IPVT: A GRAPHICAL ENVIRONMENT FOR THE DESIGN OF COMPUTER VISION APPLICATIONS

Daniel J. Gonçalves, Rui E. Cruz, Luis T. Baptista, Jorge F. Nunes
INESC - Computer Vision Group
Rua Alves Redol 9, P-1000 Lisboa
Portugal
Fax: +351-1-3145843, e-mail:dlr@vision.inesc.pt

Abstract: The need to repeatedly make experiments and test new algorithms with different parameters using common basic processing operations often arises in the Computer Vision field. This paper describes a system that helps to rapidly and easily perform experiments and create new algorithms using a graphical user interface to manipulate basic processing blocks.

1 - The IPVT System

Most computer applications in the computer vision field often use several well known and simple algorithms that are used as building blocks connected together towards a larger goal. It would be desirable to have a system that would allow a researcher to easily manipulate and interconnect these blocks during the prototyping phase of an application.

Such a system would also, be useful when dealing when team research. In such situations, the reuse of code written by different people often raises some problems, such as the need to maintain a well defined routine interface that should be strictly followed by all the researchers. Also, the data-structures of the programs would need to be shared by everyone, in order to ensure compatibility. So, the system could be seen as a means to share tools already used on past experiments, thus becoming a form of code reuse.

A preferred environment would make use of graphical user interface. This interface would facilitate the manipulation of different tools, hiding all the interconnection details from the end user.

Some applications exist that follow this approach. Mathworks' Matlab with its Symulink extensions is an example, mainly dedicated to the simulation of physical systems. The Khoros system is another example, this one targeted specifically to the image processing community. Nevertheless both these systems have some drawbacks. The simulations in Matlab are implemented on an interpreted language and this may hamper its utility, as image processing and computer vision algorithms are very demanding on computing power. The Khoros framework imposes a series of conventions upon the small programs implementing the building blocks, thus making imperative that these be specifically written to run under the Khoros environment.

The IPVT (Image Processing Visual Tool) was developed within the Computer Vision Group at INESC. It is a graphical environment allowing an easy development of image processing algorithms. The system was implemented under a UNIX operating system and uses the X window system graphical interface. Some of its most relevant features are:

- A block diagram representation of image processing algorithms.
- It allows the user to graphically build block diagrams by interconnecting existing blocks.
- It is easy to learn and use.
- It provides mechanisms to easily change the parameterization of algorithms.
- It supports multiple users in a networked environment, facilitating the sharing of algorithms.

Figure 1: The main window.

- It is possible to add multiple abstraction layers over existing algorithms by treating existing sets of blocks as a new block.

- It tries to maintain compatibility with existing programs.

The user interacts with the IPVT system as he or she would with a typical X windows application. The main way of interaction is through a window with a menu and a toolbar, like the one in figure 1.
In the following subsections we will outline some basic concepts on the IPVT system and we will describe the possible forms of interaction with the user.

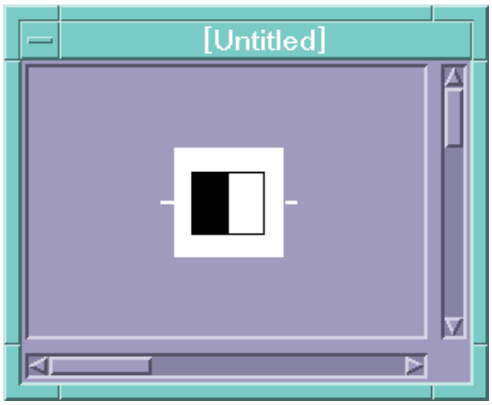

Figure 2: An example of a tool.

1.1 - The Tools

The most simple element in the system is the **tool**. A tool represents a basic processing operation. It encapsulates an executable program that performs the operations. The executable must adhere to some conventions described in section 1.5. The tool appears to the user as a rectangular box with an icon identifying it. Each tool has associated with it a name, a description string and a set of **inputs**, **outputs** and **parameters**. The inputs and outputs are depicted as small dashes on the sides of the box representing the tool. The inputs are on the left and the outputs are on the right (see figure 2).

A tool interacts with other tools through its inputs and outputs. For example, a tool performing an image filtering operation might have one input and one output. The input receives the image to be filtered and from the output comes the result of the filtering.

The parameters affect the way a tool performs its work. They may be an integer representing a threshold or a file name, for instance. All the parameters are assigned default values at design time.

