Xr: Cross-device Rendering for Vector Graphics

Carl Worth
USC, Information Sciences Institute
cworth@isi.edu

Keith Packard
Cambridge Research Laboratory, HP Labs, HP
keithp@keithp.com

Abstract

Xr provides a vector-based rendering API with output support for the X Window System and local image buffers. PostScript and PDF file output is planned. Xr is designed to produce identical output on all output media while taking advantage of display hardware acceleration through the X Render Extension.

Xr provides a stateful user-level API with support for the PDF 1.4 imaging model. Xr provides operations including stroking and filling Bézier cubic splines, transforming and compositing translucent images, and antialiased text rendering. The PostScript drawing model has been adapted for use within C applications. Extensions needed to support much of the PDF 1.4 imaging operations have been included. This integration of the familiar PostScript operational model within the native application language environment provides a simple and powerful new tool for graphics application development.

1 Introduction

The design of the Xr library is motivated by the desire to provide a high-quality rendering interface for all areas of application presentation, from labels and shading on buttons to the central image manipulation in a drawing or painting program. Xr targets displays, printers and local image buffers with a uniform rendering model so that applications can use the same API to present information regardless of the media.

The Xr library provides a device-independent API, and can currently drive X Window System[10] applications as well as manipulate images in the application address space. It can take advantage of the X Render Extension[7] where available but does not require it. The intent is to add support for Xr to produce PostScript[1] and PDF 1.4[5] output.

Moving from the primitive original graphics system available in the X Window System to a complete device-independent rendering environment should serve to drive future application development in exciting directions.

1.1 Vector Graphics

On modern display hardware, an application's desire to present information using abstract geometric objects must be translated to physical pixels at some point in the process. The later this transition occurs in the rendering process the fewer pixelization artifacts will appear as a result of additional transformation operations on pixel-based data.

Existing application artwork is often generated in pixel format because the rendering operations available to the application at runtime are a mere shadow of those provided in a typical image manipulation program. Providing sufficient rendering functionality within the application environment allows artwork to be provided in vector form which presents high quality results at a wide range of sizes.

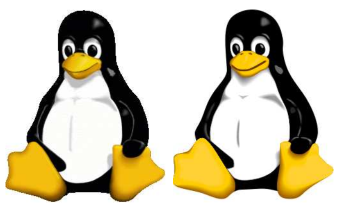

Figure 1: Raster and vector images at original size (artwork courtesy of Larry Ewing and Simon Budig)

Figures 1-3 illustrate the benefits of vector artwork. The penguin on the left of Figure 1 is the familiar image as originally drawn by Larry Ewing[3]. The penguin on the right is an Xr rendering of vector-based artwork by Simon Budig[2] intended to match Ewing's artwork as closely as possible. At the original scale of the raster artwork, the two images are quite comparable.

However, when the images are scaled up, the differences between raster and vector artwork become apparent. Figure 2 shows a portion of the original raster image scaled by a factor of 4 with the GIMP [6]. Artifacts from the scaling are apparent, primarily in the jaggies around the contour of the image. The GIMP did apply an interpolating filter to reduce these artifacts but this comes at the cost of blurring the image. Compare this to Figure 3 where Xr has been used to draw the vector artwork at 4 times the original scale. Since the vector artwork is resolution independent, the artifacts of jaggies and blurring are not present in this image.

1.2 Vector Rendering Model

The two-dimensional graphics world is fortunate to have one dominant rendering model. With the introduction of desktop publishing and the PostScript printer, application developers converged on that model. Recent extensions to that model have been incorporated in PDF 1.4, but the basic architecture remains the same. PostScript provides a simple painters model; each rendering operation places new paint on top of the contents of the surface. PDF 1.4 extends this model to include Porter/Duff image compositing [9] and other image manipulation operations which serve to bring the basic PostScript rendering model in line with modern application demands.

Figure 2: Raster image scaled 400%

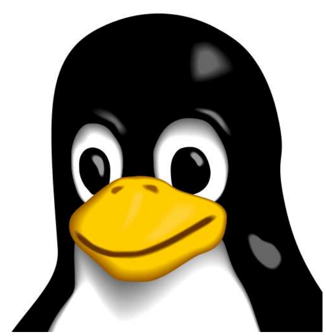

Figure 3: Vector image scaled 400%

PostScript and PDF draw geometric shapes by constructing arbitrary paths of lines and cubic Bézier splines. The coordinates used for the construction can be transformed with an affine matrix. This provides a powerful compositing technique as the transformation may be set before a complex object is drawn to position and scale it appropriately. Text is treated as prebuilt path sections which couples it tightly and cleanly with the rest of the model.

