CODEN: LUTFD2/(TFRT-5421)/1–040/(1990)

PostScript Images Generated from InterViews

Klas Lönnell

Department of Automatic Control
Lund Institute of Technology
January 1990

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
January 1990

Document Number
CODEN: LUTFD2/(TFRT-5421)/1–040/(1990)

Author(s)
Klas Lönnell

Supervisor
Dag M. Brück and Sven Erik Mattsson

Sponsoring organisation

Title and subtitle
PostScript Images Generated from InterViews

Abstract

This report describes the method of extending InterViews with a PostScript generating facility. As the current implementation of InterViews supports screen-graphics only, a mechanism that generates PostScript code would be desirable since PostScript, among other things, is often used as input for laserwriters.

InterViews is a library of C++ classes which provides for good object-oriented programming. This was taken advantage of when extending the drawing primitive of InterViews, class Painter. This class is now divided into two: class XPainter (former Painter) and class PSPainter which generates PostScript code.

The implementation was made under the condition that an already existing application should not have to altered extensively. Hence, one additional statement is needed to generate PostScript.

Unfortunately, due to the protection mechanism of C++ (information hiding) which the current implementation of InterViews uses, the brush and font representations may differ between the screen and paper (PostSript) images.

As this work can be considered as step one, further improvements are anticipated.

Key words
C++, Computer Graphics, InterViews, PostScript

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
40

Recipient's notes

Security classification

Table of Contents

1. Introduction 3
1.1 InterViews 3
1.2 PostScript 7

2. How to generate PostScript 9
2.1 Function DrawPS 9
2.2 Limitations 9

3. Implementation 11
3.1 Class Painter 11
3.2 Class XPainter 13
3.3 Class PSPainter 14
3.4 Changes in InterViews' file system 21
3.5 Different solutions 22

4. Conclusions 22

Acknowledgements 23

References 23

Program listings 24
painter.h 24
Xpainter.h 26
PSpainter.h 28
newpainter.c 29
PSpainter.c 34
PStools.c 37

1. Introduction

InterViews is a graphics package, running on top of the X Window System. The current implementation of InterViews supports screen-graphics only. The only way to 'paperize' an InterViews' image is to do an ordinary screen-dump which results in a picture with low resolution. Therefore, it is desirable to augment the InterViews library with drawing primitives that generate PostScript code. A picture extracted from PostScript code will look much better than the usual X screen dump. A typical screendump from a plotting program is shown in Figure 1.1. The improved image generated from PostScript is shown in Figure 1.2. They represent the solution of an ordinary differential equation derived by Lorenz in 1963 as an approximate model for atmospheric air currents [Elmqvist et al., 1986]. The purpose of this project was to extend the InterViews system with some PostScript generating routines which will work in parallel with those existing routines that draw on the screen. This extension was made under two conditions:

1. A minimum of changes to the current implementation of InterViews.
2. An already existing application program should not have to be altered in an extensive way.

With these two restrictions in mind, the task sounded difficult as the InterViews system was not prepared for such an extension. However, the language with which InterViews is constructed, C++, is object-oriented, and one of the big advantages with an object-oriented language is that it can be used to extend an already existing implementation easily [Lippman, 1989].

1.1 InterViews

InterViews is a library of C++ classes that provides flexible utilities for constructing user interfaces [Linton et al., 1989]. A 'user interface' might be a window containing a couple of graphics-objects, such as lines, circles, textstrings, etc. InterViews is designed and implemented by a group led by professor Mark Linton at Stanford University. The system is rather big with about 15 000 lines of C++ source code. Inter(active)Views lies on top of the X Window System. The current implementation is simpler (has fewer functions) than X but is much more easy to use. The window management of X is object-oriented too, but there is no support for this in the programming language (ordinary C), something which InterViews has in C++.

The InterViews system is constructed hierarchicaly with the base class Interactor at its root [Linton and Calder, 1987]. The user designs the apperance of his interactor by defining what drawing-primitives should be invoked when creating an Interactor-object. An example on how this is done is shown below. Later, this interactor is inserted into an object of another class, World, to make it visible on the screen. Every interactor has an associated Shape that it uses to define the desired display area characteristics, such as size, shrinkability and stretchability. The stretching and shrinking qualities *between* interactors are determined by class Glue. The actual display area is assigned to the Canvas associated with the interactor.

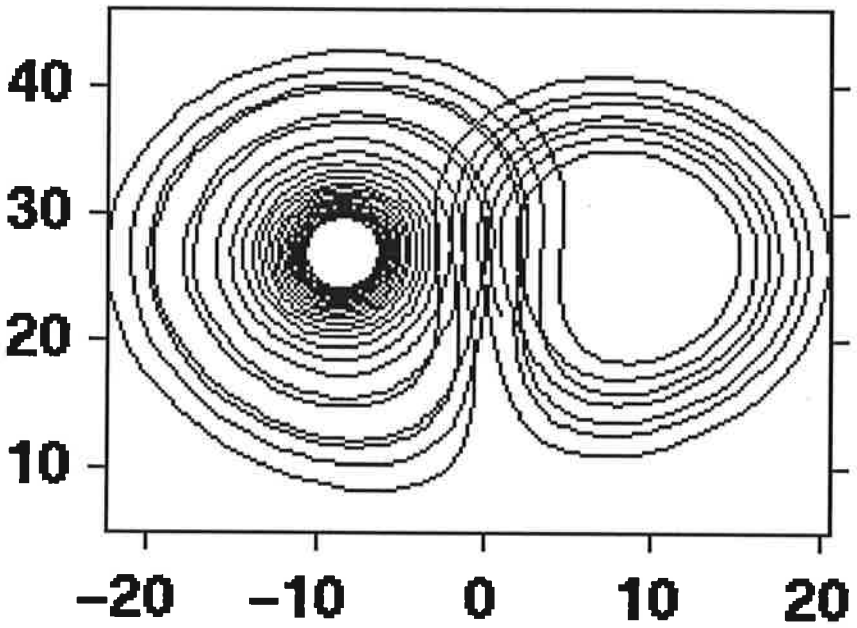

Figure 1.1 Screen dump

Class Scene defines the basic operations for managing a group of interactors. If we look at Figure 1.3 we see the scene subclass Box. This class has two subclasses, HBox and VBox. Both with the same properties, namely to insert interactors. The HBox and VBox can insert up to seven interactors, horizontally and vertically respectively. Each interactor can, in turn, contain interior interactors, just as an HBox (or VBox) does. It is when using this function one discovers the great flexibility the InterViews system offers when

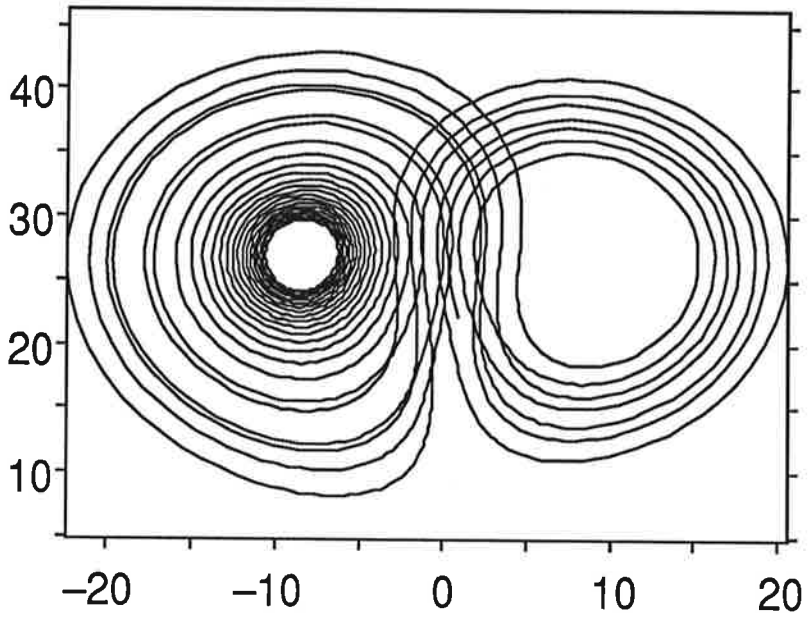

Figure 1.2 A PostScript product.

