Real-time Visualization of Interactive Parameter Changes in Image Processing Systems

Wolfgang Schneider, Wolfgang Eckstein, Carsten Steger

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

ABSTRACT

The main focus of this contribution is the description of mechanisms for an intuitive input of user-data with a direct response to gain fast parameter selection.

To build polymorph input methods for different parameter classes, a database containing knowledge about each operator is needed. This information about the syntactical and semantical structure is used to create adequate dialog elements, like sliders, list boxes, text fields, etc., each one responsible for a distinct parameter class. In addition higher-level semantics are used to aggregate data types, e.g., the user can draw a rectangle interactively, instead of only inputting the coordinates of its corners. Depending on the context, the user can also change certain subsets of the knowledge about the operator to fit special needs.

Incremental computation of the results of an operator by successively expanding regions of interest is used to gain direct feedback for every change of parameter values. So by dragging a slider for the size of a smoothing mask, e.g., the user can see the effect directly and thus select the desired value rapidly. This approach is extended to sequences of operators. In this case, more than one parameter form will be open at the same time, one for each relevant operator, to have full control of all important values.

Keywords: Data Visualization, Human-factors and Ergonomics Issues, Interaction and User Interfaces, Interactive Tools, Image Processing

1. INTRODUCTION

Commercial computer vision tasks must be solved in an effective and intuitive manner. This is why professional image processing systems frequently are shipped with some kind of program development tool wrapped in a more or less user-friendly GUI. Unfortunately, the traditional way in which the user has to interact with such a system does not correspond very well to the rapid-prototyping idea: image processing operators — the basic parts of the system — are extensively driven through *control parameters*, which have to be specified by the user, who then has to check the results. The procedure, performed in this way, is not very smooth and can be really time consuming: after specifying the parameter values, the user has to wait for the system performing its calculations, just to see the results aren't good and then repeat the whole cycle. This is classic try-and-error, if one doesn't have proper ideas about the parameter values. Another problem is that the user may be left alone with syntactic and semantic peculiarities: value range restrictions, improper subsets of the entire range and a lot more have to be taken care of, which again often has to be done by the user. As we will show, these problems can be handled in a much better way: in section 2 we'll introduce an input method for control parameter values, which can easily handle syntactic and semantic restrictions and is based on object-oriented principles. In section 3 we will discuss some ways of shortening the user feedback, keeping in mind the overall goal of solid real time visualization. Finally, in section 4 we'll describe how these concepts can be used to build an integrated program development tool for a visualization system.

2. CONCEPTS AND PRINCIPLES FOR PARAMETER DATA-INPUT

First of all, there are a couple of different types of parameter data. A typical list of often used types (in a visualization system) could look like this:

- plain numerical data: integer, real, complex, byte, long, double, etc.
- plain textual data (character strings)
- more special textual data (e.g., filenames)
- basic geometric shapes: line, rectangle, circle, polygon, etc.
- more special, visualization-related data: histogram, color map, convolution mask, etc.

In our case, images and regions are of minor interest because they can't be used to drive the operator's functionality; they act as the "raw material", which is transformed by the visualization program. We would like to call these *iconic parameters*, whereas the list given above contains *control parameters*.

2.1. Traditional data-input methods

The list shows the need to think about an adequate input method: there are many data types, the semantics of which are extremely different. A look at some established image-processing systems should give an idea, how the input-task could be performed, thus we'd like to take a quick look at the systems Khoros[2] and Stardent AVS[3]: Khoros consists of a set of *modules*, which are separately compiled mini-programs. These modules expect all their input data as parameters in the command line. To overcome this extremely unfriendly way of controlling the modules' behavior, Khoros has an additional graphical user interface,[1] where the modules' input parameters can be specified, but again, this can be done only textually via the keyboard. With AVS, we have a more improved way of specifying data; it offers various kinds of input mechanisms: a dial knob, a horizontal slider and a text field for numerical data, a colormap editor and a file selection box for the more special purposes. It can easily be seen that the Khoros method by no means fulfils the needs of a user-friendly system as far as data input is concerned; the AVS looks a lot more sophisticated, although it has problems, too: spin knobs are no very good instrument for data input, because a typical computer mouse isn't very well suited for circular movement; furthermore, if you want the knob not to react too sensitively, you have to configure it for many rotations representing the whole range. This is not possible for a slider, which therefore has to work with a minimum granularity, which cannot be turned any finer. The text field isn't very good for numerical input, also. Furthermore, every numerical input value defaults to a dial, so a possibly better alternative has to be selected manually. The solution we have chosen offers separately designed, semantically well-suited input forms for each possible data type.

