```
/*
 * Copyright (C) Rutherford Appleton Laboratory 1987
 *
 * This source may be copied, distributed, altered or used, but not
 * sold for profit or incorporated into a product except under licence
 * from the author.
 * It is not in the public domain.
 * This notice should remain in the source unaltered, and any changes
 * to the source made by persons other than the author should be
 * marked as such.
 *
 *      Crispin Goswell @ Rutherford Appleton Laboratory caag@uk.ac.rl.vd
 */
/* X11R2 modifications and fixes made from sources by Crispin Goswell,
 * Barry Shein of BU, and Jean Marie Diaz of MIT.  The resulting
 * melange, along with other fixes for color workstations, is entirely
 * my fault.
 *
 * David Jedlinsky
 * MIT Project Athena Watchmaker
 * opus@athena.mit.edu
 *
 */
#include "main.h"
#include "graphics.h"
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xatom.h>
#include <stdio.h>

#define ISWIN   1
#define ISBITMAP 2

typedef int ColorType;
#define ISPIXEL         1
#define ISHALFTONE      2

struct screen
 {
        float val;
        int sx, sy;
        struct hardware *shade;
 } *screen = NULL;

static int screen_size;

#define TRANSFER_SIZE   256

static int transfer [TRANSFER_SIZE];

int pixels_per_inch;

int single_rop [] =
 {
        ROP_FALSE, ROP_DEST, ROP_NOTDEST, ROP_TRUE,
        ROP_FALSE, ROP_DEST, ROP_NOTDEST, ROP_TRUE,
        ROP_FALSE, ROP_DEST, ROP_NOTDEST, ROP_TRUE,
        ROP_FALSE, ROP_DEST, ROP_NOTDEST, ROP_TRUE
 };

void InitTransfer (ppi) int ppi;
{
     int i;
     pixels_per_inch = ppi;
     for (i = 0; i < TRANSFER_SIZE; i++)
          transfer [i] = i;
}

static void Punt(str)
char *str;
{
    fprintf(stderr, "%s\n", str);
    exit(1);
}

static Display *dpy;
static int SCREEN;
static struct hardware *NewHardware ();
struct hardware *GraySync ();

extern HardPoint DeviceToHard ();

typedef struct _HardwareRec {
    Drawable w;
    GC gc;
    union last_color {
         unsigned long pix;
         Pixmap halftone;
    } last_color;
    short last_used;
} HardwareRec, *Hardware;

/*
 * This file describes the interface that PostScript requires to the graphics
 * system at Version 1.4.
 *
 * ''Hardware'' in this context refers to a pointer to windows and/or bitmaps
 * and is the lowest level of access that PostScript is interested in. Any
 * Hardware parameter may be expected to be NULL.
 */

/********************* CREATION OF WINDOWS AND BITMAPS *******************/

#define MIN(x, y)       (((x) < (y)) ? (x) : (y))

static GC NOP_GC = NULL;
static GC fillgc[16];
static GC BitmapGC[16];
static GC whitegc;

/*
 * InitHardware () returns a default device which PostScript may use
 * immediately (or NULL if not appropriate).  Its size and shape are
 * not defined. Most typically the user will want to start up another
 * device before it is used anyway. No attempt will be made by
 * PostScript to Destroy the resulting device.
 */
struct hardware *InitHardware ()
{
    XGCValues values;
    int i;
    DevicePoint p;

    if ((dpy = XOpenDisplay("")) == NULL)
        Punt("Could not open display");
    SCREEN = DefaultScreen(dpy);
/* DCJ:
```

```
 * This is the real ppi.  However, we might be using a screen that
 * is less than 11 inches tall, which makes life difficult.  So we
 * cheat a bit.
 */

/*    if (DisplayHeightMM(dpy, SCREEN) < 254)
         InitTransfer(DisplayHeight(dpy, SCREEN) / 14);
    else
*/       InitTransfer((int) ((float) DisplayHeight(dpy, SCREEN) * (float) 25.4 / (float)

    values.foreground = BlackPixel(dpy, SCREEN);
    values.background = WhitePixel(dpy, SCREEN);

    for (i=0 ; i<16 ; i++) {
         BitmapGC[i] = NULL;
         values.function = i;
         fillgc[i] = XCreateGC(dpy, RootWindow(dpy, SCREEN),
                               GCFunction | GCForeground | GCBackground,
                               &values);
    }

    p.dx = DisplayWidth(dpy, SCREEN);
    p.dy = DisplayHeight(dpy, SCREEN);
    return NewHardware(p);
}

static struct hardware *NewHardware(width, height)
int width, height;
{
    struct hardware *to;
    Hardware hard;

    to = (struct hardware *) malloc(sizeof(struct hardware));
    hard = (Hardware) malloc(sizeof(HardwareRec));
    to->hard.addr = (char *) hard;
    to->flags = 0;
    to->aux = to->clip = (struct hardware *) NULL;
    to->extent = NewDevicePoint(width, height);
    hard->w = (Drawable) NULL;
    hard->gc = (GC) NULL;
    to->touched = FALSE;
    to->changed = TRUE;
    hard->last_used = 0;
    return to;
}

/*
 * NewBitmapHardware () is expected to create a new bitmap. Only one plane
 * will be needed.
 *
 * NewWindowHardware () is expected to create a window on the screen. On a
 * colour system this will be expected to support full colour.
 */
struct hardware *NewBitmapHardware (width, height)
int width, height;
{
     XGCValues values;
     struct hardware *to = NewHardware(width, height);
     Hardware hard = (Hardware) to->hard.addr;
     int i;

     to->flags = ISBITMAP;

     if (width == 0 || height == 0) {
          return NULL;
     }
     hard->w = XCreatePixmap(dpy, RootWindow(dpy, SCREEN), width,
                             height, 1);

     if (! BitmapGC[GXcopy]) {
          values.foreground = 1;
          values.background = 0;

          for (i=0; i<16; i++) {
               values.function = i;
               BitmapGC[i] = XCreateGC(dpy, hard->w,
                                       GCFunction|GCForeground|GCBackground,
                                       &values);
          }
     }
     XFillRectangle(dpy, hard->w, BitmapGC[GXclear], 0, 0,
                    width, height);

     return to;
}

struct hardware *NewWindowHardware (width, height)
int width, height;
{
     struct hardware *to = NewHardware(width, height);
     Hardware hard = (Hardware) to->hard.addr;
     XEvent event;
     unsigned long vmask;
     XSetWindowAttributes xswa;

     to->flags = ISWIN;
     vmask = CWBackPixel|CWBorderPixel|CWBackingStore|
          CWBackingPlanes;
     xswa.background_pixel = WhitePixel(dpy,SCREEN);
     xswa.border_pixel = BlackPixel(dpy,SCREEN);
     xswa.backing_store = Always;
     xswa.backing_planes = AllPlanes;

     hard->w = XCreateWindow(dpy, RootWindow(dpy,SCREEN),
                             0, 0, width, height,
                             1, DefaultDepth(dpy,SCREEN),
                             InputOutput, DefaultVisual(dpy,SCREEN),
                             vmask, &xswa);

     XChangeProperty(dpy,hard->w,XA_WM_NAME,XA_STRING,8,
                     PropModeReplace,"POSTSCRIPT",10);

     XSelectInput(dpy, hard->w, ExposureMask);
     XMapWindow(dpy, hard->w);
     XNextEvent(dpy, &event);
     XSelectInput(dpy, hard->w, 0);
     return to;
}

/*
 * IsWindowHardware () should return TRUE if the hardware is a window, FALSE
 * otherwise.