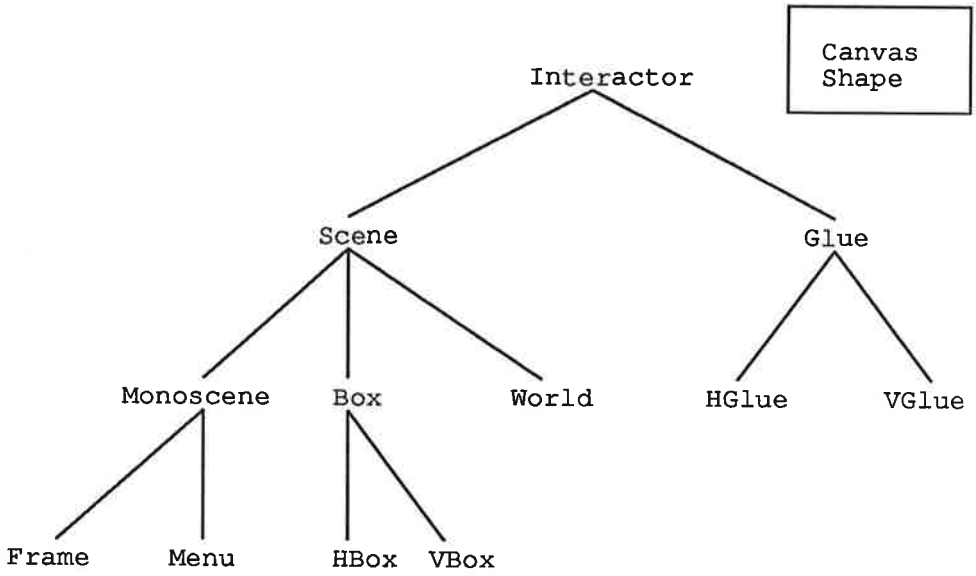

Figure 1.3 A subset of InterViews' class hierarchy: the base class Interactor and its descendants.

it arranges for up to seven interactors to fit within a given area. Below follows an illustrating example that will display the capital letter H in more than one way. The graphical output is shown in Figure 1.4.

```
class H: public Interactor {
// By declaring H as 'public Interactor'
// class H is said to be derived
// (publicly) from class Interactor.

public:
        H(); // Constructor
  void  Redraw(Coord,Coord,Coord,Coord);
};

void  H::Redraw(Coord,Coord,Coord,Coord)
{
  output->ClearRect(canvas,0,0,xmax,ymax);
  // Clear the background
  output->Line(canvas,xmax/8,ymax/8,xmax/8,7*ymax/8);
  //   left leg
  output->Line(canvas,xmax/8,4*ymax/8,7*xmax/8,4*ymax/8);
  //   middle bar
  output->Line(canvas,7*xmax/8,ymax/8,7*xmax/8,7*ymax/8);
  //   right leg
}

H::H() // Constructor
{
  shape->Square(round(inch));
  // Each interactor will now have the shape of
```

```
  // a one-inch square
}

void main()
{
  World* w= new World("test");
  HBox* h1= new HBox( new H,new HGlue(round(inch)),new H);
  HBox* h2= new HBox( new H,new H,                 new H);
  HBox* h3= new HBox( new H,new HGlue(round(inch)),new H);
  Interactor* I=new VBox(h1,h2,h3);
  w->Insert(I);
  w->Run();
}
```

Listing 1.1 An H in more than one way.

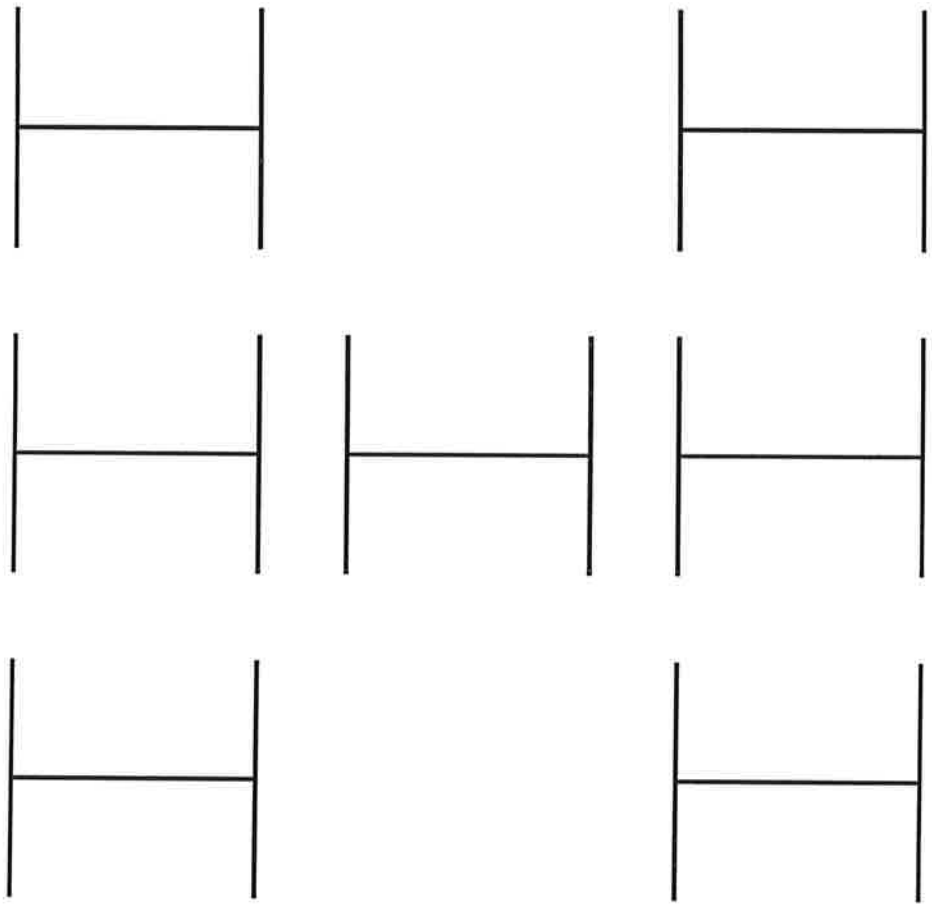

Figure 1.4 Output of the program in Listing 1.1

```
/inch 72 def % 72 pixels to an inch
/dinch 144 def % double-inch

%   Subroutines --------------------

/rectangle { 0 dinch 3 mul rlineto
 dinch 0 rlineto
 0 -3 dinch mul rlineto
 closepath } def

/circle {          % A subroutine that requires
 inch 0 360 arc    % the x- and y coordinates
 } def             % on top of stack.

% Main program  -------------------------
 inch inch moveto  % Sets the current point.
 rectangle         % Call rectangle.
 stroke            % Make it visible.

 dinch             % Paint red and green.
 3 dinch mul
 0.25 setgray      % New pattern for representing
 circle fill       % red. Note, fill instead of
                   % stroke.
 dinch dinch
 0.75 setgray      % Green
 circle  fill      % Fills the circle with a
                   % pattern correspondant to green.

 0 setgray         % Outline the three lights.
 1 1 3 {
 dinch             % Puts the coordinates for
 exch dinch mul    % the circles on the stack.
 circle stroke     % Call circle and outline it.
      } for
 showpage          % Print the page
```

Listing 1.2 A PostScript program.

1.2 PostScript

PostScript is a device independent, stack-oriented, language developed by Adobe Systems [Adobe, 1985]. It is often used as input for laserwriters. Some workstations, like Sun NeWS for instance, support it as well. It is relatively easy to use although at first sight it might look too much a low-level programming language. Listing 1.2 contains PS-code for printing a page containing a traffic-light (shown in Figure 1.5).

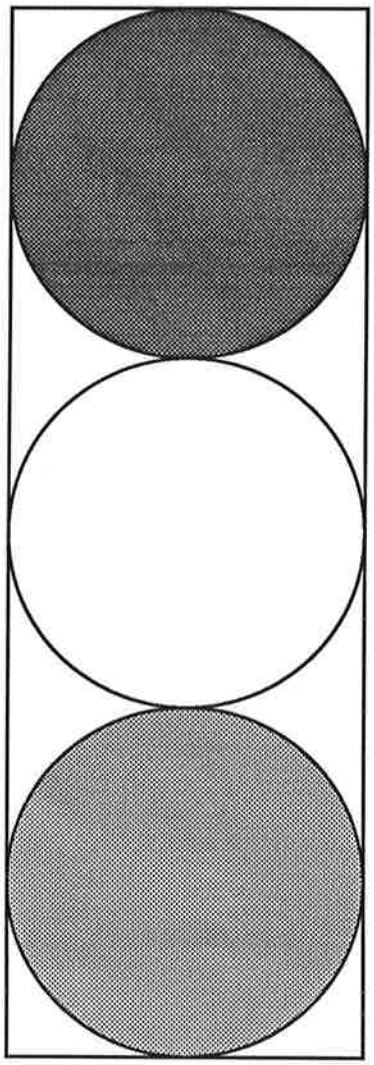

Figure 1.5 Output of the PostScript code in Listing 1.2

2. How to generate PostScript

How must a user adjust his program to generate PostScript code? What code cannot be generated?