2.2. Improved data-input methods

As we have seen that there is a need to clearly distinguish between several classes and types of parameters and to supply proper input mechanisms for them, we'd like to analyze this fact a little more precisely now. An often used way to input data is to specify them textually. This method has a couple of severe disadvantages:

- too much time is needed to type in a value, and small changes can't be done quickly (if you want to decrement from 1375.1000 by 0.0001, guess what to do).
- Humans recognize proportions and relative values much easier than absolute values: something like "in the middle of the whole range" fits human understanding much better than 17.5 (if we have a total range from 15 to 20, e.g.).
- Value restrictions may lead to errors: some convolution filters, for example, require certain parameters to be odd values; some other operator may require a parameter lying strictly between 10 and 20, e.g. These restrictions must be explicitly handled by the user.

- Some operators do not react linearly to value changes; if a parameter has logarithmic behavior, e.g., a huge part of the whole value range might produce almost none of the desired effect.
- Composite or multi-dimensional parameters are hard to handle; two-dimensional geometric objects for example cannot be moved freely without alternating at least two numerical values.
- The parameters' special semantics don't show up clearly: if we have two numerical values, the first one representing a filter threshold and the second being a x-coordinate of a rectangles upper left corner, they still are only numbers, looking alike to the user.

To solve the problems mentioned above, it would be a good idea to follow the WYSIWYG idea and have a visual representation of every parameter, allowing the values to be altered interactively by manipulating the representation object itself. For example, a good idea for a numerical input object would be a slider (dial knobs are no suitable instrument for general purpose data input, although they might be a good choice for specifying an angle). This representation has several advantages:

- The user instantly gets an intuitive overview of the whole data range.
- Range restrictions and the required stepping can be handled automatically without concerning the user.
- Logarithmic and other non-linear behavior can be eliminated, so the slider shows a linear effect to the user, regardless of what algorithm is lying underneath.
- Value changes can be done very quickly: one can do a sweep over the whole range just by dragging the slider; this of course is impossible with textual input.

But there is a disadvantage, too: if one needs to specify exactly one certain value, it may be hard to position the slider exactly in the desired position. Fortunately, this isn't as big a problem as it seem: in most cases, exact values are not necessarily needed to solve a problem — a value of 0.345 may have the same effect as 0.355 in one case, so this definitely meets the fact that humans will understand these two as "nearly the same" when represented graphically. Furthermore, it shouldn't occur too often that the desired value is known exactly. The solution we have chosen is an object consisting of two sliders (see figure 1), a main slider representing the whole range, plus a second one for fine tuning. This is necessary in some cases, as a single slider can produce hardly more than 100 distinct values, which may be too coarse.

Figure 1. Parameter selction with coarse and fine slider

On the other hand, there may be situations where only a small set of values is allowed; in this case we have chosen to offer the user a list containing possible alternatives (see figure 2).

