A GENERAL-PURPOSE LOGIC-BASED VISUALIZATION FRAMEWORK

Camil Demetrescu† [1]

Irene Finocchi‡

†Computer and Systems Science Department
University of Rome "La Sapienza"
Via Salaria 113, 00198 Rome, Italy
demetres@dis.uniroma1.it

‡Department of Computing Science
University of Rome "La Sapienza"
Via Salaria 113, 00198 Rome, Italy
finocchi@dsi.uniroma1.it

ABSTRACT

This paper describes a general-purpose visualization framework for producing images through reasoning processes. The approach is significantly different from the usual imperative one: indeed, the visualizer retrieves visual information from a knowledge base that may depend on an external source of data. This knowledge base is written by the user in a logic-based language and contains declarations about visual objects, their interrelations and their geometric and retinal features. Different algorithms for querying the knowledge base are presented, discussing both their temporal complexity and their robustness.

Keywords: visualization, logic languages.

1 INTRODUCTION

The action of a visualizer consists of building a visual representation of some kind of information, usually by creating bitmapped images. In this paper we focus our attention on the way a visualizer receives visual information, rather than on the techniques it uses for rendering an image.

We present an intelligent logic-based visualization framework that appears to be well suitable for software and data visualization purposes and for high-level visual debugging of programs (see [Price94a] and [Cresc97a]).

The great expressivity and flexibility of logic languages in specifying properties and relationships between objects make them very attractive for our aims. On the other hand, implementing a logic-based framework requires a quite complex software technology that in some cases may affect visualization's performances.

The paper is organized as follows. After describing the framework (section 2), we discuss the application of logic languages for representing visual information (section 3); then we present basic information retrieval techniques used by the visualizer (section 4) and finally we introduce a restricted graphical vocabulary and we show two visualization examples (section 5).

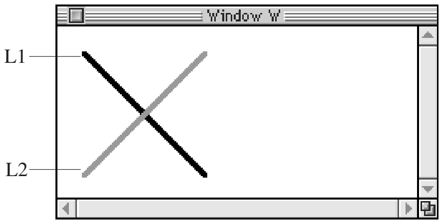
2 THE FRAMEWORK

A visualizer is usually thought as a server of graphical facilities that accepts commands from a driving client process and produces images on its behalf as shown in Fig. 1.

[1] Author supported in part by EU ESPRIT Long Term Research Project ALCOM-IT under contract no. 20244.

Knowledge base:

There is a window W
There is a line L1
There is a line L2
Line L1 is contained in window W
Line L2 is contained in window W
Coordinates of line L1 are (10,10)-(50,50)
Coordinates of line L2 are (10,50)-(50,10)
All lines standing behind any object are Blue
Line L2 is over line L1



Fragment of a possible conversation session:

Visualizer:	Which windows must be onto the screen ?
Inference engine:	Just window W.
Visualizer:	Which points must be in window W ?
Inference engine:	None.
Visualizer:	Which circles must be in window W ?
Inference engine:	None.
Visualizer:	Which lines must be in window W ?
Inference engine:	Line L1 from (10,10) to (50,50) and line L2 from (10,50) to (50,10).
Visualizer:	What's the color of line L1 ?
Inference engine:	Blue.
Visualizer:	What's the color of line L2 ?
Inference engine:	I don't know.
Visualizer:	(Then I will use my default color Grey.) Is line L1 in front of line L2 ?
Inference engine:	No.
...	