1.3 Xr Programming Interface

While the goal of the Xr library is to provide a PDF 1.4 imaging model, PDF doesn't provide any programming language interface. Xr borrows its imperative immediate mode model from PostScript operators. However, instead of proposing a complete new programming language to encapsulate these operators, Xr uses C functions for the operations and expects the developer to use C instead of PostScript to implement the application part of the rendering system. This dramatically reduces the number of operations needed by the library as only those directly involved in graphics need be provided. The large number of PostScript operators that support a complete language are more than adequately replaced by the C programming language.

PostScript encapsulates rendering state in a global opaque object and provides simple operators to change various aspects of that state, from color to line width and dash patterns. Because global objects can cause various problems in C library interfaces, the graphics state in Xr is held in a structure that is passed to each of the library functions.

The translation from PostScript operators to the Xr interface is straightforward. For example, the lineto operator translates to the XrLineTo function. The coordinates of the line endpoint needed by the operator are preceded by the graphics state object in the Xr interface.

2 API and Examples

This section provides a tour of the application programming interface (API) provided by Xr. Major features of the API are demonstrated in illustrations, and the source code for each illustration is provided in Appendix A.

2.1 Xr Initialization

```
#include <Xr.h>

#define WIDTH 600
#define HEIGHT 600
#define STRIDE (WIDTH * 4)

char image[STRIDE*HEIGHT];

int
main (void)
{
    XrState *xrs;

    xrs = XrCreate ();

    XrSetTargetImage (xrs, image,
                      XrFormatARGB32,
                      WIDTH, HEIGHT, STRIDE);

    /* draw things using xrs ... */

    XrDestroy (xrs);

    /* do something useful with image
       (eg. write to a file) */

    return 0;
}
```

Figure 4: Minimal program using Xr

Figure 4 shows a minimal program using Xr. This program does not actually do useful work—it never draws anything, but it demonstrates the initialization and cleanup procedures required for using Xr.

After including the Xr header file, the first Xr function a program must call is XrCreate. This function returns a pointer to an XrState object, which is used by Xr to store its data. The XrState pointer is passed as the first argument to almost all other Xr functions.

Before any drawing functions may be called, Xr must be provided with a target surface to receive the resulting graphics. The backend of Xr has support for multiple types of graphics targets. Currently, Xr has support for rendering to in-memory images as well as to any X Window System "drawable", (eg. a window or a pixmap).

The program calls XrSetTargetImage to direct graphics to an array of bytes arranged as 4-byte ARGB pixels. A similar call, XrSetTargetDrawable, is available to direct graphics to an X drawable.

When the program is done using Xr, it signifies this by calling XrDestroy. During XrDestroy, all data is released from the XrState object. It is then invalid for the program to use the value of the XrState pointer until a new object is created by calling XrCreate. The results of any graphics operations are still available on the target surface, and the program can access that surface as appropriate, (eg. write the image to a file, display the graphics on the screen, etc.).

2.2 Transformations

All coordinates passed from user code to Xr are in a coordinate system known as "user space". These coordinates are then transformed to "device space" which corresponds to the device grid of the target surface. This transformation is controlled by the current transformation matrix (CTM) within Xr.

The initial CTM is established such that one user unit maps to an integer number of device pixels as close as possible to 3780 user units per meter (~96 DPI) of physical device. This approach attempts to balance the competing desires of having a predictable real-world interpretation for user units and having the ability to draw elements on exact device pixel boundaries. Ideally, device pixels would be so small that the user could ignore pixel boundaries, but with current display pixel sizes of about 100 DPI, the pixel boundaries are still significant.

The CTM can be modified by the user to position, scale, or rotate subsequent objects to be drawn. These operations are performed by the functions XrTranslate, XrScale, and XrRotate. Additionally, XrConcatMatrix will compose a given matrix into the current CTM and XrSetMatrix will directly set the CTM to a given matrix. The XrDefaultMatrix function can be used to restore the CTM to its original state.

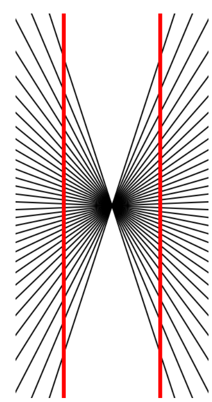

Figure 5: Hering illusion (originally discovered by Ewald Hering in 1861)[11]. The radial lines were positioned with XrTranslate and XrRotate

In Figure 5, each of the radial lines was drawn using identical path coordinates. The different angles were achieved by calling XrRotate before drawing each line. The source code for this image is in Figure 11.