Another class of parameters often used in image processing are geometric figures. Here the advantages of our WYSIWYG-like method against the textual one are even more dramatic: To specify a plain rectangle, for example, the user has to submit four numerical values, which semantically are one single parameter; specifying them textually not only means a lot of annoying typing, but also completely ignores the semantic link between

Figure 2. Parameter selection using a list of predefined values

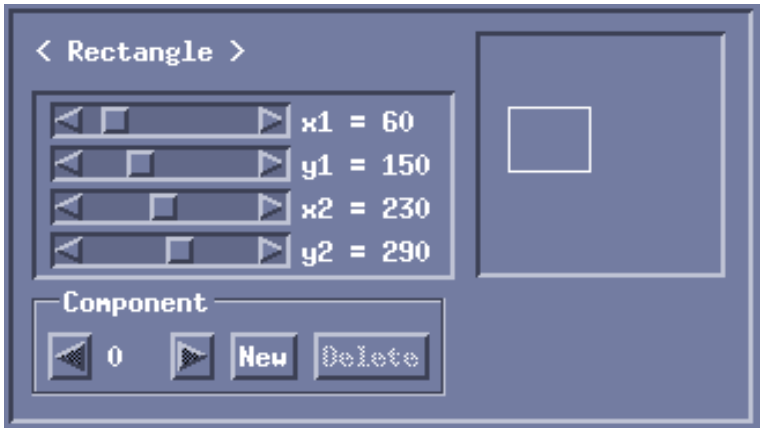

Figure 3. Selection of parameters of a rectangle

these values. Again we have chosen to supply sliders for geometric parameter input, four of them in our rectangle example (see figure 3). Additionally we have a graphical representation of the figure, which can be interactively moved and resized using the mouse. Why do we offer both ways of data input? It may be necessary to select a certain value for a single coordinate (a rectangle exacly 100 units wide, e.g.), thus we thought it was a good idea to keep the possibility to specify the values independently; on the other hand, moving and resizing the figure itself is a lot easier than keeping control of all coordinates simultaneously, so we offer this option, too. It should be mentioned that there again is the disadvantage of exact values being hard to focus; this is even worse than with numerical data, because due to lack of screen space, we have not implemented any fine tuning sliders.

There are, of course, a lot more than the numerical and geometric types, but we are not digging too deeply at this point; the principles should have become clear by now.

2.3. Using object-oriented principles for data input

All these input applets — we will call them *dialogs* from now on — are related in an object-oriented way: as they can all derive from the same base class, they have certain parts in common; this is desirable, since similar GUI parts should always look alike in order to ease the use. In our case every dialog consists of the same two to four parts (see figure 3):

- A headline containing the parameter name (and range, where possible).
- A main input area at the left: a list, some sliders or a field for textual input.

- A "representation area" at the right, where the parameter is shown graphically (with numerical data, simply the value is printed). This doesn't make sense for textual data, because a long text won't fit in, so it's omitted for textual dialogs.
- A component control area in the dialog's lower half: with the underlying $\mathcal{HORUS}$ system,[5] some operators may or must be supplied with *Tuples* of values; the `rectangle` operator, which generates a rectangular region, for example, accepts many rectangular areas simultaneously. With this component control, components can be added or deleted and the "active" component (the one which can be modified) may be selected.

An invisible feature of the representation area is its ability to offer a popup menu with a choice of default values, when clicked with the right mouse button.

To automatically generate and configure a dialog requires some knowledge about the associated parameter: the system must know

- the type of the parameter
- the data type(s) it's based on
- the correct value range(s)
- minimum and best suited stepping(s)
- default value(s)
- the funcional behavior (linear, logarithmic, etc)
- value restriction(s): values or parameters can be mutually dependent or may be restricted in any other way; for example, a rectangle may be required to always be longer than wide.

These data (and a lot more) are stored in the $\mathcal{HORUS}$ database, which holds a record for every operator in the $\mathcal{HORUS}$ system. Since some of these meta data are meant to be proposals only, it would be a good idea to make them freely configurable. For example, with numerical values we always have a minimum and a default stepping, the first of which resulting from the algorithm being used. The minimum stepping will be much too fine for most purposes, so a more suitable stepping value is used instead, which in turn may be too coarse for some other purposes. Since these two stepping values, like any other meta data, are provided by the $\mathcal{HORUS}$ database, it is necessary to cache the whole meta data record, so it can be modified by the user. This way, the user can select other steppings, another (smaller) value range etc. A dialog, with which this is achieved, can be seen in figure 4.