Figure 3: Example of an informal conversation session

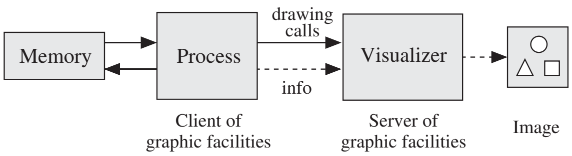


Figure 1: Procedural visualization framework

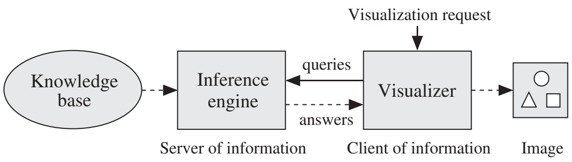


Figure 2: Knowledge-based visualization framework

In the framework we are going to present (see Fig. 2) the image generation process follows a different approach: the visualizer is the client of an inference engine that acts as a server of information retrieved from a knowledge base. This information retrieval is carried out through a tight conversation between the visualizer and the inference engine: the first one issues a sequence of queries about what to visualize and the second answers after consulting the knowledge base.

The dialogue goes on according to a precise communication scheme, that we will refer to as the *query protocol*, describing visualizer's questions and their order. In Fig. 3 a knowledge base, a possible informal conversation session and the final image produced by the visualizer are shown.

Note the knowledge base does not necessarily provide a complete description of all features of the visual world: in our example, the color of the frontmost object is left undefined. It is up to the visualizer to provide default values for all missing pieces of information before generating the final image.

It is worth observing that the image building is triggered by a visualization request coming from an external agent that is responsible of mantaining the final image consistent with its specification if the knowledge base is evolving over the time. Although this requirement is not strictly necessary, an automatic update of images allows a powerful visual feedback of knowledge base's temporal evolution.

From user's perspective, some visualization code must be written in both the knowledge-based and procedural frameworks, but in the first one the program consists of a sequence of drawing instructions, while in the second it is a repository of information expressed in a declarative form about visual objects, relations among them and their attributes.

Up to this point, a visualization request leads to the generation of a single image whose description is completely specified by the knowledge base. Hence, producing different images requires writing different knowledge bases.

However, this model is not enough powerful in most real situations where the user is interested in visualizing information stored in a computer memory. A possible solution to this problem is to parametrize the knowledge base with respect to an external source of information such as a file or a main memory segment. For the sake of concreteness, in the following we assume that the knowledge base can depend on actual values con-

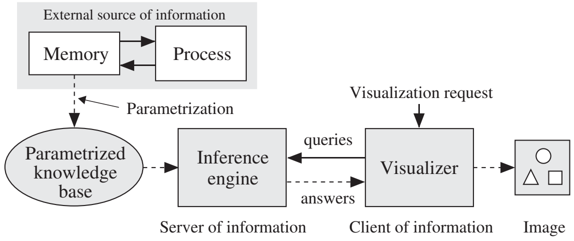


Figure 4: Parametrized knowledge-based framework

tained in the memory image of an underlying process. This means that changing even one of its locations may lead the inference engine to produce different answers to the same queries of the visualizer allowing it to generate different images. Fig. 4 is a refinement of Fig. 2 according to this concept of parametrization.

3 KNOWLEDGE REPRESENTATION USING PREDICATES

The first step for representing knowledge consists of choosing a suitable encoding language. One of the earlier solutions to this problem in the 1970s was to write sentences about objects and their properties through predicates of the first-order logic, but various criticisms against this approach have led to the development of other models, such as frame systems and semantic nets (see [Ginsb92a] for details).

However, the logic approach seems to be especially suitable for our purposes, strictly related to the visualization, since the description of graphical objects and their properties can be naturally formulated as axioms and inference rules, leading to a powerful and expressive framework for describing visual worlds.

For example, the knowledge base shown in Fig. 3 may be easily encoded as sentences of the first-order logic as follows:

$window(W)$
$line(L1)$
$line(L2)$
$in\_window(L1, W)$
$in\_window(L2, W)$
$line\_coordinates(L1, 10, 10, 50, 50)$
$line\_coordinates(L2, 10, 50, 50, 10)$
$in\_front\_of(L2, L1)$
$\forall x\ \forall y\ line(y) \wedge in\_front\_of(x, y) \rightarrow color(y, BLUE)$

All sentences in the previous knowledge base are axioms but the last one, that is a rule asserting each line y behind a generic object x is blue. Note variables (x and y) are lowercase identifiers while constants (10, 50, W, $L1$, $L2$, and $BLUE$) are uppercase alphanumeric strings.

