High Performance X Servers in the Kdrive Architecture

Eric Anholt
LinuxFund
anholt@FreeBSD.org

Abstract

The common usage patterns of 2D drivers for the X Window System have changed over time. New extensions such as Render and Composite are creating new demands for 2D rendering which do not match those for previous architectures tailored to the core protocol. This paper describes changes made to the Kdrive X server implementation to implement new 2D acceleration, improve management of offscreen memory, implement OpenGL, and implement XVideo in a manner compatible with the Composite extension. With these changes, Kdrive is far better suited as a desktop X server than before and may serve as an example for desktop X server implementations. Simple benchmarks are presented.

1. Introduction

As desktop environments have advanced, the common usage patterns of the X Window System [1] have changed. In particular, the Render extension to the protocol has allowed for new graphical operations related to blending of images [2, 3, 4]. Software implementations of these operations are too slow for them to be used extensively in the user interface, and modern graphics hardware can provide dramatic performance improvements for many operations. The new Composite extension is increasing the amount of rendering to offscreen memory, and creates new requirements for rendering of older operations. At the same time, rendering operations provided by the core protocol are being used less frequently. Improving performance of these new operations will enable significantly greater use and permit a new user experience for the Linux desktop. This paper will explore some of the changes made to the Kdrive-based X server (also known as TinyX or Xkdrive), to improve performance and create a server suitable for the Linux desktop. This introduction will describe the basic components of an X server, the features of hardware typically found in desktop computers, and describe some of the features and requirements of some X extensions related to the work here. Section 2 will describe another X server implementation, XFree86, and how it compared to Kdrive. Section 3 will cover the changes made to Kdrive as a result of research into making Kdrive-based X servers useful for the desktop. Section 4 will cover some of the tangible results of this work, and Section 5 will describe some of the changes remaining to be made to Kdrive to make it into a capable desktop X server.

1.2. X Server Architecture

The X sample server developed for the MIT X Consortium by Digital Equipment Corporation in 1987 provides a large base of shared code. Most X servers available today, including the two mentioned in this paper – XFree86 and Kdrive, take advantage of this. This shared code includes parts for handling basic X server initialization and teardown, for decoding the network protocol, for breaking rendering requests into simpler operations, for performing rendering operations to memory. It also includes extension implementations, among other things. Different X servers using this shared code base are implemented in the Device Dependent X (DDX) part of the server, located under the *hw* directory in the source tree of both XFree86 and Kdrive. The DDX is responsible for actually dealing with input and output, whether that is directly to hardware (in the case of XFree86 and Kdrive) or to some software layer (as in the xnest or the virtual framebuffer servers located in the XFree86 source tree).

1.3. Features of Commodity Hardware

Most video cards in desktop computers support the following 2D acceleration functions:

- Filling rectangles with a solid color
- Copying one area of the screen to another
- Drawing lines using the Bresenham algorithm
- Performing color expansion of monochrome bitmaps

Color expansion is the process of filling pixels of the screen with foreground/background colors based on a monochrome bitmap, which is used for non-antialiased text rendering.

Typically, each of these operations can be done with a Raster Operation (ROP) that allows various binary operations to be done using the source or foreground/background colors and the destination. A planemask can also be set that allows writing to only certain bits of each pixel.

Most cards also have the ability to take a video image (which is stored in a YUV format that requires conversion before it can be displayed on an RGB display) and scale it onto the screen in places where the color on the screen matches a color key.

The core X protocol supported these 2D capabilities well, but with the advent of 3D hardware, new possibilities are opened up for 2D rendering. By rendering rectangles using 3D hardware designed for OpenGL [5] or DirectX, we can perform blending of source images into destination images, allowing for fast rendering of effects like transparent windows, shadows, and the operations necessary for antialiased text. A card supporting 3D typically has one or two texture units, but sometimes more. Each texture unit can take input from a source image in card memory (also referred to as offscreen memory or framebuffer) or AGP memory (system memory accessible directly by the card) when performing rendering. These textures can be of varying color depths or even non-RGB formats like YUV, can include alpha channels for controlling blending, and can be scaled and mixed with each other in various ways to produce a color value to be written to the destination. With the final pixel value, various blending functions can be used based on its alpha value and the alpha value of the pixel currently in the destination, which typically map to a set of blending functions provided by OpenGL.

