has to suggest possible alternatives for the currently selected operator if the user is not satisfied with the results it yields. For the dynamic thresholding operation mentioned above, the system might suggest a simple threshold operation or a bandpass filter. Finally, there should be a pointer to other operators that might be of interest for the user in the selected context. For example, the dynamic thresholding operation can be used to detect local maxima in an image. Therefore the system should list an operator that finds local maxima directly as a possibly related operator.

Once the user has selected an appropriate operator, the system should display (upon request) a list of useful values for each parameter. This serves two purposes. If the user has little knowledge about the operator, he can get an impression which values might be appropriate and can try values contained in the list as a starting point for optimal parameter selection. Furthermore, if the values that the system suggests are well chosen, the user will rarely have to try more than two or three values from the list before finding the best value for his application. This further facilitates rapid program development.

One point that is very important for novice users as well as experienced users is an online help system that includes a context sensitive help facility. This means that if the user has selected an operator and wants to obtain information on this operator, he should only have to press one button and the system should present him with the description of the selected operator. This description should, of course, include all the relevant information in a well structured manner. Depending on the experience of the user, he should be able to configure the help system such that only information relevant to him is displayed.

7.3 Minimization of programming errors and debugging tools

In order to minimize programming errors, two aspects are of importance. First of all, the operators should be selectable by graphical elements like menus or lists. This prohibits the user from selecting operators that do not exist. Furthermore, the parameters that the user has to enter must be presented in a structured manner. This means, for example, that each parameter should have its own data entry field and that variables and values can be selected and placed into each field through the use of graphical elements, like menus or a drag-and-drop mechanism. The system then has to take care of syntactical elements like parentheses, spaces, and commas for the presentation of the program. This prohibits the user from forgetting parentheses or inadvertedly deleting a comma while editing.

To help the user to detect programming errors as quickly as possible, there has to be a direct link between the editing of the program and its execution. This means that once the user has fully parameterized an operator and finished editing, the system should immediately execute the operator and display the results of the computation. The system should be able to visualize numeric and iconic data in the most appropriate manner and only require the user to select a special mode of display if he wants to achieve special effects. If the user detects a parameter that he has set to an inappropriate value or made any other mistake, he can immediately spot the problem and fix it. Immediate execution and visualization should be optional, however, since an experienced user might wish to enter a program quickly and only run it once he has finished editing it. This can save time since some visualization processes might require complex computations.

Furthermore, in order to debug complex programs, the system must provide most of the capabilities of a good symbolic debugging tool. This means that the user must be able to set the program counter to any position he desires. Also, there has to be an execution mode in which the user steps through the program in a single step manner and a mode in which the program runs continuously. The latter mode has to be able to handle breakpoints, so the user can stop the program at certain locations. In all cases, the user must be able to change the contents of variables and the program itself, even while it is being executed.

7.4 Development of complex applications and integration into other tools

In order to enable the development of complex applications, the language that is used must contain a complete set of execution control commands, i.e., branch constructs like *if . . . then . . . else* or loop constructs like *for . . . endfor* or *repeat . . . until*. However, the completeness of the language used has to be taken into account. Since the user might want to execute calls to the underlying operating system or another library he is accustomed to, the desired level of completeness is very hard to achieve. Also, the user may want to integrate the program he has developed into a larger application as a subtask. For example, a program that finds lines in aerial images might be incorporated into a symbolic reasoning system as a subtask for extracting roads in low resolution imagery. Therefore the system should allow the user to save the program he has developed in the programming language of his choice. Once this conversion is done, the user gains the completeness of his programming language of choice, but will lose most of the advantages that the system gives to him. Therefore the option to convert the program into another language should only be used at the very end of the development process.

7.5 Life cycle of programs

Usually computer vision programs have to be maintained and adapted to new requirements, e.g., different object characteristics. Also, as new hardware platforms become available, the quick porting of programs to new architectures is important.

Therefore the system has to support all the traditional, but seldom provided, life cycle tools of software development, e.g., debugging and documantation.

In order to adapt a program to new circumstances, the concepts described in section 7.2, especially the suggestion of alternative operators and useful parameter values, help the user to quickly adapt an existing program.