2.3 Save/Restore of Graphics State

Programs using a structured approach to drawing will modify graphics state parameters in a hierarchical fashion. For example, while traversing a tree of objects to be drawn a program may modify the CTM, current color, line width, etc. at each level of the hierarchy.

Xr supports this hierarchical approach to graphics by maintaining a stack of graphics state objects within the XrState object. The XrSave function pushes a copy of the current graphics state onto the top of the stack. Modifications to the graphics state are made only to the object on the top of the stack. The XrRestore function pops a graphics state object off of the stack, restoring all graphics parameters to their state before the last XrSave operation.

This model has proven effective within structured C programs. Most drawing functions can be written with the following style, wrapping the body of the function with calls to XrSave and XrRestore:

```
void
draw_something (XrState *xrs)
{
    XrSave (xrs);
    /* draw something here */
    XrRestore (xrs);
}
```

This approach has the benefit that modifications to the graphics state within the function will not be visible outside the function, leading to more readily reusable code. Sometimes a single function will contain multiple sections of code framed by XrSave/XrRestore calls. Some find it more readable to include a new indented block between the XrSave/XrRestore calls in this case. Figure 12 contains an example of this style.

2.4 Path Construction

One of the primary elements of the Xr graphics state is the current path. A path consists of one or more independent subpaths, each of which is an ordered set of straight or curved segments. Any non-empty path has a “current point”, the final coordinate in the final segment of the current subpath. Path construction functions may read and update the current point.

Xr provides several functions for constructing paths. XrNewPath installs an empty path, discarding any previously defined path. The first path construction called after XrNewPath should be XrMoveTo which simply moves the current point to the point specified. It is also valid to call XrMoveTo when the current path is non-empty in order to begin a new subpath.

XrLineTo adds a straight line segment to the current path, from the current point to the point specified. XrCurveTo adds a cubic Bézier spline with a control polygon defined by the current point as well as the three points specified.

XrClosePath closes the current subpath. This operation involves adding a straight line segment from the current point to the initial point of the current subpath, (ie. the point specified by the most recent call to XrMoveTo). Calling XrClosePath is not equivalent to adding the corresponding line segment with XrLineTo. The distinction is that a closed subpath will have a join at the junction of the final coincident point while an unclosed path will have caps on either end of the path, (even if the two ends happen to be coincident). See Section 2.5 for more discussion of caps and joins.

It is often convenient to specify path coordinates as relative offsets from the current point rather than as absolute coordinates. To allow this, Xr provides XrRelMoveTo, XrRelLineTo, and XrRelCurveTo. Figure 6 shows a rendering of a path constructed with one call to XrMoveTo and four calls to XrRelLineTo in a loop. The source code for this figure can be seen in Figure 13.

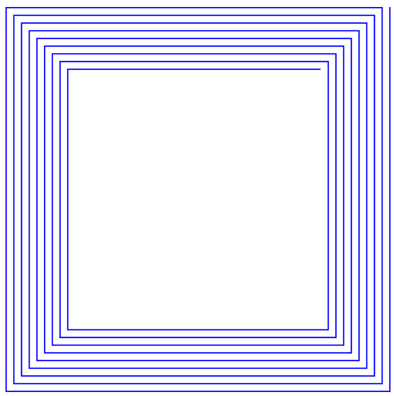

Figure 6: Nested box illusion (after a figure by Al Seckel[11]). Constructed with XrMoveTo and XrRelLineTo

As rectangular paths are commonly used, Xr provides a convenience function for adding a rectangular subpath to the current path. A call to `XrRectangle(xrs, x, y, width, height)` is equivalent to the following sequence of calls:

```
XrMoveTo    (xrs,      x, y);
XrRelLineTo (xrs,  width, 0);
XrRelLineTo (xrs, 0, height);
XrRelLineTo (xrs, -width, 0);
XrClosePath (xrs);
```

After a path is constructed, it can be drawn in one of two ways: stroking its outline (XrStroke) or filling its interior (XrFill).

