A New Rendering Model for X

Keith Packard
XFree86 Core Team, SuSE Inc.
keithp@suse.com

Abstract

X version 11 [SG92] was originally designed and implemented in 1987. In the intervening 13 years, there have been advancements in both applications and hardware, but the core of the X Window System has remained largely unchanged. The last major X server architecture changes were included in X11R4. The last wide-spread functional enhancement exported by the X server might well be the Shape extension [Pac89], designed (in the hot tub) at the 1989 Winter Usenix in San Diego.

The rise of inexpensive Unix desktop systems in the last couple of years has led to the development of new user-interface libraries, which are not well served by the existing X rendering model. A new 2D rendering model is being developed to serve this new community of applications. The problem space and proposed solutions are discussed.

1 Introduction

While a window system is more than a collection of rendering routines, the available rendering primitives constrain the capabilities of applications more than anything else. The X rendering model was developed to match the abilities of workstation hardware developed fifteen years ago and has significant limitations when applied to application development today.

As application development has advanced, the X protocol has devolved into little more than an image transport mechanism. Applications perform rendering in client-side buffers and transport the result to the screen. A shared memory mechanism for delivering images to the X server exists when the application is running on the same machine as the display, but performance suffers when attempting to run these applications over the network.

Many new graphics accelerators are providing acceleration for operations needed by new applications. Only by moving these operations into the X server can this acceleration be made accessible to X applications.

2 Origins of X Rendering

A combination of archaeology and history is needed to understand the current state of X rendering technology. Cast your mind back to 1987, and try to remember graphical workstations of that era. A 1 MIPS machine was the state of the art and one was lucky to have color on the desktop. Color, of course, was 8 bits with a palette. Those hotheads over at SGI were making noises about true color hardware, but for most that was not even a dream. Hardware acceleration was available, but frequently no faster than software, and a huge pain to code for.

The state of the art in 2D rendering was PostScript [Ado85]. The definition of objects by precise mathematical formulae was compellingly beautiful to engineers. PostScript provided sophisticated font technology embedded inside the printers of the era, but left the desktop with only bitmap versions of the same fonts.

Into this stepped a group of networking protocol and hardware hackers intent on updating their latest offering, the X Window System. Not a single one of them had even been introduced to a computational geometer, nor did they have the resources of the modern internet to help with the design. Of course a constant refrain was to get the darn thing finished and out the door. Digital, who was funding the sample implementation, had product schedules to meet. Meanwhile, back at MIT, Project Athena was deploying more and more X10 boxes.

So they picked up the PostScript "Red Book" and started writing a specification. Of course their new window system was extensible; with any luck, limitations in the original design would be masked by clever add-ons in the future. What they failed to realize was that the Red Book inadequately described the actual implementation of some primitives. The developers also lacked foresight about how difficult it would be to create consensus around future rendering standards.

One big limitation of PostScript in that era was in image manipulation. Printers were black-and-white, so PostScript didn't need any complex image compositing operators. Besides, X was an interactive protocol: alpha blending a full-screen image looked like slugs racing down the monitor.

And then there were lumpy lines. The Red Book describes a beautifully pure line stroking algorithm: a circular pen is dragged along the path and illuminates pixels within the circle. Too bad that the results look ugly—the apparent width of the line varies along the length of the line. Lacking understanding of the problem, Adobe kludged around it. John Hobby had recently solved the problem [Hob85], but his solution had not yet been published outside of Stanford and was not discovered by the X community for several years.

Instead of providing PostScript paths, X provided only straight lines and axis-aligned ellipses. Why axis-aligned? Because there was a rumor that the rendering algorithm for thin non-axis aligned ellipses was patented and there was agreement that X should be free of patented technologies. This rumor was unfounded; the algorithm (published many years ago [Pit67]) was unencumbered.

At one meeting, members of the X11 team looked around the table and discovered that not one of them had any clue about splines. Instead of doing something wrong, they left them out. Sub-pixel positioning was deemed an extravagant use of network bandwidth, since it would double the payload of each rendering primitive by requiring the use of 32 bits for each coordinate instead of 16.

The expectation was that these issues could be left for future development in the form of an extension. However, the usage of X expanded and compatibility between X servers was deemed a market necessity. Creating an extension that existed in only some X servers would create application interoperability problems. Thus the rendering model has stagnated.

