Experimenting with a Constructive Page-description Language.

Juan José Cabezas
PEDECIBA Informática
Instituto de Computacion, Universidad de la República,
Casilla de Correo 16120, Montevideo, Uruguay
Email: jcabezas@fing.edu.uy Fax: (5982) 711 04 69

July 6, 2004

Abstract

We present in this paper the results of a experiment with a prototype implementation of a constructive page-description language. The language is based on the concept of *Typed Windows*.

Typed Windows generalizes Computer Graphics standard concept of window, associating graphic representation rules with types of a typed programming language.

The graphic representation rules are inspired by the "Constructive Universalism" of the Uruguayan painter Joaquín Torres García and the type system theoretical framework is Per Martin-Löf's Type Theory.

In this paper, we present an informal introduction to the concepts of typed window, view and oriented port and a description of the graphic representation rules.

Finally, the results of a prototype implementation of *Typed Windows* are discused including the experimentation with a collection of examples and compared performance with other page-description language.

1 Introduction

The art of describing in a computer programming language the appearance of graphical shapes, text and sampled images on a page is a complex task.

We usually describe these two-dimensional graphic objects expressing their geometry trough a Cartesian coordinate system. This fact implies that page-description programs are often composed of an important amount of numerical information and non trivial algorithms, producing a poor readability and reusability of this kind of software. The use of large graphics libraries with functions including a multitude of parameters does not help to modify the situation.

It can be argued that, in most cases, page-description programs are automatically produced by graphics editors or other software pieces, so that programmers do not need to deal with this *low-level coding.* From this point of view, the page-description 'assemblers' could be an acceptable solution.

We think that the existence of high level page-description languages is a necessary condition for the developing of high-quality and low-cost graphics software.

A good example supporting this assertion is the concept of window. This concept –a mapping of a rectangular region of a world-coordinate system onto the page coordinates– is an important methodological tool for obtaining a better programming environment. A remarkable property of programs produced with this methodology is that they can be page-coordinate independent or, in other words, device independent.

The main goal of the *Typed Windows* Project is the developing and experimentation of new methodologies for the construction of page-description programs, based on the concept of *Typed Windows.* This concept is a generalization of the standard window concept.

In this paper, we present an informal introduction to the concepts of typed window, view and oriented port and a description of the graphic representation rules. Finally, the results of a prototype implementation of *Typed Windows* are discused including the experimentation with a collection of examples.

1.1 The standard concept of window.

Computer graphics programs are usually constructed based on the concepts of window and port.

These concepts can be explained as follows:

- *" ... specifying a window in the world-coordinate space surrounding the information we wish displayed." [1]*
- *"In addition to the window, we can define a port (or viewport), a rectangle on the screen where we would like the windows contents displayed." [1]*
- *"We use the window to define what we want to display; we use the port to specify where on the screen to put it". [1]*

The *screen* –or *page* in our case– is a $m \times n$ rectangular array of dots, called *pixels* (picture elements).

Pixels are individually addressed by pairs of a Cartesian coordinate system of natural subranges according to the array sizes. We will suppose the existence of a procedure for setting the color of a pixel. A screen (page) including such procedure is called a *pixel machine.*

A port is a rectangular region of the screen defined by the address of two pixels.

Windows are rectangular regions defined by two points of the world-coordinate space, usually a real (floating point notation) Cartesian coordinate system. When a window is associated with a port, a mapping between the world-coordinate space and a page region can be defined. By this way an abstraction of the page (the graphics hardware) is obtained.

1.2 Typed Windows.

However, the type of the graphic objets is limited to a Cartesian product of reals or integers. This limitation may conduce to an unaceptable increment of the programming complexity when graphic objects of other types are required. In this case, it is necessary to reduce (translate) them to (lists of) pairs of reals or integers, before they can be printed through a certain window.

From a methodological point of view, it would be desirable that for every object a of a certain type A in the *graphic universe* , there exists a window of type A that accepts the object a for printing.

A window system with these properties called *Typed Windows* was first presented in [3]. The paper described a generalization of the traditional concept of window by means of associating graphic representation rules with the types of a small functional programming language.

The resulting programming language is a constructive page-description language, that is, a language capable of describing the appearance of text, graphical shapes and sampled images on a page. As a reference, *Postcript* [4] can be considered the best-known page-description language. *PostCript* is a trademark of Adobe Inc.

Typed Windows introduces an original programming concept for page-description languages:

- To define a window is equivalent to define the type of the graphic objects to be printed on that window, extended with some visibility information.
- A graphic object is an expression (program) of the programming language where *Typed Windows* has been defined, and a program (in such programming language) is a graphic object in *Typed Windows.*

1.3 The theoretical background.

Typed Windows takes as theoretical framework the Type Theory of the Swedish mathematician Per Martin-Löf.

This theory is a formalization of the constructive mathematics with concepts and properties interesting for Computing Science.

In this theory, the idea of specification and program can be associated with the idea of type and element respectively. The theory allows to express both using the same language and formally verify the correctness of a program to its specification.

These ground ideas of Martin-Löfs Type Theory are introduced in [7] as follows:

"The judgement $a \in A$ in type theory can be read in at least the following ways:

- *a is an element in the set A.*
- *a is a proof object for the proposition A.*
- *a is a program satisfying the specification A.*

- *a is a solution to the problem A.*

The reason for this is that the concepts set, proposition, specification and problem can be explained in the same way."

The window concept can be explained in the same way too and therefore we extend the list introduced above with:

- *a is an object that can be printed in the window A.*

As a consequence, this extension makes possible to use this theory in the Computer Graphics area (figure 1).

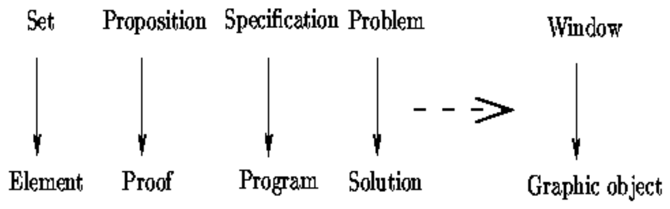

Figure 1: *Window* - *Graphic Object* relation.

1.4 Torres García's art conception.

In *Typed Windows*, the task of printing an object through a window of type A is determined by the graphic rules inferenced from the type A.