2.5 Path Stroking

XrStroke draws the outline formed by stroking the path with a pen that in user space is circular with a radius of the current line width, (as set by XrSetLineWidth). The specification of the XrStroke operator is based on the convolution of polygonal tracings as set forth by Guibas, Ramshaw and Stolfi [4]. Convolution lends itself to efficient implementation as the outline of the stroke can be computed within an arbitrarily small error bound by simply using piece-wise linear approximations of the path and the pen.

As subsequent segments within a subpath are drawn, they are connected according to one of three different join styles, (bevel, miter, or round), as set by XrSetLineJoin. Closed subpaths are also joined at the closure point. Unclosed subpaths have one of three different cap styles, (butt, square, or round), applied at either end of the path. The cap style is set with the XrSetLineCap function.

Figure 7 demonstrates the three possible cap and join styles. The source code for this figure (Figure 12) demonstrates the use of XrSetLineJoin and XrSetLineCap as well as XrTranslate, XrSave, and XrRestore.

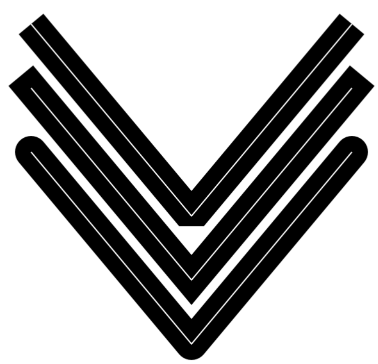

Figure 7: Demonstration of cap and join styles

2.6 Path Filling

XrFill fills the area on the “inside” of the current path. Xr can apply either the winding rule or the even-odd rule to determine the meaning of “inside”. This behavior is controlled by calling XrSetFillRule with a value of either XrFillRuleWinding or XrFillRuleEvenOdd.

Figure 8 demonstrates the effect of the fill rule given a star-shaped path. With the winding rule the entire star is filled in, while with the even-odd rule the center of the star is considered outside the path and is not filled. Figure 15 contains the source code for this example.

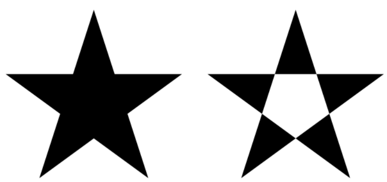

Figure 8: Demonstration of the effect of the fill rule

2.7 Controlling Accuracy

The graphics rendering of Xr is carefully implemented to allow all rendering approximations to be performed within a user-specified error tolerance, (within the limits of machine arithmetic of course). The XrSetTolerance function allows the user to specify a maximum error in units of device pixels.

The tolerance value has a strong impact on the quality of rendered splines. Empirical testing with modern displays reveals that errors larger than 0.1 device pixels are observable. The default tolerance value in Xr is therefore 0.1 device pixels.

The user can increase the tolerance value to tradeoff rendering accuracy for performance. Figure 9 displays the same curved path rendered several times with increasing tolerance values. Figure 14 contains the source code for this figure.

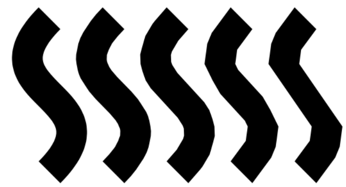

Figure 9: Splines drawn with tolerance values of .1, .5, 1, 5, and 10

2.8 Paint

The example renderings shown so far have all used opaque “paint” as the source color for all drawing operations. The color of this paint can be controlled with the XrSetRGBColor function.

Xr supports more interesting possibilities for the paint used in graphics operations. First, the source color need not be opaque; the XrSetAlpha function establishes an opacity level for the source paint. The alpha value ranges from 0 (transparent) to 1 (opaque).

When Xr graphics operations combine translucent surfaces, there are a number of different ways in which the source and destination colors can be combined. Xr provides support for all of the Porter/Duff compositing operators as well as the extended operators defined in the X Render Extension. The desired operator is selected by calling XrSetOperator before compositing. The default operator value is XrOperatorOver corresponding to the Porter/Duff OVER operator.

Finally, the XrSetPattern function allows any XrSurface to be installed as a static or repeating pattern to be used as the "paint" for subsequent graphics operations. The pattern surface may have been initialized from an external image source or may have been the result of previous Xr graphics operations.

Figure 10 was created by first drawing small, vertical black and white rectangles onto a 3X2 surface. This surface was then scaled, filtered, and used as the pattern for 3 XrFill operations. This demonstrates an efficient means of generating linear gradients within Xr.