3.1 Standard predicates

In the logic approach, a single query of the visualizer consists of a goal theorem that the inference engine tries to prove starting from the sentences in a knowledge base Δ provided by the user as a specification of the visualization.

For example, if Δ is a Horn base, the inference engine may use a resolution algorithm (see [Lloyd87a]) to check out if a given goal predicate (e.g. $line(x)$) is entailed by Δ. In this case the unification process may bear variable assignments that validate the predicate with respect to the given axiomatic system.

Therefore, the information the visualizer receives is both the truth value of the goal predicate (e.g. *true* if there is at least a line in the visual world, *false* otherwise) and possibly a list of binds between variables and values obtained by literals unification (e.g. *which* are the lines).

Indeed, if Δ is the base given above with the last rule rewritten as the Horn clause

$$color(y, BLUE) \vee \neg line(y) \vee \neg in\_front\_of(x, y)$$

the conversation session of Fig. 3 may be driven by the sequence of queries shown in Fig. 5. Table's truth values and binding lists have been computed through the resolution algorithm for each query of the visualizer.

Note for some goals the inference procedure is run until all binding lists validating them have been found, since we are often interested not only in proving a theorem, but also in getting values. For this reason, the query $line(l)$? has two answers, $l = L1$ and $l = L2$, this meaning that there are actually two lines in the visual world. Thanks to this feature, the visualizer can easily get sets of objects to build up complex images.

As we have pointed out in section 2, visualizer's queries follow a precise communication protocol built in the visualization system. According to it, goal predicates asked by the visualizer have a predefined signature (name and list of arguments) that the user must agree with in writing the knowledge base.

From now on we will refer to predicates matching

Query	Truth value	Returned values
$window(w)$?	$true$	$w = W$
	$false$	-
$line(l)$?	$true$	$l = L1$
	$true$	$l = L2$
	$false$	-
$in\_window(L1, W)$?	$true$	-
$in\_window(L2, W)$?	$true$	-
$line\_coordinates(L1, x_1, y_1, x_2, y_2)$?	$true$	$x_1 = 10, y_1 = 10, x_2 = 50, y_2 = 50$
$line\_coordinates(L2, x_1, y_1, x_2, y_2)$?	$true$	$x_1 = 10, y_1 = 50, x_2 = 50, y_2 = 10$
$in\_front\_of(L1, L2)$?	$false$	-
$color(L1, c)$?	$true$	$c = BLUE$
$color(L2, c)$?	$false$	-

Figure 5: Queries of the visualizer to a Horn database processed through the resolution algorithm

against visualizer's goals as *standard predicates*. Basically, they belong to two main categories:

- those concerning the existence of visual objects, called *constructors*;
- those related to properties of visual objects (retinal and geometric features and interrelations), called *descriptors*.

In our previous example, *window*, *line*, *point*, and *circle* are constructors, while *in_window*, *line_coordinates*, *in_front_of* and *color* are descriptors.

3.2 Expressing predicates in Alpha

In this section we informally introduce Alpha, a logic-based programming language that will serve as a basis for our next examples.

An Alpha program is a set of definitions of boolean functions that we will refer to as *predicates*. Each one has a signature (*head*), defining its name and its formal parameters, and one or more *bodies*, specifying how its results must be computed.

Arguments may be either "by name" (if prefixed by the keyword `Out`) or "by value" and have integer type.