If a program has to be maintained or adapted, good documentation is essential. Based on the knowledge-base that contains facts about the primitive operators, the system can aid the user to generate the documentation for the application. The part of the documentation that deals with parameters, types, and assertions of the program can be generated mostly automatically. The semantics of the program has to be entered by the user. Since the knowledge about the operators is highly structured, the system can direct the user to missing information.

Portability of developed programs relies heavily on the fact that the underlying image processing library must be portable. The same argument holds for the graphical user interface. Here, standard libraries have to be used. Finally, the system must support many different programming languages on output (see section 7.4), depending on the availability and convenience on the target machine.

8 IMPLEMENTATION OF THE SYSTEM

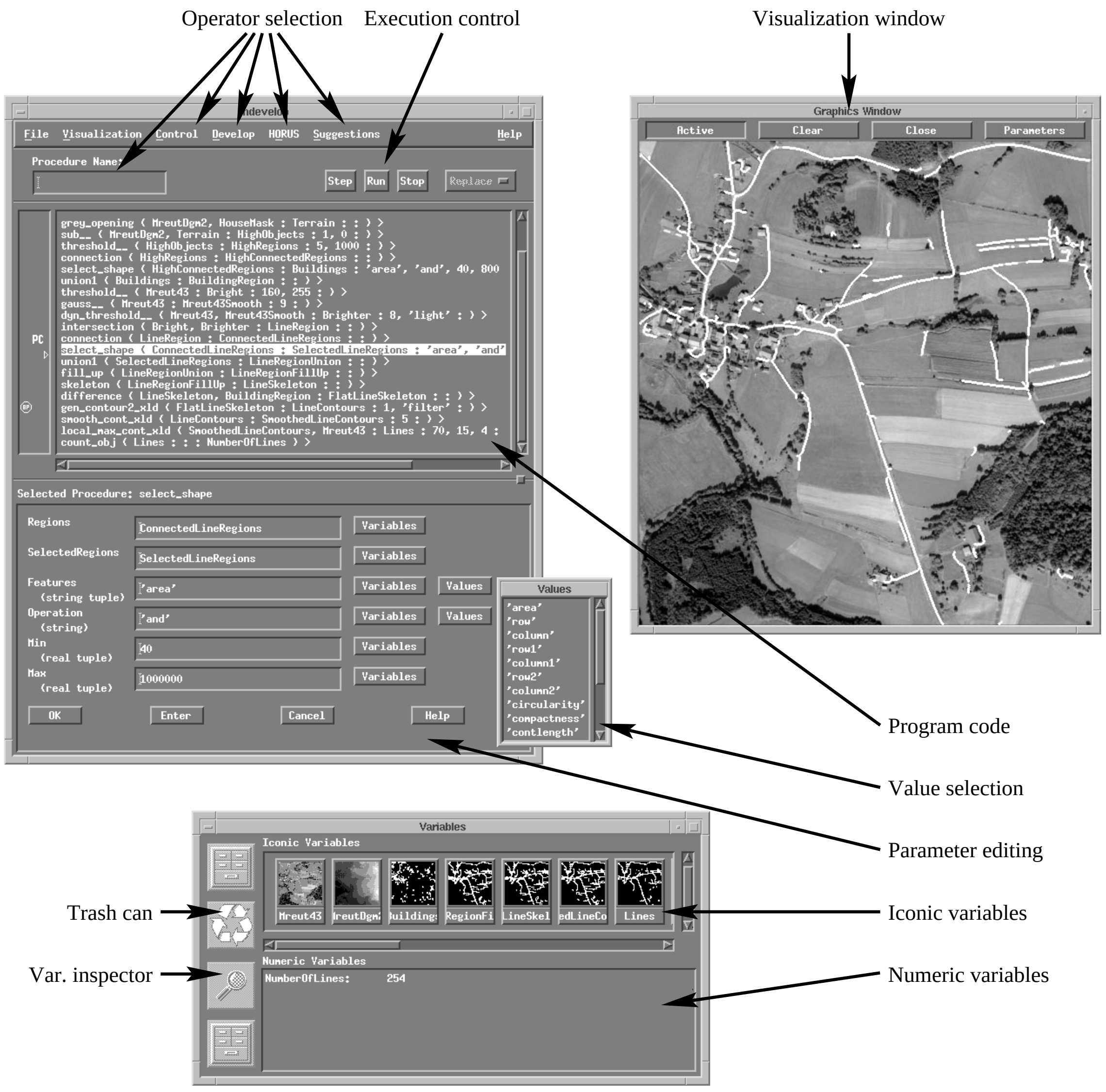
Most of the concepts that have been described in section 7 have been incorporated into a working programming environment called $\mathcal{HORUS}$*Develop*. This tool is based on the $\mathcal{HORUS}$ image processing library[4] which provides about 600 operators, mainly in the low- and mid-level domain, as well as a knowledge base with facts about the available operators. The remainder of this section will describe how the concepts developed in section 7 have been put into practise.