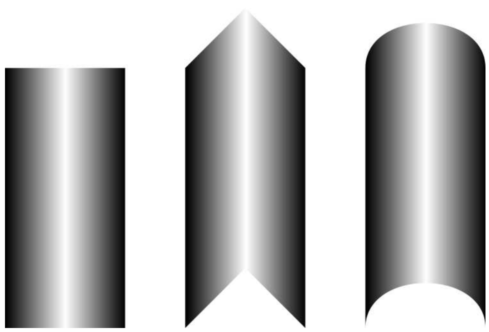

Figure 10: Outline affects perception of depth from shading, (after an illustration by Isao Watanabe[14]). This example uses XrFill with XrSetPattern

2.9 Images

In addition to the vector path support, Xr also supports bitmapped images as primitive objects. Images are transformed, (and optionally filtered), by the CTM in the same manner as all other primitives. In order to display an image, an XrSurface object must first be created for the image, then the surface can be displayed with the XrShowSurface function. XrShowSurface places an image of the given width and height at the origin in user space, so XrTranslate can be used to position the surface.

In addition to the CTM, each surface also has its own matrix providing a transformation from user space to image space. This matrix can be used to transform a surface independently from the CTM.

The XrShowSurface function has another important use besides allowing the display of external images. When using the Porter/Duff compositing operators, it is often desirable to combine several graphics primitives on an intermediate surface before compositing the result onto the target surface. This functionality is similar to the notion of transparency groups in PDF 1.4 and can be achieved with the following idiom:

```
XrSave (xrs);
XrSetTargetSurface (xrs, surface);
/* draw to intermediate surface with
   any Xr functions */
XrRestore (xrs);
XrShowSurface (xrs, surface);
```

In this example an intermediate surface is installed as the target surface, and then graphics are drawn on the intermediate surface. When XrRestore is called, the original target surface is restored and the resulting graphics from the intermediate surface are composited onto the original target.

This technique can be applied recursively with any number of levels of intermediate surfaces each receiving the results of its "child" surfaces before being composited onto its "parent" surface.

Alternatively, images can be constructed from data external to the Xr environment, acquired from image files, external devices or even the window system. Because the image formats used within Xr are exposed to applications, this kind of manipulation is easy and efficient.

3 Implementation

As currently implemented, Xr has good support for all functions described here. The major aspects of the PostScript imaging model that have not been discussed are text/font support, clipping, and color management. Xr does include some level of experimental support for text and clipping already, but these areas need further development.

The Xr system is implemented as 3 major library components: libXr, libXc, and libIc. LibXr provides the user-level API described in detail already.

LibXc is the backend of the Xr system. It provides a uniform, abstract interface to several different low-level graphics systems. Currently, libXc provides support for drawing to the X Window System or to in-memory images. The semantics of the libXc interface are consistent with the X Render Extension so it is used directly whenever available.

LibIc is an image compositing library that is used by libXc when drawing to in-memory images. LibIc can also be used to provide support for a low-level system whose semantics do not match the libXc interface. In this case, libIc is used to draw everything to an in-memory image and then the resulting image is provided to the low-level system. This is the approach libXc uses to draw to an X server that does not support the X Render Extension.

The libIc code is based on the original code for the software fallback in the reference implementation of the X Render Extension. It would be useful to convert any X server using that implementation to instead use libIc.

These three libraries are implemented in approximately 7000 lines of C code.

4 Related Work

Of the many existing graphics systems, several relate directly to this new work.

4.1 PostScript and Display PostScript

As described in the introduction, Xr adopts (and extends) the PostScript rendering model. However, PostScript is not just a rendering model as it includes a complete programming language. Display PostScript embeds a PostScript interpreter inside the window system. Drawing is done by generating PostScript programs and delivering them to the window system.

One obvious benefit of using PostScript everywhere is that printing and display can easily be done with the same rendering code, as long as the printer supports PostScript. A disadvantage is that images are never generated within the application address space making it more difficult to use where PostScript is not available.

Using the full PostScript language as an intermediate representation means that a significant fraction of the overall application development will be done in this primitive language. In addition, the PostScript portion is executed asynchronously with respect to the remaining code, further complicating development. Integrating the powerful PostScript rendering model into the regular application development language provides a coherent and efficient infrastructure.

4.2 Portable Document Format

PDF provides a very powerful rendering model, but no application interface. Generating PDF directly from an application would require some kind of PDF API along with a PDF interpreter. The goal for Xr is to be able to generate PDF output files while providing a clean application interface.

A secondary goal is to allow PDF interpreters to be implemented on top of Xr. As Xr is missing some of the less important PDF operations, those will need to be emulated within the interpreter. An important feature within Xr is that such emulation be reasonably efficient.