 *
 * IsBitmapHardware () should return TRUE if the hardware is a bitmap,
 * FALSE otherwise.
 */

int IsWindowHardware (h) struct hardware *h;
```

```
{
     return h->flags & ISWIN;
}

#define IsWindowHardware(h) ((h)->flags & ISWIN)

#define IsBitmapHardware(h) ((h)->flags & ISBITMAP)

/*
 *
 * DestroyHardware () should release the resources required by the hardware,
 * bitmap or window.  This should cause a window device to vanish. NULL is not
 * an error (does nothing).
 */

void DestroyHardware (h)
struct hardware *h;
{
     if (h) {
          Hardware hard = (Hardware) h->hard.addr;
          if (IsWindowHardware(h))
               XDestroyWindow(dpy, hard->w);
          else
               if (IsBitmapHardware(h))
                    XFreePixmap(dpy, hard->w);
          Free(h);
     }
}

/*
 *
 * DeviceMatrix () should return a matrix appropriate to a device of the given
 * height and width.  For a typical display with a graphics origin at the top
 * left of a window, an appropriate definition would be:
 *
 * Matrix DeviceMatrix (width, height)
 * int width, height;
 * {
 *     return NewMatrix (PIXELS_PER_INCH / 72.0, 0.0, 0.0,
 *                       -PIXELS_PER_INCH / 72.0, 0.0, (float) height);
 * }
 */

Matrix DeviceMatrix (width, height) int width, height;
 {
        return NewMatrix (pixels_per_inch / 72.0, 0.0, 0.0,
                          -pixels_per_inch / 72.0, 0.0, (float) height);
 }

/*
 * HardwareExtent () returns a DevicePoint describing the width and height of
 * the argument.  NULL has extent NewDevicePoint (0, 0).
 */

DevicePoint HardwareExtent (h) struct hardware *h;
 {
        if (h)
                return h->extent;
        else
                return NewDevicePoint (0, 0);
 }

static NeedAux (h) struct hardware *h;
{
     DevicePoint p;

     if (h->aux)
          return;
     p = HardwareExtent(h);
     h->aux = NewBitmapHardware (p.dx, p.dy);
}

/*************************** OUTPUT PRIMITIVES ******************************/
/*
 *
 * BitBlt () is a full function RasterOp. The 'rop' argument will have values
 * as described in the header file hard.h. If the from argument is NULL it is
 * taken to be a bitmap full of ones the shape of the fromPoint and extent. If
 * the to argument is NULL, this is a no-op.
 *
 * Paint () is an addition to BitBlt. Bits that are set in the source are
 * Painted into the destination in the given colour with a copying rasterop so
 * that they replace pixels previously there. If the machine does not support
 * colour windows, half-toning should be performed.  Colour values have hue,
 * saturation and brightness components. On a black and white or greyscale
 * system the brightness value will be a FP value between 0.0 (black) and 1.0
 * (white), which can be used as a grey level.
 *
 * Paint is expected to mask with the clip mask. BitBlt is not,
 */

void BitBlt (from, to, fromPoint, toPoint, extent, rop)
struct hardware *from, *to;
DevicePoint toPoint, fromPoint, extent;
int rop;
{
     Hardware fromhard, tohard;
     static int count = 0;
     GC gc = NULL;
     XGCValues values;

     if (to == NULL) return;
     /*
      * Copying from full color window to monochrome bitmap is the
      * wrong idea.
      */
     if (IsWindowHardware(from) && !IsWindowHardware(to)) return;

     tohard = (Hardware) to->hard.addr;
     if (IsWindowHardware(to)) {
          if (from == NULL) {
               XFillRectangle(dpy, tohard->w, fillgc[rop], toPoint.dx,
                              toPoint.dy, extent.dx, extent.dy);
          } else {
               fromhard = (Hardware) from->hard.addr;
               if (IsWindowHardware(from)) {
                    XCopyArea(dpy, fromhard->w, tohard->w, fillgc[rop],
                              fromPoint.dx, fromPoint.dy,
                              extent.dx, extent.dy,
                              toPoint.dx, toPoint.dy);
               } else
               if (IsBitmapHardware (from)) {
                    XCopyPlane(dpy, fromhard->w, tohard->w, fillgc[15-rop],
                               fromPoint.dx, fromPoint.dy,
                               extent.dx, extent.dy,
                               toPoint.dx, toPoint.dy, 1);
               }
          }
```

```
	} else
	if (IsBitmapHardware(to)) {
		if (from == NULL) {
			XFillRectangle(dpy, tohard->w, BitmapGC[rop], toPoint.dx,
					toPoint.dy, extent.dx, extent.dy);
		} else {
			if (IsBitmapHardware(from)) {
				fromhard = (Hardware) from->hard.addr;
				XCopyArea(dpy, fromhard->w, tohard->w, BitmapGC[rop],
					fromPoint.dx, fromPoint.dy,
					extent.dx, extent.dy,
					toPoint.dx, toPoint.dy);
			}
		}
	}

	to->touched = TRUE;
	to->changed = TRUE;

	if (gc != NULL)
		XFreeGC(dpy, gc);
	if (count++ % 50 == 0) XSync(dpy, FALSE);
}

void Paint (from, to, fromPoint, toPoint, extent, colour)
struct hardware *from, *to;
DevicePoint fromPoint, toPoint, extent;
Colour colour;
{
	struct hardware *temp = NULL, *gray = NULL;
	Hardware hard, grayhard, tohard;
	int valuemask = 0, created = FALSE;
	XGCValues values;
	GC gc = NULL;
	unsigned long pix;
	ColorType ret;
	static int count = 0;
	Drawable root = RootWindow(dpy, SCREEN);

	if (!IsBitmapHardware(from) && from != NULL)
		Punt("Paint() called with non-bitmap source.\n");

	ret = RealColor(colour, &pix, &gray);

	if (to == NULL)
		return;

	if (!from->touched)
		return;

	if (IsBitmapHardware(to)) {
		root = ((Hardware) to->hard.addr)->w;
		if (ret == ISPIXEL)
			BitBlt(from, to, fromPoint, toPoint, extent,
				(pix == WhitePixel(dpy,SCREEN))?ROP_NOTAND:ROP_OR);
		else if (ret == ISHALFTONE) {

			printf("Paint calling BitBlt halftone !?\n");

			BitBlt(from, to, fromPoint, toPoint, extent, ROP_SOURCE);
		}
		return;
	}
	UpdateControl (to, FALSE);

	if (ret == ISPIXEL) {
		values.function = GXcopy;