2.1 Function DrawPS

The desired PostScript code is generated by calling the new Interactor-function 'DrawPS'. DrawPS opens a new file, sets the Painter's PSflag, calls Draw which calls the interactor's and all interior interactors' Redraw, resets the PSflag and closes the file. Hence, to get PostScript code from the 'H'-program in Chapter 1, the following code has to be supplied.

```
w->Insert(I);
I->DrawPS("Outfile.PS"); // new
w->Run();
```

This one extra line of code leads to the opening and writing of file 'Outfile.PS', which properly sent to the printer will result in the hardcopy of Figure 1.4. However, in a real application, PostScript code would typically be generated when the user presses a mouse button. An input handler function can look like this:

```
void  H::Handle(Event& e)
{
    if (e.eventType == DownEvent)
       DrawPS("Outfile.PS");
}
```

Another feature of `DrawPS` is that it supplies the "BoundingBox" for the picture. The `BoundingBox` is essential when allotting pictures to the TeX word processer.

2.2 Limitations

As explained, only the basic drawing-primitives are implemented in class PSPainter. The lack of "set"-functions (SetPattern, SetBrush etc.) does not affect us that much since we have an XPainter object from which we can extract much of the current XPainter-status.

Orientation

The three orientation functions Translate, Scale and Rotate do affect us negatively while they only deal with the underlying X system. If, for example, we were to call "output->Rotate(90)" together with "output->Translate(inch,0)" in H's constructor in the above example. The output would be an H consisting of I-bars instead of Hs, while the paper-image still would result in the same H.

Fonts

Since we are not able to extract the exact typefaces, the font's size might differ in the string representation. All fixed-width fonts are translated to Courier; for all other fonts the PostScript representation will be Helvetica.

Brush

The InterViews user can set a dashed line:

```
Brush* b= new Brush(xf0f0,10); // (pattern, width)
```

The hexa-decimal number F0F0 results in a 16-bit line-pattern with bits 4-7 and 12-15 set. We are, unfortunately, restricted to the default solid line. Fortunately, the brush's width is used.

Pattern

The user can, as with the brush, set the fill-pattern arbitrarily. However, there are five pre-defined fill-patterns: solid (default), darkgray, gray, lightgray and clear. Only the pre-defined patterns are detected and translated into greyscales.

Other features

Some other features of the Painter have no PostScript equivalent: points, splines, xor-drawing and clearing painted areas. Anyone interested is welcome to try to augment the PSPainter. For example, suppose you want to implement the Painter-function 'Point'. The two things you will have do are:

1. Construct the PSPainter-function, `PSPainter::Point(Canvas*,Coord,Coord)`.
2. Call this function in class Painter's implementation code (file new-painter.c).

3. Implementation

As mentioned, all user interface objects are derived from class Interactor. An interactor can perform output on its canvas using a Painter. A Painter provides drawing operations and manages graphics state such as font, fill pattern, foreground and background colours. Also, Painters can perform arbitrary coordinate transformations composed of translations, rotations and scalings. Now, this project's main purpose is to complement the Painter's drawing operations with functions that generate the equivalent PostScript code. One of the solutions of doing this is presented here (and has been implemented at the department), later a discussion on different solutions follows.

The idea is to rename current class Painter to XPainter, as it uses X Window System. A new class for generating PostScript is introduced: class PSPainter. The new class Painter will keep its place in InterViews' class hierarchy so that the environment will sense no difference from before. The new Painter's functions will all call an XPainter-function, and in some, the basic drawing routines, a call to the corresponding PSPainter-function is made in order to generate PostScript code. Thus, the Painter will serve only as a delegater:

```
Painter::Line(Canvas* c,
              Coord x1, Coord y1, Coord x2, Coord y2)
{
   if (PSflag)
      PS->Line(c,x1,y1,x2,y2);
   X->Line(c,x1,y1,x2,y2);
}
```

And where no corresponding PS-function exists:

```
Painter::Rotate(float angle)
{
   X->Rotate(angle);
}
```

3.1 Class Painter

Figure 3.1 Another subset of the class hierarchy: An interactor's attributes.

3.1.1 Declaration of class Painter

The new class Painter will look to the environment precisely as it did before. Hence, no application programs have to be altered. However, there are some modifications made in the declaration of class Painter (file painter.h).

Added variables:

```
static boolean PSflag
XPainter*  X
PSPainter* PS
```

Removed members (variables and functions):

```
All members declared as private are removed since no other
class object than Painter itself can address them. The one
exception is the function BeginXor, which is used by class
Rubberband.
```

This leads us to the new declaration of class Painter:

```
class Painter: public Resource {
   public:
   static boolean PSflag;      /* new */
   /* Functions as before, see appendix    */
   private:
   friend class Rubberband;    /* As before */

   void BeginXor();            /* As before */
   XPainter* X                 /* new */
   PSPainter* PS               /* new */
};
```

"Information hiding" is a C++ term meaning that non-members of a class can not access non-public (protected or private) members. The way to circumvent this is to declare an object as 'friend' inside the class declaration. In this case, members of class Rubberband will have access to the four private members of class Painter, although the only one it *will* access is BeginXor. PSflag will be accessed by an Interactor function. Since the boolean PSflag is new it implies that the accessing Interactor function also is new (Chapter 2).

3.1.2 Implementation of class Painter

The member functions of class Painter are implemented in a new file called 'newpainter.c'. The functions will look about the same as those shown above with a pointer to a PSPainter function in the following functions:
`Line, MultiLine,`
`Rect, FillRect,`
`Polygon, FillPolygon,`
`Circle, FillCircle,`
and the four different types of `Text`.
The interesting part concerns the two constructors:

```
Painter::Painter(){
   X=  new XPainter;
   PS= new PSPainter(X);
```

```
}

Painter::Painter(Painter* copy){
   X=  new XPainter(copy->X);
   PS= new PSPainter(X);
}
```

The reason for sending an XPainter-object along with the new PSPainter is that the PSPainter has no "set"-functions as the XPainter has. With this X-pointer the PSPainter can call various "get"- functions to access the current font, brush and pattern.

3.2 Class XPainter

Since class XPainter is nothing but former class Painter, its class declaration and implementation code look as class Painter's used to do, except for two things:

1. All instances where the word 'Painter' used to be present, now reads 'XPainter' instead. For example, former

```
Painter::Line(Canvas* c,
              Coord x1,Coord y1, Coord x2, Coord y2)
{ /* ..  */ }
```

is now

```
XPainter::Line(Canvas* c,
              Coord x1,Coord y1, Coord x2, Coord y2)
{ /* ..  */ }
```

2. Since former class Painter had class Rubberband as a friend, class Painter is now declared as friend of class XPainter, so that the Rubberband-member still can access 'BeginXor' through the Painter's corresponding function.

3.3 Class PSPainter

```
class PSPainter   {
public:
  static Interactor* i;

  void Line(Canvas*, Coord ,Coord ,Coord ,Coord  );
  void Rect(Canvas* , Coord ,Coord ,Coord ,Coord  ) ;
  void Circle(Canvas* , Coord ,Coord ,int);
  void FillCircle(Canvas* , Coord ,Coord ,int);
  void MultiLine(Canvas* , Coord x[],Coord y[],int);
  void Polygon(Canvas* , Coord x[],Coord y[],int);
  void FillPolygon(Canvas* , Coord x[],Coord y[],int);
  void FillRect(Canvas* , Coord ,Coord ,Coord ,Coord  ) ;
  void Text(Canvas* , const char*, Coord, Coord);
  void Text(Canvas* , const char*);
  void Text(Canvas* , const char*,int,Coord, Coord);
  void Text(Canvas* , const char*,int);
  PSPainter(XPainter* );
  ~PSPainter();
protected:
  void SetPoint(Coord, Coord);
  void DrawLine(Coord, Coord);
  void Activate(char*, boolean);
  void PolyLine( Coord x[],Coord y[],int);
  void UpDatePosition(Canvas* ,
       Coord, Coord, Coord&, Coord&);
  void UpDateShade();
  void UpDateBrush();
  void UpDateFont();
  void SetSolidShade();
  void GetLineStatus( int&);
  char* GetTextStatus(  int&);
  void GetShadeStatus(  float& );
  void SetWidth( int );
  void SetText(const char*, int );
  void SetShade(float  );
  void Convert(Coord, Coord, Coord, Coord,
       Coord x[] , Coord y[]);
  void WorldRelative(Coord&, Coord&);
  void GetCoordinates(Canvas*, Coord&, Coord&);
private:
  int CurrentWidth;
  float CurrentShade;
  char* FontType;
  int FontHeight;
  XPainter* pp;
};
```

Figure 3.2 Declaration of the PSPainter

Figure 3.2 shows the members of class PSPainter. The public functions (except for the constructor and destructor) we recognize as those listed in section 3.1.2. The protected ones are auxillary-functions and are implemented in a seperate file.
When a PSPainter function is called (only by a Painter function) it uses the class' auxillary-functions to produce PostScript code that eventually will be written to an external file. The private members are variables that hold the current state associated with each PSPainter.