2.1 Problems with the Core Protocol

Even ignoring new rendering techniques, the core protocol rendering architecture has some fundamental problems:

Lack of a stenciling operator

X10 provided a stenciling operator for solid fills, even this operator is missing from X11. A stencil can be emulated using a ClipMask, but the sample implementation of ClipMasks is inefficient, making this impractical.

Stenciling can be used to accelerate missing rendering primitives, the application generates the appropriate shape in a monochrome bitmap and uses that to stencil the result to the screen. The implementors of the sample server knew this and included a stenciling operator inside the server for use by higher level primitives.

Separation of lines and arcs

As useless as axis-aligned arcs are, they are made even less useful by being separated from lines. This means there is no way to join a sequence of lines and arcs together. As a special case, zero width/height arcs are defined to be equivalent to lines, making it possible to render an axis-aligned rounded rectangle.

No vertical escapement for text

This is all that is needed to render Asian text and to allow for rotated fonts.

2.2 Features of the Core Protocol

In building a new rendering system, it would be unwise to ignore the best parts of the existing system:

Precise pixelization

Each X operator, with the exception of thin primitives, has exactly specified pixelization requirements. This not only allows for reproducible rendering across X server implementations, but probably more importantly allows for automated testing of the rendering code. The rules themselves may be broken, but their existence is of vital importance.

Pixel values, not colors
Providing an underlying pixel value basis for the rendering system allows for the implementation of a color-based system in user space. The reverse is not true. Additionally, the only way to make boolean pixel operators usable by applications is to expose the pixel values.

Allow all rendering permutations
X allows applications to render stippled text using a variety of raster-ops (such as XOR). Such combinations work with all primitives other than ImageText. This makes it possible to dither everything on the screen in a consistent manner or to apply a reversible XOR raster-op.

3 Reasons for a New Model

The strongest argument for building a new rendering model is in evidence on almost every Linux machine these days. The combination of KDE, Gnome, and Enlightenment demonstrate that the world of 2D graphics is rapidly leaving the X Window System behind. These applications use sophisticated rendering primitives like outlined text and cubic splines. They improve image quality with anti-aliasing and blend images together with alpha compositing.

It is no longer a question of what kind of rendering will be done. The question now is where that rendering should happen. Applications will advance, and X must either keep up or get out of the way. One thing working in favor of an extension today is that many new applications are being written using a higher-level rendering model provided by a toolkit. Providing new X server functionality that matches the rendering model in the toolkit allows for a gradual adoption of the extension as the toolkits are modified: the toolkits can accelerate operations using the extension when available and still fall back to client-side rendering for older X servers.

4 Components of a New Rendering System

The current generation of 2D applications are similar in their demands on the rendering system. By analyzing existing usages and choosing primitives with care, a reasonably consistent system can be built which will be useful for many applications. The existence of applications with well-understood requirements provides an opportunity lacking in the initial protocol design.

4.1 Alpha Compositing

Alpha compositing is the blending together of images with a per-pixel (α) value controlling an arithmetic combination of the colors. There are many reasonable functions for this operator. The most common is a translucency operation, in which the colors are combined as $v = \alpha v_1 + (1 - \alpha) v_2$. As images are composited with this operator, they appear as translucent overlays on the original image.

Alpha compositing is also useful in approximating anti-aliasing. A suitable function and constraints on both the structure and order of the rendering primitives can yield satisfactory results.

3D applications make significant use of alpha compositing, so graphics hardware now commonly supports some of the most popular alpha compositing functions of OpenGL. Giving applications access to hardware compositing will provide dramatic performance improvements.

There are many different ways of presenting image data along with alpha channel information. At 32 bits per pixel, the alpha channel is frequently delivered in the unused upper byte. For 16 bit images sometimes the alpha channel is embedded as one of four 4-bit components and sometimes the alpha channel is in a separate 8-bit image.

For applications to be able to take maximal advantage of the available acceleration, the characteristics of the hardware must be exposed to the application. This significantly complicates the toolkit, which must match rendering requests with available resources.[1]

[1] Better architectural ideas are welcome.

Alpha compositing is easy to describe in a TrueColor environment, but more problematic in PseudoColor where there is no linear relation between pixel and color. Fortunately, most modern machines are able to display in TrueColor, making it tempting to provide this functionality only in that case.

4.2 Anti-Aliasing

Anti-aliasing is the application of signal processing in rasterization. It reduces the high-frequency quantization noise generated by imprecisely positioned object edges. Conceptually, anti-aliasing is performed by oversampling the image and resampling at the screen resolution.

