The GNU `libxmi` 2-D Rasterization Library

Version 2.0

Robert S. Maier

Short Contents

Table of Contents

1 The libxmi 2-D Rasterization Library

This is the documentation for version 2.0 of the GNU libxmi 2-D rasterization library, which is used by C and C++ programmers. It converts 2-D geometrical objects, such as polylines, polygons, and arcs, to raster patterns. There is support for setting drawing attributes, including line width, join style, cap style, and a multicolored dash pattern. The objects may be unfilled or filled. If the latter, the filling may be a solid color, a stipple pattern, or a texture. There is support for sophisticated color-merging between separately drawn objects.

1.1 An overview of libxmi

With the aid of the GNU libxmi library, a C or C++ programmer may rasterize two-dimensional geometric objects; that is, draw them on a two-dimensional array of pixels. The supported objects are points, polylines, filled polylines (i.e., polygons), rectangles, filled rectangles, and 'arcs': segments of ellipses whose principal axes are aligned with the coordinate axes. Like polylines and rectangles, arcs may be unfilled or filled.

The corresponding rendering functions in the libxmi API (application programming interface) are miDrawPoints, miDrawLines, miFillPolygon, miDrawRectangles, miFillRectangles, miDrawArcs, and miFillArcs. Each of these takes an array, rather than a single object, as an argument. For example, one of the arguments of miDrawLines is an array of points, interpreted as the vertices of a polyline. The polygon filled by miFillPolygon is specified similarly. And the final four functions render lists of objects, rather than single objects.

Actually, libxmi provides a two-stage graphics pipeline. In the first stage, an opaque object called a miPaintedSet is drawn onto. Each of the eight drawing functions takes a pointer to a miPaintedSet as its first argument. Conceptually, a miPaintedSet is an unordered set of points with integer coordinates, partitioned by pixel value. The datatype representing a pixel value is miPixel, which is normally typedef'd as unsigned int. Each of the drawing functions takes a pointer to a miGC, or graphics context, as its second argument. The miGC specifies such drawing attributes as line width, join style, cap style, and dashing style, and also the pixel value(s) to be used in the painting operation.

In the first stage of the pipeline the Painter's Algorithm is used, so that a repeatedly-painted point in a miPaintedSet acquires the pixel value applied to it in the final drawing operation. In the second stage, more sophisticated pixel-merging algorithms may be applied. In this stage, a miPaintedSet is copied ('merged') onto a miCanvas, by invoking miCopyPaintedSetToCanvas. A miCanvas is a structure that includes a miPixmap: a two-dimensional array of miPixels that may be initialized by the user, and read, pixel by pixel, after the merging is performed. By default, miCopyPaintedSetToCanvas uses the Painter's Algorithm too, so that the source pixel in the miPaintedSet replaces the destination pixel in the miCanvas. But the merging process may be controlled in much finer detail. A stipple bitmap and a texture pixmap, and binary and ternary pixel-merging functions, may be specified.

The *interpretation* of pixel values is left up to the user. A miPixel could be an index into a color table. It could also be an encoding of a color according to the RGB scheme, the RGBA scheme, or some other scheme. Even though a miPixel is normally an unsigned int, this may be altered, if desired, at the time libxmi is installed. Any scalar type or nonscalar type, including a structure or a union, could be substituted.

libxmi is intended for use both as a standalone library and as a rendering module that may be incorporated in other packages. To facilitate its use in other packages, it may be extensively customized at installation time. Besides customizing the definition of miPixel, it is possible to customize the definition of the miPixmap datatype, which by default is an array of pointers to rows of miPixels. The default merging algorithm used by miCopyPaintedSetToCanvas may

also be altered: it need not be the Painter's Algorithm. For instructions on customization, see the comments in the libxmi header file 'xmi.h'.