Figure 5 shows the general appearance of the system. In this case $\mathcal{HORUS}$*Develop* was used to develop a program that detects roads in aerial images with a ground resolution of 2 m. In this resolution roads can be modeled as lines that are brighter than their surroundings. However, buildings should not be detected by the program. From these considerations a program was developed that first extracts objects that are higher than their surroundings from a digital terrain model of the area. In the second part of the program lines that are brighter than their surroundings are extracted. The final result of the process can be seen in the graphics window in figure 5.

The $\mathcal{HORUS}$*Develop* system consists of three major windows. In the main window, shown on the top left of figure 5, the user can edit the program code in textual description with the help of a graphical editor. To enter a line of code, he can select operators from four menus or enter the operator name in a text field. The menu *Control* contains loop and branch constructs as described in section 7.4. The menus *Develop* and *HORUS* allow the user to select operators from the $\mathcal{HORUS}$ image processing library. The menus are structured in a chapter/section fashion to enable the user to select the appropriate operator quickly. The *Suggestions* menu allows the user to look for alternative operators to the currently selected one, for possible predecessor and successor operators, and for other operators that might be interesting for him. These menus facilitate rapid prototyping since the user can try different operators very quickly.

Once the user has selected an operator, the parameters are presented to him in a structured manner in the parameter editing area of the main window. The layout of this area is determined dynamically from the knowledge about operators, including types, value lists, assertions, ranges, and default values. Consequently, no changes to the user interface have to be coded when adding a new operator. Each parameter has a *Variables* button associated with it, so that the user can bring up a list of variables used so far and enter or replace the current variable name. Another possibility to enter a variable name into a parameter field is to drag it from the variable window into the appropriate field. Furthermore, most parameter fields have a *Values* button associated with it. When one of these buttons is pressed a list with useful values for the parameter will be displayed, as shown in figure 5 for the parameter *Features*. With this scheme, the user rarely has to enter data on the keyboard. Since the parameters are presented in a manner in which the user does not have to take care about the syntax of the language, programming errors that result from missing parentheses and other similar error sources are avoided. Additionally, the system takes care of the layout of the program. Hence, the time to write the program code will be significantly reduced.

If the user has fully parameterized the operator he can select *OK*, in which case the operator will be executed immediately and the results will be shown in the visualization window. If he presses *Enter* the operator will be added to the program but not be executed. This is useful for quickly editing a program. Additionally, the user can press *Help* to get a manual page



```
Procedure Name:

grey_opening ( MreutDgm2, HouseMask : Terrain : : ) >
sub__ ( MreutDgm2, Terrain : HighObjects : 1, 0 : ) >
threshold__ ( HighObjects : HighRegions : 5, 1000 : ) >
connection ( HighRegions : HighConnectedRegions : : ) >
select_shape ( HighConnectedRegions : Buildings : 'area', 'and', 40, 800
union1 ( Buildings : BuildingRegion : : ) >
threshold__ ( Mreut43 : Bright : 160, 255 : ) >
gauss__ ( Mreut43 : Mreut43Smooth : 9 : ) >
dyn_threshold__ ( Mreut43, Mreut43Smooth : Brighter : 8, 'light' : ) >
intersection ( Bright, Brighter : LineRegion : : ) >
connection ( LineRegion : ConnectedLineRegions : : ) >
select_shape ( ConnectedLineRegions : SelectedLineRegions : 'area', 'and'
union1 ( SelectedLineRegions : LineRegionUnion : : ) >
fill_up ( LineRegionUnion : LineRegionFillUp : : ) >
skeleton ( LineRegionFillUp : LineSkeleton : : ) >
difference ( LineSkeleton, BuildingRegion : FlatLineSkeleton : : ) >
gen_contour2_xld ( FlatLineSkeleton : LineContours : 1, 'filter' : ) >
smooth_cont_xld ( LineContours : SmoothedLineContours : 5 : ) >
local_max_cont_xld ( SmoothedLineContours, Mreut43 : Lines : 70, 15, 4 :
count_obj ( Lines : : : NumberOfLines ) >

Selected Procedure: select_shape

Regions                ConnectedLineRegions     Variables
SelectedRegions        SelectedLineRegions      Variables
Features               'area'                   Variables   Values
  (string tuple)
Operation              'and'                    Variables   Values
  (string)
Min                    40                       Variables
  (real tuple)
Max                    1000000                  Variables
  (real tuple)

OK          Enter          Cancel          Help

Values
'area'
'row'
'column'
'row1'
'column1'
'row2'
'column2'
'circularity'
'compactness'
'contlength'

NumberOfLines:      254
```