3.3.1 The data members of class PSPainter

`int CurrentWidth:`	The brush's width in pixels, i.e. how wide the lines are.
`float CurrentShade:`	The pattern for filled areas. Takes a value between zero and one. Also, lines and fonts are affected by the fill pattern.
`char* FontType:`	The style with which you write text. Examples of fonts are: Helvetica, Courier, Times.
`int FontHeight:`	The characters' height in pixels.
`XPainter* pp:`	Used to update the four members above.
`static Interactor* i:`	Points to the interactor representing the total display area.

Of the above members, all but the last are private.

3.3.2 The public functions of class PSPainter

Lets begin by looking at the constructor. A constructor must be declared as public, otherwise a new class-object cannot be created by non-friend objects.

```
PSPainter::PSPainter(XPainter* p): pp(p) {
  CurrentWidth=-1;
  CurrentShade=-1;
  FontType=" ";
  FontHeight=0;
}
```

The constructor is called automaticly when creating a class object. Therefore it is suitible to use the constructor as "init"-function. In this case the five private members are initialized, each with an unreasonable value except for the XPainter* pp, which points to the Painter's XPainter. The notation " pp(p) " is another way for writing " pp=p; " inside the function body.
And at last, lets have a look at the actual drawing primitives.

Line

```
void PSPainter::Line(Canvas* c,
     Coord x1,Coord y1,Coord x2,Coord y2 )
{
  Coord x,y;
  UpDatePosition(c,x1, y1, x, y);
  UpDateBrush();
```

```
    SetPoint(x, y);
    DrawLine(x2-x1, y2-y1);
    Activate();
  }
```

While the corresponding XPainter-function just calls the underlying XLine, a cumbersome procedure takes place in order to generate the appropriate PostScript code.

`UpDatePosition:`	Get the actual coordinates. Screen coordinates x1,y1 are transformed to x,y.
`UpDateBrush:`	Checks if the brush's attributes have changed since the last reference. If so, the appropriate "set"-functions are 'called', i.e. they will be called when the code is sent to the printing device.
`SetPoint:`	Sets the actual starting point for the line.
`DrawLine:`	Draws a line from the point previously set.
`Activate:`	Makes it visible.

Of the above five functions, all but UpDatePosition generate PostScript code.

MultiLine

```
void PSPainter::MultiLine(Canvas* c,
  Coord X[], Coord Y[],int n)
{
  Coord x, y;
  UpDatePosition(c,X[0], Y[0], x, y);
  UpDateBrush();
  SetPoint(x, y);
  PolyLine(X,Y,n);
  Activate();
}
```

PolyLine:	Calls DrawLine n times.

Polygon

```
void PSPainter::Polygon(Canvas* c,
        Coord X[], Coord Y[],int n)
{ const boolean closepath=true;
  Coord x, y;
  UpDatePosition(c,X[0], Y[0], x, y);
  UpDateBrush();
  SetPoint(x, y);
  PolyLine(X,Y,n);
  Activate("stroke" , closepath);
}
```

Polygon is a variant on MultiLine. The missing last line is activated by the PostScript command "closepath".

FillPolygon

```
void PSPainter::FillPolygon(Canvas* c,
    Coord X[], Coord Y[],int n)
```

```
{ const boolean closepath=true;
  Coord x, y;
  UpDatePosition(c,X[0], Y[0], x, y);
  UpDateShade();
  SetPoint(x, y);
  PolyLine(X,Y,n);
  Activate("fill" , closepath);
}
```

UpDateShade: Updates the fill pattern.

The command "fill" is used instead of "stroke" (Listing 1.2).

Rect

```
void PSPainter::Rect(Canvas* c,
          Coord x1, Coord y1, Coord x2, Coord y2 )
{ // Four calls to PSPainter's Line is avoided.
  Coord X[4], Y[4];
  Convert( x1 ,y1 ,x2 ,y2, X, Y);
  Polygon(c,X,Y,4);
}
```

Since a rectangle is a polygon, it's better to call Polygon instead of MultiLine or succesive calls to Line.

FillRect

As Rect, with FillPolygon instead of Polygon.

Circle

```
void PSPainter::Circle(Canvas* c, Coord x,Coord y,int r)
{
  Coord Cx, Cy;
  UpDatePosition(c,x, y, Cx, Cy);
  UpDateBrush();
  file<< Cx<<" "<<Cy<<" "<<r<<" "<<0 <<" "<<360<<" arc\n";
  Activate();
}
```

The PostScript command for drawing a circle (or an arc) requires five arguments: the coordinates for the circle's centre, its radius, the start and stop angles.

FillCircle

The same relationship exists between Circle and FillCircle as with Polygon and FillPolygon.

Text

There are four different Text functions in InterViews' system.

```
void PSPainter::Text(Canvas* c, const char* str, int n,
     Coord x,Coord y)
{
  Coord Cx, Cy;
  UpDatePosition(c,x, y, Cx, Cy);
  SetPoint(Cx,Cy);
  UpDateFont();

  file <<"(";
```

```
    for (int i = 0; i < n && str[i] != 0; i++)
      switch (str[i]) {
      case '-':
        // Character '-' is a minus sign in the X fonts,
        // but a hyphen in the LaserWriter;
        // special fix to make hardcopy look like X.
        file << "\\261";
        break;
      case '(' :
      case ')':
      case '\\':
        file << "\\";
        /* fall through */
      default:
        file.put(str[i]);
      }
    file <<")  show newpath\n";
}
```

UpDateFont: Updates the font's type and height.

```
void PSPainter::Text(Canvas* c, const char* str)
{
 Coord Cx, Cy;
 pp->GetPosition(Cx,Cy);  // curx and cury
 Text(c,str,strlen(str),Cx,Cy);
}


void PSPainter::Text(Canvas* c, const char* str, int n)
{
 Coord Cx, Cy;
 pp->GetPosition(Cx,Cy);   // curx and cury
 Text(c,str,n,Cx,Cy);
}

Void PSPainter::Text(Canvas* c, const char* str,
     Coord x, Coord y)
{
 Text(c,str,strlen(str),x,y);
}
```

The last three functions are implemented almost exactly as are the XPainter's.

3.3.3 The protected functions of class PSPainter

Members declared as protected (and public) can be accessed by subclasses. It is said that the subclass "inherits" its baseclass' abilities. If an expansion of class PSPainter takes place in the future, there might be need for subsequent classes. Thus, the current implementation leaves room for further development. Below follows some protected functions worth commenting.

UpDatePosition
Gets the actual coordinates.
Screen coordinates x and y are transformed to paper coordinates newx and newy.

```
void PSPainter::UpDatePosition(Canvas* c,
                Coord x, Coord y, Coord& newx, Coord& newy)
{ // Sets the position for where a drawing begins,
 //  i.e. a line's first point or a circle's centre.
 const int XMinMargin=20;
 const int YMinMargin=30;
 int OrigX, OrigY;

 pp->GetOrigin(OrigX, OrigY);// offx and offy
 newx=x + OrigX+ XMinMargin;
 // MinMargin lifts up the drawing onto the printer's paper.
 newy=y + OrigY+ YMinMargin;
//  The below part makes it possible to display
//  multiple interactors on the same paper.
 Coord X, Y, ry, rx;

 WorldRelative(rx,ry);
 GetCoordinates(c,X,Y);
 newx += X+rx;
 newy += Y+ry- c->Height();
}
```

`WorldRelative:` Gives the coordinates for the total display area, which is represented by the static Interactor* i.

This is why it's declared as static. A static variable holds the same value in every class object. That is, if 'i' were to be assigned another interactor, all PSPainters' 'i's would change to the new pointer. Compare class Painter's PSflag.

`GetCoordinates:` Gives the coordinates for the Canvas c.

WorldRelative returns negative values and GetCoordinates returns the y-value for the top of the canvas.
UpDateFont, UpDateShade and UpDateBrush compare the XPainter pp's current attributes against the PSPainter's. If altered, the appropriate "set"-code is generated.

UpDateFont
Updates the font's type and height.

```
void PSPainter::UpDateFont()
{
  int height;
  char* name=  GetTextStatus( height);
  // Get the XPainter's current status.
  if (height!= FontHeight  ||  strcmp(name,FontType))
  // Compare it against the PSPainter's
    { FontType= name;       // Update
```

```
        FontHeight= height;
        SetText(name,height);// Generate 'Set'-code
      }
    SetSolidShade();
    // Text must have a solid fill-pattern.
}
```

UpDateShade and UpDateBrush in a similar way. Lets have a closer look at GetTextStatus, GetLineStatus (used by UpDateBrush) and GetShadeStatus.