Multiple bodies for the same predicate may be introduced either through the keyword `Moreover` or by repeating its head. Each body consists of a (possibly empty) sequence of `Assign`, `If` and `For` statements whose execution is carried on according to the following computation rules:

- the `Assign` statement puts values into parameters and/or local variables (e.g. `Assign X=10`) and lets the computation proceed to the next statement;
- the `For` statement invokes a predicate, allowing it to put values into passed variables (e.g. `For N : myPred(N)`). If the called predicate returns *true*, then the computation proceeds to the next statement, else backtracks to the previous `For` statement;
- the `If` statement checks for a condition (e.g. `If X+Y<=Z/2`). If this one holds then the computation proceeds to the next statement, else backtracks as above;
- if there is no next statement to proceed on, the predicate returns *true*;
- if there is no previous `For` statement to backtrack on, another body of the predicate, if any, is computed. Otherwise the predicate returns *false*.

In other words, the predicate computation returns *true* when and only when the last statement of a body has been successfully completed. A subsequent call to the predicate from an external caller will then proceed from that point on, exploring new computation possibilities, if any. It is worth noticing that predicates are able to return different values on different calls thanks to the presence of more bodies and to the backtracking mechanism.

In Fig. 6 a simple example of two Alpha predicates, `Enum` and `OddEnum`, is given. The first one has three bodies and is able to enumerate values 1, 2, and 3 on subsequent calls. The second one invokes `Enum` and filters its result through an `If` statement in order to produce only odd values. Three successive calls to `OddEnum` are traced in the same figure.

Definition of Enum and Oddenum:

```
Enum(Out N)                          OddEnum(Out N)
    Assign N=1 Moreover                  For N:Enum(N)
    Assign N=2 Moreover                  If N%2==1;
    Assign N=3;
```

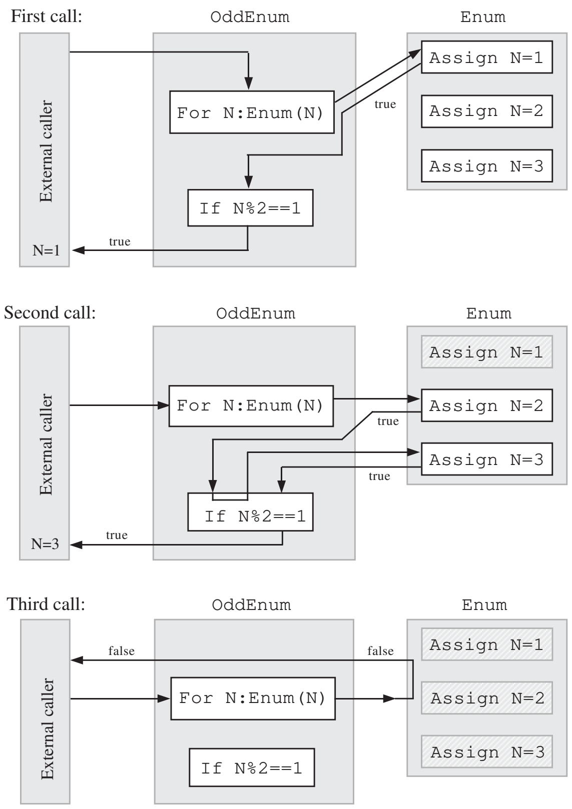


Figure 6: Definition of **OddEnum** and **Enum** and computation of **OddEnum**

For simplicity, from now on we will use the shortcuts "`myPred(1) ...;`" instead of "`myPred(N) If N==1 ...;`" and "`myPred(Out 1) ...;`" instead of "`myPred(Out N) ... Assign N=1;`". Moreover, we will prefix ALPHA constant identifiers with the symbol '_'.

In the context of our knowledge-based framework, a query of the visualizer consists now of an ALPHA standard predicate computation request. The inference engine evaluates the goal predicate according to the computation rules given above (that are a procedural interpretation of the underlying theorem proving process) and the results - truth value and possibly generated values - are returned to the visualizer.

3.3 Parametrizing ALPHA predicates

ALPHA is expressly designed for supporting the parametrization with respect to variables of an underlying program. In particular, we assume to deal with C programs that fill-in their own variables with information to be visualized.