These graphic representation rules will be expressed in terms of a certain abstract *graphic machine* that is constructed accordingly to a certain *imaging model*.

Traditional graphic machines are often based on the *painting model*:

"In PostCript the imaging model is based on the notion of painting with opaque paint on a plane. The paint is applied by a pen or brush of a user-specified width..." [2]

As a consequence, the main instructions of such graphic machine are:

- select a pen (color, width).
- move the pen to a certain point of the plane.
- paint a line or a circle or write a letter or ...

This idealized machine –too close to the pixel machine– is inadequate to express the graphic representation rules in a simple and easy to understand way.

Instead of this painting model, *Typed Windows* has a constructive graphic machine based on the color plane concept of the Uruguayan painter Joaquín Torres García (1874-1949).

At the same time, the graphic representation rules were designed inspired by the *Constructive Universalism* of the Uruguayan painter.

Torres García has proposed an art conception that stands out for understand the constructive painting like a symbols structure. [6]

He produced an art movement based on two concepts:

Structure : in order to give a unity to the construction (*"Color planes and lines combined with art, will build a real structure."*[5]).

Abstraction : since he withdrew form imitation of nature, he defined ideograms to represent things and simple ideas in order to use universal representations (*"The painter is not interested in the object, he is interested in the color plane and the geometry of its structure."*[5]).

Torres García created a constructive painting based on a composition (structure) of rectangles (color planes) and ideograms.

A constructive imaging model, following Torres García ideas, can be presented in this way:

- construct the color planes of the page.
- construct a rectangle structure representing the image structure.
- for every ractangle of the structure, stamp an ideogram or construct a structure representing the rectangle image ...

  Continue this structuring process until the desired image is obtained.

This imaging conception –taking color rectangle structures as basic graphic objects– unifies the foreground-background duality; the classical duality of the painting model:

"In the unity of the composition, the idea of thing and background should disappear. ... Then, there are not the thing and the background, all is thing and all is background."[5].

And when this duality disappear, the duality point-plane of the graphic machine disappear too, producing an important change: it is possible to design abstract graphic machines with independence of the pixel machine.

Torres García's concept of color plane has inspired the graphic machine of *Typed Windows* : *the oriented port machine.*

This concept of port is a generalization (unification) of the traditional concepts of port and pixel in Computer Graphics.

In Torres García's paintings we can find **structures**, **ideograms** and **color planes**. In *Typed Windows* these concepts are implemented by *windows (and views)*, *graphic objects (programs)* and *oriented ports*.

2 Ports, Windows and Views.

In this section, the concepts of oriented port, window and view, and the graphic representation rules are informally introduced.

2.1 Oriented Port

A port is an oriented rectangular region of the *page* including color information. There is four different orientations: *Up*,*Right*,*Down* and *Left*.

Notice that it is possible to define four different ports with the same color information and rectangular region on the page.

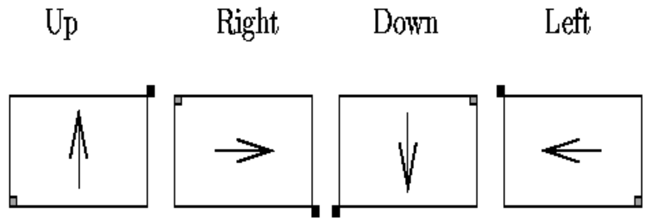

Figure 2:

There are seven port constructors:

1. *page_port*(cl) : construct a port from the page rectangle, *Right* oriented and with a color list *cl* representing the color planes of the port. The actual color plane of the port is represented by the head of the list *cl*. If *cl* is a null list, the color plane of the port is white.

2. *null_port*$()$: construct a null port. Printing a null port produces no changes on the page.

3. *rot_orient*(p) : construct a port rotating the orientation clockwise. The rectangular region and color information are taken from p.

4. *intersection*(p_a, p_b) : construct a port representing the intersection of the rectangular regions of p_a and p_b and with the color and orientation of port p_a. If the intersection is a line, a point or empty, a null port is constructed.

5. *sel_split*(n, i, p) : If $i \leq n$ and $i > 0$ split the port p in n sub-rectangles and construct a port from the *ith* rectangle. Otherwise, construct a null port. The orientation and color information are taken from p. Figure 3 shows the four different results of *sel_split* depending of the four possible orientations of p.

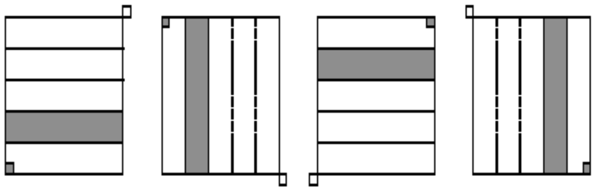

Figure 3: sel_split example for $n = 5$ and $i = 2$.

6. *partition*(r, p) : If $0 < r < 1$ then split the port p in two rectangles proportionally to the real number r and construct a port from the first rectangle. Otherwise, construct a null port if $r \leq 0$ and a p equall port if $r \geq 1$. The orientation and color information are taken from port p. Figure 4 shows the four different results of *partition* depending of the four possible orientations of p ($r = 0.75$).

7. *next_color*(p) : Construct a port with the tail of the color list of p. The rectangular region and orientation are taken from p.

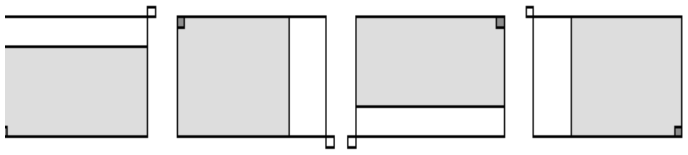

Figure 4: Partition example for $r = 0.75$

2.1.1 Ports equality.

Let us suposse that *port_eq* is defined so that ports p_1 and p_2 are equall ports if they represent the same rectangle on the *page* and have equall orientation and color list.

