An Approach to Typed Windows

Juan José Cabezas[*]
Instituto de Computación Facultad de Ingeniería
Montevideo URUGUAY
email: jcabezas@incouy.edu.uy

June, 1991.

abstract

The concept of window in Computer Graphics has been increasing its importance during the lasts years.

From a methodological point of view, it would be desirable that for every object of a certain type in the "graphics" universe, there exists a window of a certain type that accepts that object for displaying. A window system with those properties can inprove the quality of the programming environment in Computer Graphics, extending to the graphics area, the power of type systems in recent programming languages.

In this paper, three possible strategies for obtaining such window environment are evaluated and a simple typed window system is presented.

1 The Window Model

The ideas supporting the concept of window are simple and easy to understand:

- " ... specifying a window in the world-coordinate space surrounding the information we wish displayed." [Spr83]
- "In addition to the window, we can define a viewport, a rectangle on the screen where we would like the windows contents displayed." [Spr83]
- "We use the window to define what we want to display ; we use the viewport to specify where on the screen to put it". [Spr83]

The window (the world-coordinate space and the viewport associated with it) may be seen as an abstraction of the screen and the pixel (the graphics hardware). This window model [Spr83] is a powerful tool that allows a better programming methodology in Computer Graphics.

[*]The author is partially supported by a PEDECIBA-Informática grant.

However, the type of the objets of a certain window is usually limited to a cartesian product of subranges of reals or integers. As a consequence, when programming with graphic objects of different types, it is necessary to reduce (translate) them to a list of objects of the window type, before they could be displayed.

From a methodological point of view, it would be desirable that for every object of a certain type in the "graphics" universe, there exists a window of a certain type that accepts that object for displaying. A window system with those properties can increase the quality of the programming environment in Computer Graphics, extending to the graphics area, the power of type systems in recent programming languages.

This approach implies a more abstract concept of window presenting some theoretical questions:

- which types have a graphical meaning and therefore could be accepted by a window?
- what is the concept of clipping for such windows?
- what does it means type checking between objects and windows?

2 Three possible strategies

In this section, three possible strategies for obtaining a typed window environment are evaluated.

2.1 Function oriented windows

A possible strategy for obtaining more power and simplicity is to associate a set of windows with different functions in the graphic domain. These windows could include a library for the specified task. Some examples of these types of windows could be:

Type of the window	use
2D_W = 2DWindow(h_1,..,h_n)	display(2D_W,Translate(ob1:2D),p_1,..,p_n)
3D_W = 3DWindow(h_1,..,h_n)	display(3D_W,Rotate(ob2:3D),p_1,...,p_n)
Me_W = MenuWdw(h_1,..,h_n)	select(Me_W,ob3:Menu,p_1,..,p_n)
Fn_W = FuncWindow(h_1..,h_n)	funPlot(Fn_W,ob4:Function,p_1,..,p_n)

where p_1,..,p_n and h_1,...,h_n could be understand as other parameters. In this case, we have four window types (2DWindow, 3DWindow, MenuWdw and FuncWindow) including one graphic library each, supporting a high level graphics functionality.

The basic idea of this strategy is to make an abstraction of the window by means of specialization, obtaining a finite set of powerful window types.

This strategy may be applied when the window types are predefined since the programmer is not able to construct new window types from them. It can be considered as a refinement of a graphic library but not a window type system. In some cases this limitation is not desirable.

2.2 The Infinite Window

This design strategy , by means of abstraction of the concept of window, tries to obtain with no compromise, the most simple programming environment. As we will see later, the basic power of the window concept is lost.

Only one function, the function *view* is available for the programmer. *view* has two parameters, the first is the object to be displayed in the screen and the second contains the information needed to define the viewport on the screen.

Let a be an object of type A and p defining a viewport.

$$view(a,p) \qquad\qquad [a \in A \quad p \in Vport\_inf]$$

we say that the object a will be displayed in the viewport defined by p and will be seen through a window of type A. The type A is obtained by a type inference algorithm from a.

Example: if $a =< tuesday, sun >$ so that $a \in A$ and A is the cartecian product of the enumerated sets $Week \times Weather$

$Week = \{sunday, monday, tuesday, wednesday, thursday, friday, saturday\}$
and
$Weather = \{sun, cloud, rain\}$

then $view(< tuesday, sun >, p)$ will display the object $< tuesday, sun >$ on the viewport p through a window of type $(Week \times Weather)$. The reader can easily imagine a possible graphic representation of this window type.