		values.foreground = pix;
		valuemask = valuemask|GCFunction|GCForeground;
		if (pix == BlackPixel(dpy, SCREEN))
			gc = fillgc[GXcopy];
		else if (pix == WhitePixel(dpy, SCREEN))
			gc = whitegc;
	} else if (ret == ISHALFTONE) {
		values.function = GXcopy;
		values.foreground = BlackPixel(dpy, SCREEN);
		values.background = WhitePixel(dpy, SCREEN);
		grayhard = (Hardware) gray->hard.addr;
		values.fill_style = FillOpaqueStippled;
		values.stipple = grayhard->w;
		valuemask = valuemask|GCFunction|GCForeground|GCBackground|GCFillStyle|GCStippl
		gc = grayhard->gc;
	} else {
		UpdateControl (to, TRUE);
		return;
	}

	if (from) {
		if (to->clip) {
			temp = NewBitmapHardware(extent.dx, extent.dy);
			BitBlt(to->clip, temp, toPoint, NewDevicePoint(0, 0),
				extent, ROP_SOURCE);
			BitBlt(from, temp, fromPoint, NewDevicePoint(0, 0),
				extent, ROP_AND);

			hard = (Hardware) temp->hard.addr;
			values.clip_mask = hard->w;
			values.clip_x_origin = toPoint.dx;
			values.clip_y_origin = toPoint.dy;
			valuemask = valuemask|GCClipMask|GCClipXOrigin|GCClipYOrigin;
			gc = NULL;
		} else {
			temp = NewBitmapHardware(extent.dx, extent.dy);
			BitBlt(from, temp, fromPoint, NewDevicePoint(0, 0),
				extent, ROP_SOURCE);
			hard = (Hardware) temp->hard.addr;
			values.clip_mask = hard->w;
			values.clip_x_origin = toPoint.dx;
			values.clip_y_origin = toPoint.dy;
			valuemask = valuemask|GCClipMask|GCClipXOrigin|GCClipYOrigin;
			gc = NULL;
		}
	} else {
		if (to->clip) {
			hard = (Hardware) to->clip->hard.addr;
			values.clip_mask = hard->w;
			valuemask = valuemask|GCClipMask;
			gc = NULL;
		}
	}

	if (!gc) {
		created = TRUE;
		gc = XCreateGC(dpy, root, valuemask, &values);
	}
	tohard = (Hardware) to->hard.addr;
```

```
	XFillRectangle(dpy, tohard->w, gc, toPoint.dx, toPoint.dy,
			extent.dx, extent.dy);

	if (temp)
		DestroyHardware(temp);

	if (created)
		XFreeGC(dpy, gc);

	UpdateControl (to, TRUE);

	to->touched = TRUE;
	to->changed = TRUE;

	if (++count % 100 == 0) {
		XSync(dpy, FALSE);
		printf("%d Paints.\n", count);
	}
}

int FillPath(to, path, rule)
struct hardware *to;
Path path;
int rule;
{
	register Path p;
	static XPoint *vertices = NULL;
	static int counter = 0;
	int ret, ok = FALSE, i = 0, valuemask = 0;
	XGCValues values;
	GC gc;
	unsigned long pix;
	struct hardware *gray = NULL;
	Hardware hard, grayhard;

	if (!vertices) {
		if ((vertices = (XPoint *) malloc(MAXPATHELEMENTS * sizeof(XPoint)))
		    == NULL)
			Punt("Not enough memory for filling.\n");
	}

	if (to->clip)
		printf("Can't clip (yet).\n");

	if (EmptyPath (path))
		return TRUE;

	path = FlattenPath (path);

	if (!CloseAll (path)) {
		PathFree (path);

		return FALSE;
	}

	ret = RealColor(gstate->colour, &pix, &gray);

	if (ret == ISPIXEL) {
		values.function = GXcopy;
		values.foreground = pix;
		valuemask = valuemask|GCFunction|GCForeground;
		if (pix == BlackPixel(dpy, SCREEN))
			if (IsWindowHardware(to))
				gc = fillgc[GXcopy];
			else
				gc = BitmapGC[GXcopy];
		if (pix == WhitePixel(dpy, SCREEN))
			if (IsWindowHardware(to))
				gc = whitegc;
			else
				gc = BitmapGC[GXcopyInverted];
		hard = (Hardware) to->hard.addr;
		if (hard->last_used == ISPIXEL && hard->last_color.pix == pix)
			ok = TRUE;
	} else if (ret == ISHALFTONE) {
		values.function = GXcopy;
		values.foreground = BlackPixel(dpy, SCREEN);
		values.background = WhitePixel(dpy, SCREEN);
		values.fill_style = FillOpaqueStippled;
		grayhard = (Hardware) gray->hard.addr;
		values.stipple = grayhard->w;
		valuemask = valuemask|GCFunction|GCForeground|GCBackground|GCFillStyle|GCStippl
		gc = grayhard->gc;

		hard = (Hardware) to->hard.addr;
		if (hard->last_used == ISHALFTONE &&
		    hard->last_color.halftone == grayhard->w)
			ok = TRUE;

		if (!gc)
			printf("HalfTone GC not set!\n");

	} else return;

/*	if (to->clip) {
		hard = (Hardware) to->clip->hard.addr;
		values.clip_mask = hard->w;
		valuemask = valuemask|GCClipMask;

		if (ok)
			gc = ((Hardware) to->hard.addr)->gc;

		if (to->clip->changed) {
			to->clip->changed = FALSE;
			printf("Clip changed.\n");
			gc = NULL;
		}
	}
*/
	if (!gc) {
		gc = XCreateGC(dpy, RootWindow(dpy, SCREEN), valuemask, &values);
	}

	hard = (Hardware) to->hard.addr;
/*	hard->last_used = ret;
	hard->gc = gc;

	if (created)
		XFreeGC(dpy, gc);
*/

	UpdateControl (to, FALSE);

	for (p = path->next; p != path; p = p->next)
	switch (p->ptype) {
	case EMove:
		if (i > 1)
			XFillPolygon(dpy, hard->w, gc, vertices, i,
```

```
                    Complex, CoordModeOrigin);
          vertices[0].x = (short) p->pe.point.hx;
          vertices[0].y = (short) p->pe.point.hy;
          i = 1;
          break;
     case ELine:
          vertices[i].x = (short) p->pe.point.hx;
          vertices[i++].y = (short) p->pe.point.hy;
          break;
     case EClose:
          XFillPolygon(dpy, hard->w, gc, vertices, i,
                    Complex, CoordModeOrigin);
          i = 0;
          break;
     }

     UpdateControl (to, TRUE);

     PathFree (path);

     if (++counter % 100 == 0) {
          XSync(dpy, FALSE);
          printf("%d FillPaths.\n", counter);
     }

     to->touched = TRUE;
     to->changed = TRUE;

     return TRUE;
}

static short Xvalue (ax, ay, bx, by, cy) int ax, ay, bx, by, cy;
{
     if (ay - by == 0) {
          printf("Xvalue: ax=%d, ay=%d, bx=%d, by=%d, cy=%d\n",
                 ax, ay, bx, by, cy);
          return bx + (cy - by) * (ax - bx);
     }
     return bx + (cy - by) * (ax - bx) / (float) (ay - by);
}

/*
 * BitBltTrapezoid () and PaintTrapezoid () render a complete trapezoidal
 * shape.  The corners of the trapezoid may lie far outside the range of
 * interesting scan-lines, but the slope of the line should be clipped by the
 * top and bottom. The coordinates are half-open.