A direct approach would create an oversampled version of the image in memory, and resample the completed image either to the frame buffer or (ideally) as it is delivered to the screen. The prospect of multiplying the amount of video memory by some large amount and reducing rendering performance by a similar amount have led to a search for inexpensive incremental approximations.

When displaying a single convex primitive, the simple alpha compositing operator described above can be used to accurately approximate anti-aliasing. By generating an alpha channel containing the output of the resampling filter, the primitive can be composited onto the screen. However, when more than one primitive is involved the task becomes more difficult, as the alignment of the edges of each primitive is lost in the compositing operation.

OpenGL contains a set of more complicated alpha operations, which ameliorate the errors in this approximation when used properly. A reasonable subset of these operations will be included in the new system.

As mentioned above, the alpha channel is filled with the output of the resampling filter. Most existing anti-aliasing systems simply compute the amount of the pixel covered by the object and use that as the alpha value; for the edges of a polygon, the system has a measure of that value computed as it walks the edge. A more sophisticated anti-aliasing system uses the output of a 2D filter to fill the alpha channel. This filter can even take into account the response characteristics of the electron beam displaying the image: systems built with such techniques work quite well.

Given that this alpha blending technique is only approximate and that sophisticated techniques are likely to be a performance problem in the near term, only the simple coverage model is currently planned. Provisions will be made for adding new anti-aliasing mechanisms in the future.

4.3 Coordinate System

The current rendering system uses a 16-bit integer coordinate space, which is fine for describing rectangles but imprecise when drawing text lines and polygons. Sub-pixel positioning is essential when compositing polygons into larger shapes, to avoid visible discontinuities along edges.

Sub-pixel positioning allows applications to more precisely position objects on the screen. To render an object using the core protocol, the coordinates must be rounded to the nearest pixel boundary. This mispositions the object by as much as 1/2 pixel. While this may not seem serious, the cumulative visual effect of many 1/2 pixel errors is quite noticeable. Scott Nelson describes this problem in more detail [Nel96], including an example showing the improvement offered by sub-pixel positions even in the absence of anti-aliasing.

One obvious coordinate representation is IEEE 32-bit floating point numbers. The 24 bit mantissa specified by IEEE would provide at least 8 bits of sub-pixel position within the 16-bit X coordinate space, and would be easy for applications to manage.

However, it is desirable for objects to be translationally invariant. As objects move to larger coordinates, IEEE floats will slowly drop bits of sub-pixel position information. This is especially important as windows move around the screen. While IEEE floats could probably be made to work by artificially limiting their precision for smaller values, using fixed-point numbers eliminates this problem entirely.

The next question is how many bits of fraction to use. Four is enough for most applications, but eight will suffice for all but the most particular uses. Applications which use larger coordinate spaces will still need to perform clipping operations during the transformation to X coordinates, but with 8-bits of sub-pixel position, it should suffice for most to simply truncate objects at the boundary of the X coordinate space.

For these reasons, 32-bit fixed-point coordinates with 8 fractional bits will be used.

4.4 Rendering Primitives

One thing missing from the core protocol is a simple server primitive that could be used to render geometrical objects not defined by the protocol. Inside a PostScript interpreter, the primitive used is a horizontal trapezoid—that is, the top and bottom edges are horizontal. LibArt, the rendering library for the Gnome project, uses an equivalent primitive called sorted edge lists.

So, at a minimum, this new primitive will be included. Unlike the core polygon request, this request will be able to draw many trapezoids at a time.

A question remains as to whether PostScript-style paths should be included. Doing so would significantly reduce the wire traffic but would complicate the implementation. The paths would include lines, cubic splines and character elements.

Paths would be rasterized by cracking them into trapezoids as described above, using a settable error value to describe the polygonalization of curves. By making this rendering mechanism explicit, it would be possible to precisely specify pixelization of the path in relatively simple terms, and to exactly replicate this pixelization on the client side if necessary.

4.5 Text

The original X design was done before outline rasterizers were used to generate screen fonts. The only fonts available were bitmaps, and the idea of providing scaled versions of those for the screen lacked appeal.

The resulting design does not match the realities of outline fonts well at all. Even the XLFD specification (and its extensions to support scalable fonts, font subsetting, and glyph rotation) is difficult or impossible to use with outlined fonts.