We have seen that it is possible and necessary to retrieve meta data about the parameters of an oparator. With this ability we can easily automate the dialog creation process and group all parameter dialogs for a certain operator in a single window, which will be called *form*. Hence there are operators which have many parameters and we also have the need to pop up several operator forms simultaneously (we'll explain this later on, in section 4), we have chosen to supply a mechanism with which the user can hide single parameter dialogs selectively (see figure 5). Each form gets a menu bar as well, where again some default values can be retrieved via the *values*-command. There is another command in the menu bar, named *options*, and an action bar containing some push buttons, which we will discuss in the next sections. Now we would like to take a look at the feedback the user gets when manipulating parameter values; the next section will introduce the principles and concepts behind this issue; but we'll also find out that there still is no fully satisfying way to supply a real-time feedback.

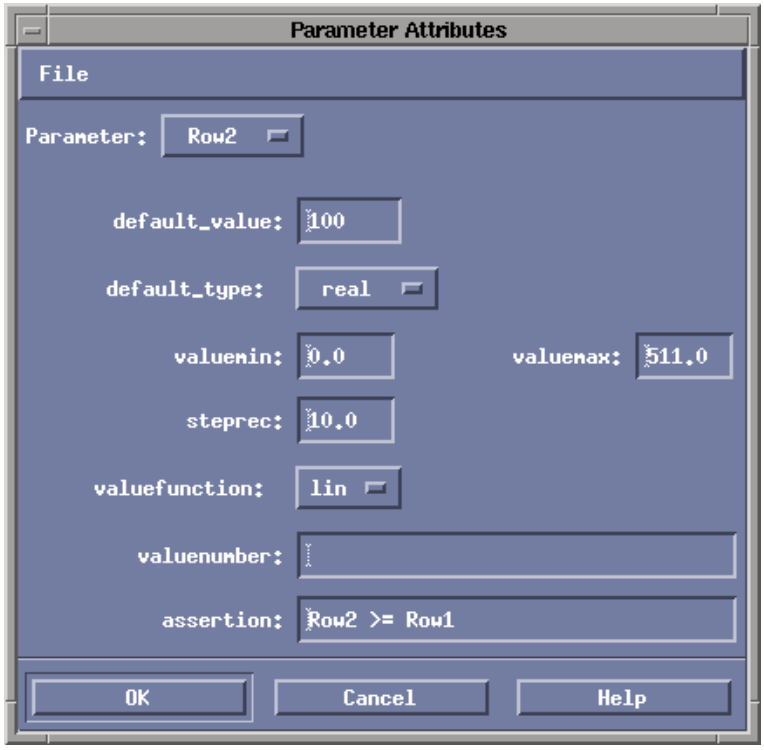

Figure 4. Modification of the default values from the $\mathcal{HORUS}$ database

3. INTERACTIVE VISUALIZATION CONCEPTS

To remember the situation so far, the "traditional" way to experimentally find out proper parameter values is try-and-error. This fact we cannot alter, of course but we can improve the overall time needed to get the desired values. The problem is that the specification of parameter values and the visualization of results are decoupled: after selecting values, one usually has to push some button to start calculation and then, after a short delay, the results are displayed. In a situation where the user doesn't have much experience, or when the problem is too complicated to achieve proper results within a few repetitions, the summarized calculation time can slow down the whole process far too much. Furthermore, the decoupling of data input and result displaying again reduces the potential power of the above introduced input method: the numerical slider from the section before, for example, is ment to be an aid to experimentally find a desired value, not to specify a value already known; it would be most useful when any value manipulation would show a result immediately, just like turning the frequency knob on an old radio receiver. It should be mentioned at this point that this goal cannot be achieved with standard hardware at this time, because single processor systems are not yet fast enough to provide the required raw computing power. We would like to mention, however, that the point at which this could become possible, may not be too far away: multiprocessor workstations and modern multi-threading operating systems have become definitely affordable and we are modifying $\mathcal{HORUS}$ for usage in parallel environments at the time, so in some future version of the system a solid real-time visualization is likely to become possible. But for now, we have to take a look at what can be done so far.