2.1.2 The oriented port machine.

We call the following set of functions the *oriented port machine (opm)*:

Let $n, i : Nat$; $r : Real$; $lc : List(Color)$; $lp, lp1, lp_2 : List(Port)$; in

$$\begin{array}{rcl}
opm\_page(cl) & = & page\_port(cl) : [] \\
opm\_rot(0, lp) & = & lp \\
opm\_rot(succ(n), lp) & = & opm\_rot(n, map\ rot\_orient\ lp) \\
opm\_intersection([], lp) & = & [] \\
opm\_inters(a.lp_1, lp_2) & = & map\ \lambda x.intersection(a, x)\ lp_2 \\
 & = & ++\, opm\_inters(lp_1, lp_2) \\
opm\_selsplit(n, i, lp) & = & map\ \lambda x.sel\_split(n, i, x)\ lp \\
opm\_partition(r, lp) & = & map\ \lambda x.partition(r, x)\ lp \\
opm\_next\_color(lp) & = & map\ \lambda x.next\_color(x)\ lp
\end{array}$$

2.2 Windows

The types defined for this *Typed Windows* version are enumerations, natural and real numbers, cartesian product, disjoint union , lists and functions.

A window is a type expression. In the case of natural and real numbers and lists the type expression will be extended with visibility information.

In the case of natural numbers the visibility information is a subrange of natural numbers. The following example

$$Nat(2..4)$$

defines a window that accepts natural numbers but only the numbers 2,3 and 4 will be visible through this window.

Real number windows includes visibility information as a real number subrange:

$$Real(-1.0..1.0)$$

defines a window that accepts real numbers but only the numbers greater or equall than -1.0 or less or equall than 1.0 will be visible through this window.

In the case of lists, visibility information is a pair of the enumeration $rot_0, rot_1, rot_2, rot_3$ and a subrange of natural numbers. The second member of the pair determines which elements of a list will be visible, and the first gives the relative orientation that the elements are visualized. :

$$list((rot_1, 2..4), A)$$

is a window that accepts lists of type A but only the second, third and fourth element of the lists will be visible with rotated orientation (rot_1).

Examples of window expressions are:

$$Bool \times Nat(5..11) \rightarrow list((rot_0, 1..100), Bool)$$
$$Nat(1..56) \times list((rot_3, 1..4), T + Real(0.5..9.0))$$

2.3 Views

A view is an ordered pair of type

$$Wdw \times List(Port)$$

where the first member is a window and the second a port list. We call this notation *restricted views (Rviews).* When a more general way for constructing views is required, the Venum, Vnat, Vreal, Vprod, Vunion, Vlist and Vfunction constructors can be used. We call these expressions *general views (Gviews).*

The following is a brief description of the *Gviews* primitives:
Let $m, n : Nat$; $r, s : Real$; $rot : rot0, rot1, rot2, rot3$; $V_A, V_B : View$ where V_A and V_B accept objects of type A and B respectively; $lp : List(Port)$ in

1. $Venum(\{a_0, .., a_n\}, lp)$: construct a view that will accept an object of type $\{a_0, .., a_n\}$ for printing on port list *lp*.
2. $Vnat((m, n), lp)$: construct a view that will accept an object of type Nat for printing on port list *lp* with visibility range (m, n).
3. $Vreal((r, s), lp)$: construct a view that will accept an object of type Real for printing on port list *lp* with visibility range (r, s).
4. $Vprod(V_A, V_B)$: construct a view that will accept an object of type $A \times B$ for printing.
5. $Vunion(V_A, V_B)$: construct a view that will accept an object of type $A + B$ for printing.
6. $Vlist((rot, (m, n)), V_A)$: construct a view that will accept an object of type $List(A)$ for printing with visibility range (m, n) and relative rotated orientation *rot*.
7. $Vfunction(V_A, V_B)$: construct a view that will accept an object of type $A \rightarrow B$ for printing.

While *Rviews* are mainly intend to define (specify) ideograms, *Gviews* expressions support the design of graphic structures.

Since *Rviews* is a *Gviews* subset, it is possible to translate *Rviews* expressions into *Gviews* expressions:

Let $m, n : Nat$; $r, s : Real$; $rot : rot0, rot1, rot2, rot3$; $A, B : Wdw$; $lp : List(Port)$ in

$$\begin{aligned}
Gv(<\{a_0, .., a_n\}, lp>) &= Venum(\{a_0, .., a_n\}, lp) \\
Gv(< Nat(m..n), lp>) &= Vnat((m, n), lp) \\
Gv(< Real(r..s), lp>) &= Vreal((r, s), lp) \\
Gv(< A \times B, lp>) &= Vprod(Gv(< A, map_rot\_orient(lp)>), Gv < B, lp>) \\
Gv(< A + B, lp>) &= Vunion(Gv(< A, lp>), Gv(< B, opm\_next\_color(lp)>)) \\
Gv(< list((rot, m..n), A), lp>) &= Vlist((rot, (m, n)), Gv(< A, lp>)) \\
Gv(< list((rot, m..), A), lp>) &= Vlist((rot, (m, 0)), Gv(< A, lp>)) \\
Gv(< A \rightarrow B, lp>) &= Vfunction(Gv(< A, lp>), Gv(< B, lp>))
\end{aligned}$$

2.4 The Graphic Representation Rules.

In *Typed Windows*, images are represented by port list expressions constructed by the *oriented port machine* .

The function *gr_rep* constructs the image of a graphic object applying to a port list expression the graphic representation rules inferenced from a view expression:

$$gr\_rep : View \times Obj \rightarrow List(Port)$$

where $View$ is a view and Dbj is a graphic object (a program).

Let $m, n, i : Nat$ and $m < n$; $lp : List(Port)$; $a_0, .., a_i, .., an : Id$; $V_A, V_B : View$; $c : A \times B$; $[e_0, .., e_m, .., e_n, ..], a.l : List(A)$; $u : A + B$; $f : A \rightarrow B$; in