4.3 OpenGL

OpenGL[12] provides an API with much the same flavor as Xr; immediate mode functions with an underlying stateful library. OpenGL doesn't provide the PostScript rendering model, and doesn't purport to support printing or the local generation of images.

As Xr provides an abstract interface atop many graphics architectures, it should be possible to layer Xr on OpenGL.

5 Future Work

The Xr library is in active development. Everything described in this paper is currently working, but much work remains to make the library generally useful for application development.

5.1 Text Support

Much of the current design effort has been focused on the high-level drawing model and some low-level rendering implementation for geometric primitives. This design effort was simplified by the adoption of the PostScript model. PostScript offers a few useful suggestions about handling text, but applications require significantly more information about fonts and layout. The current plan is to require applications to use the FreeType [13] library for font access and the Fontconfig [8] library for font selection and matching. That should leave Xr needing only relatively primitive support for positioning glyphs and will push issues of layout back on the application.

5.2 Printing Backend

Xr is currently able to target the X Window System, (with or without the X Render Extension), as well as local images. Still missing is the ability to generate PostScript or PDF output files. Getting this working is important not only so that applications can print, but also because there may be unintended limitations in both the implementation and specification caused by the essential similarity between the two existing backends.

One of the goals of Xr is to have identical output across all output devices. This will require that Xr embed glyph images along with the document output to ensure font matching across all PostScript or PDF interpreters. Embedding TrueType and Type1 fonts in the output file should help solve this problem.

5.3 Color Management

Xr currently supports only the RGB color space. This simplifies many aspects of the library interface and implementation. While it might become necessary to add support for more sophisticated color management, such development will certainly await a compelling need. One simple thing to do in the meantime would be to reinterpret the device-dependent

RGB values currently provided as sRGB instead. Using ICC color profiles would permit reasonable color matching across devices while not adding significant burden to the API or implementation.

6 Availability

Xr is free software released under an MIT license from `http://xr.xwin.org`.

7 Disclaimer

Portions of this effort sponsored by the Defense Advanced Research Projects Agency (DARPA) under agreement number F30602-99-1-0529. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

References

[1] Adobe Systems Incorporated. *PostScript Language Reference Manual*. Addison Wesley, 1985.

[2] Simon Budig. The linux-pinguin again. `http://www.home.unix-ag.org/simon/penguin`.

[3] Larry Ewing. Linux 2.0 penguins. `http://www.isc.tamu.edu/~lewing/linux`.

[4] Leo Guibas, Lyle Ramshaw, and Jorge Stolfi. A kinetic framework for computational geometry. In *Proceedings of the IEEE 1983 24th Annual Symposium on the Foundations of Computer Science*, pages 100–111. IEEE Computer Society Press, 1983.

[5] Adobe Systems Incorporated, editor. *PDF Reference: Version 1.4*. Addison-Wesley, 3rd edition, 2001.

[6] Peter Mattis, Spencer Kimball, and the GIMP developers. The GIMP: The GNU image manipulation program. `http://www.gimp.org`.

[7] Keith Packard. Design and Implementation of the X Rendering Extension. In *FREENIX Track, 2001 Usenix Annual Technical Conference*, Boston, MA, June 2001. USENIX.

[8] Keith Packard. Font Configuration and Customization for Open Source Systems. In *2002 Gnome User's and Developers European Conference*, Seville, Spain, April 2002. Gnome.

[9] Thomas Porter and Tom Duff. Compositing Digital Images. *Computer Graphics*, 18(3):253–259, July 1984.

[10] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, third edition, 1992.

[11] Al Seckel. *The Great Book of Optical Illusions*. Firefly Books Ltd., 2002.

[12] Mark Segal, Kurt Akeley, and Jon Leach (ed). *The OpenGL Graphics System: A Specification*. SGI, 1999.

[13] David Turner and The FreeType Development Team. The design of FreeType 2, 2000. `http://www.freetype.org/freetype2/docs/design/`.

[14] Isao Watanabe. 3-d shape and outline. `http://www.let.kumamoto-u.ac.jp/watanabe/Watanabe-E/Illus-E/3D-E/index.%html`.