 */

void BitBltTrapezoid (to, lefttop, leftbottom, righttop, rightbottom,
                      top, bottom, rop)
struct hardware *to;
DevicePoint lefttop, leftbottom, righttop, rightbottom;
int top, bottom, rop;
{
     int i, j, temp;
     static int left [1024], right [1024];

     int
          ltx = Xvalue (lefttop.dx, lefttop.dy, leftbottom.dx,
                        leftbottom.dy, top),
          rtx = Xvalue (righttop.dx, righttop.dy, rightbottom.dx,
                         rightbottom.dy, top),
          lbx = Xvalue (lefttop.dx, lefttop.dy, leftbottom.dx,
                         leftbottom.dy, bottom),
          rbx = Xvalue (righttop.dx, righttop.dy, rightbottom.dx,
                         rightbottom.dy, bottom);

     if (ltx == lbx && rtx == rbx) {
          if (rtx < ltx)
               temp = rtx, rtx = ltx, ltx = temp;

          BitBlt ((struct hardware *) NULL, to,
                  NewDevicePoint (0, 0), NewDevicePoint (ltx, top),
                  NewDevicePoint (rtx - ltx + 1, bottom - top + 1), rop);

          return;
     }

     for (i = top, j = 0; i <= bottom; i++, j++) {
          int
               lx = Xvalue (lefttop.dx, lefttop.dy, leftbottom.dx,
                             leftbottom.dy, i),
               rx = Xvalue (righttop.dx, righttop.dy, rightbottom.dx,
                             rightbottom.dy, i);

          if (rx < lx)
               temp = rx, rx = lx, lx = temp;

          left [j] = lx; right [j] = rx;
     }
     BitBltBlob (to, top, bottom - top, left, right, single_rop [rop]);

     to->touched = TRUE;
     to->changed = TRUE;
}

/*void BitBltTrapezoid (to, lefttop, leftbottom, righttop, rightbottom,
                      top, bottom, rop)
struct hardware *to;
DevicePoint lefttop, leftbottom, righttop, rightbottom;
int top, bottom, rop;
{
     int i, j, temp;
     static int left [1024], right [1024];

     int
          ltx = Xvalue (lefttop.dx, lefttop.dy, leftbottom.dx,
                        leftbottom.dy, top),
          rtx = Xvalue (righttop.dx, righttop.dy, rightbottom.dx,
                         rightbottom.dy, top),
          lbx = Xvalue (lefttop.dx, lefttop.dy, leftbottom.dx,
                         leftbottom.dy, bottom),
          rbx = Xvalue (righttop.dx, righttop.dy, rightbottom.dx,
                         rightbottom.dy, bottom);

     if (ltx == lbx && rtx == rbx) {
          if (rtx < ltx)
               temp = rtx, rtx = ltx, ltx = temp;

          BitBlt ((struct hardware *) NULL, to,
                  NewDevicePoint (0, 0), NewDevicePoint (ltx, top),
                  NewDevicePoint (rtx - ltx + 1, bottom - top + 1), rop);

          return;
     }

     for (i = top, j = 0; i <= bottom; i++, j++) {
          int
```

```
               lx = Xvalue (lefttop.dx, lefttop.dy, leftbottom.dx,
                            leftbottom.dy, i),
               rx = Xvalue (righttop.dx, righttop.dy, rightbottom.dx,
                            rightbottom.dy, i);

          if (rx < lx)
               temp = rx, rx = lx, lx = temp;

          left [j] = lx; right [j] = rx;
     }
     BitBltBlob (to, top, bottom - top, left, right, single_rop [rop]);
}
*/

void PaintTrapezoid (to, lefttop, leftbottom, righttop, rightbottom,
                     top, bottom, colour)
struct hardware *to;
DevicePoint lefttop, leftbottom, righttop, rightbottom;
int top, bottom;
Colour colour;
{
     struct hardware *gray = NULL;
     Hardware hard, grayhard, tohard;
     int valuemask = 0, ok = FALSE;
     XGCValues values;
     GC gc = NULL;
     unsigned long pix;
     ColorType ret;
     XPoint coord[4];
     static int counter = 0;

     ret = RealColor(colour, &pix, &gray);

     /* This should be done better: Halftones aren't handled... */
     if (IsBitmapHardware(to)) {
          BitBltTrapezoid(to, lefttop, leftbottom, righttop, rightbottom,
                          top, bottom, (pix == WhitePixel(dpy, SCREEN)) ? ROP_FALSE : ROP
          return;
     }

     if (to == NULL)
          return;

     coord[0].x = Xvalue (lefttop.dx, lefttop.dy, leftbottom.dx,
                          leftbottom.dy, bottom);
     coord[0].y = bottom;
     coord[1].x = Xvalue (lefttop.dx, lefttop.dy, leftbottom.dx,
                          leftbottom.dy, top);
     coord[1].y = top;
     coord[2].x = Xvalue (righttop.dx, righttop.dy, rightbottom.dx,
                          rightbottom.dy, top);
     coord[2].y = top;
     coord[3].x = Xvalue (righttop.dx, righttop.dy, rightbottom.dx,
                          rightbottom.dy, bottom);
     coord[3].y = bottom;

     UpdateControl (to, FALSE);

     if (ret == ISPIXEL) {
          values.function = GXcopy;
          values.foreground = pix;
          valuemask = valuemask|GCFunction|GCForeground;
          if (pix == BlackPixel(dpy, SCREEN))
               gc = fillgc[GXcopy];
          if (pix == WhitePixel(dpy, SCREEN))
               gc = whitegc;
          hard = (Hardware) to->hard.addr;
          if (hard->last_used == ISPIXEL && hard->last_color.pix == pix)
               ok = TRUE;
     } else if (ret == ISHALFTONE) {
          values.function = GXcopy;
          values.foreground = BlackPixel(dpy, SCREEN);
          values.background = WhitePixel(dpy, SCREEN);