Note that the system makes no assumptions on the format of data that flows from one tool to another. The user is responsible for maintaining the coherence of the data between tools by only connections outputs and inputs that generate and expect the same type of data. This allows the system to be used with all kinds of tools, since the imposing of a specific set of pre-defined data-types would result in a very important loss of genericity of the system. Without imposing any constraints on the data-types that can flow from one tool to another, the system can even use programs for which we don't have access to the source code, if they follow the conventions defined in section 1.5

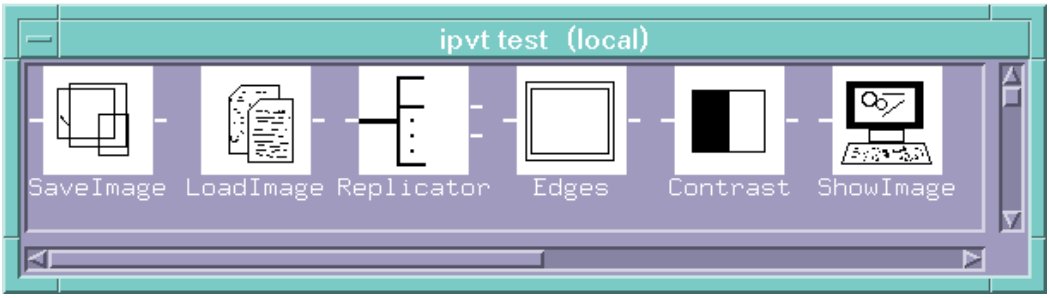

Figure 3: An example of a library.

1.2 - The Libraries

The tools are stored in one or more **libraries**. A library is a set of tools sharing a common functionality. For example, there could be a library of image filters and another of edge detectors. It is possible to create and delete libraries as needed. A tool is assigned to a library at design time. As a way to support multiple users each library is either **local** or **global**. Local libraries may be created by any user. A local library may only be used or manipulated by the user that created it. Global libraries can only be created by a system manager, who is also responsible for their maintenance (adding and removing tools, etc). However, all users are allowed to use tools from global libraries. It is thus possible to share tools among all users while protecting them from unwanted alterations.

A user sees a library as a window with several tools inside it (see figure 3).

1.3 - The Applications

An application is a composition of tools without any dangling connections. To build an IPVT application the user places the desired tools on a canvas. The tools are selected from a library by clicking on its icon in the library window. Multiple instances of a tool may exist in one application. The tools are connected by selecting one of its inputs or outputs and dragging the mouse pointer to the output or input, respectively, of another tool. To represent a connection a line is drawn between the tools. See figure 4 for an example of an application.

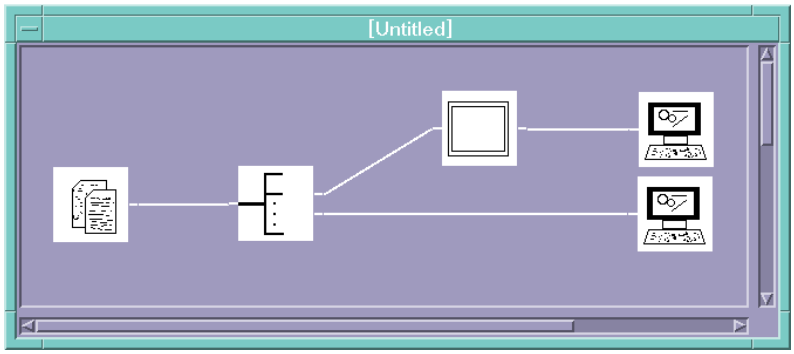

Figure 4: An example of an application.

After all connections are made the user can order the system to run the application by pressing the appropriate button in the main window (figure 1). Each tool underlying process will then be spawned by the system. The use can abort the execution at any time before the normal completion of the application. If one of the tools aborts its operation the user will be informed of the cause of failure in a pop-up window.

The user can change the parameters of a tool by clicking on it with the right mouse button. A dialog box will then appear allowing the user to define the new values for the parameters. When altering the parameters of a tool, the user is aided by a small description of the meaning of each parameter, written when the tool was created.