1.2 A sample libxmi program

The following C program uses libxmi to create a miPaintedSet, draws a dashed polyline and a dashed elliptic arc on the miPaintedSet, and transfers the painted pixels to a miCanvas that includes a pixmap of specified size. Finally, it writes the pixmap to standard output.

```
#include <stdio.h>
#include <stdlib.h>
#include <xmi.h>            /* public libxmi header file */

int main ()
{
  miPoint points[4];        /* 3 line segments in the polyline */
  miArc arc;                /* 1 arc to be drawn */
  miPixel pixels[4];        /* pixel values for drawing and dashing */
  unsigned int dashes[2];   /* length of 'on' and 'off' dashes */
  miGC *pGC;                /* graphics context */
  miPaintedSet *paintedSet; /* opaque object to be painted */
  miCanvas *canvas;         /* drawing canvas (including pixmap) */
  miPoint offset;           /* for miPaintedSet -> miCanvas transfer */
  int i, j;

  /* define polyline: vertices are (25,5) (5,5), (5,25), (35,22) */
  points[0].x = 25;  points[0].y = 5;
  points[1].x = 5;   points[1].y = 5;
  points[2].x = 5;   points[2].y = 25;
  points[3].x = 35;  points[3].y = 22;

  /* define elliptic arc */
  arc.x = 20; arc.y = 15;   /* upper left corner of bounding box */
  arc.width = 30;           /* x range of box: 20..50 */
  arc.height = 16;          /* y range of box: 15..31 */
  arc.angle1 = 0 * 64;      /* starting angle (1/64'ths of a degree) */
  arc.angle2 = 270 * 64;    /* angle range (1/64'ths of a degree) */

  /* create and modify graphics context */
  pixels[0] = 1;            /* pixel value for 'off' dashes, if drawn */
  pixels[1] = 2;            /* default pixel for drawing */
  pixels[2] = 3;            /* another pixel, for multicolored dashes */
  pixels[3] = 4;            /* another pixel, for multicolored dashes */
  dashes[0] = 4;            /* length of 'on' dashes */
  dashes[1] = 2;            /* length of 'off' dashes */
  pGC = miNewGC (4, pixels);
  miSetGCAttrib (pGC, MI_GC_LINE_STYLE, MI_LINE_ON_OFF_DASH);
  miSetGCDashes (pGC, 2, dashes, 0);
  miSetGCAttrib (pGC, MI_GC_LINE_WIDTH, 0); /* Bresenham algorithm */

  /* create empty painted set */
  paintedSet = miNewPaintedSet ();
```

```
  /* paint dashed polyline and dashed arc onto painted set */
  miDrawLines (paintedSet, pGC, MI_COORD_MODE_ORIGIN, 4, points);
  miDrawArcs (paintedSet, pGC, 1, &arc);

  /* create 60x35 canvas (initPixel=0); merge painted set onto it */
  canvas = miNewCanvas (60, 35, 0);
  offset.x = 0; offset.y = 0;
  miCopyPaintedSetToCanvas (paintedSet, canvas, offset);

  /* write canvas's pixmap (a 60x35 array of miPixels) to stdout */
  for (j = 0; j < canvas->drawable->height; j++)
    {
      for (i = 0; i < canvas->drawable->width; i++)
        /* note: column index precedes row index */
        printf ("%d", canvas->drawable->pixmap[j][i]);
      printf ("\n");
    }

  /* clean up */
  miDeleteCanvas (canvas);
  miDeleteGC (pGC);
  miClearPaintedSet (paintedSet); /* not necessary */
  miDeletePaintedSet (paintedSet);

  return 0;
}
```

This program illustrates how `miPaintedSet`, `miGC`, and `miCanvas` objects are created and destroyed, as well as manipulated. Each of these types has a constructor and a destructor, named `miNew...` and `miDelete...`, respectively.

When creating a `miGC` with `miNewGC`, an array of `miPixel`s of length at least 2 must be passed as the second argument. The first argument, `npixels`, is the length of the array. The default color for drawing operations will be pixel number 1 in the array. The other pixel colors in the array will only be used when dashing. In normal (on/off) dashing, the colors of the 'on' dashes will cycle through the colors numbered 1,2,...,npixels-1 in the array. In so-called double dashing, the 'off' dashes will be drawn too, in color number 0.