The parametrization is realized by allowing first an ALPHA program to be embedded into a C source code, and then ALPHA statements to syntactically refer to its variables, according to their scoping rules. A good choice for making ALPHA predicates semantically "invisible" to the underlying program is to embed them as special comments delimited by `/**` and `**/` separators.

The following simple example shows how the background color of a window may depend on the sign of an underlying C variable, provided that `Window` and `WindowColor` are ALPHA standard predicates:

```
int i=0;
/**
   Window(Out _W);
   WindowColor(_W,Out C)
      Assign C = (i>0) ?  BLUE : RED;
**/
```

`Window(Out _W)` declares a window whose label is _W, while `WindowColor(_W,Out C)` declares its color. Note after each assignment to the variable `i` made by the process, the visualizer should ask the inference engine to re-evaluate the `WindowColor` standard predicate, allowing window's background to be red as long as `i` $<= 0$ and switching its color to blue as soon as `i` becomes > 0.

4 QUERY PROTOCOLS

Query protocols are the backbone of the knowledge retrieval process. They specify: (a) the algorithm followed by the visualizer for evaluating standard predicates (the *query algorithm*); (b) the signature of standard predicates; (c) the meaning of their parameters.

One of the main features of our logic-based framework is the total freedom in designing query protocols. It is not surprising that the visualizer can receive the same information in many different ways. Indeed, supporting various types of built-in query strategies may turn out to be extremely expressive.

In the following we present four different basic query protocols that may be combined together in order to design more complex ones. In particular, we focus our attention in using ALPHA predicates to produce generic sets of values, that in our context may represent either visual objects or features for them.

4.1 Different models of query protocols

Let Ω be an arbitrary set of elements and $S \subseteq \Omega$ a generic subset. We propose four basic protocols for retrieving S through successive evaluations of ALPHA predicates and we call them (a) *enumeration*, (b) *cardinality definition*, (c) *linked structure* and (d) *characteristic function* protocols, respectively.

Fundamental criteria for evaluating them are:

- their fitness for a particular purpose;
- their robustness;
- the temporal complexity of their query algorithms.

Informally, a protocol is *fragile* if a mistake by the user in writing the body of a predicate may introduce logic inconsistences that compromise the visualization process. In general, the robustness of a protocol can be enforced by adding consistency checkings in its query algorithm at the prize of some temporal complexity overhead.

If the body of a standard predicate `Pred` evaluated by a query algorithm $\mathcal{A}$ depends on an external parameter p (e.g. a C variable), we denote with $\mathcal{C}_{\texttt{Pred}}(p)$ the worst-case complexity of a single evaluation of `Pred` during the execution of $\mathcal{A}$, and with $\mathcal{C}_{\mathcal{A}}(p)$ the worst-case complexity of $\mathcal{A}$.

In Fig. 7 we sketch each basic query protocol, denoting them by `Enum`, `Card`, `Link` and `Char`, respectively. For each of them we provide the signature of standard predicates, the skeleton of the query algorithm and its temporal complexity. Finally, we give examples of predicates definitions and we show the sets of values they produce. Note that our description is general and the names of involved standard predicates are not conceptually relevant.

The enumeration protocol

The enumeration strategy takes advantage of the backtracking-based computation mechanism of ALPHA predicates: items of S are generated by an enumerative predicate, `SetItem`, that returns them one by one in its output parameter. Values are fetched by the query algorithm as long as `SetItem` is able to produce a new one. Lacks of robustness of this protocol are due to the possibility of the predicate `SetItem` both to enumerate values out of Ω and to generate the same item more than once.

The cardinality definition protocol

It can be used when S is a set of consecutive ordered items, such as the interval $\{1,\ldots,|S|\}$. In this case, the simple declaration of $|S|$ is enough to define the whole set. This strategy appears to be robust and easy to use, but is not as flexible as the previous one.