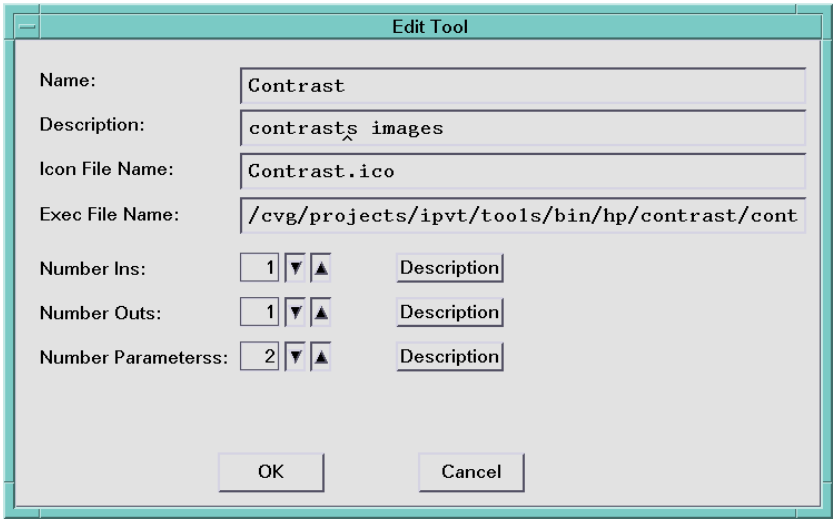

Figure 5: The dialog box for creating a new tool

1.4 - The Composite Tools

One of the most important features of the system is the possibility of creating **composite tools**. A composite tool is one that does not directly encapsulates a program but is instead composed of a set of connected tools. As an example let us imagine that a user has a tool that acts as a filter and another one that extracts the edges of an image. Furthermore he or she frequently uses these tools connected to each other. It is possible to create a new composite tool using these two, calling it *filtered edges*, for instance. From now on the user needs not to manually couple the two independent tools anymore as he or she may now manipulate them as single entity. Once created the composite tools are treated by the system as any other tools. It is possible to store them in libraries for future use, create multiple instances of them or even create new composite tools with other composite tools inside them.

To create a composite tool the user must select the appropriate option from the toolbar in the main window. In an application window the user selects a set of tools by dragging the mouse pointer. As the user drags a bounding box rectangle identifies the selected tools. The selected tools and its connections will be removed from the application window and will be replaced with a composite tool. The inputs and outputs of the new tool will correspond to the inputs and outputs on the previously selected tools not yet connected and to the connections ending outside the bounding rectangle.

Ordinary tools and composite tools are represented with different colors in the application window. By pressing the middle mouse button over a composite tool a new application window will appear containing the tools forming the composite tool. The user may then edit each one of them to modify the respective parameters, if needed.

1.5 - The Creating New Tools

The user may create a new tool by selecting the appropriate menu command in the main window. A dialog box like the one in figure 5 will be displayed and the user is expected to fill in the fields. Some of the fields, like the name and description, are only used for informative purposes. The most important field is the executable name. It will contain the name of the executable file of the program that implements the operations of the tool. Other relevant fields are the number of inputs, outputs and parameters of the tool.

The user must specify a library where the new tool will be placed.

The program that implements the tool must adhere to some simple rules. These rules specify only the semantics of the arguments passed to the program at execution time. The program will receive as arguments the name of the files it will treat as inputs, the name of files it will use as outputs and the values of the parameters. Thus, the program must be prepared to read its input from files and to write its output to files. No other restrictions are imposed.

2 - Technical Considerations

In this section we describe some technical details about the system implementation. The reader may skip this section if he or she is not interested in such technicalities.

The system was developed under the HP-UX~9.01 operating system. It was written in C++ using the Gnu compiler. The graphical interface is based on the Motif GUI libraries.

2.1 - The Tools

An application consisting of various tools and its connections is internally represented as a graph. In this graph tools are the nodes an the connections are the arcs. Two classes are derived from a basic *Tool* class. One of them represents ordinary tools and the other one represents composite tools. Each tool has (among others, related with the graphics display, such as screen coordinates, etc., and, thus, not relevant to this discussion) the following attributes:

- identification number, that uniquely identifies the tool in the system;
- number of inputs and outputs and their descriptions;
- number parameters, their default values and their descriptions;
- two arrays (one for the inputs and another for the outputs) of *Connections*.

Connection is a class that stores information that refers to a given input or output of a given tool (for example, (2,1), to represent the first input or output of the tool with id 2). The disambiguation of the second value is achieved by the use given to the variable: if in the inputs array, it indicates the output to which that input is connected and vice-versa. It is, in a practical sense, the way used to represent the graph's arcs;

An ordinary tool corresponds, in the UNIX environment, to a process that is to be run. Thus, it has another field, used to store the executable's name. The processes are launched by the system every time the user decides to run the application. When all the processes that compose an application terminate, so will the application. In addition if the application's execution is aborted, they will all be terminated.