          values.fill_style = FillOpaqueStippled;
          grayhard = (Hardware) gray->hard.addr;
          values.stipple = grayhard->w;
          valuemask = valuemask|GCFunction|GCForeground|GCBackground|GCFillStyle|GCStippl
          gc = grayhard->gc;

          hard = (Hardware) to->hard.addr;
          if (hard->last_used == ISHALFTONE &&
              hard->last_color.halftone == grayhard->w)
               ok = TRUE;

          if (!gc)
               printf("HalfTone GC not set!\n");

     } else return;

     if (to->clip) {
          hard = (Hardware) to->clip->hard.addr;
          values.clip_mask = hard->w;
          valuemask = valuemask|GCClipMask;

          if (ok)
               gc = ((Hardware) to->hard.addr)->gc;

/*        if (to->clip->changed) {
               to->clip->changed = FALSE;
               printf("Clip changed.\n");
               gc = NULL;
          }
*/
          gc = NULL;
     }

     if (!gc) {
          gc = XCreateGC(dpy, RootWindow(dpy, SCREEN), valuemask, &values);
     }
     tohard = (Hardware) to->hard.addr;

     XFillPolygon(dpy, tohard->w, gc, coord, 4, Complex, CoordModeOrigin);

     hard = (Hardware) to->hard.addr;
     hard->last_used = ret;
     hard->gc = gc;

/*     if (created)
          XFreeGC(dpy, gc);
          */

     UpdateControl (to, TRUE);

     if (++counter % 100 == 0) {
          XSync(dpy, FALSE);
          printf("%d Trapezoids.\n", counter);
     }
```

```
	to->touched = TRUE;
	to->changed = TRUE;
}

/*
void PaintTrapezoid (to, lefttop, leftbottom, righttop, rightbottom,
					top, bottom, colour)
struct hardware *to;
DevicePoint lefttop, leftbottom, righttop, rightbottom;
int top, bottom;
Colour colour;
{
	int col;

	if (to == NULL)
		return;

	col = IsWindowHardware (to) ? HardColour (colour) : 0;

	UpdateControl (to, FALSE);

	if (to->clip)
		NeedAux (to);
	if (col == 0)
		BitBltTrapezoid (to, lefttop, leftbottom, righttop, rightbottom,
					top, bottom, ROP_TRUE);
	else if (col == TransferSize () - 1)
		BitBltTrapezoid (to, lefttop, leftbottom, righttop, rightbottom,
					top, bottom, ROP_FALSE);
	else {
		DevicePoint origin, extent;
		struct hardware *gray;

		int left = lefttop.dx < leftbottom.dx ? lefttop.dx : leftbottom.dx,
		right = righttop.dx > rightbottom.dx ? righttop.dx : rightbottom.dx;

		if (righttop.dx < left)
			left = righttop.dx;
		if (rightbottom.dx < left)
			left = rightbottom.dx;
		if (lefttop.dx > right)
			right = lefttop.dx;
		if (leftbottom.dx > right)
			right = leftbottom.dx;
		origin = NewDevicePoint (left, top);
		extent = NewDevicePoint (right - left, bottom - top);

		gray = GraySync (col);
		NeedAux (to);
		BitBlt ((struct hardware *) NULL, to->aux, origin, origin, extent,
			ROP_FALSE);
		BitBltTrapezoid (to->aux, lefttop, leftbottom, righttop, rightbottom,
					top, bottom, ROP_TRUE);
		if (to->clip)
			BitBlt (to->clip, to->aux, origin, origin, extent, ROP_AND);
		BitBlt (to->aux, to, origin, origin, extent, ROP_NOTAND);
		RasterTile (gray, to->aux, origin, extent, ROP_AND);
		BitBlt (to->aux, to, origin, origin, extent, ROP_OR);
	}
	UpdateControl (to, TRUE);
}
*/

/*
 *
 *	BitBltLine () is expected to draw a line between the given points
 *	with the given RasterOp and colour masking.
 *	The line should be one pixel wide and half-open.
 *	[Thicker lines are done with BitBlt.]
 *
 *	PaintLine () is expected to Paint a line by analogy with Paint
 *	and BitBlt.
 */

void BitBltLine (h, fromPoint, toPoint, rop)
struct hardware *h;
DevicePoint fromPoint, toPoint;
int rop;
{
    if (h) {
	Hardware hard = (Hardware) h->hard.addr;
	if (IsWindowHardware(h)) {
		XDrawLine(dpy, hard->w, fillgc[rop], fromPoint.dx, fromPoint.dy,
			toPoint.dx, toPoint.dy);
	} else
	if (IsBitmapHardware(h)) {
		XDrawLine(dpy, hard->w, BitmapGC[rop], fromPoint.dx, fromPoint.dy,
			toPoint.dx, toPoint.dy);
	}

	h->touched = TRUE;
	h->changed = TRUE;
   }
}

void PaintLine (h, fromPoint, toPoint, colour)
struct hardware *h;
DevicePoint fromPoint, toPoint;
Colour colour;
{
	int col;

	if (h == NULL)
		return;

	col = IsWindowHardware (h) ? HardColour (colour) : 0;

	UpdateControl (h, FALSE);

	if (h->clip == NULL && (col == 0 || col == TransferSize () - 1))
		BitBltLine (h, fromPoint, toPoint,
			(col == 0 ? ROP_TRUE : ROP_FALSE));
	else {
		int mx = Min (fromPoint.dx, toPoint.dx), my = Min (fromPoint.dy,
								toPoint.dy),
		Mx = Max (fromPoint.dx, toPoint.dx), My = Max (fromPoint.dy,
								toPoint.dy);
		DevicePoint orig, ex;

		orig = NewDevicePoint (mx, my);
		ex = NewDevicePoint (Mx - mx + 1, My - my + 1);

		NeedAux (h);
		BitBlt ((struct hardware *) NULL, h->aux, orig, orig, ex, ROP_FALSE);

		BitBltLine (h->aux, fromPoint, toPoint, ROP_TRUE);

		BitBlt (h->clip, h->aux, orig, orig, ex, ROP_AND);
```

```
        BitBlt (h->aux, h, orig, orig, ex, ROP_NOTAND);

        if (col != 0 && col != TransferSize () - 1) {
             struct hardware *gray = GraySync (col);
             RasterTile (gray, h->aux, orig, ex, ROP_AND);
        }
        BitBlt (h->aux, h, orig, orig, ex, ROP_OR);
    }
    UpdateControl (h, TRUE);

    h->touched = TRUE;
    h->changed = TRUE;
}

/*
 * BitBltBlob () takes a set of pixel coordinates and fills the trapezon
 * figure half open.