Figure 5: Main components of the HORUSDevelop system

about the currently selected operator. The manual page is presented in a highly structured manner. However, it is not yet possible to configure which parts of the documentation are shown.

Once the user has developed a program, it can be debugged through the use of the execution control buttons. These provide single-step and continuous execution until the user stops the program or a breakpoint is reached. Breakpoints and the program counter can be set in the area to the left of the program code window. During program execution all results are immediately visualized. Iconic data is displayed in the graphics window and in symbolic form in the variable window. Numeric data is only displayed in the variable window. Thus, the user has immediate feedback about the results of each operation and will be able to detect programming errors very quickly. Also, by this mechanism the effort necessary to maintain and adapt an application is greatly reduced. Finally, if the user is satisfied with the results of his program and needs a specialized application, e.g., a fancy user interface, he can output the program in a programming language of his choice. At the moment the system can produce C and C++ code.

9 CONCLUSIONS

In this paper concepts for the visualization of iconic data are presented in a systematic manner. From these concepts, a tool $\mathcal{HORUS}$*Inspector* has been developed that integrates most of them into a working environment and is currently in use. The novelty of the presented approach lies in the organization of the display objects into a hierarchy in which messages are passed between displays with a well defined semantics. This makes the system easy to handle. It allows fast and easy building of complex visualization programs in an interactive and intuitive way. A clear structure is presented to the user, arising from a highly adapted object oriented design.

Furthermore, the requirements and concepts that allow the rapid development of computer vision applications have been presented. A tool $\mathcal{HORUS}$*Develop* has been developed that integrates most of these concepts into a working rapid prototyping environment and is currently in use. However, at the moment the tool does not provide facilities to support the user in documenting his programs. One additional area of improvement that will further speed up program development is the input and visualization of parameters. If the semantics of a group of parameters is known, e.g., that four particular parameters describe a rectangle, a specialized input field can be displayed to the user. These two concepts will be implemented in the near future. Furthermore, the visualization of results while the user is in the process of selecting a parameter can be of great help. For example, if the user can select the parameters of a threshold operation using sliders and the results are immediately displayed when the user moves the slider, the time to find appropriate parameters for an operator will be greatly reduced. Finally, the suggestion of operators can be extended. One can envision a system that automatically determines sequences of operators from the input data and the result the user desires. Research in this area is planned.

10 REFERENCES

1. K. Konstantinides, J. R. Rasure, “The Khoros Software Development Environment For Image And Signal Processing”, *IEEE Transactions on Image Processing*, Vol. 3, No. 3, pp. 243-252, May 1994.

2. Amerinex Artificial Intelligence, Inc., “General Support Tools for Image Understanding”, *Amerinex Artificial Intelligence, Inc. technical report*, 1992.

3. C. Upson, T. Faulhaber, Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, A. van Dam, “The Application Visualization System: A Computational Environment for Scientific Visualization”, *IEEE Computer Graphics & Applications*, pp. 30–42, July 1989.

4. B. Radig, W. Eckstein, K. Klotz, T. Messer, J. Pauli, “Automatization in the Design of Image Understanding Systems”, *Proc. 5th International Conference IEA/AIE*, Paderborn, Germany, 1992.

5. A. R. Pope, D. G. Lowe, “Vista: A Software Environment for Computer Vision Research”, *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pp. 768–772, 1994.

6. J. L. Mundy and the IUE Committee, “The Image Understanding Environment: Overview”, *Proc. DARPA Image Understanding Workshop*, pp. 283–288, 1993.