A composite tool is represented as a sub-graph of the main graph. This sub-graph will be treated as a node of its super-graph. It can be stored as any other tool by means of inclusion polymorphism. It will have some new fields, such as the sub-graph in itself and the mappings from the higher level inputs and outputs and the actual inputs and outputs of the graph (more on this below). A composite tool will react differently to some actions. When an ordinary tool is told to run it simply launches the appropriate process. But when a composite tool is told to run it will act as a graph, telling the tools inside it to run. This is a recursive structure. It allows the user to keep on creating progressively higher abstraction levels without any problems. This way the complexity of an application can be greatly reduced when a large number of tools is involved.

When the user creates a new composed tool, the selected sub-graph will be removed from the application and stored as the graph of the composed tool. The tools retain their ID numbers and two special arrays are created to map between the inputs and outputs of those tools and those of the composed tool (for example, to indicate, in a specific situation, that the third input of the composed tool corresponds to the first input of the fifth tool of the sub-graph).

2.1 - The Communications

The connections between tools were implemented using UNIX named pipes. They were used to allow processes to access them from their names, without having to know their descriptor beforehand. This allows named pipes to be used as ordinary files would, although only sequential access is allowed. This will enhance the genericity of the system, since it can use general purpose programs, and not only those specially designed for it. Also, they are directly created in the Unix *inode* tables, thus acting more efficiently than normal pipes.

When preparing to run the application the system will automatically create the named pipes and their names will be given as arguments to the programs associated with the tools.

Figure 6: An empty working area.

When a process tries to open a pipe for reading, it will be suspended until another process opens the same pipe for writing. Similarly, when a process opens a pipe for writing, it will be blocked until the other end of the pipe is opened for reading. These facts were used to our advantage, since they allow the synchronization between the tools to be automatically handled by the operating system.

This is, of course, a great advantage, since the system needs not to worry about synchronizing the processes. Note, however, that, in some special cases, where the application has cycles, if the tools are not prepared for that situation they might not run properly. This can happen if a tool is suspended, waiting for an input that will only be available when the tool has opened it's outputs (and those are not yet opened, of course). This type of applications is, however, very rare.

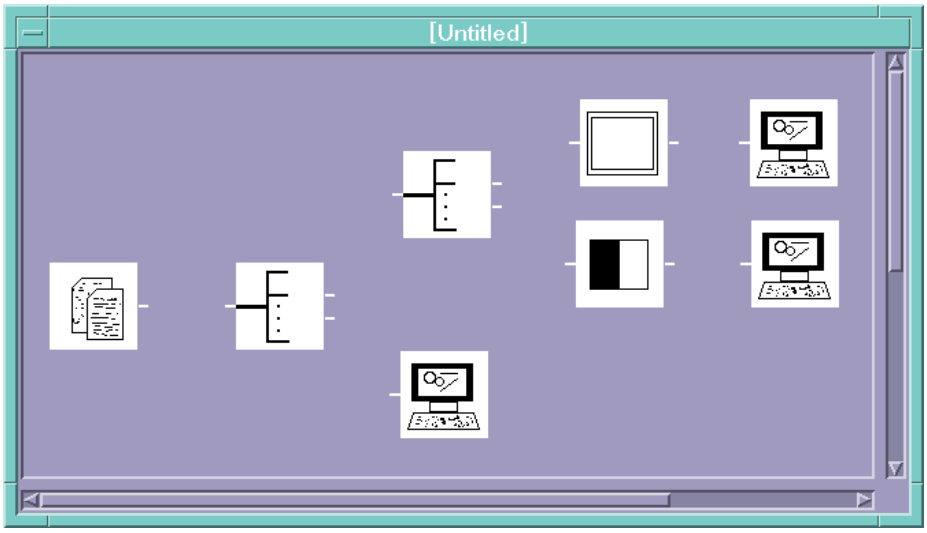

Figure 7: An application where the tools are not yet connected.

3 - An Example

We will now describe an example where we use the IPVT system to build and run an application. We intend to build an application that changes the contrast of an image and, simultaneously, performs an edge detection operation on the same image. The relevant tools are assumed to have been already developed.