 */

void BitBltBlob (to, top, height, left, right, rop)
struct hardware *to;
int top, height, *left, *right, rop;
{
    int i;
    DevicePoint p1, p2;
    for (i=0 ; i<height ; i++) {
         p1 = NewDevicePoint (left[i], top + i);
         p2 = NewDevicePoint (right[i], top + i);
         BitBltLine(to, p1, p2, rop);
    }
}

/*
 * RasterTile () replicates the whole of ‘‘from’’ over ‘‘to’’, but clipped by
 * the rectangle bounded by ‘‘toPoint’’ and ‘‘extent’’.
 */
void RasterTile (from, to, toPoint, extent, rop)
struct hardware *from, *to;
DevicePoint toPoint, extent;
int rop;
{
    Hardware fromhard, tohard;
    static GC wgc = NULL, bgc = NULL;
    XGCValues values;
    int valuemask;
    if (to == NULL) return;
    if (from == NULL || IsWindowHardware(from))
        Punt("Can only RasterTile from Bitmap.");

    fromhard = (Hardware) from->hard.addr;
    tohard = (Hardware) to->hard.addr;
    values.tile = fromhard->w;
    values.fill_style = FillTiled;
    values.function = rop;
    valuemask = GCFunction | GCTile | GCFillStyle;
    if (IsWindowHardware (to)) {
         if (wgc == NULL) {
              wgc = XCreateGC(dpy, RootWindow(dpy, SCREEN),
                              valuemask, &values);
         } else {
              XChangeGC(dpy, wgc, valuemask, &values);
         }
         XFillRectangle(dpy, tohard->w, wgc, toPoint.dx, toPoint.dy,
                        extent.dx, extent.dy);
    } else
    if (IsBitmapHardware(to)){
         if (bgc == NULL) {
              bgc = XCreateGC(dpy, tohard->w, valuemask, &values);
         } else {
              XChangeGC(dpy, bgc, valuemask, &values);
         }
         XFillRectangle(dpy, tohard->w, bgc, toPoint.dx, toPoint.dy,
                        extent.dx, extent.dy);
    }
    to->touched = TRUE;
    to->changed = TRUE;
}

/******************* BRIGHTNESS TRANSFER FUNCTION ***********************/
/*
 *
 * TransferSize () and SetTransfer () control the mapping function between
 * user brightnesses and device brightnesses. The interface is expected to
 * perform this mapping of brightnesses to a sufficient resolution.
 * SetTransfer takes a table of floating point numbers between 0 and 1. User
 * brightnesses are scaled to the size of this table and mapped through it.
 * The argument table given to SetTransfer () will be deleted after use.
 * TransferSize () simply enquires the required size of the table.
 *
 * It may be appropriate to half-tone on a grayscale or colour device to
 * improve rendering if it is not too expensive. TransferSize () returns the
 * size of the pattern table.
 */

GC GetGC (color, src, dest)
Colour color;
struct hardware *src, *dest;
{
     int bright, ret;
     Hardware hard, grayhard, cliphard;
     struct hardware *halftone;
     unsigned long pixelvalue = BlackPixel(dpy, SCREEN);
     Pixmap gray;
     XGCValues values;
     int valuemask = 0;
     GC gc;
     Drawable root = RootWindow(dpy, SCREEN);

     /* Assumes dest is either a bitmap or window. */

     /* Handles only monochrome for now.  Fix for color later. */

     /*
      * This routine should take a color, a destination, and a source
      * hardware.  Simplest case:
      * Solid color, No clip, No from.
      *
      * Worst case:
      * Halftone, to clip, from non-null.
      */

     if (NOP_GC == NULL) {
          values.function = GXnoop;
          NOP_GC = XCreateGC(dpy, RootWindow(dpy, SCREEN),
                             GCFunction, &values);
     }

     bright = transfer[(int)((TRANSFER_SIZE - 1) * color.brightness + .5)];
```

```
	if (bright == 0) {
		pixelvalue = BlackPixel(dpy, SCREEN);
		ret = ISPIXEL;
	} else if (bright == TRANSFER_SIZE - 1) {
		pixelvalue = WhitePixel(dpy, SCREEN);
		if (!whitegc) {
			values.function = GXcopy;
			values.foreground = WhitePixel(dpy, SCREEN);
			values.background = BlackPixel(dpy, SCREEN);
			whitegc = XCreateGC(dpy, RootWindow(dpy, SCREEN),
					GCFunction|GCForeground|GCBackground,
					&values);
		}
		ret = ISPIXEL;
	} else {
		halftone = GraySync(bright);
		grayhard = (Hardware) halftone->hard.addr;
		ret = ISHALFTONE;
	}

	if (dest == NULL)
		return NOP_GC;

	if (src && !src->touched)
		return NOP_GC;

	if (IsBitmapHardware(dest)) {
		root = ((Hardware) dest->hard.addr)->w;
		values.function = GXcopy;
		values.foreground = 1;
		values.background = 0;
		valuemask = GCFunction|GCForeground|GCBackground;
		gc = BitmapGC[GXcopy];
		if (src) {
			if (IsWindowHardware(src))
				return NOP_GC;
			if (dest->clip) {
				if (ret == ISHALFTONE) {
					temp = NewBitmapHardware(extent.dx, extent.dy);
					BitBlt(to->clip, temp, toPoint, NewDevicePoint(0, 0),
						extent, ROP_SOURCE);
					BitBlt(from, temp, fromPoint, NewDevicePoint(0, 0),
						extent, ROP_AND);

					hard = (Hardware) temp->hard.addr;
					values.clip_mask = hard->w;
					values.clip_x_origin = toPoint.dx;
					values.clip_y_origin = toPoint.dy;
					valuemask = valuemask|GCClipMask|GCClipXOrigin|GCClipYOrigin;
					gc = NULL;
				} else {
					/* clip = clip, Stippled = src */
					cliphard = (Hardware) to->clip->hard.addr;
					values.clip_mask = cliphard->w;
				}
			} else {
				if (IsBitmapHardware(src)) {
					hard = (Hardware) src->hard.addr;
					values.fill_style = FillStippled;
					values.stipple = hard->w;
					valuemask = valuemask|GCFillStyle|GCStipple;
					gc = NULL;
				}
			}
		} else {
			if (dest->clip) {
				/* clip=clip */
				cliphard = (Hardware) to->clip->hard.addr;
				values.clip_mask = cliphard->w;
				valuemask = valuemask|GCClipMask;
				gc = NULL;
			} else {
				if (ret == ISHALFTONE) {
					values.fill_style = FillTiled;
					values.tile = grayhard->w;
					valuemask = valuemask|GCFillStyle|GCStipple;
					gc = NULL;
				}
			}
		}
	} else {
		root = RootWindow (dpy, SCREEN);
		values.function = GXcopy;
		values.foreground = pixelvalue;
		values.background = WhitePixel (dpy, SCREEN);
		gc = fillgc[GXcopy];
		if (src) {
			if (dest->clip) {
				if (ret == ISHALFTONE) {

				} else {
					/* clip = clip, OpaqueStippled = src */

				}
			} else {
				if (IsBitmapHardware(src)) {
					hard = (Hardware) src->hard.addr;