In the program, the first call to `miGCSetAttrib` sets the line mode to `MI_LINE_ON_OFF_DASH` rather than `MI_DOUBLE_DASH`. This replaces the default, which is `MI_LINE_SOLID`, meaning no dashing (only color number 1 in the pixel array is used). An array of dash lengths is then specified by calling `miSetGCDashes`. (The default dash length array, which is replaced, is `{4,4}`). When dashing, the specified dash length array will be cyclically used. The first dash will be 'on', the second 'off', and so forth. The third argument to `miSetGCDashes` specifies an initial offset into this cyclic pattern. Currently, the offset must be nonnegative.

The second call to `miGCSetAttrib` sets the line width in the graphics context. If the specified line width is positive, lines and arcs will be drawn with a circular brush whose diameter is equal to the line width. All pixels within the brushed region will be painted. If the line width is zero, as it is here, a so-called Bresenham algorithm will be used. Bresenham lines and arcs are drawn with fewer pixels than is the case for lines and arcs with width 1, and many people prefer them.

`miDrawLines` and `miDrawArc` do the actual drawing. They are passed a polyline (i.e., an array of `miPoint`s) and a `miArc`, respectively. The definitions of the `miPoint` and `miArc` structures appear in the header file '`xmi.h`', which is worth examining. The third argument of `miDrawLines`,

MI_COORD_MODE_ORIGIN, specifies that the points of the polyline, after the first, are expressed in absolute rather than relative coordinates.

Finally, the program transfers the painted pixels to a miCanvas, and copies the pixels from it to standard output. A miCanvas, unlike a miPaintedSet and a miGC, is not an opaque object, so its elements may be read (and written). In fact, a miCanvas may be constructed by hand and passed to the miCopyPaintedSetToCanvas function. However, it is usually easiest to use the constructor miNewCanvas and the destructor miDeleteCanvas. Any miCanvas created with miNewCanvas is allocated on the heap, with malloc. It includes a pixmap (an array of miPixels, of specified size) that is itself allocated on the heap.

It is only when the painted pixels are transferred from miPaintedSet to miCanvas that clipping to a pixmap takes place. Drawing to a miPaintedSet is entirely unclipped: at least in principle, the miPaintedSet is of potentially infinite extent. However, the pixmap in the miCanvas created by miNewCanvas (60, 35, 0) has upper left corner (0,0) and lower right corner (59,34). Out-of-bound painted pixels, if any, will not be transferred.

The third argument of miNewCanvas is its initPixel argument: the pixel value to which each miPixel in the pixmap is initialized. Since this value is '0', the pixmap that is sent to standard output will display the dashed polyline and arc in the '2', '3', and '4' colors, on a background of zeroes.

1.3 The libxmi API

1.3.1 Opaque data structures

The drawing functions used in the first stage of the libxmi graphics pipeline paint pixels on a miPaintedSet. A miPaintedSet should be thought of as an unordered set of points with integer coordinates, partitioned according to pixel value. Any pixel value is a miPixel, which in most libxmi installations is typedef'd as an unsigned int.

A miPaintedSet is an opaque object that must be accessed through a pointer. The functions

```
miPaintedSet * miNewPaintedSet (void);
void miDeletePaintedSet (miPaintedSet *paintedSet);
```

are the constructor and destructor for the miPaintedSet type. The function

```
void miClearPaintedSet (miPaintedSet *paintedSet);
```

clears all pixels from a miPaintedSet, i.e., makes it the empty set.