$$\begin{aligned}
gr\_rep(Venum(\{a_0, .., a_n\}, lp), a_i) &= opm\_sel\_split(n, i, lp) \\
gr\_rep(Vnat((m, n), lp), i) &= opm\_sel\_split(n - m, i - m, lp) \\
gr\_rep(Vreal((r, s), lp), i) &= opm\_partition(i - r/s - r, lp) \\
gr\_rep(Vprod(V_A, V_B), c) &= opm\_inters(gr\_rep(V_A, fst(c)), \\
&\quad gr\_rep(V_B, snd(c))) \\
gr\_rep(Vudisj(V_A, V_B), u) &= when(u, \lambda a.gr\_rep(V_A, a), \\
&\quad \lambda b.gr\_rep(V_B, b)) \\
gr\_rep(Vlist((rot, (m, n)), Gv(A, lp)) & \\
, [e_0, .., e_m, .., e_n, ..]) &= gr\_rep(Gv(A, opm\_rot(rot, gr\_rep(\\
&\quad (Nat(m, n), lp), m))), e_m) \\
&\quad + + gr\_rep(Gv(A, opm\_rot(rot, gr\_rep(\\
&\quad (Nat(m, n), lp), m + 1))), e_{m+1})
\end{aligned}$$

$$
\begin{array}{rcl}
 & & ++\ \ldots\ldots\ldots \\
 & & ++\, gr\_rep(Gv(A, opm\_rot(rot, gr\_rep(\\
 & & \ (Nat(m,n), lp), n))), e_n) \\
gr\_rep(Vlist((rot,(n,m)), Gv(A,lp)), []) & = & [] \\
gr\_rep(Vlist((rot,(n,m)), Gv(A,lp)), a.l) & = & gr\_rep(Gv(A, opm\_rot(rot, lp)), a) \\
 & & ++\ \ gr\_rep(Vlist((rot,(n,m)), Gv(A,lp))), l) \\
gr\_rep(Vfunc(V_A, V_B), f) & = & null\_port() : []
\end{array}
$$

In order to give a better understanding of the rules, figures 5, 6 and 7 show the graphic representation of simple examples.

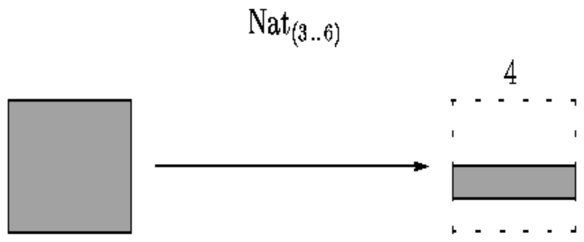

Figure 5:

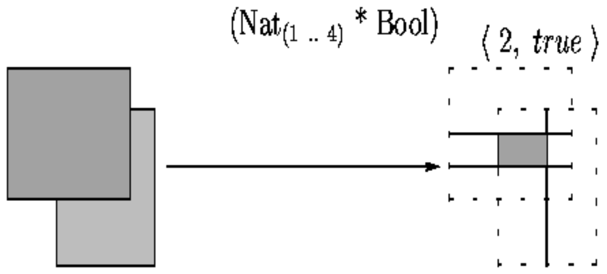

Figure 6:

2.5 Some interesting properties.

- Recursive graphic design.

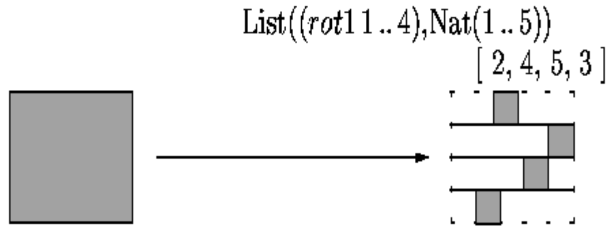

Figure 7:

In *Typed Windows*, the recursive graphic design is an easy to use design methodology.

The following example

$$gr\_rep((W_A, gr\_rep((W_A, lp), a)), a)$$

is the graphic representation of the object a on a port list that is the graphic representation of the object a on the port list lp.

- Cartesian coordinate compatibility.

  There are some wellknown traditional conventions for the graphic representation of Cartesian coordinate systems.

  These conventions are naturally followed by the window

  $$Nat(a..b) \times Nat(c..d) \text{ where } a, b, c, d : Nat$$

- Equality of figures.

  In *Typed Windows*, a figure can be represented by the pair

  $$<< W_A, pl >, a >$$

  where W_A is a window of type A , lp a port list and $a : A$ a graphic object.

  The equality of figures can be defined if the equality of windows, ports and graphic objects is defined. Ports equality has been introduced previously in this paper and windows and graphic objects equality can be defined in the same way that types and elements equality is defined in Martin Löf's Type Theory [7]. However, in the case of windows an extra condition must be added: two windows are equal if the type expressions and *the visibility information* are equal.

3 Experimenting with Typed Windows

Although the theoretical properties of *Typed Windows* look interesting, there are questions that only practice can answer:

- Has *Typed Windows* enough programming power to express in an acceptable way shapes, text and sampled images?
- Is it possible to program in *Typed Windows* complex figures using ideogram libraries instead of function libraries? How necessary are function libraries in this kind of page-description languages ?
- Are these programs easy to read and reuse compared with other page-description languages? And what about the amount of numerical information in such programs?

Pablo Queirolo has made a prototype implementation for testing the practical performance of *Typed Windows* [8].

In order to give an overview of this experiment, we have selected eight examples covering most of the tested performance of *Typed Windows* .

3.1 Example 1: Hello World

The first example is the classic text printing of *Hello World* where *Typed Windows* shows its ability for recursive graphic design, displaying the word "HELLO" on ports where the word "WORLD" has been programmed.

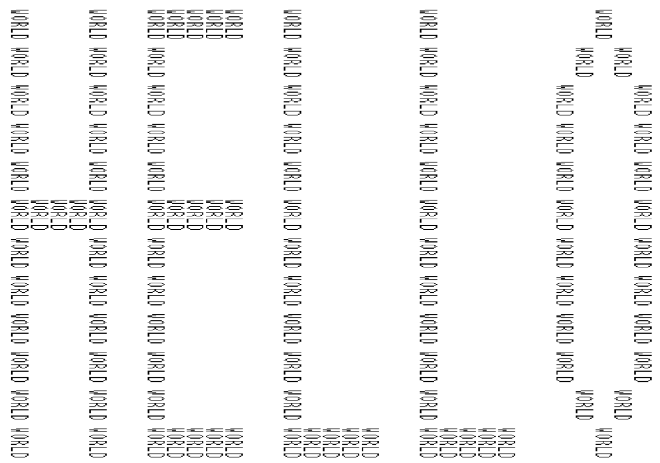

Figure 8: Example 1: Hello World.