But, which representation could be accepted for the following example?

If $a = 4$ and $A \equiv N$ then $view(4,p)$ will display the object 4 on the viewport p through a window of type natural numbers. Natural numbers is an infinite set. What is the graphical meaning of such a window?

In this case, the concept of window is lost. Informally, a window could be represented by a finite subset of a certain set (the window universe), but not the entire universe when it is infinite.

In conclusion, this strategy has two important limitations:

- the concept of window is lost if infinite sets are considered.
- Although this strategy offers to the programmer the possibility of construct different types of windows, the programmer is not able to define different windows for an object of a certain type because the window is automatically defined by the type of the object to be displayed.

2.3 Typed Windows

The third design strategy , by means of abstraction of the concept of window, tries to obtain a friendly programming environment and, at the same time, includes the possibility of defining different types of windows in a certain universe.

For every object a of type A it is possible to define a window w of type W_A, where W_A is a restriction of A.

Example: if the type of a is $A \equiv (N \times N)$ then the type W_A of the window w could be $W_A \equiv (\{10..20\} \times \{5..40\})$.

The objects $< 1, 100 >$ and $< 15, 10 >$ are of type A, but only the second is of type W_A. In this case, the window w accepts both objects, but only the second is displayed. So, a window can be constructed if we know the type W_A representing the restrictions over the type of the objects to be displayed. The basic types considered in this paper are: enumerated sets, natural numbers subranges of natural numbers, cartesian product, disjoint union, lists and functions. Subranges of natural numbers are specially important when defining window types (type W_A). Notice that this set of types is the set of predefined types in most programming languages.

Some examples of different types of windows and objects:

Window type	Object type (deduced from W_A)
$(\{1..10\} \times \{4..90\}) + list(Bool)$	$(N \times N) + list(Bool)$
$list(\{1..600\} \rightarrow \{0..400\})$	$list(N \rightarrow N)$
$(\{a1, a2\} \times \{b1, b2\}) \times list(\{0..4\})$	$(\{a1, a2\} \times \{b1, b2\}) \times list(N)$

3 The function *view*.

In this section, a simple typed window system based on the third strategy, is proposed.

Accordingly to this type system, the function *view* is programmed in Martin-Löf's Type Theory.

Martin-Löf's Type Theory is a formalism for program construction developed by the swedish mathematicien Per Martin-Löf. It is well suited as a theory for program construction since it is possible to express both specifications and programs within the same formalism.([NPS89])

3.1 Function oriented windows

3.2 Views and Types

In Computer Graphics the concept of view can be introduced in the following way:

1. define a window w of a certain universe (usually a cartesian product of real or integer subranges).

2. define a viewport vp.

3. define a function *chgcoor* so that, $chgcoor(w, vp, < x, y >)$ where x and y are the coordinates of a point in [1], gives the coordinates of a pixel $< a, b >$ in the viewport defined in [2].

We call *view* a procedure consisting of the above three steps:

$$view < W_A, p > (a) \qquad [a \in A; \quad W_A \text{ is a restriction of } A]$$

so that when $view < W_A, p >$ (*view* applied to a window type W_A and a viewport defined by p) applies to an object a of type A, yields the list of pixels to be modified on the screen.

$View$, the type of *view*, is defined as follows::

$View \equiv (\forall\ wp \in (Wdw\_type \times Vp\_coor))\ Obj\_type(fst(wp)) \rightarrow list(Vport(snd(wp)))$
where

$Wdw\_type \equiv U$ (U is the set of small types. [NPS89])
$Vp\_coor \equiv list(Screen \times Screen)$
$Screen \equiv \{0..M_x\} \times \{0..M_y\}$
$Obj\_type(t) \equiv obj(set(t))$ (obj is defined in [3.2])
$Vport(m) \equiv \{ffst(hd(m))..fsnd(hd(m))\} \times \{sfst(m))..ssnd(m)\}$ $[m \in Vp\_coor]$

so that if $< A, p > \in (Wdw\_types \times Vp\_coor)$ and there exists a function $view \in View$ then $view < A, p >$ yields a view vw (a function) of the type

$$vw \equiv view(< A, p >) \qquad vw \in Obj\_type(A) \rightarrow list(Vport(p))$$

and if $a \in Obj\_type(A)$ (where $A \in Wdw\_type$) then $vw(a)$ yields a list lp of pairs $< m, n >$ of type $list(vport(p))$. lp is the list of pixels to be modified or, in other words, we say that the object a will be displayed through a window of type A on the viewport defined by p.