All drawing functions that paint pixels on a miPaintedSet take a pointer to a graphics context as an argument. A graphics context is an opaque object, called a miGC, that contains drawing parameters. The functions

```
miGC * miNewGC (int npixels, const miPixel *pixels);
void miDeleteGC (miGC *pGC);
miGC * miCopyGC (const miGC *pGC);
```

are the constructor, destructor, and copy constructor for the miGC type. The arguments of miNewGC specify an array of miPixels, which must have length at least 2. The default color for drawing operations will be pixel number 1 in the array. The other pixel colors in the array will only be used when dashing. In normal (on/off) dashing, the colors of the 'on' dashes will cycle through the colors numbered 1,2,. . . ,npixels-1. In so-called double dashing, the 'off' dashes will be drawn too, in color number 0.

The array of pixel colors may be modified at any later time, by invoking the function miSetGCPixels.

```
void miSetGCPixels (miGC *pGC, int npixels, const miPixel *pixels);
```

is the declaration of this function.

The lengths of dashes, when dashing, may be set by invoking miSetGCDashes. It has declaration

```
void miSetGCDashes (miGC *pGC, int ndashes, const unsigned int *dashes,
                    int offset);
```

Here dashes is an array of length ndashes. The array is cyclically used. The first dash in a polyline or arc will be an 'on' dash of length dashes[0]; the next dash will be an 'off' dash of length dashes[1]; and so forth. The default dash length array in any miGC is {4,4}. The dash length array need not have the same length as the pixel color array, and it need not have even length. offset, if nonzero, is an initial offset into the dash pattern. For example, if it equals dashes[0] then the first dash will be an 'off' dash of length dashes[1]. Currently, offset must be nonnegative.

Besides the array of pixel colors and the array of dash lengths, any miGC contains several attributes whose values are enums, and an integer-valued line width attribute. Any one of these may be set by invoking miSetGCAttrib, and any list of them by invoking miSetGCAttribs. The declarations of these functions are:

```
typedef enum { MI_GC_FILL_RULE, MI_GC_JOIN_STYLE, MI_GC_CAP_STYLE,
               MI_GC_LINE_STYLE, MI_GC_ARC_MODE, MI_GC_LINE_WIDTH
             } miGCAttribute;
void miSetGCAttrib (miGC *pGC, miGCAttribute attribute, int value);
void miSetGCAttribs (miGC *pGC, int nattributes,
                     const miGCAttribute *attributes,
                     const int *values);
```

These attributes are similar to the drawing attributes used in the X Window System. The allowed values for the attribute MI_GC_FILL_RULE, which specifies the rule used when filling polygons and arcs, are:

```
enum { MI_EVEN_ODD_RULE, MI_WINDING_RULE };
```

The default is MI_EVEN_ODD_RULE. The allowed values for the attribute MI_GC_JOIN_STYLE, which specifies the rule used when joining polylines and polyarcs, are:

```
enum { MI_JOIN_MITER, MI_JOIN_ROUND, MI_JOIN_BEVEL,
       MI_JOIN_TRIANGULAR };
```

The default is MI_JOIN_MITER. The allowed values for the attribute MI_GC_CAP_STYLE, which specifies the rule used when capping the ends of polylines and arcs, are:

```
enum { MI_CAP_NOT_LAST, MI_CAP_BUTT, MI_CAP_ROUND, MI_CAP_PROJECTING,
       MI_CAP_TRIANGULAR };
```

The default is MI_CAP_BUTT. The allowed values for the attribute MI_GC_LINE_STYLE, which specifies whether or not dashing should take place when drawing polylines and arcs, are:

```
enum { MI_LINE_SOLID, MI_LINE_ON_OFF_DASH, MI_LINE_DOUBLE_DASH };
```

The default is MI_LINE_SOLID. The allowed values for the attribute MI_GC_ARC_MODE, which specifies how arcs should be filled, are:

```
enum { MI_ARC_CHORD, MI_ARC_PIE_SLICE };
```

The default is MI_ARC_PIE_SLICE.

Finally, the value for the line width, i.e., for the MI_GC_LINE_WIDTH attribute, may be any nonnegative integer. The default is 0, which has a special meaning. Zero-width lines and arcs are not invisible. Instead, they are drawn with a Bresenham algorithm, which paints fewer pixels than is the case for lines with width 1.