Figure 8 shows the resulting page of the following program.

```
-- Including character fonts from an ideagram library.
-- "char_window" is the font window type of the library.
-- char_window= List((0,1..),Integer(0..13) * Integer(0..6));
#include "TyWin_LIBRARY/char.5.12.all";

-- defining a five char text window type
text_window = List((1,1..5),char_window);

-- defining a port list
lport = opm_page([Black]);

-- the graphic objects
HELLO = [ ch_H, ch_E, ch_L, ch_L, ch_O ];
WORLD = [ ch_W, ch_O, ch_R, ch_L, ch_D ];

-- constructing and printing the image
HELLO_ports = gr_rep(Gv(text_window, lport), HELLO);
showpage(gr_rep(Gv(text_window, HELLO_ports), WORLD));
```

Notice that the font (ideogram) library file *char.5.12.all* is included in the first line of the program. An ideogram library is a file including a window type declaration and graphic objects of that type. In the case of fonts, the window type is identified with the name *char_window* and the fonts with *ch_* followed by the selected character.

It is important to remark that both words *HELLO* and *WORLD* use the same library. The rest six lines of the program construct the graphic structure by means of windows, views and ports showing the resulting page.

3.2 Example 2: Function visualization

This example presents a five lines program that visualizes the *sin* function. The program includes the types disjoint union, real numbers and one element enumeration.

Figure 9 shows the resulting page of the following program:

```
-- defining a two color port list
sin_lp = opm_page([SkyBlue, Red]);

-- defining the window type
-- "T" is the one element enumeration {tt}.
sin_window = List((0,1..),(T + List((3,1..100),Real(-1.0..1.0)));

-- constructing the graphic object .  pi=3.1415...
sin_obj = [inl(tt), inr(map sin [0.0,pi/50..pi*2])];

-- constructing and printing the image
showpage(gr_rep(Gv(sin_window,sin_lp),sin_obj);
```

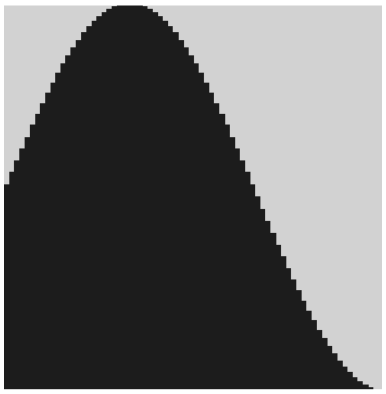

Figure 9: Example 2: Function visualization.

The union list *sin_window* is used to obtain two color planes (SkyBlue and Red) on the same rectangle.

3.3 Example 3: Ancient frieze.

The example presents the construction of a frieze in two steps:

- Construct the frieze structure: an 8×8 rectangular array of three color planes rectangles (mosaics).
- stamp the frieze pattern on these rectangles.

Figure 10 shows the resulting page of the program:

```
-- constructing the frieze port list.
frieze_lpt = opm_page([LightYellow,White,Black]);

-- Constructing an 8 element list of 8 element "tt" list.
ls_8x8= [[tt|x<-[1..8]]|x<-[1..8]];

-- constructing the frieze structure (an 8 x 8 matrix of ports)
frieze_struct= gr_rep(Gv(List((1,1..8),List((1,1..8),T)),
                         light_lpt), ls_8x8);

-- defining the frieze pattern window
wdw16x16= Integer(1..16) * Integer(1..16);
frieze_wdw= List((0,1..),wdw16x16);
```

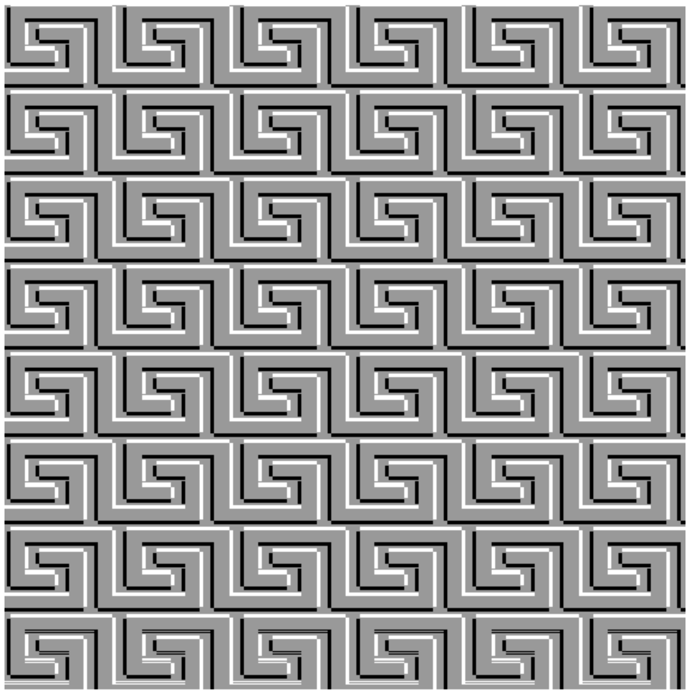

Figure 10: Example 3: Ancient frieze.

```
-- constructing the frieze pattern (graphic object)
hor_lines= (map \x->(1,x) [1..13])++(map \x->(1,x) [14..15])
          ++(map \x->(5,x) [3..9])++(map \x->x.(9,x) [7..10])
          ++(map \x->(13,x) [7..15]);
ver_lines= (map \x->(2,x) [2..13])++(map \x->(6,x) [6..8])
          ++(map \x->(10,x) [10..11])++(map \x->(14,x) [6..15]);
frieze_obj= [inl(tt),inr(inl(hor_lines)),inr(inr(inl(hor_lines))),
      inr(inr(inr(inl(ver_lines)))),inr(inr(inr(inr(ver_lines))))];

-- constructing and printing the frieze
showpage(gr_rep(Vlist((1,0),Vunion(Gv(T,frieze_struct),
     Vunion(Gv(pattern_wdw,opm_next_color(frieze_struct)),
     Vunion(Gv(pattern_wdw,opm_rot(1,opm_next_color(frieze_struct))),
     Vunion(Gv(pattern_wdw,opm_rot(2,
             opm_next_color(opm_next_color(frieze_struct)))),
     Gv(pattern_wdw,opm_rot(3,opm_next_color(
             opm_next_color(frieze_struct))))))))),
     frieze_obj));
```

Notice that the *General View* constructor *Vunion* is used instead of the disjoint union window (the + type) in order to obtain enough power to deal with the three color graphic structure and frieze pattern rotations.