A Example Source Code

This appendix contains the source code that was used to draw each figure in Section 2. Each example contains a top-level "draw" function that accepts an XrState pointer, a width, and a height. The examples here can be made into complete programs by adding the code from the example program of Figure 4 and inserting a call to the appropriate "draw" function.

```
void
draw_hering (XrState *xrs,
             int width, int height)
{
#define LINES 32.0
#define MAX_THETA (.80 * M_PI_2)
#define THETA (2 * MAX_THETA / (LINES-1))

  int i;

  XrSetRGBColor (xrs, 0, 0, 0);
  XrSetLineWidth (xrs, 2.0);

  XrSave (xrs);
  {
    XrTranslate (xrs, width / 2, height / 2);
    XrRotate (xrs, MAX_THETA);

    for (i=0; i < LINES; i++) {
      XrMoveTo (xrs, -2 * width, 0);
      XrLineTo (xrs, 2 * width, 0);
      XrStroke (xrs);

      XrRotate (xrs, - THETA);
    }
  }
  XrRestore (xrs);

  XrSetLineWidth (xrs, 6);
  XrSetRGBColor (xrs, 1, 0, 0);

  XrMoveTo (xrs, width / 4, 0);
  XrRelLineTo (xrs, 0, height);
  XrStroke (xrs);

  XrMoveTo (xrs, 3 * width / 4, 0);
  XrRelLineTo (xrs, 0, height);
  XrStroke (xrs);
}
```

Figure 11: Source for Hering illusion of Figure 5

```
void
draw_caps_joins (XrState *xrs,
                 int width, int height)
{
    static double dashes[2] = {10, 20};
    int line_width = height / 12 & (~1);

    XrSetLineWidth (xrs, line_width);
    XrSetRGBColor (xrs, 0, 0, 0);

    XrTranslate (xrs, line_width, line_width);
    width -= 2 *line_width;

    XrSetLineJoin (xrs, XrLineJoinBevel);
    XrSetLineCap (xrs, XrLineCapButt);
    stroke_v_twice (xrs, width, height);

    XrTranslate (xrs, 0, height/4-line_width);
    XrSetLineJoin (xrs, XrLineJoinMiter);
    XrSetLineCap (xrs, XrLineCapSquare);
    stroke_v_twice (xrs, width, height);

    XrTranslate (xrs, 0, height/4-line_width);
    XrSetLineJoin (xrs, XrLineJoinRound);
    XrSetLineCap (xrs, XrLineCapRound);
    stroke_v_twice (xrs, width, height);
}

void
stroke_v_twice (XrState *xrs,
                int width, int height)
{
    XrMoveTo (xrs, 0, 0);
    XrRelLineTo (xrs, width/2,  height/2);
    XrRelLineTo (xrs, width/2, -height/2);

    XrSave (xrs);
    XrStroke (xrs);
    XrRestore (xrs);

    XrSave (xrs);
    {
        XrSetLineWidth (xrs, 2.0);
        XrSetLineCap (xrs, XrLineCapButt);
        XrSetRGBColor (xrs, 1, 1, 1);
        XrStroke (xrs);
    }
    XrRestore (xrs);

    XrNewPath (xrs);
}
```

Figure 12: Source for cap and join demonstration of Figure 7

```
void
draw_spiral (XrState *xrs,
             int width, int height)
{
  int wd = .02 * width;
  int hd = .02 * height;
  int i;

  width -= 2;
  height -= 2;

  XrMoveTo (xrs, width - 1, -hd - 1);
  for (i=0; i < 9; i++) {
    XrRelLineTo (xrs, 0, height-hd*(2*i-1));
    XrRelLineTo (xrs, -(width-wd*(2*i)), 0);
    XrRelLineTo (xrs, 0,-(height-hd*(2*i)));
    XrRelLineTo (xrs,  width-wd*(2*i+1), 0);
  }

  XrSetRGBColor (xrs, 0, 0, 1);
  XrStroke (xrs);
}
```

Figure 13: Source for nested box illusion of Figure 6

```
void
draw_splines (XrState *xrs,
              int width, int height)
{
    int i;
    double tolerance[5] = {.1,.5,1,5,10};
    double line_width = .08 * width;
    double gap = width / 6;

    XrSetRGBColor (xrs, 0, 0, 0);
    XrSetLineWidth (xrs, line_width);

    XrTranslate (xrs, gap, 0);
    for (i=0; i < 5; i++) {
        XrSetTolerance (xrs, tolerance[i]);
        draw_spline (xrs, height);
        XrTranslate (xrs, gap, 0);
    }
}

void
draw_spline (XrState *xrs, double height)
{
    XrMoveTo (xrs, 0, .1 * height);
    height = .8 * height;
    XrRelCurveTo (xrs,
                  -height/2, height/2,
                  height/2, height/2,
                  0, height);
    XrStroke (xrs);
}
```