Any miGC also contains a miter-limit attribute, which, if the join mode attribute has value MI_JOIN_MITER and the line width is at least 1, will affect the drawing of the joins in polylines and polyarcs. At any join point, the 'miter length' is the distance between the inner corner

and the outer corner. The miter limit is the maximum value that can be tolerated for the miter length divided by the line width. If this value is exceeded, the miter will be 'cut off': the MI_JOIN_BEVEL join mode will be used instead.

The function

```
void miSetGCMiterLimit (miGC *pGC, double miter_limit);
```

sets the value of this attribute. The specified value must be greater than or equal to 1.0. That is because the miter limit is the cosecant of one-half of the minimum join angle for mitering, so values less than 1.0 are meaningless. The default value for the miter limit is 10.43, as in the X Window System. 10.43 is the cosecant of 5.5 degrees, so by default, miters will be cut off if the join angle is less than 11 degrees.

1.3.2 The first stage of the graphics pipeline

In the first stage of the libxmi graphics pipeline, one or more of the core drawing functions is invoked. Each drawing function takes pointers to a miPaintedSet and a miGC (a graphics context) as its first and second arguments. It will paint pixels in the miPaintedSet, according to the drawing parameters in the graphics context.

The drawing functions fall into three groups: (1) functions that draw points, polylines, and polygons, (2) functions that draw rectangles, and (3) functions that draw 'arcs' (segments of ellipses whose principal axes are aligned with the coordinate axes). We discuss these three groups in turn.

The point/polyline/polygon group includes:

```
void miDrawPoints (miPaintedSet *paintedSet, const miGC *pGC,
                   miCoordMode mode, int npts, const miPoint *pPts);
void miDrawLines (miPaintedSet *paintedSet, const miGC *pGC,
                  miCoordMode mode, int npts, const miPoint *pPts);
void miFillPolygon (miPaintedSet *paintedSet, const miGC *pGC,
                    miPolygonShape shape,
                    miCoordMode mode, int npts, const miPoint *pPts);
```

The final three arguments of each are a coordinate mode, a specified number of points, and an array that contains that number of points. miDrawPoints draws the array as a cloud of points, miDrawLines draws a polyline defined by the array, and miFillPolygon fills a polygon defined by the array. The mode argument specifies whether the points in the array, after the first, are in absolute or relative coordinates. Its possible values are:

```
typedef enum { MI_COORD_MODE_ORIGIN,
               MI_COORD_MODE_PREVIOUS } miCoordMode;
```

The miPoint structure is defined by

```
typedef struct
{
  int x, y; /* integer coordinates, y goes downward */
} miPoint;
```

The additional shape argument of miFillPolygon is advisory. Its possible values are:

```
typedef enum { MI_SHAPE_GENERAL, MI_SHAPE_CONVEX } miPolygonShape;
```

They indicate whether the polygon is (1) unconstrained (i.e., not necessarily convex, with self-intersections allowed), or (2) convex and not self-intersecting. The latter case can be drawn more rapidly.

The rectangle group includes

```
void miDrawRectangles (miPaintedSet *paintedSet, const miGC *pGC,
                       int nrects, const miRectangle *pRects);
```

```
void miFillRectangles (miPaintedSet *paintedSet, const miGC *pGC,
                       int nrects, const miRectangle *pRects);
```

The final two arguments of each are a specified number of rectangles and an array that contains that number of rectangles. `miDrawRectangles` draws the outline of each rectangle, and `miFillRectangle` fills each rectangle. The `miRectangle` structure is defined by

```
typedef struct
{
  int x, y;                   /* upper left corner of rectangle */
  unsigned int width, height; /* dimensions: width>=1, height>=1 */
} miRectangle;
```

The rectangle group is redundant, since a rectangle is a special sort of polyline, defined by a five-point point array in which the last point is the same as the first.