GetTextStatus
Gets the current font's type and height.

```
char* PSPainter::GetTextStatus( int& height)
{  Font* f= pp->GetFont();
   height= f->Height();
   if (f->FixedWidth())  // The sole distinguisher so far
     return "Courier";
   else
     return "Helvetica";
}
```

A user of the InterViews system can choose among a great number of different fonts. However, due to information hiding in class Font, we are not able to extract the exact font. Therefore, we make the best of it and distinguish what we can.

GetLineStatus
Gets the current brush's width.

```
void PSPainter::GetLineStatus( int& BrushWidth)
{
  BrushWidth= pp->GetBrush()->Width();
  if ( BrushWidth==0)
      BrushWidth=1;
}
```

Information hiding in class Brush makes it impossible to see whether the current brush is dashed. We will have to settle with a solid brush.

GetShadeStatus
Gets the current pattern.

```
void PSPainter::GetShadeStatus(float& shade)
{ // Returns a fill pattern composed of a mixture
  // between the XPainter's pattern and the intensities
  // of its foreground colours.
const int full= 65535;
float sh=0; // default= solid
float fg;
int FGcol[3]; // red, green and blue
Pattern* pa=  pp->GetPattern();
```

```
pp->GetFgColor()->DisplayIntensities(
          FGcol[0], FGcol[1], FGcol[2]);
fg=(3*full- FGcol[0]- FGcol[1]- FGcol[2])/(3*full);
if (pa== clear)
   sh= 1;
 else if (pa== lightgray )
   sh= 0.875;
 else if (pa== gray )
   sh= 0.5;
 else if (pa== darkgray)
   sh= 0.125;
  // else solid
 sh=1-sh;
shade=  1- fg*sh;
}
```

The same thing about information hiding goes for class Pattern as well. We are, however, able to check if the current pattern equals any of the five pre-defined fill-patterns.

3.4 Changes in InterViews' file system

The InterViews system is extended with a couple of files containing the code described above.

3.4.1 New files

PSpainter.h
The PSPainter's class declaration.

PSpainter.c
The implementation code of class PSPainter. Contains the public functions. Makes include on file PStools.c.

PStools.c
Contains the auxillary-functions of class PSPainter, declared as protected.

newpainter.c
The implementation code of the new class Painter.

Xpainter.h
Former painter.h. Now with a couple of extra Xs.

3.4.2 Modified files

painter.h
The pointers XPainter* X, PSPainter* PS and the boolean PSflag are added while all private members are removed except for function BeginXor.

painter.c and X11.c
As class Painter is renamed XPainter an extra 'X' is added as preamble on all function headings.

interactor.h and interactor.c
Class Interactor is extended with the new drawing-facility: function 'DrawPS.'

Various header files
Those '.h' files that used to have class Painter declared as a friend now have class XPainter as friend instead. These include: 'brush.h,' 'color.h,' 'font.h,' 'pattern.h' and 'transformer.h.'

3.5 Different solutions

The one alternative to the solution presented is to avoid the creation of the new class XPainter. But instead exploit the existing Painter by adding one or two lines of code (correspondant to the code in newpainter.c) in the concerned functions of class Painter. Such a solution would make for less interference with the existing system as the only files tampered would be files painter.c and X11.c. Also, the new PostScript-generating functions must, of course, be added as well. That is, Section 3.4.1 would shrink to *PSpainter.h , PSpainter.c* and *PStools.c*, while Section 3.4.2 would shrink to *painter.c, X11.c, interactor.h* and *interactor.c*. However, that kind of sneaking requires a great knowledge of how the implementation code works, so that one is sure where to add the extra code. Consider; since the existing Painter class is rather nested, wrongly added code could mean tragical consequences.
With the solution presented it is quite easy to follow the chain of action from the application program's Redraw-function to the paper-hardcopy.

4. Conclusions

With this project, the InterViews system is extended with drawing primitives that "draw on paper". This extension has made the InterViews system grow with one class and five files, making up to 1000 lines of code. The alteration consists roughly of class Painter being divided into two: class XPainter (using the underlying X window system as before) and class PSPainter which generates PostScript code. This 'division' was made easily thanks to InterViews' object-oriented design. However, only the basic drawing-primitives of class Painter (such as lines, rectangles and text) are represented with a PostScript generating facility. Transformations are not, in any way, regarded. Also, due to information hiding, the types of font, brush and pattern might differ between the paper and screen images. These imperfections are hopefully dealt with in a future development.

The goal to extend InterViews with a small number of changes was fulfilled; applications need only one additional statement to generate PostScript.

Acknowledgements

This work was done as a Master's thesis, supervised by Dag M. Brück. Other people whom I am grateful to are the lot at the department.

References

ELMQVIST, HILDING, KARL JOHAN ÅSTRÖM and TOMAS SCHÖNTHAL (1986): *SIMNON User's Guide for MS-DOS Computers*, Department of Automatic Control, Lund Institute of Technology, Sweden.

LINTON, MARK A. and PAUL R. CALDER (1987): "The Design and Implementation of InterViews," *Proc. USENIX C++ Workshop*, November 9-10 1987, Santa Fe, NM, USA.

LINTON, MARK A., JOHN M. VLISSIDES and PAUL R. CALDER (1987): "Composing User Interfaces with InterViews," *IEEE Computer*, **22**, 2, February 1989.

LIPPMAN, STANLEY B. (1989): *A C++ Primer*, Addison-Wesley.

ADOBE (1985): *PostScript Language — Tutorial and Cookbook*, Addison-Wesley.

ADOBE (1985): *PostScript Language — Reference Manual*, Addison-Wesley.

Program listings

This appendix contains program listings for the new and the modified files. Below is the function DrawPS which is added to class Interactor.

```
ofstream file;
void  Interactor::DrawPS(const char* name="Out.PS")  {
/* creating a file for PostScript code */

     PSPainter::i=this;
     file.open(name, ios::out);
     file<<"%%BoundingBox: "<<0<<" "<<0<<" ";
     file<<xmax<<" "<<ymax<<" \n";
     file<<"newpath \n";
     Painter::PSflag=true;
     Draw();
     Painter::PSflag=false;
     file<<"showpage \n";
     file.close();
}
```

painter.h

```
/*
 * Graphics interface
 */

#ifndef painter_h
#define painter_h

#include <InterViews/defs.h>
#include <InterViews/resource.h>

class Canvas;
class Color;
class PainterRep;
class Pattern;
class Brush;
class Font;
class Transformer;
class Bitmap;
class Raster;
class XPainter;
class PSPainter;

class Painter : public Resource {
public:
    static boolean PSflag;
    Painter();
    Painter(Painter*);
    ~Painter();
    void FillBg(boolean);
    boolean BgFilled();
    void SetColors(Color* f, Color* b);
    Color* GetFgColor();
    Color* GetBgColor();
    void SetPattern(Pattern*);
```

```
    Pattern* GetPattern();
    void SetBrush(Brush*);
    Brush* GetBrush();
    void SetFont(Font*);
    Font* GetFont();
    void SetStyle(int);
    int GetStyle();
    void SetTransformer(Transformer*);
    Transformer* GetTransformer();
    void MoveTo(int x, int y);
    void GetPosition(int& x, int& y);
    void SetOrigin(int x0, int y0);
    void GetOrigin(int& x0, int& y0);

    void Translate(float dx, float dy);
    void Scale(float x, float y);
    void Rotate(float angle);

    void Clip(Canvas*, Coord left, Coord bottom, Coord right, Coord top);
    void NoClip();
    void SetOverwrite(boolean);
    void SetPlaneMask(int);

    void Text(Canvas*, const char*);
    void Text(Canvas*, const char*, int);
    void Text(Canvas*, const char*, Coord, Coord);
    void Text(Canvas*, const char*, int, Coord, Coord);
    void Stencil(Canvas*, Coord x, Coord y, Bitmap* image, Bitmap* mask = nil);
    void RasterRect(Canvas*, Coord x, Coord y, Raster*);
    void Point(Canvas*, Coord x, Coord y);
    void MultiPoint(Canvas*, Coord x[], Coord y[], int n);
    void Line(Canvas*, Coord x1, Coord y1, Coord x2, Coord y2);
    void Rect(Canvas*, Coord x1, Coord y1, Coord x2, Coord y2);
    void FillRect(Canvas*, Coord x1, Coord y1, Coord x2, Coord y2);
    void ClearRect(Canvas*, Coord x1, Coord y1, Coord x2, Coord y2);
    void Circle(Canvas*, Coord x, Coord y, int r);
    void FillCircle(Canvas*, Coord x, Coord y, int r);
    void Ellipse(Canvas*, Coord x, Coord y, int r1, int r2);
    void FillEllipse(Canvas*, Coord x, Coord y, int r1, int r2);
    void MultiLine(Canvas*, Coord x[], Coord y[], int n);
    void Polygon(Canvas*, Coord x[], Coord y[], int n);
    void FillPolygon(Canvas*, Coord x[], Coord y[], int n);
    void BSpline(Canvas*, Coord x[], Coord y[], int n);
    void ClosedBSpline(Canvas*, Coord x[], Coord y[], int n);
    void FillBSpline(Canvas*, Coord x[], Coord y[], int n);
    void Curve(Canvas*,
Coord x0, Coord y0, Coord x1, Coord y1,
Coord x2, Coord y2, Coord x3, Coord y3
    );
    void CurveTo(Canvas*,
Coord x0, Coord y0, Coord x1, Coord y1, Coord x2, Coord y2
    );
    void Copy(
Canvas* src, Coord x1, Coord y1, Coord x2, Coord y2,
Canvas* dst, Coord x0, Coord y0
    );
    void Read(Canvas*, void*, Coord x1, Coord y1, Coord x2, Coord y2);
    void Write(Canvas*, const void*, Coord x1, Coord y1, Coord x2, Coord y2);


private:
    friend class Rubberband;
    void Begin_xor();

    XPainter* X;
    PSPainter* PS;

};


#endif
```