3.3 A graphics semantic for *view*

A possible definition of view is the following:

$view < A, p > \equiv \lambda m.coorchg(code(A, m), gen(A, m), subwin < A, p > (a))$
where

$coorchg \in Base\_Wcoor \times list(Base\_Wcoor) \times Vp\_coor \rightarrow Vp\_coor$
$code \in (\forall t \in Wdw\_type)Obj_type(t) \rightarrow Base\_Wcoor$
$Base\_Wcoor \equiv (N \times N) \times (N \times N)$
$gen \in (\forall t \in Wdw\_type)Obj_type(t) \rightarrow list(Base\_Wcoor)$
$subwin \in View$

The functions *code*, *gen*, *subwin* and *obj* are defined for the predefined types:

1. **Subranges**

   Subranges are denoted {a..b} where $a \leq b$ and $a, b \in N$

   $code(\{a..b\}) = \lambda n. << a, succ(b) >, < a, succ(b) >>$
   $gen(\{a..b\}) = \lambda n. << n, n >, < succ(n), succ(n) >> .nil$
   $subwin < \{a..b\}, p >= \lambda n.p$
   $obj(\{a..b\}) = N$

2. **Enumerations**

   Enumerations are denoted $\{i_1, .., i_n\}$ where i_k is an identifier.

   $code(\{a_1, .., a_n\}) = \lambda e.code(\{0..n\})$
   $gen\{a_1, .., a_n\} = \lambda e. < case(e, 0, 1, .., n), case(e, 0, 1, .., n) >,$
   $< succ(case(e, 1, .., n)), succ(case(e, 1, ..., n)) > .nil$
   $subwin < \{a_1, .., a_n\}, p >= \lambda e.p$
   $obj(\{a_1, .., a_n\}) = \{a_1, .., a_n\}$

3. **Cartesian Product**

$$\begin{array}{l} code(A \times B) = \lambda m. < fst(code(A)), snd(code(B)) > \\ gen(\ A \times B) = \lambda m.pairs(map(fst, genA(fst(m))), map(snd, genB(snd(m)))) \\ subwin < A \times B, p >= \lambda m.case(cl(A, B), \\ \qquad view < snd(code(A)) \times fst(code(B)), p > \\ \qquad < snd(genA(fst(m))), fst(genB(snd(m))) >, \\ \qquad view < snd(code(A)) \times fst(code(T)), p >< snd(genA(fst(m))), tt >, \\ \qquad view < snd(code(T)) \times fst(code(B)), p >< tt, fst(genB(snd(m))) >, \\ \qquad p) \\ obj(A \times B) = obj(A) \times obj(B) \end{array}$$

4. **Disjoint Union**

$$\begin{array}{l} code(A + B) = \lambda u.when(u, \lambda a.code(A), \lambda b.code(B)) \\ gen(A + B) = \lambda u.when(u, gen(A), gen(B)) \\ subwin < A + B, p >= \lambda u.when(u, subwin < A, p >, subwin < B, p >) \\ obj(A + B) = obj(A) + obj(B) \end{array}$$

5. **Lists**

$$\begin{array}{l} code(list(A)) = \lambda l.code(A) \\ gen(list(A)) = \lambda e.listrec(e, nil, \lambda xyz.concat(genA(x), z)) \\ subwin < list(A), p >= \lambda e.listrec(e, nil, \lambda xyz.concat(subwin < A, p > (x), z)) \\ obj(list(A)) = list(obj(A)) \end{array}$$

6. **A → B**

$$\begin{array}{l} code(A \to B) = \lambda s.code(list(A \times B)) \\ gen(A \to B) = \lambda f.gen(list(A \times B), pairmap(f, explode(A))) \\ subwin < A \to B, p >= \lambda f.subwin < list(A \times B), p > pairmap(f, explode(A))) \\ obj(A \to B) = obj(A) \to obj(B) \end{array}$$

7. **Natural numbers**

   Since N has no meaning when defining window types, it will be considered as the subrange $\{1..0\}$.

$$\begin{array}{ll} code(N) = code(\{0..1\} & gen(N) = gen\{0..1\} \\ subwin < N, p >= \lambda n.p & obj(N) = N \end{array}$$

4 Examples

4.1 Bar graph

Window type:

$$Bars \equiv list(Bar \times \{1..16\})$$

where
$Bar \equiv list(T \times \{1..8\})$
$T \equiv \{tt\}$

Object to be displayed:
$bar\_graph \equiv pairs(bars, [2;4;6;8;10;12;14])$
where
$bars \equiv map(\lambda n.bar(n), [4,2,3,6,5,2,4])$
$bar(n) \equiv map(\lambda x.pair(tt,x), [1..n])$