Figure 14: Source for splines drawn with varying tolerance as in Figure 9

```
void
draw_stars (XrState *xrs,
            int width, int height)
{
  XrSetRGBColor (xrs, 0, 0, 0);

  XrSave (xrs);
  {
    XrTranslate (xrs, 5, height/2.6);
    XrScale (xrs, height, height);
    star_path (xrs);
    XrSetFillRule (xrs, XrFillRuleWinding);
    XrFill (xrs);
  }
  XrRestore (xrs);

  XrSave (xrs);
  {
    XrTranslate (xrs,
                 width-height-5, height/2.6);
    XrScale (xrs, height, height);
    star_path (xrs);
    XrSetFillRule (xrs, XrFillRuleEvenOdd);
    XrFill (xrs);
  }
  XrRestore (xrs);
}

void
star_path (XrState *xrs)
{
    int i;
    double theta = 4 * M_PI / 5.0;

    XrMoveTo (xrs, 0, 0);
    for (i=0; i < 4; i++) {
        XrRelLineTo (xrs, 1.0, 0);
        XrRotate (xrs, theta);
    }
    XrClosePath (xrs);
}
```

Figure 15: Source for stars to demonstrate fill rule as in Figure 8

```
void
draw_gradients (XrState *xrs,
                int img_width, int img_height)
{
    XrSurface *gradient;
    double width, height, pad;

    width = img_width / 4.0;
    pad = (img_width - (3 * width)) / 2.0;
    height = img_height;

    gradient=make_gradient(xrs,width,height);

    XrSetPattern (xrs, gradient);
    draw_flat (xrs, width, height);
    XrTranslate (xrs, width + pad, 0);
    XrSetPattern (xrs, gradient);
    draw_tent (xrs, width, height);
    XrTranslate (xrs, width + pad, 0);
    XrSetPattern (xrs, gradient);
    draw_cylinder (xrs, width, height);

    XrRestore (xrs);

    XrSurfaceDestroy (gradient);
}

XrSurface *
make_gradient (XrState *xrs,
               double width, double height)
{
    XrSurface *g;
    XrMatrix *matrix;

    XrSave (xrs);

    g = XrSurfaceCreateNextTo (
                 XrGetTargetSurface (xrs),
                 XrFormatARGB32, 3, 2);
    XrSetTargetSurface (xrs, g);

    XrSetRGBColor (xrs, 0, 0, 0);
    XrRectangle (xrs, 0, 0, 1, 2);
    XrFill (xrs);

    XrSetRGBColor (xrs, 1, 1, 1);
    XrRectangle (xrs, 1, 0, 1, 2);
    XrFill (xrs);

    XrSetRGBColor (xrs, 0, 0, 0);
    XrRectangle (xrs, 2, 0, 1, 2);
    XrFill (xrs);

    XrRestore (xrs);

    matrix = XrMatrixCreate ();
    XrMatrixScale (matrix,
                   2.0/width, 1.0/height);
    XrSurfaceSetMatrix (g, matrix);
    XrSurfaceSetFilter (g, XrFilterBilinear);
    XrMatrixDestroy (matrix);

    return g;
}
```

```
void
draw_flat (XrState *xrs, double w, double h)
{
    double hw = w / 2.0;

    XrRectangle (xrs, 0, hw, w, h - hw);

    XrFill (xrs);
}

void
draw_tent (XrState *xrs, double w, double h)
{
    double hw = w / 2.0;

    XrMoveTo    (xrs,   0,  hw);
    XrRelLineTo (xrs,  hw, -hw);
    XrRelLineTo (xrs,  hw,  hw);
    XrRelLineTo (xrs,   0,  h - hw);
    XrRelLineTo (xrs, -hw, -hw);
    XrRelLineTo (xrs, -hw,  hw);
    XrClosePath (xrs);

    XrFill (xrs);
}

void
draw_cylinder (XrState *xrs, double w, double h)
{
    double hw = w / 2.0;

    XrMoveTo (xrs, 0, hw);
    XrRelCurveTo (xrs,   0, -hw,
                  w, -hw,   w, 0);
    XrRelLineTo (xrs, 0, h - hw);
    XrRelCurveTo (xrs,   0, -hw,
                  -w, -hw,  -w, 0);
    XrClosePath (xrs);

    XrFill (xrs);
}
```

Figure 16: Source for 3 gradient-filled shapes of Figure 10