The arc group includes

```
void miDrawArcs (miPaintedSet *paintedSet, const miGC *pGC,
                 int narcs, const miArc *parcs);
void miFillArcs (miPaintedSet *paintedSet, const miGC *pGC,
                 int narcs, const miArc *parcs);
```

The final two arguments of each are a specified number of arcs and an array that contains that number of arcs. `miDrawArcs` draws each arc. It will join successive arcs, if they are contiguous, according to the 'join mode' in the graphics context. Similarly, `miFillArcs` will fill each arc according to the 'arc mode' in the graphics context. Either a pie slice or a chord will be filled.

The `miArc` structure is defined by

```
typedef struct
{
  int x, y;           /* upper left corner of ellipse's bounding box */
  unsigned int width, height; /* dimensions: width>=1, height>=1 */
  int angle1, angle2; /* initial angle, angle range (in 1/64 degrees) */
} miArc;
```

`x`, `y`, `width`, `height` specify a rectangle aligned with the coordinate axes, and `angle1`, `angle2` specify an angular range (a 'pie slice') of an ellipse inscribed in the rectangle. By convention, they are the starting polar angle and angle range of the circular arc that would be produced from the elliptic arc by squeezing the rectangle into a square.

`miDrawArcs` maintains a cache of rasterized ellipses. This cache is persistent, and internal to `libxmi`; accordingly, `miDrawArcs` is not reentrant. For applications in which reentrancy is important, a reentrant counterpart is provided. It is

```
void miDrawArcs_r (miPaintedSet *paintedSet, const miGC *pGC,
                   int narcs, const miArc *parcs,
                   miEllipseCache *ellipseCache);
```

The caller of `miDrawArcs_r` must supply a pointer to a `miEllipseCache` object as the final argument. A pointer to such an object, which is opaque, is returned by `miNewEllipseCache`. After zero or more calls to `miDrawArcs_r`, the object would be deleted by a call to `miDeleteEllipseCache`. The declarations

```
miEllipseCache * miNewEllipseCache (void);
void miDeleteEllipseCache (miEllipseCache *ellipseCache);
```

are supplied in the header file '`xmi.h`'.

1.3.3 The second stage of the graphics pipeline

In the second state of the graphics pipeline, the pixels in a `miPaintedSet` are transferred ('merged') to a `miCanvas`, by invoking `miCopyPaintedSetToCanvas`. It is only when the painted pixels are transferred to a `miCanvas` that clipping to a pixmap takes place.

A `miCanvas` is a structure that includes a pixmap and several parameters that control the transfer of pixels. Since it is not opaque, it may be constructed and modified by hand, if necessary. The `miCanvas` type has definition

```
typedef struct
{
  miCanvasPixmap *drawable;   /* the pixmap */

  miBitmap *stipple;          /* a mask, if non-NULL */
  miPoint stippleOrigin;      /* placement of upper left corner */

  miPixmap *texture;          /* a texture, if non-NULL */
  miPoint textureOrigin;      /* placement of upper left corner */

  miPixelMerge2 pixelMerge2;  /* binary merging function, if non-NULL */
  miPixelMerge3 pixelMerge3;  /* ternary counterpart, if non-NULL */
} miCanvas;
```

Here, the `miBitmap` and `miPixmap` types are defined by

```
typedef struct
{
  int **bitmap;               /* each element is 0 or 1 */
  unsigned int width;
  unsigned int height;
}
miBitmap;

typedef struct
{
  miPixel **pixmap;           /* each element is a miPixel */
  unsigned int width;
  unsigned int height;
}
miPixmap;
```

That is, each of them contains an array of pointers to rows (of integers or pixel values, as the case may be). In most installations of `libxmi`, `miCanvasPixmap` is typedef'd as `miPixmap`. The typedefs

```
typedef miPixel (*miPixelMerge2) (miPixel source, miPixel destination);
typedef miPixel (*miPixelMerge3) (miPixel texture, miPixel source,
                                  miPixel destination);
```

define the datatypes of the binary and ternary pixel-merging function members.