1.4. Render

The Render extension provides several new rendering operations in X servers to handle the new demands of desktop applications, designed to target the 3D capabilities of newer hardware. It begins by creating a new type, the Picture, which wraps around an X drawable (a pixmap or a window). A Picture adds a fourth color channel, alpha, which indicates the amount of opacity. The Picture also adds the ability to apply transformations to the coordinates of pixels requested from the drawable and to make coordinates outside of the drawable wrap around (“repeat”). The new rendering commands that can be performed on Pictures include:

- blending of a constant color into a destination rectangle
- blending of rectangles from images into the destination
- blending of triangles from images into the destination
- blending of trapezoids from images into the destination
- blending of a series of glyphs into the destination

In the sample implementation of the Render extension used by both XFree86 and Kdrive, all of those graphical operations are implemented using the same basic composite operation for blending of rectangles. This call (“Composite”) takes the following arguments:

- a source picture and starting x/y coordinates
- an optional mask picture and starting x/y coordinates
- a destination picture and starting x/y coordinates
- the width and height of the rectangle to be rendered into the destination
- a composite operation

For each pixel in the destination rectangle, a source pixel is chosen based on the offset within the destination currently being rendered, after modification by the transform and repeat requirements of the source picture structure. If a mask is included, the source pixel is multiplied by the alpha value of a pixel chosen similarly from the mask picture (unless component alpha is being used, to be discussed in the "Complications for Render Acceleration" section). This calculated source pixel value is then blended into the destination pixel according to the composite operation chosen, as described in *Design and Implementation of the X Rendering Extension* by Keith Packard [2].

1.5. Composite

The Composite extension (not to be confused with Render's Composite operation) is an X server extension that was first implemented in the Kdrive X server. It allows an X client to control the compositing of a window hierarchy into a parent window. It does this by redirecting the rendering of that window hierarchy (or, to simplify the discussion, a single window) into an offscreen pixmap. This X client (a compositing manager) can then use the Damage extension to be notified when an area of the window changes, and draw the appropriate area in the parent window as necessary. Because the client controls the drawing into the destination, it can add effects such as translucency, shadows, or reshaping of windows. Composite can also be used to simply get at the pixels of a window that would otherwise be obscured. The first publicly available compositing manager was xcompmgr, which adds shadows to windows and provides translucent menus. The use of xcompmgr also eliminates the delay between a window being exposed (for example after a window on top of it is moved) and the redraw of the window underneath. This provides a smoother user experience, with less of the flicker associated with moving and resizing of windows.

1.6. XVideo

The XVideo extension is used to perform conversion and scaling of video formats such as YUV in the X server. These conversions can be very expensive if done in software, and they are the primary time consumer in video playback.

1.7. OpenGL

OpenGL support is necessary for X servers due to its widespread use in games and other applications on the desktop. The GLX extension allows sending OpenGL commands over the X protocol to be rendered by an X server. Because the wrapping of OpenGL commands into GLX requests creates significant overhead, it is desirable to avoid that process. To avoid the overhead, direct rendering is implemented to give the client direct access to the video card. However, direct rendering introduces security risks because many cards can issue DMA requests to read or write system memory, and thus access has to be limited. Direct rendering also requires synchronization of access to the hardware as well as kernel assistance in submitting DMA requests and handling interrupts.

2. Related Work

XFree86 is the most popular open-source X server, and is the de facto standard for comparison of other X servers. This section describes significant differences between XFree86 and Kdrive.

2.1. XAA Versus KAA

Part of the DDX is responsible for handling hardware acceleration of drawing. In XFree86 and Kdrive this piece is called the XFree86 Acceleration Architecture (XAA) or Kdrive Acceleration Architecture (KAA) [6]. The design decisions behind the two implementations vary greatly.

XAA manages offscreen memory by treating it as a large 2D area at a set number of bits per pixel (bpp). Pixmaps (offscreen images) of any other bpp therefore cannot be stored in offscreen memory. Acceleration hooks are implemented as two or more callbacks to the driver. One is a Setup hook which sets up the hardware for the planemask and ROP, and performs other preparation for rendering that operation. The Subsequent hook can then be called one or more times to actually perform rendering. This division is an advantage because it avoids repeated Setup calls in the case where clipping results in the operation needing to be done in separate pieces. The video card is set up so that the visible screen (offset 0) is the source and destination offset, the screen's width is the pitch, and offscreen areas are simply areas with "y" values beyond the height of the screen. The Setup hook is not allowed to fail, so many flags are available for drivers to tell XAA about the hardware's features, such as planemask and transparency support or limitations on the direction of screen-to-screen copies. Also, accelerators have limits on coordinates for 2D acceleration, so often the memory for offscreen pixmaps has to be limited so that offscreen areas with "y" values beyond the limits are not used. XAA has hooks for copying areas, filling rectangles and trapezoids, drawing different types of lines, color expansion of monochrome bitmaps, uploading images from system memory to screen memory, and Composite operations from memory (described in Section 2.2).

In contrast, in KAA the offscreen memory is managed as a linear area. This allows pixmaps of different bpp to be stored offscreen, which is a requirement for fast acceleration of Render operations, where using pixmaps of different bpp from the screen is the norm.