3.4 Example 4: Text handling

The example shows the word *TyWin* printed in three different modes:

- Monospaced text. The direct and simple way of printing text from a font library.
- Multispaced (variable pitch) text. In this case, the use of a font library is combined with a simple text processing function library.
- Rotated multispaced text. In this case, the use of a font library is combined with a simple text processing and a simple 2D graphic function library.

Figure 11 presents the resulting page of the program:

```
-- constructing the port list.
text_lp = opm_rot(3,opm_page([LemonChiffon,MidnightBlue]));

-- defining the page places for the 3 text examples
3x5 = Integer(1..5) * Integer(1..3);
example1_lp = opm_rot(1,gr_rep(Gv(5x3,text_lp),(4,2)));
example2_lp = opm_rot(1,gr_rep(Gv(5x3,text_lp),(3,2)));
example3_lp =        gr_rep(Gv(5x3,text_lp),(2,2));
```

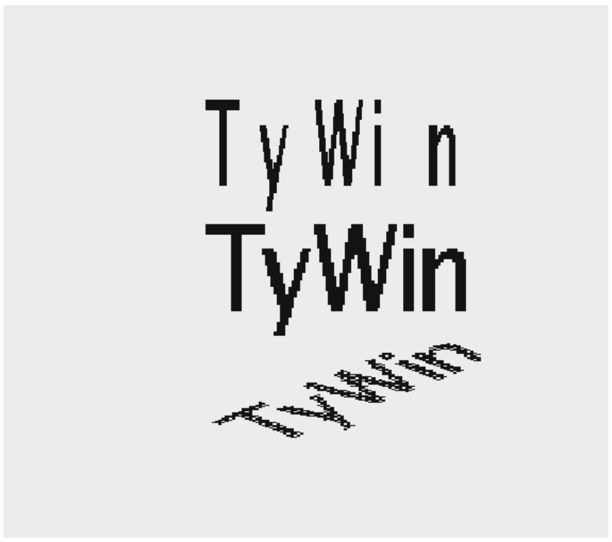

Figure 11: Example 4: Standard, proportional and rotated text.

```
--  Including the character font library
#include "$TW_LIBRARY/fonts/lucida";

-- Example 1 "TyWin" in fixed pitch text format
text_window1 = List((4,1..5),char_window);
tywin1 = [ ch_T , ch_y , ch_W , ch_i , ch_n ];

-- Example 2 "TyWin" in variable pitch text format
text_window2 =  List((0,1..),Integer(0..95) * Integer(0..35));
-- "TyWin" space information
TyWin_sp= [ sp_T , sp_y , sp_W , sp_i , sp_n ];
TyWin2= multispaced(TyWin1,TyWin_sp));

-- Example 3 rotated "TyWin" in variable pitch text
text_window3 =  List((0,1..),Integer(20..110)*Integer(0..80));
TyWin3 = map_rot (pi/4) din_str(TyWin2));

-- constructing and printing the page
text123_obj= [inl(tt),inr(inl(tywin1)),inr(inr(inl(tywin2))),
              inr(inr(inr(TyWin3)))];
showpage(gr_rep(Vlist((1,0),Vunion(Gv(T,text_lp),
         Vunion(Gv(text_window1,opm_next_color(example1_lp)),
         Vunion(Gv(text_window2,opm_next_color(example2_lp)),
         Vunion(Gv(text_window3,opm_next_color(example3_lp))))),
         text123_obj)));
```

The *multispaced* function constructs a pair list representing the text in multisapaced format:

```
mapsum n [] = []
mapsum n a.l = (n + a):(mapsum (n + a) l)
map_chs offs  [] = []
map_chs offs c.chs=(map \x->(((hd offs)+(fst x)),(snd x)) c)++
                         (map_chs (tl offs) chs)
multispaced chls spls = (map_chs  0:(mapsum 0 spls) chls)
```

The *maprot* function applies a rotation matrix to a vector list:

```
srot (a,b) sina cosa = (round ((a*cosa)+(b*sina)),
                        round((-a*sina)+(b*cosa)))
maprot angle pairls = map (\x->srot x (sin angle)
                        (cos angle)) pairls
```

3.5 Torres García's paintings.

The following figures shows three examples of designs where Torres García's paintings were taken as models. In order to test the expressivity power of *Typed*

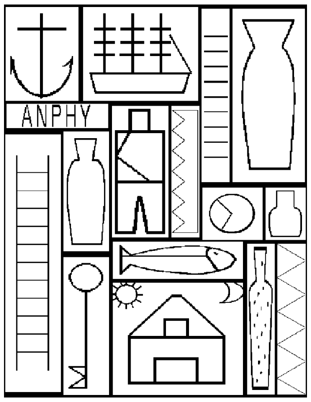

Figure 12: Ex.5:"Constructivo con ancla y barco - Anphy" (1932 - 12 × 9.cm.).

Windows, the examples do not use funtion libraries. Just only ideogram libraries were allowed.

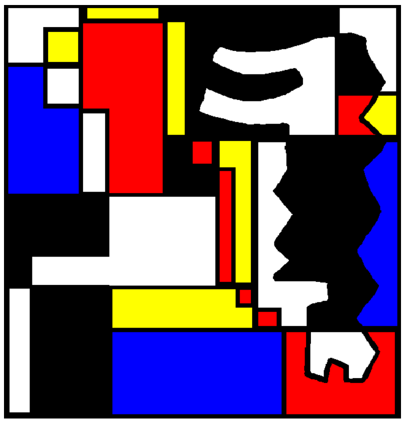

Figure 13: Example 6:"Estructura a cinco tonos con dos formas intercaladas" (1948 - 52 × 50 cm.).

In figure 12 (" Constructive with anchor and ship - Anphy "), the programmer constructs a simple structure and stamps the ideograms (man , ship, key, etc.) from a library.

The same constructive methodology, but with increasing complexity, is applied in figures 13 (" Five tones structure with two intercalated shapes. ") and 14 (" Construction ").