The functions

```
miCanvas * miNewCanvas (unsigned int width, unsigned int height,
                        miPixel initPixel);
void miDeleteCanvas (miCanvas *pCanvas);
miCanvas * miCopyCanvas (const miCanvas *pCanvas);
```

are the constructor, destructor, and copy constructor for the `miCanvas` type. Rather than a `miCanvas` being created by hand, `miNewCanvas` is usually used instead. The `initPixel` argu-

ment is a miPixel value with which the newly allocated pixmap should be filled. The stipple and texture pointers in a newly created miCanvas are NULL, as are the pixel-merging function members. The four convenience functions

```
void miSetCanvasStipple (miCanvas *pCanvas, const miBitmap *pStipple,
                         miPoint stippleOrigin);
void miSetCanvasTexture (miCanvas *pCanvas, const miPixmap *pTexture,
                         miPoint textureOrigin);
void miSetPixelMerge2 (miCanvas *pCanvas, miPixelMerge2 pixelMerge2);
void miSetPixelMerge3 (miCanvas *pCanvas, miPixelMerge3 pixelMerge3);
```

may be used to set these members.

The miCopyPaintedSetToCanvas function, which implements the second stage of the graphics pipeline, can now be discussed. It has declaration

```
void miCopyPaintedSetToCanvas (const miPaintedSet *paintedSet,
                               miCanvas *canvas, miPoint origin);
```

where origin is the point on the miCanvas to which the point (0,0) in the miPaintedSet is mapped. (It could equally well be called offset.)

The semantics of miCopyPaintedSet boil down to a single issue: how a 'source' pixel in a miPaintedSet is merged onto the corresponding 'destination' pixel in a miCanvas to form a new pixel. The simplest case is when no texture is specified (the corresponding pointer in the miCanvas is NULL). In that case, if the binary pixel-merging function member of the miCanvas is NULL, a default merging algorithm will be used. In most libxmi installations this is the Painter's Algorithm: the new pixel in the miCanvas will simply be the source pixel. If, on the other hand, the binary pixel-merging function in the miCanvas is non-NULL, it will be used to compute the new pixel.

A texture pixmap may be specified. If so, it will be extended periodically so as to cover the miCanvas, and its value at the location of the destination pixel will affect the merging process. If the ternary pixel-merging function member of the miCanvas is NULL, a default merging algorithm, appropriate to the case when a texture is present, will be used. In most libxmi installations this is a variant of the Painter's Algorithm: the new pixel in the miCanvas will be the texture pixel, rather than the source pixel. If, on the other hand, the ternary pixel-merging function in the miCanvas is non-NULL, it will be used to compute the new pixel.

Any miCanvas may also include a pointer to a stipple bitmap. If so, it will be extended periodically so as to cover the miCanvas, and its value at the location of the destination pixel will determine whether or not its replacement by a new pixel, according to one of the preceding rules, will take place. If the stipple value is zero, it will not; otherwise it will. Stipple bitmaps are useful for creating so-called screen door patterns, and more generally for protecting, or masking off, part of a miCanvas.

Acknowledgements

`libxmi` is based on the machine-independent 2-D graphics routines in the X11 sample server code. Those routines fill polygons and draw wide polygonal lines and arcs. They were written by Brian Kelleher, Joel McCormack, Todd Newman, Keith Packard, Robert Scheifler and Ken Whaley, who worked for Digital Equipment Corp., MIT, and/or the X Consortium, and are copyright © 1985–89 by the X Consortium.

In 1998–99, Robert Maier extracted the graphics routines from the sample server code in the X11R6 distribution, converted them to ANSI C, and added extensive comments. He also introduced data structures appropriate for a two-stage graphics pipeline, and converted the graphics routines to use them. He added some new rendering features, such as support for multicolored dashing.

The modified code was first released by being incorporated in version 2.2 of the GNU `plotutils` package, as a rendering module for export of PNM and pseudo-GIF files. See the `plotutils` home page. After further modifications, the code was released as the standalone `libxmi` library.