3.1. Using data reduction to speed up calculation

First of all, how fast should the response be? Dragging a slider for example, the user can easily produce a hundred values per second data rate; if we wanted to have a smooth coupling with the result output, the system would have to do the calculation and display the result in less than 10 milliseconds. Assuming a standard

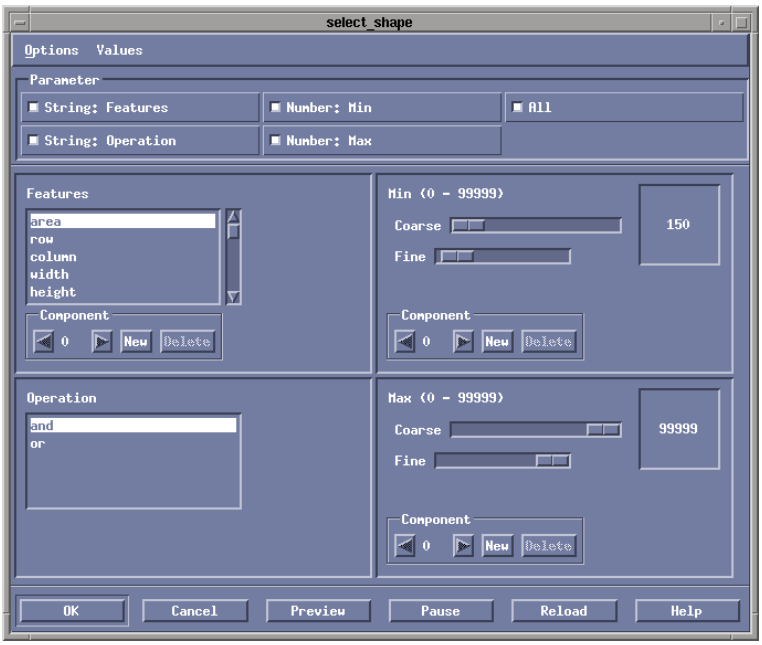

Figure 5. Structure of a *form* with four *dialogs*

512×512 pixel greyscale image, which means 256 KBytes of data, we would need a processing power of more than 256 MIPS, if we had a fictive 10 operations per pixel (including the result output!). But even if we'd drop the input data rate to 10 values per second, it still would work only for the simplest operators, because ten CPU operations per pixel can hardly be reached. If we switch to multi-channel color images, guess what happens then. So if we like to have it working at all, we must dramatically reduce the amount of data, which have to be processed; we now will analyze some strategies, how this can be done:

1. **Data compression.** This doesn't work for images, because the system can't perform any calculations on Lempel-Ziv-, Huffman- or other-encoded image material without decompressing it first; but there is a compression algorithm for regions which can be used: the RLE (run-length encoding). Representing regions with lines, which can be stored as starting position/length-tuples, significantly reduces memory usage and the material doesn't need pre-decompression to be processed. $\mathcal{HORUS}$ uses this compression method by default with regions, so region based operations may be fast enough, but unfortunately not always are.

2. **Virtual simulation of the operation.** We could transform the graphic adapters colormap to simulate contrast or brightness modifications, for example. Of course, there is no transformation done at all with the image material, hence we have a true real-time processing capability.

3. **Subsampling the image material.** By skipping a certain percentage of the image's pixels, data can be reduced significantly. Unfortunately, with this technique we change the information contained in the image, so the processing results probably won't be the same afterwards.

4. **Cropping the image to a small region of interest.** As the amount of processed data behaves proportional to the area being calculated, a reduction of the region of interest (called *ROI*) can speed up calculation a lot. Furthermore, this needn't be a severe restriction: when performing a single operation, we are very often only interested in a certain detail of an image anyway.

5. **Repetition of the cropping or subsampling method until the whole image material is processed.** If we do so, the total calculation time can't be lowered, of course, but the user has the ability to interrupt the process by supplying new values, if the old ones turn out to be improper and apart from that a fairly smooth impression of something being calculated at all is produced.