					values.fill_style = FillStippled;
					values.stipple = hard->w;
					valuemask = valuemask|GCFillStyle|GCStipple;
					gc = hard->gc;
				} else {
					hard = (Hardware) src->hard.addr;
					values.fill_style = FillTiled;
					values.tile = hard->w;
					valuemask = valuemask|GCFillStyle|GCTile;
					gc = NULL;
				}
			}
		} else {
			if (dest->clip) {
				/* clip=clip */
			} else {
				if (ret == ISHALFTONE) {
					values.fill_style = FillOpaqueStippled;
					values.stipple = grayhard->w;
					valuemask = valuemask|GCFillStyle|GCStipple;
					gc = grayhard->gc;
				}
			}
		}
	}

/*
	if (!gc) {
		created = TRUE;
		gc = XCreateGC(dpy, root, valuemask, &values);
```

```
    }
*/

    return gc;
}

ColorType RealColor (color, pixelvalue, halftone)
Colour color;
unsigned long *pixelvalue;
struct hardware **halftone;
{
    int bright;
    XGCValues values;

    bright = transfer[(int)((TRANSFER_SIZE - 1) * color.brightness + .5)];

    if (bright == 0) {
        *pixelvalue = BlackPixel(dpy, SCREEN);
        return ISPIXEL;
    } else if (bright == TRANSFER_SIZE - 1) {
        *pixelvalue = WhitePixel(dpy, SCREEN);
        if (!whitegc) {
            values.function = GXcopy;
            values.foreground = WhitePixel(dpy, SCREEN);
            values.background = BlackPixel(dpy, SCREEN);
            whitegc = XCreateGC(dpy, RootWindow(dpy, SCREEN),
                                GCFunction|GCForeground|GCBackground,
                                &values);
        }
        return ISPIXEL;
    }

    *halftone = GraySync(bright);
    return ISHALFTONE;
}

struct hardware *GraySync (col) int col;
{
    col = col * (float) screen_size / TRANSFER_SIZE + 0.5;

    return screen[col].shade;
}

int HardColour (colour) Colour colour;
{
    /* DCJ:
     * This is a hack.  This is only a hack.  If this had been real
     * code, you would have been notified of it through email.
     */
    if (BlackPixel(dpy, SCREEN) == 0)
        return (TRANSFER_SIZE - 1 - transfer [(int) ((TRANSFER_SIZE - 1) * colour.brigh
    else
        return transfer [(int) ((TRANSFER_SIZE - 1) * colour.brightness + .5)];
}

int TransferSize ()
{
    return TRANSFER_SIZE;
}

#define TransferSize() TRANSFER_SIZE

void SetTransfer (tran) float *tran;
{
    int i;

    for (i = 0; i < TRANSFER_SIZE; i++)
        transfer [i] = (TRANSFER_SIZE - 1) * tran[i] + .5;
}

/********************** BITMAP CONVERSION ********************************/
/*
 *
 * StringFromHardware () produces a string from its argument which describes
 * the bitmap.  The bitmap is returned in row-major order with the leftmost
 * bit of each byte in the most significant position. Rows are padded to byte
 * boundaries. Only single plane bitmaps are used.
 *
 * HardwareFromString () performs the inverse mapping, generating a bitmap
 * from a set of bits, given a width and height. Only single plane bitmaps are
 * used.
 */

/*char *StringFromHardware (h)
struct hardware *h;
{
    XImage *image;
    Hardware hard;
    char *result, *ptr, c;
    int x, y, i, hdx, hdy;
    if (h == NULL) return NULL;
    hard = (Hardware) h->hard.addr;
    hdx = h->extent.dx;
    hdy = h->extent.dy;
    image = XGetImage(dpy, hard->w, 0, 0, hdx, hdy, AllPlanes, ZPixmap);
    result = Malloc(((hdx + 7) / 8) * hdy);
    ptr = result;
    for (y=0 ; y<hdy ; y++) {
        for (x=0 ; x<hdx ; x+=8) {
            c = 0;
            for (i=0 ; i<8 ; i++) {
                c = c << 1;
                if (x+i < hdx)
                    c |= XGetPixel(image, x+i, y);
            }
        }
        *ptr++ = c;
    }
    Free((char *) image);
    return (char *) result;
}

struct hardware *HardwareFromString (s, width, height)
char *s;
int width, height;
{
    struct hardware *h = NewBitmapHardware(width, height);
    Hardware hard = (Hardware) h->hard.addr;
    XImage *image;
    if (s == NULL) Punt("HardwareFromString called with NULL string!");
    image = XCreateImage(dpy, DefaultVisual(dpy, SCREEN),
                         DefaultDepth(dpy, SCREEN), ZPixmap, 0, s,
                         h->extent.dx, h->extent.dy, 8, 0);
    image->bitmap_bit_order = MSBFirst;
    XPutImage(dpy, hard->w, fillgc[GXcopy], image, 0, 0, 0, 0,
              h->extent.dx, h->extent.dy);
    free((char *) image);
    return h;
```

```
}
*/
char *StringFromHardware (h)
struct hardware *h;
{
    XImage *image;
    Hardware hard;
    char *result, *ptr, c;
    int x, y, i, hdx, hdy;

    printf("StringFromHardware called.\n");

    if (h == NULL || !IsBitmapHardware(h)) return NULL;

    printf("and is getting the image.\n");

    hard = (Hardware) h->hard.addr;
    hdx = h->extent.dx;
    hdy = h->extent.dy;
    image = XGetImage(dpy, hard->w, 0, 0, hdx, hdy, AllPlanes, XYPixmap);
    result = Malloc(((hdx + 7) / 8) * hdy);
    ptr = result;
    for (y=0 ; y<hdy ; y++) {
        for (x=0 ; x<hdx ; x+=8) {
            c = 0;
            for (i=0 ; i<8 ; i++) {
                c = c << 1;
                if (x+i < hdx)
                    c |= XGetPixel(image, x+i, y);
            }
        }
        *ptr++ = c;
    }
    Free((char *) image);
    return (char *) result;
}

struct hardware *HardwareFromString (s, width, height)
char *s;
int width, height;
{
    struct hardware *h = NewBitmapHardware(width, height);
    Hardware hard = (Hardware) h->hard.addr;
    XImage *image;

    printf("HardwareFromString called.\n");

    if (s == NULL) Punt("HardwareFromString called with NULL string!");
    image = XCreateImage(dpy, DefaultVisual(dpy, SCREEN),
                         1, XYBitmap, 0, s,
                         h->extent.dx, h->extent.dy, 8, 0);
    image->bitmap_bit_order = MSBFirst;
    XPutImage(dpy, hard->w, BitmapGC[GXcopy], image, 0, 0, 0, 0,
              h->extent.dx, h->extent.dy);
    free((char *) image);
    h->touched = TRUE;
    h->changed = TRUE;
    return h;
}

/************************** HALF-TONE SCREEN *******************************/
/*
 * ScreenSize () allows PostScript to determine how large an array of sample
 * points to expect.  It should return the length of the side of the sample
 * square.