The linked structure protocol

According to this protocol the user must define a predicate `First` that generates the first item of S and a predicate `Next` that takes an element $i \in S$ and yields its successor, if any. The proposed query algorithm simply iterates on the predicate `Next` to collect items of S. Although this protocol is expecially well suitable for describing linked structures, it is extremely fragile, for it does not converge if items' successors generated by `Next` form a cycle.

The characteristic function protocol

A simple way for describing a set is through its characteristic function. This can be done by defining a predicate, `InSet`, able to classify items of the superset Ω as belonging to S or not. A natural technique for retrieving S proceeds by evaluating `InSet` for each element in Ω. The characteristic function protocol is the most robust among those presented so far, but it may be extremely time-consuming if Ω is too large.

Observe that protocols (a) and (c) allow the user to define an explicit ordering on S, while strategies (b) and (d) rely on a built-in evaluation order specified by the protocols themselves. This issue may be important when the ordering of items is relevant for the visualization.

5 DESCRIBING VISUAL WORLDS IN ALPHA

In this section, we provide a set of ALPHA standard predicates for describing visual objects. Due to the lack of space, we cannot present a complete general-purpose graphical vocabulary and we assume to deal only with windows, some elementary geometric objects (points, lines, rectangles and circles), and raster images. Signatures of standard predicates for describing them are given below:

Prot.	Signatures	Query algorithm	Temp. complexity	Definition examples	Defined sets
`Enum`	`SetItem(Out s)`	$S \leftarrow \emptyset$ while (`SetItem`(s)= $true$) do $S \leftarrow S \cup \{s\}$	$\mathcal{C}_{\texttt{Enum}}(p) =$ $O(\lvert S\rvert \cdot \mathcal{C}_{\texttt{SetItem}}(p))$	`SetItem`(Out s) For s : `OddEnum`(s);	$S = \{1, 3\}$

`Card`	`NumItem(Out n)`	$S \leftarrow \emptyset$
		if (`NumItem`(n)= $true$) then
		for i=1 to n do
		$S \leftarrow S \cup \{i\}$
			$\mathcal{C}_{\texttt{Card}}(p) =$
			$O(\lvert S\rvert + \mathcal{C}_{\texttt{NumItem}}(p))$
				`NumItem`(Out 8);	$S = \{1, 2, \ldots, 8\}$

`Link`	`First(Out` f)
	`Next(`f,`Out` n)
		$S \leftarrow \emptyset$
		if (`First`(f)= $true$) then
		begin
		$S \leftarrow S \cup \{f\}$
		while (`Next`(f,n)= $true$)
		begin
		$S \leftarrow S \cup \{n\}$;
		$f \leftarrow n$
		end
		end
			$\mathcal{C}_{\texttt{Link}}(p) =$
			$O(\mathcal{C}_{\texttt{First}}(p)+$
			$\lvert S\rvert \cdot \mathcal{C}_{\texttt{Next}}(p))$
				`First`(Out 0);
				`Next`(f, Out n)
				If ($f < 8$)
				Assign $n = f + 2$;
					$S = \{0, 2, 4, 6, 8\}$

`Char`	`InSet`(ω)	$S \leftarrow \emptyset$ $\forall \omega \in \Omega$ if (`InSet`(ω)= $true$) then $S \leftarrow S \cup \{\omega\}$	$\mathcal{C}_{\texttt{Char}}(p) =$ $O(\lvert\Omega\rvert \cdot \mathcal{C}_{\texttt{InSet}}(p))$	`InSet`(ω) If ($\omega \geq 0$) && ($\omega \leq 8$) && ($\omega \% 2 == 0$);	$\Omega = \{0, \ldots, 100\}$ $S = \{0, 2, 4, 6, 8\}$