The first method, of course, is no general purpose solution, for it doesn't work with images and will work only with simple regions (which can be compressed a lot). Furthermore, it is used already within the *HORUS* system, so every enhancement it can do, is already done. The second method also isn't general, so it can't be used. The third technique could be a solution, but it won't work with regions due to their RLE representation, and it would be a doubtful choice for images, too, because the calculation result could probably dramatically differ from the original output. The fourth and fifth approach are the ones we have chosen to use: as already mentioned, the situations, where only a small ROI is needed to judge the correctness of certain parameter values, seem to be rather often. For example, if one has an image of an animal and is interested in analyzing a certain attribute (let's say the eyes), it might be o.k. to only process a close area around this attribute. To process a whole image, the repetition method is the best we can offer; the user doesn't have to wait until the whole source data is processed, but can interrupt the calculation and specify new values, if desired. In this case, the repetition cycle must be restarted, of course. Figure 6 shows, how the repetition mechanism works: small rectangular stripes are sequentially processed from the center towards the outside. The point where the calculation starts, defaults to the middle of the image, but this may be changed by the user. Thus, each operator form contains commands to influence the way the mechanism works: in the options menu, a new ROI, which also acts as the starting area for the repetition mechanism, can be drawn inside the visualization window, or it may be extended to fit the whole image. There are three different visualization policies, also: `continuously` means repetition switched on and restart with each new value supplied; `simple` means the same, but the process is restarted only when a slider is released and `ROI only` means repetition switched off.

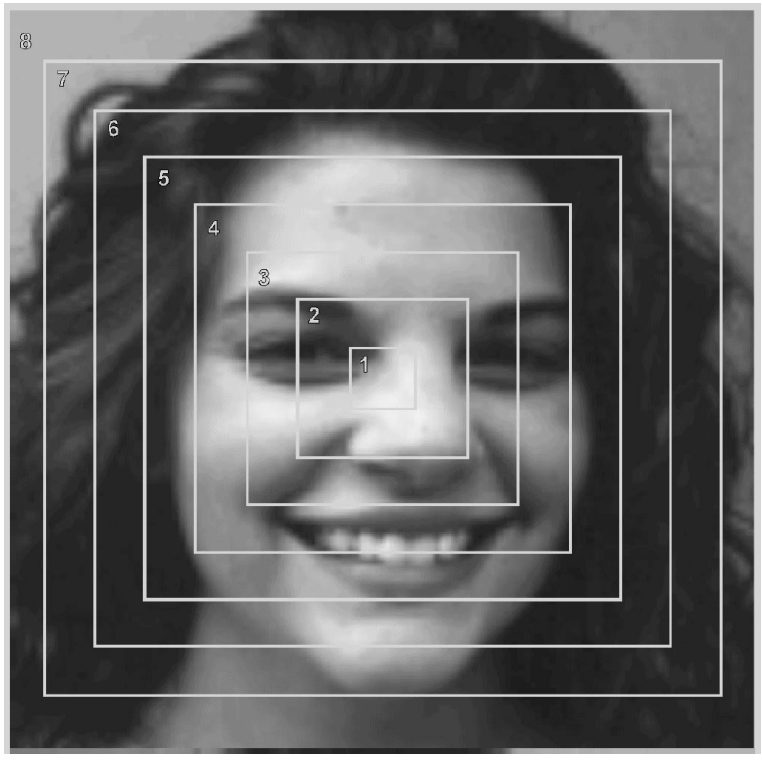

Figure 6. Stepwise expanded execution of an operator

Unfortunately, like with subsampling, the repetition method sometimes produces strange results: some operators behave anomalous at the image borders, which, of course, becomes visible when smaller image parts are processed sequentially. Now that we have the basic techniques to supply proper input mechanisms tightly coupled with result output, we will go a little further and describe how these can be integrated to create a user-friendly program development environment for image processing tasks.