We can now display the object *bar_graph* on the viewport defined by p through a window of type *Bars*:

$$view < Bars, p > (bar\_graph)$$

```
8------------------
7|                |
6|       I        |
5|       I I      |
4| I     I I   I  |
3| I   I I I   I  |
2| I I I I I I I  |
1| I I I I I I I  |
 ------------------
  1              16
```

4.2 Function plotting

Window type:
$2P\_fun \equiv \{1..4\} \times \{1..3\} \rightarrow \{1..16\}$
Object to be displayed:
$ft1 \equiv \lambda xy.(x * y + 1)$

The function $ft1$ will be displayed on the viewport defined by p through a window of type $2P\_fun$:

$$view < 2P\_fun, p > (ft1)$$

```
     ----------------
   4|            *   |
x  3|         *      |
   2|      *    (y=3)|
   1|...*............|
   4|        *       |
x  3|      *         |
   2|    *      (y=2)|
   1|..*.............|
   4|    *           |
x  3|   *           |
   2|  *       (y=1)|
   1|_*_____________|
     1            16
           ft1(x,y)
```

4.3 The text window

Suppose we have defined fonts for graphic respresentation of characters and the function chr_to_font:

$Font = \{0..10\}x\{0..10\}$
$charfonts = [font\_A; font\_B; font\_C; ....; font\_z]$

$chr\_to\_font \in Char \times list(Font) \rightarrow Font$
f. e. $chr\_to\_font('B', charfonts) = font\_B$

We define now the funcion str_fts :
$str\_fts \in list(Char) \times list(Font) \rightarrow list(Font)$
$str\_fts(st, fonts) = map(\lambda c.chr\_to\_font(c, fonts), st)$

Window type:

$Wtext(n, m) \equiv \{1..n\} \times String(m)$
$String(m) \equiv list(Font \times \{0..m\})$

Object to be displayed:

$hello \equiv < 4, str("Helloworld", 11) >$
where $str(st, n) = pairs(str\_fts(st, charfonts), [0..n])$

$$view < Wtext(4, 20), p > (hello)$$

```
  ----------------------
 4|Hello world         |
  |                    |
  |                    |
 1|                    |
  ----------------------
  0                   20
```

4.4 A composed window

Suppose we want to compose the following window

```
---------------------------------
|text1          .C              |
|               .  text2        |
|               .               |
| example[4.2]  ................|
|               .                 |
|               .  example[4.1]   |
|A              .B                |
-----------------------------------
```

so that *example*[4.2] and *text*1 will be displayed on the rectangle A, *example*[4.1] on the rectangle B and the *text*2 on C.

text1 = "Function ft1"

text2="Bar Graph"

Window type:

$Wcomp \equiv list(\ ((Wtext(12,20) + Bargraph) \times Bool)+$
$\quad ((Wtext(3,20) + 2P\_fun) \times (Bool \times Bool))$

where $Bool = \{false, true\}$

Object to be displayed:

$comp\_obj \equiv [obj1; obj2; obj3; obj4]$

where

$obj1 \equiv inl(inl(<< 6, str("BarGraph", 9) >, false >))$

$obj2 \equiv inl(inr(< bargraph, false >))$

$obj3 \equiv inr(inl(<< 1, str("Functionft1", 12) >, < true, true >>)$

$obj4 \equiv inr(inr(< ft1, < false, true >>)$

Finally, the composed object is displayed:

$$view < Wcomp, p > (comp\_obj)$$

```
 -------------------------------
|Function ft1*  .               |
|         *     .               |
|      *        .               |
|...*............               |
|        *      .Bar graph      |
|      *        ................|
|    *          .       I       |
|..*.............       I I     |
|    *          . I     I I   I |
|   *           . I   I I I   I |
|  *            . I I I I I I I |
|_*_______________I_I_I_I_I_I_I_|
```

References

[NPS89] Bengt Nordstrom, Kent Petersson, and Jan M. Smith. *Programming in Martin Lof's Type Theory.* Programming Methodology Group - Department of Computer Sciences - University of Gothembourg / Chalmers - S-412 96 Gothembourg, Sweden, 1989.

[Spr83] William M. Newman Robert F. Sprouil. *Principles of Interactive Computer Graphics*. Mc Graw Hill, 1983.