Xpainter.h

```
/*
 * Graphics interface
 */

// The old painter.h
// now: Xpainter.h

#ifndef xpainter_h
#define xpainter_h

#include <InterViews/defs.h>
#include <InterViews/resource.h>


class Canvas;
class Color;
class PainterRep;
class Pattern;
class Brush;
class Font;
class Transformer;
class Bitmap;
class Raster;

class XPainter : public Resource {
public:
    XPainter();
    XPainter(XPainter*);
    ~XPainter();
    void FillBg(boolean);
    boolean BgFilled();
    void SetColors(Color* f, Color* b);
    Color* GetFgColor();
    Color* GetBgColor();
    void SetPattern(Pattern*);
    Pattern* GetPattern();
    void SetBrush(Brush*);
    Brush* GetBrush();
    void SetFont(Font*);
    Font* GetFont();
    void SetStyle(int);
    int GetStyle();
    void SetTransformer(Transformer*);
    Transformer* GetTransformer();
    void MoveTo(int x, int y);
    void GetPosition(int& x, int& y);
    void SetOrigin(int x0, int y0);
    void GetOrigin(int& x0, int& y0);

    void Translate(float dx, float dy);
    void Scale(float x, float y);
    void Rotate(float angle);

    void Clip(Canvas*, Coord left, Coord bottom, Coord right, Coord top);
    void NoClip();
    void SetOverwrite(boolean);
    void SetPlaneMask(int);

    void Text(Canvas*, const char*);
    void Text(Canvas*, const char*, int);
    void Text(Canvas*, const char*, Coord, Coord);
    void Text(Canvas*, const char*, int, Coord, Coord);
    void Stencil(Canvas*, Coord x, Coord y, Bitmap* image, Bitmap* mask = nil);
    void RasterRect(Canvas*, Coord x, Coord y, Raster*);
    void Point(Canvas*, Coord x, Coord y);
    void MultiPoint(Canvas*, Coord x[], Coord y[], int n);
    void Line(Canvas*, Coord x1, Coord y1, Coord x2, Coord y2);
    void Rect(Canvas*, Coord x1, Coord y1, Coord x2, Coord y2);
```

```
    void FillRect(Canvas*, Coord x1, Coord y1, Coord x2, Coord y2);
    void ClearRect(Canvas*, Coord x1, Coord y1, Coord x2, Coord y2);
    void Circle(Canvas*, Coord x, Coord y, int r);
    void FillCircle(Canvas*, Coord x, Coord y, int r);
    void Ellipse(Canvas*, Coord x, Coord y, int r1, int r2);
    void FillEllipse(Canvas*, Coord x, Coord y, int r1, int r2);
    void MultiLine(Canvas*, Coord x[], Coord y[], int n);
    void Polygon(Canvas*, Coord x[], Coord y[], int n);
    void FillPolygon(Canvas*, Coord x[], Coord y[], int n);
    void BSpline(Canvas*, Coord x[], Coord y[], int n);
    void ClosedBSpline(Canvas*, Coord x[], Coord y[], int n);
    void FillBSpline(Canvas*, Coord x[], Coord y[], int n);
    void Curve(Canvas*,
               Coord x0, Coord y0, Coord x1, Coord y1,
               Coord x2, Coord y2, Coord x3, Coord y3);
    void CurveTo(Canvas*,
                 Coord x0, Coord y0, Coord x1, Coord y1,
                 Coord x2, Coord y2);
    void Copy(Canvas* src,
              Coord x1, Coord y1, Coord x2, Coord y2,
              Canvas* dst, Coord x0, Coord y0);
    void Read(Canvas*, void*, Coord x1, Coord y1, Coord x2, Coord y2);
    void Write(Canvas*, const void*, Coord x1, Coord y1, Coord x2, Coord y2);

    PainterRep* Rep();
private:
    friend class Painter;

    Color* foreground;
    Color* background;
    Pattern* pattern;
    Brush* br;
    Font* font;
    int style;
    Coord curx, cury;
    int xoff, yoff;
    Transformer* matrix;
    PainterRep* rep;

    void Init();
    void Copy(XPainter*);
    void Begin_xor();
    void End_xor();
    void Map(Canvas*, Coord x, Coord y, Coord& mx, Coord& my);
    void Map(Canvas*, Coord x, Coord y, short& sx, short& sy);
    void MapList(Canvas*, Coord x[], Coord y[], int n, Coord mx[], Coord my[]);
    void MapList(Canvas*, float x[], float y[], int n, Coord mx[], Coord my[]);
    void MultiLineNoMap(Canvas* c, Coord x[], Coord y[], int n);
    void FillPolygonNoMap(Canvas* c, Coord x[], Coord y[], int n);
};

inline PainterRep* XPainter::Rep () { return rep; }

#endif
```

PSpainter.h

```
//   PSpainter.h                    ====================================
#ifndef pspainter_h
#define pspainter_h

#include <InterViews/defs.h>

#include <iostream.h>
#include <fstream.h>
extern ofstream file;
```

```
class Interactor;
class XPainter;
class Canvas;
class PSPainter {
public:
  static Interactor* i;


  void Line(Canvas*, Coord ,Coord ,Coord ,Coord  );
  void Rect(Canvas* , Coord ,Coord ,Coord ,Coord  ) ;
  void Circle(Canvas* , Coord ,Coord ,int);
  void FillCircle(Canvas* , Coord ,Coord ,int);
  void MultiLine(Canvas* , Coord x[],Coord y[],int);
  void Polygon(Canvas* , Coord x[],Coord y[],int);
  void FillPolygon(Canvas* , Coord x[],Coord y[],int);
  void FillRect(Canvas* , Coord ,Coord ,Coord ,Coord  ) ;
  void Text(Canvas* , const char*, Coord, Coord);
  void Text(Canvas* , const char*);
  void Text(Canvas* , const char*,int,Coord, Coord);
  void Text(Canvas* , const char*,int);
  PSPainter(XPainter* );

  ~PSPainter();

protected:  // In PStools
  void SetPoint(Coord, Coord);
  void DrawLine(Coord, Coord);
  void Activate(char*, boolean);
  void PolyLine( Coord x[],Coord y[],int);
  void UpDatePosition(Canvas* , Coord, Coord,Coord&, Coord&);
  void UpDateShade();
  void UpDateBrush();
  void UpDateFont();
  void SetSolidShade();
  void GetLineStatus( int&);
  char* GetTextStatus(  int&);
  void GetShadeStatus(  float& );
  void SetWidth( int );
  void SetText(const char*, int );
  void SetShade(float  );
  void Convert(Coord, Coord, Coord, Coord,Coord x[] , Coord y[]);
  void WorldRelative(Coord&, Coord&);
  void GetCoordinates(Canvas*, Coord&, Coord&);
private:

  int CurrentWidth;
  float CurrentShade;
  char* FontType;
  int FontHeight;
  XPainter* pp;   // To use for status information
};  // class PSPainter

#endif
```