Figure 7: Basic query protocols

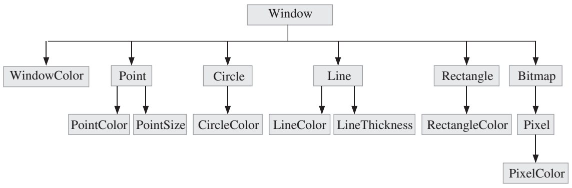


Figure 8: Partial evaluation order of standard predicates

```
Window(Out WinID)
WindowColor(WinID,Out Color)
Point(WinID,Out PointID,Out X,Out Y)
PointColor(WinID,PointID,Out Color)
PointSize(WinID,PointID,Out Size)
Line(WinID,Out LineID,
     Out X1,Out Y1,Out X2,Out Y2)
LineColor(WinID,LineID,Out Color)
LineThickness(WinID,LineID,Out Thickness)
Rectangle(WinID,Out RectID,
          Out X,Out Y,Out Width,Out Height)
RectangleColor(WinID,RectID,Out Color)
Circle(WinID,Out CircleID,
       Out X,Out Y,Out Radius)
CircleColor(WinID,CircleID,Out Color)
Bitmap(WinID,Out BitmapID,Out Palette,
       Out X,Out Y,Out Width,Out Height)
Pixel(WinID,BitmapID,I,J)
PixelColor(WinID,BitmapID,I,J,Out Color)
```

These predicates are logically organized as in the tree of Fig. 8. Since it does not make sense asking for a point before asking for a window containing it, or for the color of a point before asking for the point itself, a pre-order visit of the tree yields a correct evaluation order for standard predicates. For this reason, it represents the logical foundation of the query algorithm shown in Fig. 9.

According to this algorithm, the visualizer collects values by evaluating standard predicates

$S_{Win} \leftarrow \emptyset$
while (`Window`(w)= $true$)
begin
$S_{Win} \leftarrow S_{Win} \cup \{w\}$
if (`WindowColor`(w,c)= $true$)
then $WindowCol[w] \leftarrow c$
else $WindowCol[w] \leftarrow WHITE$
...
$S_{Line}(w) \leftarrow \emptyset$
while (`Line`(w,l,x_1,y_1,x_2,y_2)= $true$)
begin
$S_{Line}(w) \leftarrow S_{Line}(w) \cup \{< l, x_1, y_1, x_2, y_2 >\}$
if (`LineColor`(w,l,c)= $true$)
then $LineCol[w, l] \leftarrow c$
else $LineCol[w, l] \leftarrow GREY$
if (`LineThickness`(w,l,t)= $true$)
then $LineThick[w, l] \leftarrow t$
else $LineThick[w, l] \leftarrow 1$
end
...
$S_{Bitmap}(w) \leftarrow \emptyset$
while (`Bitmap`(w,b,p,x,y,l,h)= $true$)
begin
$S_{Bitmap}(w) \leftarrow S_{Bitmap}(w) \cup \{< b, p, x, y, l, h >\}$
$S_{Pixel}(w, b) \leftarrow \emptyset$
$\Omega_{Bitmap}(w, b) \leftarrow \{0 \ldots l-1\} \times \{0 \ldots h-1\}$
$\forall (i, j) \in \Omega_{Bitmap}(w, b)$
if (`Pixel`(w,b,i,j)= $true$) then
begin
$S_{Pixel}(w, b) \leftarrow S_{Pixel}(w, b) \cup \{< i, j >\}$
if (`PixelColor`(w,b,i,j,c)= $true$)
then $PixelCol[w, b, i, j] \leftarrow c$
else $PixelCol[w, b, i, j] \leftarrow BLACK$
end
end
...
end

Figure 9: Fragment of query algorithm

```
unsigned char v[200][300];
/**
   Window(Out _W);
   Bitmap(_W, Out _B1,Out GREY256,
          Out 10,Out 10,Out 200,Out 300);
   Pixel(_W,_B1,I,J);
   PixelColor(_W,_B1,I,J,Out Color)
       Assign Color=v[I][J];
   Bitmap(_W, Out _B2,Out BLACKWHITE,
          Out 220,Out 10,Out 200,Out 300);
   Pixel(_W,_B2,I,J) If v[I][J]<255*0.35;
**/
```