 *
 * BuildScreen () returns a set of sampling coordinates to PostScript to hand
 * to the users spot-function
 *
 * SetScreen () allows PostScript to set the thresholds for each sample point
 * so that half-tone bitmaps can be made.
 */

static int FreqSize (freq) float freq;
{
     int i = pixels_per_inch / freq + 0.5;

     if (i < 2)
          return 2;
     return i;
}

int ScreenSize (freq, rot) float freq, rot;
{
     int size = FreqSize (freq);

     return size * size;
}

void BuildScreen (freq, rotation, x, y) float freq, rotation, *x, *y;
{
     int size = FreqSize (freq);
     int i, j;

     for (i = 0; i < size; i++)
          for (j = 0; j < size; j++)
               *x++ = (2 * i - size + 1) / (float) size,
               *y++ = (2 * j - size + 1) / (float) size;
}

static sgn (a) float a;
{
     if (a == 0)
          return 0;
     else if (a < 0)
          return -1;
     else
          return 1;
}

static int screen_cmp (a, b) char *a, *b;
{
     struct screen *aa = (struct screen *) a, *bb = (struct screen *) b;

     return sgn (aa->val - bb->val);
}

void SetScreen (freq, rotation, thresh) float freq, rotation, *thresh;
{
     struct hardware *temp;
     int i, j, size = FreqSize (freq);
     struct screen *p;
     int valuemask;
     XGCValues values;
     Hardware hard;

     if (screen) {
          for (i = 0; i < screen_size; i++)
```

```c
			DestroyHardware (screen [i].shade);
		free ((char *) screen);
	}
	p = screen = (struct screen *) Malloc ((unsigned) (((screen_size = size * size) + 1)
/*	screen_side = size;*/
	for (i = 0; i < size; i++)
		for (j = 0; j < size; j++) {
			p->val = *thresh++;
			p->sx = i;
			p->sy = j;
			++p;
		}
	qsort ((char *) screen, screen_size, sizeof (struct screen), screen_cmp);
	temp = NewBitmapHardware (size, size);
	BitBlt ((struct hardware *) NULL, temp, NewDevicePoint (0, 0),
		NewDevicePoint (0, 0), NewDevicePoint (size, size), ROP_TRUE);

	for (i = 0; i < screen_size; i++) {
		screen [i].shade = NewBitmapHardware (size, size);
		BitBlt (temp, screen[i].shade,
			NewDevicePoint (0, 0), NewDevicePoint (0, 0),
			NewDevicePoint (size, size), ROP_SOURCE);
		BitBlt ((struct hardware *) NULL, temp,
			NewDevicePoint (0, 0),
			NewDevicePoint (screen[i].sx, screen[i].sy),
			NewDevicePoint (1, 1), ROP_FALSE);
		values.function = GXcopy;
		values.foreground = BlackPixel(dpy, SCREEN);
		values.background = WhitePixel(dpy, SCREEN);
		values.fill_style = FillOpaqueStippled;
		hard = (Hardware) screen[i].shade->hard.addr;
		values.stipple = hard->w;
		valuemask = GCFunction|GCForeground|GCBackground|GCFillStyle|GCStipple;
		hard->gc = XCreateGC(dpy, RootWindow(dpy, SCREEN), valuemask,
					&values);
		screen[i].shade->touched = TRUE;
		screen[i].shade->changed = TRUE;
	}
	screen[screen_size].shade = temp;
	values.function = GXcopy;
	values.foreground = BlackPixel(dpy, SCREEN);
	values.background = WhitePixel(dpy, SCREEN);
	values.fill_style = FillOpaqueStippled;
	hard = (Hardware) temp->hard.addr;
	values.stipple = hard->w;
	valuemask = GCFunction|GCForeground|GCBackground|GCFillStyle|GCStipple;
	hard->gc = XCreateGC(dpy, RootWindow(dpy, SCREEN), valuemask,
				&values);
	screen[screen_size].shade->touched = TRUE;
	screen[screen_size].shade->changed = TRUE;
}

/************************* CLIPPING ******************************************/
/*
 *
 * SetClipHardware sets hardware which is a clip mask for BitBlt. This mask
 * should be ANDed with any output operation. If clip is NULL, masking will
 * not be needed.
 */

void SetClipHardware (h, clip) struct hardware *h, *clip;
{
	if (h) {
		h->clip = clip;
		if (h->clip)
			h->clip->changed = TRUE;
	}
}

/************************ UPDATE CONTROLS ***********************************/
/*
 * HardUpdate is a hook to allow devices which do output buffering to flush
 * that buffering when appropriate.  This allows an interactive user to see
 * completed graphics between prompts (it is called as a side-effect of the
 * PostScript flush operator). Typically is is a no-op.
 */

void HardUpdate ()
{
    XFlush(dpy, 0);
}

/*
 * This call can be used to enable batching of output operations.
 * UpdateControl (h, FALSE) means ‘‘start of batching’’ UpdateControl (h,
 * TRUE) means ‘‘end of batching’’. It is used to improve performance on
 * machines where screen updates have a high locking overhead. It may be a
 * no-op.  The operation should nest if batching is already in progress: FALSE
 * increments a counter, TRUE decrements a counter. Display changes are
 * allowed when the counter is non-zero.
 */

void UpdateControl (h, on)
struct hardware *h;
int on;
{}

#define UpdateControl (h, on) {}

DevicePoint ConvertToDevicePoint (x, y) int x, y;
 {
	DevicePoint res;

	res.dx = (int) ((float) x * (float) pixels_per_inch / 72.0);
	res.dy = (int) ((float) y * (float) pixels_per_inch / 72.0);

	return res;
 }
```