newpainter.c

```
#include <InterViews/painter.h>
#include <InterViews/Xpainter.h>
#include <InterViews/PSpainter.h>
#include <InterViews/font.h>



boolean Painter::PSflag=false;
```

```
void Painter::  FillBg(boolean b)
{
     X->FillBg(b);
}



void Painter::SetColors(Color* f, Color* b)
{
     X->SetColors( f,  b);
}

void Painter::SetPattern(Pattern* p)
{
     X->SetPattern(p);
}

void Painter:: SetBrush(Brush* b)
{
     X->SetBrush(b);
}

void Painter::SetFont(Font* f)
{
     X->SetFont(f);
}

void Painter::SetStyle(int i)
{
     X->SetStyle(i);
}

void Painter::SetTransformer(Transformer* t)
{
     X->SetTransformer(t);
}



void Painter::MoveTo(int x, int y)
{
     X->MoveTo(x,  y);
}


void Painter::  GetPosition(int& x, int& y)
{
     X->GetPosition( x, y);
}


void Painter::SetOrigin(int x0, int y0)
{
     X->SetOrigin(x0, y0);
}


void Painter::GetOrigin(int& x0, int& y0)
{
     X->GetOrigin( x0, y0);
}


void Painter::Translate(float dx, float dy)
{
     X-> Translate(dx, dy);
}

void Painter::Scale(float x, float y)
{
```

```
    X->Scale( x, y);
}

void Painter::Rotate(float angle)
{
    X->Rotate( angle);
}

void Painter::Clip(Canvas* c, Coord left, Coord bottom, Coord right, Coord top)
{
    X->Clip(c, left,bottom, right,  top);
}

void Painter::NoClip()
{
    X->NoClip();
}


void Painter::SetOverwrite(boolean b)
{
    X->SetOverwrite(b);
}

void Painter::SetPlaneMask(int i)
{
    X->SetPlaneMask(i);
}


void Painter::Stencil(Canvas* c, Coord x, Coord y, Bitmap* image, Bitmap* mask)
{

    X->Stencil(c, x, y, image, mask);

}

void Painter::RasterRect(Canvas* c, Coord x, Coord y, Raster* r)
{

    X->RasterRect(c, x, y, r);

}

void Painter::Point(Canvas* c, Coord x, Coord y)
{

    X->Point(c, x, y);

}

void Painter::MultiPoint(Canvas* c, Coord x[], Coord y[], int n)
{

    X->MultiPoint(c, x, y, n);

}



void Painter::Ellipse(Canvas* c, Coord x, Coord y, int r1, int r2)
{

    X->Ellipse(c, x, y, r1,  r2);

}


void Painter::FillEllipse(Canvas* c, Coord x, Coord y, int r1, int r2)
{
```

```
    X->FillEllipse(c, x, y, r1,  r2);

}

void Painter::BSpline(Canvas* c, Coord x[], Coord y[], int n)
{

    X->BSpline(c,x,y,n);

}


void Painter::ClosedBSpline(Canvas* c, Coord x[], Coord y[], int n)
{

    X->ClosedBSpline(c,x,y,n);

}


void Painter::FillBSpline(Canvas* c, Coord x[], Coord y[], int n)
{

    X->FillBSpline(c,x,y,n);

}

void Painter::Curve(Canvas* c,
Coord x0, Coord y0, Coord x1, Coord y1,
Coord x2, Coord y2, Coord x3, Coord y3)
{

    X->Curve(c, x0, y0,x1,y1,x2,y2,x3,y3);

}

void Painter::CurveTo(Canvas* c,
Coord x0, Coord y0, Coord x1, Coord y1, Coord x2, Coord y2)
{

    X->CurveTo(c, x0, y0,x1,y1,x2,y2);

}

void Painter:: Copy(
Canvas* src, Coord x1, Coord y1, Coord x2, Coord y2,
Canvas* dst, Coord x0, Coord y0)
{

    X->Copy(src,x1,y1,x2,y2,dst,x0,y0);

}

void Painter::Read(Canvas* c, void* v, Coord x1, Coord y1, Coord x2, Coord y2)
{

    X->Read(c, v, x1, y1, x2, y2);

}

// one private painter
void Painter::Begin_xor()
{

   X->Begin_xor();
}
void Painter::Write(Canvas* c,const void* v,Coord x1,Coord y1,Coord x2,Coord y2)
{

    X->Write(c, v, x1, y1, x2, y2);
```

```
}
// Painters with return type. ============================


boolean Painter::BgFilled()
{

  return   X->BgFilled();

}

 Color* Painter::GetFgColor()
{

  return   X->GetFgColor();

}

 Color* Painter::GetBgColor()
{

  return   X->GetBgColor();

}

  Pattern* Painter::GetPattern()
{

  return   X->GetPattern();

}

 Brush* Painter::GetBrush()
{

  return   X->GetBrush();

}

 Font* Painter::GetFont()
{

  return   X->GetFont();

}

 int Painter::GetStyle()
{

  return   X->GetStyle();

}

Transformer* Painter::GetTransformer()
{

  return   X->GetTransformer();

}



// Painters containing pointers to PostScript-generating functions.

void Painter::Text(Canvas* c, const char* str)
{
  if (PSflag)
     PS->Text(c, str);
  X->Text(c, str);
}
```

```
void Painter::Text(Canvas* c, const char* str, int i)
{
   if (PSflag)
     PS->Text(c, str,i);
   X->Text(c, str,i);
}

void Painter::Text(Canvas* c, const char* str, int i, Coord x, Coord y)
{
   if (PSflag)
     PS->Text(c, str,i,x,y);
   X->Text(c, str,i,x,y);
}

void Painter::Text(Canvas* c,const char* str,  Coord x ,Coord y)
{
  if (PSflag )
     PS->Text(c,str,x,y);
  X->Text(c,str,x,y);
}

void Painter::Line(Canvas* c, Coord x1,Coord y1,Coord x2,Coord y2 )
{
   if (PSflag )
        PS->Line(c, x1,y1,x2,y2);
   X->Line(c , x1, y1, x2, y2);
}


void Painter::FillRect(Canvas* c, Coord x1,Coord y1,Coord x2,Coord y2 )
{

   if (PSflag )
      PS->FillRect(c, x1, y1, x2, y2);
   X->FillRect(c , x1, y1, x2, y2);
}



void Painter::ClearRect(Canvas* c, Coord x1,Coord y1,Coord x2,Coord y2 )
{
   X->ClearRect(c,x1, y1, x2, y2);
}


void Painter::Rect(Canvas* c, Coord x1,Coord y1,Coord x2,Coord y2 )
{
   if (PSflag )
      PS->Rect(c, x1,y1,x2,y2);
   X->Rect(c , x1, y1, x2, y2);
}


void Painter::Polygon(Canvas* c,  Coord XA[],Coord Y[],int n)
{
  if (PSflag )
     PS->Polygon(c,XA,Y,n);
  X->Polygon(c,XA,Y,n);
}



void Painter::FillPolygon(Canvas* c,Coord XA[], Coord Y[], int n)
{
  if (PSflag )
     PS->FillPolygon(c,XA,Y,n);
  X->FillPolygon(c,XA,Y,n);
}


void Painter::MultiLine(Canvas* c,   Coord XA[],Coord Y[],int n)
```

```
{
  if (PSflag )
     PS->MultiLine(c,XA,Y,n);
  X->MultiLine(c,XA,Y,n);
}


void Painter::Circle(Canvas* c, Coord x,Coord y,int r)
{
   if (PSflag )
       PS->Circle(c, x, y, r);
   X->Circle(c, x, y, r);
}

void Painter::FillCircle(Canvas* c, Coord x,Coord y,int r)
{
   if (PSflag )
       PS->FillCircle(c, x, y, r);
   X->FillCircle(c, x, y, r);
}



Painter::  Painter()
{ // constructor
 X=  new XPainter();
 PS= new PSPainter(X);
}


Painter::  Painter(Painter* copy)
{ // constructor
 X=  new XPainter(copy->X);
 PS= new PSPainter(X);
}


Painter::  ~Painter()
{// destructor
   if (LastRef()) {
     delete PS;
     delete X;
   }
}
```