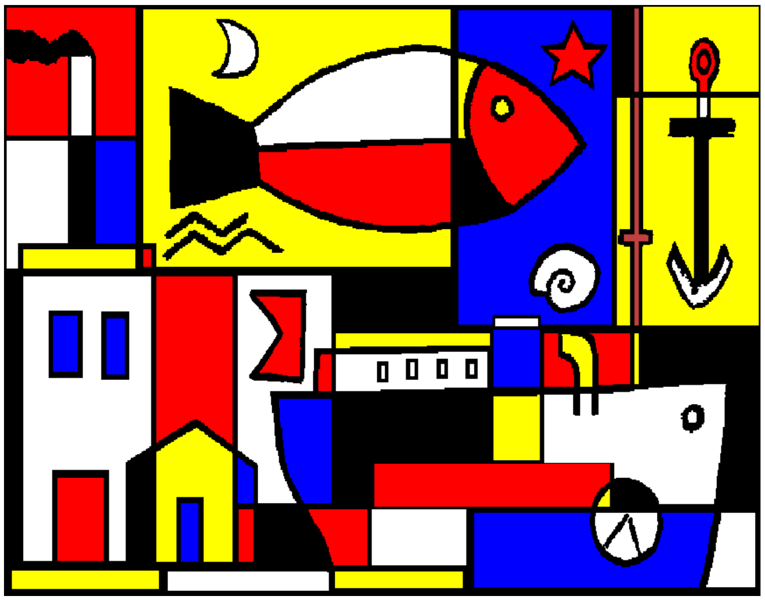

Figure 14: Example 7:“Construcción” (1944 - 54 × 82 cm.).

3.6 Example 8: Computer screen.

Figure 15 shows a possible screen on a user graphic interface. The *lady* has been translated to *Typed Windows* from a raster format file. Function libraries were not used in this example.

3.7 The size of the programs.

Example Nr.	Hello 1	Func 2	Frieze 3	Text 4	Anphy 5	5Tones 6	Constr 7	Screen 8
Typed Windows programs sizes								
Wdw decl.	42	67	108	206	457	130	534	455
Digits (chars)	6	16	74	71	990	800	1049	520
Total (chars)	301	203	1088	1036	9774	9576	10050	10349
Ideogram library sizes								
Ideogram cant.	0	0	0	0	13	1	11	3
Digits (chars)	0	0	0	0	11509	650	9830	2384
Total (chars)	0	0	0	0	55411	15621	86929	66782

Table 1: Typed Windows program and ideogram library sizes.

Table 1 shows the program size of the eight examples. We will point some interesting results extracted from the table:

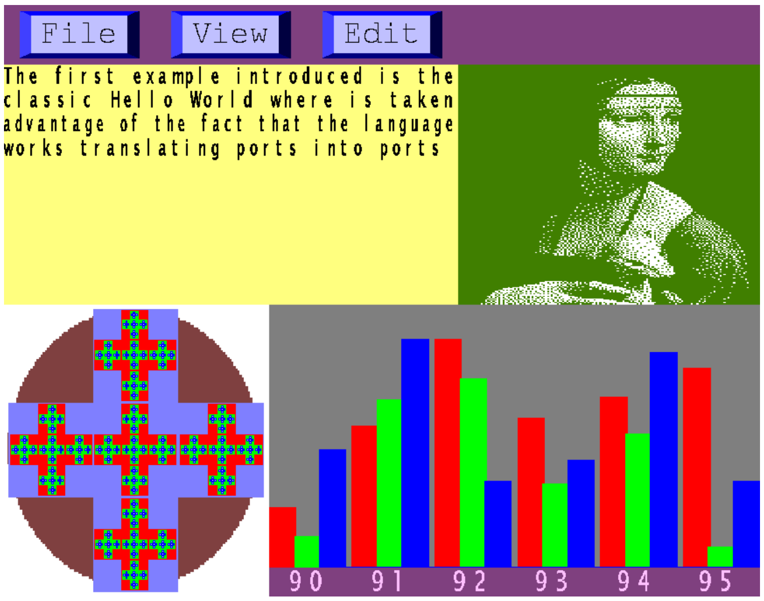
File
View
Edit
The first example introduced is the classic Hello World where is taken advantage of the fact that the language works translating ports into ports
90
91
92
93
94
95

Figure 15: Example 8: Screen Design.

1. The size of window (type) declarations is in the same order than the numerical information of the programs, and together are less than 30% of the total size of the programs (except example 2).
2. The size of the programs is not directly dependent of the complexity of the images. However, the size of ideogram libraries is clearly dependent of the image complexity.

Notice that font libraries are not included in the tables. We suposse that font libraries are standard in page-description languages.

3.8 Numerical information in page-description programs.

Table 2 shows the amount of numerical information of the eight examples in *Typed Windows* and *PostCript* . The size of *PostCript* programs must be considered just a relative reference. We are not expert *PostCript* programmers and it is possible that the eight examples can be programmed in a more efficient way.

		Typed Windows			PostCript		
Example		Prog. Size	Digits		Prog. Size	Digits	
Name	Nr.	Chars	Chars	%	Chars	Chars	%
Hello World	1	301	6	2	3051	581	19
Func.Visual.	2	203	16	8	3101	934	30
Ancient Fri.	3	1088	108	10	2930	1342	45
Text	4	1036	206	20	234	17	7
Anphy	5	65185	12499	19	30571	17954	59
Five tones..	6	25197	1450	6	13306	8413	63
Construction	7	96979	10879	11	28560	16098	56
Comp.Screen	8	77131	2904	4	71519	46013	64

Table 2: Numerical information in page-decription programs.

Even accepting the previous consideration, we detect some results that appear to be independent of the programmer's expertise:

1. The numerical information size in a *Typed Windows* program is less or equal than 20% of the program size in all cases. We cannot detect a relationship between this size and the image complexity.

2. In most cases, the numerical information size in a *PostCript* program is around the half of the program size and grows with the image complexity. There are two examples where the size is less than 20%. Both are simple images based on font libraries.

3. In most cases (five), *PostCript* programs are smaller than *Typed Windows* programs.

4. In most cases (seven), the numerical information of *Typed Windows* programs is smaller than *PostCript* numerical information.

3.9 The results of the experiment.

- About the programming power.