One problem to be solved is in the naming and accessing of fonts. A simple mechanism could be added to provide more control over which font is selected, and to provide more than a simple string to identify fonts. Another issue is access to additional metrics about the font, such as pair kerning tables, glyph names, and more precise glyph metrics.

A requirement for modern applications is that the application and the X server share access to the raw outline data and metrics. This allows the application to augment the text rendering provided by the X server with fancier versions on the client side. An easy way to provide this is to extend the X Font Services Protocol [Ful94] to include this additional information.

Another issue with the core protocol is in accessing glyph metrics. The core protocol provides only the QueryFont request which retrieves metrics for all glyphs in a font at once. This allows the client to quickly compute the extents for any set of glyphs without consulting the server in the future. However, it also requires that the metrics for every glyph in the font be available when the request is made. For scalable fonts, this means that the entire font must be rasterized; for most scalable technologies, generating X metrics is a side effect of rasterizing glyphs.

Most applications issue a QueryFont for each font that they open, this means that in normal usage, the X server rasterizes every glyph in every font used by applications.

Additionally, the ListFontsWithInfo request returns bounding metrics for all glyphs in the font. Computing the bounding metrics requires the complete set of metrics for the font.

New font information requests are needed. A request to query the metrics for a list of glyphs along with a new font listing function which provides as much information about the font as can be gathered without rasterizing every glyph.

Better rendering primitives are required as well, allowing for rotation of glyphs and baselines, sub-pixel positioning, and anti-aliasing.

Direct support of glyph outlines may be addressed at some point. This is somewhat difficult given the multiplicity of outline font formats, the lack of high-quality Type1 rasterizers and the additional rendering infrastructure required.

5 Strategy

Building a new rendering system will take some time, and feedback during the process is essential to make it successful. To make this possible, the system will be developed in stages, with each stage building on the previous stages. Some enhancements will be available soon, while others wait both for resources to implement them and for consensus to be built supporting the particular design.

1. **Alpha Compositing**
   Many applications need this today but are suffering with unaccelerated client-side implementations. This is an operation that graphics hardware can improve by a huge amount, forming the basis for anti-aliased graphics.

2. **Trapezoids**
   Moving these primitives to the server will reduce the demands placed on the bus between the CPU and the graphics adapter.

3. **Paths**
   Moving these into the server will reduce wire traffic, but not provide any dramatic performance improvements except in a networked environment.

5. **Font Information**
   Reducing the work required to open and list fonts will improve the ability of the system to cope with the increasing availability of outline and 16-bit fonts.

4. **Font Access**
   Improving the mechanisms by which the X server and application share access to the same fonts will allow for improvements in management and deployment of applications, especially in a complex networked environment.

5. **Text Rendering**
   Adding the ability to display outline fonts with the option of anti-aliasing has been on the "wish list" for a long time.

Each of these systems will be implemented first in software, and then hardware acceleration will be provided for some common graphics chips. Where possible, existing graphics systems can be used to avoid a duplication of effort. In particular, OpenGL will make this task easier for chips which have appropriate support in place.

6 Conclusion

The existing X rendering model was rushed to completion by people who understood their limitations and expected it to be quickly augmented with suitable extensions. No credible 2D graphics extensions have been developed in the intervening 13 years, but the world has recently changed. The advent of new toolkits that provide advanced rendering models abstracted from the core protocol opens a new opportunity to improve the X Window System. A new rendering model, designed to solve specific performance and network transparency issues of these new toolkits, has the promise of significantly increasing the power of the X desktop environment.

References

[Ado85] Adobe Systems Incorporated. *PostScript Language Reference Manual*. Addison Wesley, 1985.

[Ful94] Jim Fulton. The x font service protocol. X consortium standard, Network Computing Devices, Inc., 1994.

[Hob85] John D. Hobby. *Digitized Brush Trajectories*. PhD thesis, Stanford University, 1985. Also *Stanford Report STAN-CS-85-1070*.

[Nel96] Scott R. Nelson. Twelve characteristics of correct antialiased lines. *Journal of Graphics Tools*, 1(4):1–20, 1996.

[Pac89] Keith Packard. X nonrectangular window shape extension protocol. X consortium standard, MIT X Consortium, 1989.

[Pit67] M. L. V. Pitteway. Algorithm for drawing ellipses or hyperbolae with a digital plotter. *The Computer Journal*, 10(3):282–289, November 1967.

[SG92] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, third edition, 1992.