Figure 10: Visualizing a greyscale bitmap and its binary version with threshold 35%

and builds data structures to contain them. In particular, information taken from constructors (`Window`, `Line`, `Bitmap` and `Pixel`) is stored in four different kinds of sets (S_{Win}, S_{Line}, S_{Bitmap} and S_{Pixel} respectively). S_{Win} is the set of labels of all declared windows; S_{Line} contains both the label and the end-points coordinates of each line; similarly, S_{Bitmap} describes for each bitmap its label, its color palette, its upper-left corner position and its width and height in pixels; finally, S_{Pixel} is the set of the coordinates of bitmap's active pixels.

Note that each window $w \in S_{Win}$ has its own set of lines ($S_{Line}(w)$) and of bitmaps ($S_{Bitmap}(w)$); moreover, each bitmap $b \in S_{Bitmap}(w)$ has its own set of active pixels ($S_{Pixel}(w, b)$). Furthermore, sets S_{Win}, $S_{Line}(w)$, $S_{Bitmap}(w)$ are built according to an enumeration strategy, while $S_{Pixel}(w, b) \subseteq \Omega_{Bitmap}(w, b)$ is generated by consulting its characteristic function (`Pixel`). Objects' features are stored in data structures $WindowCol$, $LineCol$, $LineThick$ and $PixelCol$.

To conclude this paper we show two simple examples of ALPHA programs and the images they produce (see Fig. 10 and 11).

The first one concerns the visualization of a 8-bit greyscale image stored in a 200×300 matrix of bytes `v`. In particular, a window containing two bitmaps is declared. The first bitmap is the direct representation of `v`'s content (note all pixels are active), while the second one is a black and white binary image obtained from `v` by applying a threshold filter at 35% (note only black pixels are active).

```
int a[100]={47,68,19,82,25,90,6,72,43,55};
int num=10;
/**
   Window(Out _W);
   Rectangle(_W,Out RecID,Out X,Out Y,Out L,Out H)
       For RecID:InRange(RecID,0,num-1)
       Assign X=10+RecID*20    Y=10
              L=15             H=a[RecID];
   RectangleColor(_W,RecID,Out DARKGREY)
       If a[RecID]<50;
   Line(_W,Out _L,
        Out 5,Out 10,Out 10+num*20,Out 10);
**/
```

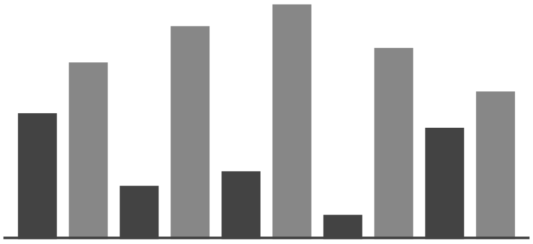


Figure 11: Visualizing a histogram

The second ALPHA program concerns the visualization of a histogram. Bars are declared as rectangles whose heights depend on the content of an external array `a`. Note rectangles' labels belong to the interval $[0, \texttt{num} - 1]$ and are generated through the enumerative predefined predicate `InRange(Out N,From,To)`. Moreover, in order to highlight a semantic property of data, the color of rectangle `i` is dark grey if `a[i]`< 50 .

REFERENCES

[Cresc97a] Crescenzi, P., Demetrescu, C., Finocchi, I., Petreschi, R.: *Leonardo: a software visualization system,* in Proceedings WAE'97, pp. 146–155, 1997

[Ginsb92a] Ginsberg, M.: *Essentials of Artificial Intelligence, Morgan Kaufmann Publishers,* pp. 252–272, 1992

[Lloyd87a] Lloyd, J.W.: *Foundations of logic programming, Springer Verlag,* second extended edition, pp. 35–69, 1987

[Price94a] Price, B.A., Baecker, R.M., Small, I.S.: *A principled taxonomy of software visualization, Journal of visual languages and computing,* 4, pp. 211–266, 1994