  The examples show that *Typed Windows* is power enough to express complex images (see examples 7 and 8).

- About function and ideogram libraries.

  The language do not require large function libraries to reach an acceptable expressivity power.

  During the experiment, ideogram libraries were constructed, used and reused in an easy way, However, in some cases (see example 4) the programming task is strongly simplified using function libraries.

- About the readibility.

The examples show that *Typed Windows* programs have a low level of numerical information compared with page-description languages supporting the standard concept of window (see PostCript in *Table 2*).

This fact and the window declarations (see *Table 1*) -a specification of the graphic objects- conduce to an important improvement of program readability.

- About the reusability.

  Graphic objects in *Typed Windows* are not a special class (or classes) of objects. This property of the language makes very easy the reuse of graphic objects in a program or trough ideogram libraries.

4 Related work

Beside the related works referenced in the introduction of this paper, we want to point some interesting connections between *Typed Windows* and the art conception of the Dutch painter Piet Mondrian, the Constructive Solid Geometry , the Functional Geometry and the *printf* function.

- Piet Mondrian.

  Avoiding the traditional primitives of the painting model like *lineto* , *polyline* or *curve*, *Typed Windows* has only one primitive geometric object: the normal rectangle, that is, a rectangle with vertical and horizontal sides.

  The preference for this geometric object is typical for most constructivist painters, but the Dutch painter Piet Mondrian (1872-1944) with his austere art of black lines and colored rectangles placed against white backgrounds conceived the normal rectangle as the unique geometric object [15].

  In his perception, vertical and horizontal lines joined at right angles -the angle of perfect equilibrium- were the 'primitives' to express an universal, spiritual, and harmonious art conception.

  Following Mondrian ideas, the oriented port concept in *Typed Windows* was developed taken the colored normal rectangle as the primitive graphic object.

- Constructive Solid Geometry.

  Constructive methodologies are not new in Computer Graphics. Constructive Solid Geometry (CSG) has been widely used in 3D Solid Modeling. The main idea in CSG is to describe a solid object as a composition of primitive objects (cylinders, spheres, cubes) combined with Boolean set operators such as union, intersection and difference. An objet is stored as a tree with operators at the internal nodes and simple primitives at the leaves [9] [10] .

Although CSG is a simple and compact way of representing solids that induces to a constructive thinking when defining a 3D object, it is neither a constructive programming language nor a formal system with well-known properties.

The function *intersection* introduced in the second chapter of this paper is inspired on the CSG concepts.

- Functional geometry.

  There are several approaches to express pictures with structured datatypes and functional programming producing low-cost prototypes that are easy-to-use for non expert graphics programmers [11] [12] [13].

  In most cases the primitive drawing elements of these packages are empirically selected taken the line segment as the basic geometric element.

  These approaches are an interesting innovation from the programming methodologies side but the lackness of a coherent imaging or page-description model reduced them to a friendly interface of standard graphics packages or page-description languages like PostCript.

  However, these experiences convinced us about the convenience of using functional programming when prototyping. The first *Typed Windows* prototype was implemented in a functional programming environment.

- The *printf* function.

  The *gr_rep* function introduced previously in this paper is strong influenced by *printf*, the most common C language function that uses information from the first argument to determine the type and visibility (print format) of the others [14]. The types accepted by the *printf* function are limited to the atomic types: int, float and char.

  In the case of *gr_rep* the first argument accepts a type expression with no limitattions. From this point of view, it would be possible to understand *gr_rep* like a generalization of the *printf* function.

5 Conclusions and future work

In this paper, we have presented in detail the results of a experiment with a prototype implementation of a page-description language which is based on the *Typed Windows* concept.

The results of our experiment indicates that

- *Typed Windows* is power enough to express complex images.
- The language do not require large function libraries to reach an acceptable expressivity power.
- the readability and reusability of the language is better compared with other page-description languages supporting the standard concept of window.

Although the implementation and experimental studies look promising, at least two questions remain to be answered:

- How efficient can be *Typed Windows* when requiring memory and processor resources? This kind of efficience aspects were not considered in the actual implementation.
- Is it possible to extend procedural programming languages like Pascal, C or C++ with *Typed Windows*? The actual implementation is based on a small functional language.

References

[1] William M. Newman and Robert F. Sprouil. *Principles of Interactive Computer Graphics.*
Mc Graw Hill, 1983.

[2] J. Foley, A. van Dam, S. Feiner and J. Hughes. *Computer Graphics Principles and Practice.*
Adisson Wesley, 1990.

[3] Juan José Cabezas. *An approach to typed windows.*
Proceedings of the Compugraphics91 Conference, Portugal, 1991.

[4] Adobe *Postcript Language Reference Manual.*
Adisson Wesley.

[5] Joaquín Torres García. *Lo aparente y lo concreto en el arte (1947).*
Capítulo Oriental 41 - Centro Editor de América Latina, 1969.

[6] Gabriel Paluffo Linari. *Historia de la Pintura Uruguaya. Torres García : de Barcelona a París.*
Ediciones de la Banda Oriental, 1992.

[7] Bengt Nordström, Kent Petersson and Jan M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction..*
Oxford University Press, 1990.

[8] Pablo Queirolo *Typed Windos: An Implementation of a Programming Language for Graphic Design.* .
Reporte Técnico INCO 97-02, Uruguay, 1997.

[9] Goldstein, R.A. and Nagel R. *3D Visual Simulation.*
Simulation, January 1971, pags. 25-31.

[10] Geoff Wyvill and Tosiyasu Kunii, *Space Division for Ray Tracing in CSG.*
IEEE Computer Graphics & Applications, April 1986.

[11] Henderson, Peter *Functional Geometry* .
1982 ACM Symposium on Lisp and Functional Programming.

[12] Chaillaux, E. and Cousineau, G. *Programming Images in ML* .
Proccedings of the ACM SIGPLAN workshop on ML and its aplications, 1992.

[13] Finne, S and Peyton Jones, S *Picture: A Simple Structured Greaphics Model* .
http://www.dcs.gla.ac.uk/fp/software/haggis

[14] Kernighan, Ritchie *The C Programming Language*
pag 71, Prentice Hall.

[15] Seuphor, Michel *Piet Mondrian, life and work.*
N.Y. Harry N. Abrams.