PSpainter.c

```
#include <InterViews/PSpainter.h>


#include "PStools.c"



void PSPainter::Line(Canvas* c,  Coord x1,Coord y1,Coord x2,Coord y2 )
{
   Coord x,y;

  UpDatePosition(c,x1, y1, x, y);
  UpDateBrush();
  SetPoint(x, y);
  DrawLine(x2-x1, y2-y1);
  Activate();
}


void PSPainter::MultiLine(Canvas* c,  Coord X[],Coord Y[],int n)
{
```

```
 Coord x, y;
  UpDatePosition(c,X[0], Y[0], x, y);
  UpDateBrush();
  SetPoint(x, y);
  PolyLine(X,Y,n);
  Activate();
}


void PSPainter::Polygon(Canvas* c,       Coord X[],Coord Y[],int n)
{ const boolean closepath=true;
  Coord x, y;
  UpDatePosition(c,X[0], Y[0], x, y);
  UpDateBrush();
  SetPoint(x, y);
  PolyLine(X,Y,n);
  Activate("stroke" , closepath);
}


void PSPainter::FillPolygon(Canvas* c,     Coord X[],Coord Y[],int n)
{ const boolean closepath=true;
 Coord x, y;
  UpDatePosition(c,X[0], Y[0], x, y);
  UpDateShade();
  SetPoint(x, y);
  PolyLine(X,Y,n);
  Activate("fill" , closepath);
}


void PSPainter::Rect(Canvas* c, Coord x1,Coord y1,Coord x2,Coord y2 )
{ // Four calls to PSPainter's Line is avoided.
  Coord X[4], Y[4];
  Convert( x1 ,y1 ,x2 ,y2, X, Y);
  Polygon(c,X,Y,4);

}


void PSPainter::FillRect(Canvas* c, Coord x1,Coord y1,Coord x2,Coord y2 )
{ // Four calls to PSPainter's Line is avoided.
  Coord X[4], Y[4];
  Convert( x1 ,y1 ,x2 ,y2, X, Y);
  FillPolygon(c, X,Y,4);
}


void PSPainter::Text(Canvas* c, const char* str, int n, Coord x,Coord y)
{
  Coord Cx, Cy;
  UpDatePosition(c,x, y, Cx, Cy);
  SetPoint(Cx,Cy);
  UpDateFont();

  file <<"(";
  for (int i = 0; i < n && str[i] != 0; i++) {
    switch (str[i]) {
    case '-':
      // Character '-' is a minus sign in the X fonts, but a hyphen in
      // the LaserWriter; special fix to make hardcopy look like X.
      file << "\\261";
      break;
    case '(' :
    case ')':
    case '\\':
      file << "\\";
      /* fall through */
    default:
      file.put(str[i]);
    }
```

```
  }
  file <<")  show newpath\n";

}


void PSPainter::Text(Canvas* c, const char* str)
{
 Coord Cx, Cy;
 pp->GetPosition(Cx,Cy);
 Text(c,str,strlen(str),Cx,Cy);
}


void PSPainter::Text(Canvas* c, const char* str, int n)
{
 Coord Cx, Cy;
 pp->GetPosition(Cx,Cy);
 Text(c,str,n,Cx,Cy);
}



void PSPainter::Text(Canvas* c, const char* str,  Coord x, Coord y)
{
 Text(c,str,strlen(str),x,y);
}


void PSPainter::Circle(Canvas* c,        Coord x,Coord y,int r)

{
 Coord Cx, Cy;
  UpDatePosition(c,x, y, Cx, Cy);
  UpDateBrush();
  file<< Cx <<" "<<Cy <<" "<< r<<" "<<0 <<" "<<360<<" arc\n";
  Activate();
}


void PSPainter::FillCircle(Canvas* c,   Coord x,Coord y,int r)

{
 Coord Cx, Cy;
  UpDatePosition(c,x, y, Cx, Cy);
  UpDateShade();
  file<< Cx <<" "<<Cy <<" "<< r<<" "<<0 <<" "<<360<<" arc\n";
  Activate("fill");
}



PSPainter::  PSPainter(XPainter* p): pp(p) // pp gets initialized

{ // constructor
 // A file for PostScript code is opened in DrawPS
   CurrentWidth=-1;
   CurrentShade=-1;
   FontType=" ";
   FontHeight=0;


}


PSPainter::  ~PSPainter()
{// destructor

   delete pp;
}
```

PStools.c

```
#include <strings.h>
#include "itable.h"
#include <InterViews/canvas.h>
#include <InterViews/pattern.h>
#include <InterViews/painter.h>
#include <InterViews/world.h>
#include <InterViews/brush.h>
#include <InterViews/font.h>
#include <InterViews/color.h>
#include <InterViews/Xpainter.h>
class WorldRep;
extern "C" {
#include <InterViews/X11/worldrep.h>
#include <InterViews/X11/Xlib.h>
#include <InterViews/X11/Xutil.h>
#include <X11/Xatom.h>
#ifdef __cplusplus
#endif
}



// The PSPainter's tools:

extern WorldRep* _world;


void PSPainter::GetCoordinates(Canvas* c, Coord& x, Coord& y)
{
  Coord rx, ry;
  Window child;
  x=0;
  y=0;
  XTranslateCoordinates(_world->display(),
(Window)c->Id(), (Window)_world->root(),
x, y, &rx, &ry, &child);
  x = rx;
  y =900-ry; // The screen is 900 pixels high.
}



void PSPainter::SetPoint( Coord x,Coord y)
{


 file <<x << " "<<y  << " moveto \n";
}



void PSPainter::SetWidth(int w)
{
 file<< w  << " setlinewidth \n";
}

void PSPainter::SetText(const char* name, int height)
{
file  << "/"<< name<<  " findfont \n";

file  << height<<  " scalefont setfont \n";
}



void PSPainter::SetShade(float sh)
{
 file  << sh<<  " setgray \n";
```

```
}


void PSPainter::DrawLine( Coord x,Coord y)
{
 // Draws a line from the current point

  file <<x << " "<<y  << " rlineto \n";
}


void PSPainter::PolyLine( Coord X[],Coord Y[], int n)
{
  for (int i=1; i<n; i++)
    DrawLine(X[i]-X[i-1], Y[i]-Y[i-1]);
}



void PSPainter::Activate(char* ActType="stroke",  // stroke or fill
 boolean close=false)

{// make it visible
  if ( close)
     file << "closepath ";
  file << ActType<<" \n";
}

void PSPainter::GetShadeStatus(float& shade)
{ // Returns a fill pattern composed of a mixture
  // between the XPainter's pattern and the intensities
  // of its foreground colours.
const int full= 65535;
float sh=0; // default= solid
float fg;
int FGcol[3]; // red, green and blue
Pattern* pa;

pp->GetFgColor()->DisplayIntensities(FGcol[0], FGcol[1], FGcol[2]);
fg=(3*full- FGcol[0]- FGcol[1]- FGcol[2])/(3*full);
pa=  pp->GetPattern();

if (pa== clear)
   sh= 1;
 else if (pa== lightgray )
   sh= 0.875;
 else if (pa== gray )
   sh= 0.5;
 else if (pa== darkgray)
   sh= 0.125;
 // else solid
 sh=1-sh;

shade=  1- fg*sh;
}



void PSPainter::GetLineStatus( int& BrushWidth)
{
  BrushWidth= pp->GetBrush()->Width();
  if ( BrushWidth==0)
       BrushWidth=1;
}


char* PSPainter::GetTextStatus( int& height)
{  Font* f= pp->GetFont();
```

```
    height= f->Height();

    if (f->FixedWidth())  // The sole distinguisher so far
      return "Courier";
    else
      return "Helvetica";

}


void PSPainter::Convert(Coord x1, Coord y1, Coord x2, Coord y2,
                Coord Ax[], Coord Ay[])
{

 Ax[0] =x1 ;
 Ay[0] =y1 ;

 Ax[1] =x1 ;
 Ay[1] =y2 ;

 Ax[2] =x2 ;
 Ay[2] =y2 ;

 Ax[3] =x2 ;
 Ay[3] =y1 ;
}


void PSPainter::SetSolidShade()
{
  if (CurrentShade!=0.0)
    {  CurrentShade= 0.0;
       SetShade(0.0 );
    }
}

void PSPainter::UpDateFont()
{
  int height;
  char* name=  GetTextStatus( height);
  // Get the XPainter's current status.
  if (height!= FontHeight  ||  strcmp(name,FontType))
  // Compare it against th PSPainter's
    { FontType= name;       // Update
      FontHeight= height;
      SetText(name,height);// Generate "Set"-code
    }
  SetSolidShade();
  // Text must have a solid fill-pattern.
}

void PSPainter::UpDateShade()
{ float Shade;

  GetShadeStatus(Shade );
  if (Shade !=CurrentShade )
    {  CurrentShade=Shade;
       SetShade(Shade );
    }
}



void PSPainter::UpDateBrush()
{ int Width ;

  GetLineStatus(Width );
  if (Width !=CurrentWidth )
    {  CurrentWidth=Width;
       SetWidth(Width );
    }
```

```
  SetSolidShade();
}



void PSPainter:: WorldRelative(Coord& x, Coord& y)
{ x=0;
  y=0;
  i->GetWorld()->GetRelative(x,y,i);
}



void PSPainter::UpDatePosition(Canvas* c,Coord x,Coord y,Coord& newx,Coord& newy)
{ // Sets the position for where a drawing begins,
 //  i.e. a line's first point or a circle's centre.
 const int XMinMargin=20;
 const int YMinMargin=30;
 int OrigX, OrigY;

 pp->GetOrigin(OrigX, OrigY);// offx and offy
 newx=x + OrigX+ XMinMargin; // MinMargin lifts up the drawing onto the
 newy=y + OrigY+ YMinMargin; // printer's paper.
//  The below part makes it poosible to display
//  multiple interactors on the same paper.
 Coord X, Y, ry, rx;

 WorldRelative(rx,ry);
 GetCoordinate(c,X,Y);
 newx += X+rx;
 newy += Y+ry- c->Height();
}
```