4. INTEGRATION OF INTERACTIVE PARAMETER INPUT

The principles and concepts described so far can't be used stand-alone, of course; instead we have to think about a way, in which they can be used to enhance the rapid prototyping process. We have decided to add this extra funcionality to our existing *HORUS Develop*, which we have introduced in another SPIE paper before.[4] *HORUS Develop* is an integrated program development system for image processing tasks. Formerly, it did support only a textual input method, which we have kept, because some experienced users may prefer it. With the textual method, the user has to choose an operator which may be, after all parameter values are supplied, inserted into the program currently being developed. With the new input method, however, the semantics has become slightly different: as mentioned before, the user still can rely on textual input, and in this case, nothing changes. But instead of typing in the values, one can press a *parameter*-button, which then raises one of the above described parameter forms. Since the system already knows the operator and all the parameter types, this form can be automatically configured in the way described in section two.

Once the form is raised, the user can start experimenting immediately, without changing anything in the program. This is a rather important fact indeed, because the technique is definitely meant to be an experimental aid in finding parameters, and should not interfere with anything (like variables, e.g.) in the program developed so far. Once the desired result is achieved, the user may press the *Ok*-button in the form and the program is updated with the newly found parameter values; this has exactly the same effect, as if the values would have been typed in textually. Some other useful functions are also accessible from the form: of course the window can be dismissed via a *Cancel*-button; since it will be desirable to check the results for the whole image, if only a smaller ROI was used, we have supplied a *Preview*-button; there also is the possibility to *Reload* the original image, we have a *Pause/Continue*-button to temporarily stop the repetition process and a *Help*-button.

An important question is, how the form should behave, if the user presses the *Ok*-button and thus updates the program; we have chosen to keep the form active in this case. Since the form is still attached to the program line it was called for, it can be used to manipulate the corresponding operator at a later time; when this is done, a jump is made to the point, where the new values arise. Thus, the forms act just like data sources, which additionally can be used to manipulate the programs control flow. There is another feature, which is useful due to the fact that some operators may require pre- or post-processing: the user can select a certain amount of program lines (in the options menu), which are processed before and/or after the current line. With this feature, a whole block of code can be evaluated, while the control data is sent to any of the block's program lines.

5. CONCLUSION

In this paper, we have seen the need for an improved data-input technique. The out-of-date method of specifying data textually not only is annoying work, but also completely ignores the semantic characteristics of different data types. We have seen that semantically well-suited input dialogs can be automatically built for each parameter using object-oriented principles, so the user can supply control parameter values much more easily and intuitively. Furthermore, we found that supplying this improvement also requires an adequate coupling between input and output, so the user gets feedback from the system while simultaneously experimenting with input values. We took these concepts and updated the already existing *HORUS Develop* application, which is a development tool for image analysis programs. This way it was possible to significantly enhance the way the user can interact with the system: the time needed to find proper values for operator parameters can be lowered a lot and the paramters' semantics can be seen more clearly. The ability to tailor some of the parameters' attributes can be very useful; a stepping or a value range, which perfectly fits one situation, can be totally inadequate in another case and vice versa. Several input forms can be open simultaneously, so the user can pick

some critical operations anywhere in the program context and can freely experiment with control parameter data for them; the programs control flow then will be behave accordingly to the data the user supplies.

It also showed up that there still is no fully satisfying way to handle the output: because of the immense processing power, which is required for real-time visualization, we had to find another solution. We chose to split the source image material into several adjacent stripes, which are calculated sequentially, thus the user can interrupt the process. Another method is to use only a small segment of the whole image, which meets the fact that often a smaller region of interest will be o.k. to solve the problem. Finally, the goal to have a more solid real-time visualization may be achieved in the near future, since we currently are working on a multiprocessor-ready version of $\mathcal{HORUS}$.

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. W. Eckstein, C. Steger, "Interactive Data Inspection and Program Development for Computer Vision", *Visual Data Exploration and Analysis III, Georges G. Grinstein, Robert F. Erbacher (Editors), Proc. SPIE 2656*, pp. 296–309, Feb. 1996.
5. B. Radig, W. Eckstein, K. Klotz, T. Messer, J. Pauli, "Automatization in the Design of Image Understanding Systems", *Proc. 5th International Conference IEA/AIE 92*, Paderborn, 1992.