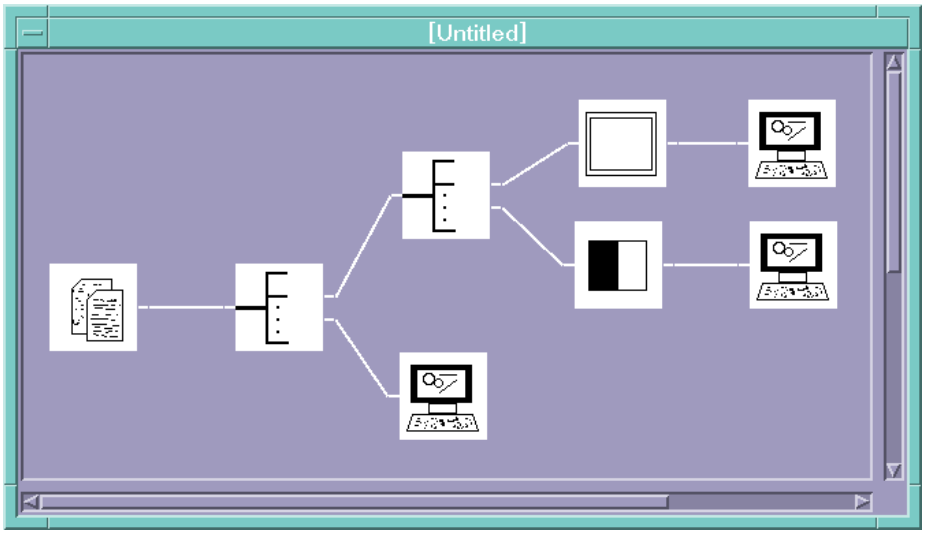

Figure 8: A complete application.

We start by creating a new application choosing the appropriate option in the file menu. We are then presented with a blank working area (see figure 6) where we proceed to place the desired tools. To do so we select the tools from an open library (see figure 3) containing the tools we need. Besides the *Contrast* and *Edges* tools that perform the required operations, we will also need some other tools. The *Replicator* simply copies its input to both its outputs. The *LoadImage* tool reads an image from a file and dumps it on its output. The *ShowImage* tool expects an image at its input and opens up a window showing that image.

To place a tool on the working area (figure 6) we select it in the library window (figure 3) by clicking it with the left mouse button. We then enter the tool inserting mode by pressing the appropriate button on the tool bar (figure 1). While we are in the tool inserting mode a rectangle the size of the tool will follow de cursor whenever it is inside the working area. The tool is placed in the working area by clicking the left mouse button over the desired position. In figure 7 we show the tools already positioned in place.

We may now create the connections between the tools. We start by pressing the connection mode button on the tool bar. A connection is made by clicking near the output of a tool and clicking again at the input of another tool (or vice-versa). You can see the final result in figure 8.

Before we order the application to run we must specify the file with the image to be processed. We do that by changing the appropriate parameter in the *LoadImage* tool. We now order the application to run by pressing a button on the tool bar. As a result three windows appear containing the results (see figure 9). These windows were created by the *ShowImage* tool. If we wished to make experiments with different parameters, like a different threshold for the edge detection algorithm, we would only have to edit the tool's relevant parameters and command the application to run again.

Figure 9: The results after running the application

4 - Conclusions

We have described a system that allows a rapid and easy prototyping of image processing algorithms using already existing basic processing elements. This system will be used within the Computer Vision Group at INESC to support research on image processing and the development of software. Some improvements are already under study, such as the enhancement of the graphical interface, giving it additional functionalities as, for instance, easier ways to get information about a tool, context sensitive help.

Also, support for iterative processes is also under study. For instance, have the system to consistently re-launch some tools to repeat the same operation over different data (processing the different frames of a movie with the same filter used for a single image, for instance).

Acknowledgments

The authors would like to thank Prof. Jorge Marques of the Computer Vision Group at INESC for his useful suggestions.

Bibliography

[1] BERLAGE, Thomas. *OSF/Motif - Concepts and Programming*. Addison-Wesley Publishing Company, 1991.

[2] GLASS, Graham. *UNIX for Programmers and Users - A Complete Guide*. Prentice Hall International, 1993.

[3] JAIN, A. *Fundamentals of Digital Image Processing,* Prentice Hall International, 1989

[4] LIPPMAN, Stanley B. *C++ Primer*. AT&T Bell Laboratories, Addison-Wesley Publishing Company, 1989.

[5] MARQUES, Jose Alves and GUEDES, Paulo. *Fundamentos de Sistemas Operativos*. Editorial Presenca, 1990.

[6] YOUNG, Douglas A. *The X Window System - Programming and Applications with Xt - OSF/Motif Edition*. Prentice Hall Inc., 1990.