Because the offscreen memory cannot be represented as a single 2D area, the pixmaps have to be passed in to the acceleration hooks, and the drivers then set up the hardware's offset and pitch registers for the source and destination each time. This has the added benefit that there are not the same limitations on the total offscreen memory due to the size of the hardware's 2D coordinates. To avoid the complexity of flags for KAA to determine whether an operation can be accelerated, the Prepare hook (equivalent to Setup in XFree86) is allowed to fail, which tells KAA to fall back to software rendering. An additional Done hook is included to allow for any cleanup necessary after a rendering operation, but the Done hook is a stub in most drivers. KAA implemented only two types of acceleration before the work for this paper, which were screen-to-screen copies and solid fills. It was decided that the other types of acceleration supported in XAA, such as lines, trapezoids and color expansion were used rarely enough in modern desktops that they did not justify the additional complexity to accelerate.

2.2. XAA Implementation of Render Acceleration

XAA implements a small subset of Render operations. It handles only cases where the destination picture is located in offscreen memory and the source picture is not. Because XAA's 2D offscreen memory layout prevents the offscreen storage of pixmaps with a bpp different from the screen, many source images will be located in system memory anyway. This means that typically the source picture has to be uploaded into a scratch area each time before the hardware can perform acceleration from it. Furthermore, the only case where composite with a mask is supported is when the source is a 1x1 repeating picture (a solid color). The hooks for this acceleration of compositing are implemented for Matrox Gx00 cards in the mga driver, the vmware emulator in the vmware driver, and possibly for SiS 300-series cards in the sis driver.

2.3. OpenGL in XFree86

In XFree86, OpenGL is implemented using the Direct Rendering Infrastructure (DRI). The DRI consists of several components: a kernel module specific to the video card ("Direct Rendering Module" or DRM), a DRI-aware 2D driver in the DDX, the GLX extension, the XF86DRI extension to the X protocol (used for communicating information about the DRM and hardware setup to the client), and the 3D driver itself, which is a card-specific shared library opened by the OpenGL library (libGL) .

The kernel module is used by the X server to set up a shared memory area that contains information about the video card. That area includes a lock, which is used cooperatively by clients and the server to arbitrate access to the card, along with card-specific state that is managed by the clients and server. Though clients could misbehave and abuse the lock, this can at worst lead to a lockup of the hardware and should not allow a security compromise. The DRM also often allows allocation of DMA buffers and submission of sets of commands by DMA, waiting for hardware interrupts, and other interactions with the hardware which cannot be handled or are difficult to handle in userspace.

Because of the availability of source to the DRI drivers in XFree86, many of the drivers are very similar in their structure. Most drivers allocate memory for OpenGL's back and depth buffers statically at server startup or when the first 3D client is started. Clipping is used so that all clients can share the back and depth buffers and only render to the same rectangles in the back buffer as they would on the visible screen. Because the design decision to use these static buffers was made originally, pbuffers (pixel buffers not associated with the screen) are not implemented, though they are in demand. The SiS300 driver is different in that it allocates back and depth buffers per client from a static block of card memory managed by the kernel module. However, its design is not a good model for other drivers because it cannot handle failure to allocate memory for buffers or textures.

In XFree86, all indirect rendering (OpenGL commands sent through GLX) is performed in software. Although the ability to hardware-accelerate indirect rendering is desired, it has not been implemented yet. The primary reason is that the interface between the server GLX code and the current software rendering core ("libGLcore") is very different from that between libGL and a DRI 3D driver.

2.4. XVideo in XFree86

In XFree86, XVideo is almost always implemented using the card's overlay scaler. With this method, the window where the video is to be shown is painted with a color key. The video image is loaded into offscreen memory, and the hardware is set up to paint the image appropriately into the card's video output where the color key matches. Most, or most likely all, hardware has only a single overlay scaler, meaning only one video can be played at once.

One exception is the driver for the Matrox Gx00 series hardware, which offers textured video as an option along with the overlay scaler video, though enabling textured video is exclusive of the DRI and overlay video. With textured video, the 3D hardware is used to perform the conversion and scaling of the video, and supports multiple ports (many videos being displayed at once).

3. Acceleration in Modern X Servers

In the past, Kdrive has been targeted for X server development and not for general desktop use. The changes researched for this paper were oriented towards making Kdrive an option for a desktop server by implementing new acceleration and making an example of acceleration for other servers to follow. This consisted of creating a better architecture for Render acceleration, implementing GLX in the server, improving management of offscreen memory, and implementing XVideo in a manner more appropriate for the Composite extension.

3.1. Complications for Render Acceleration

The Composite operation implemented by Render is complicated, and it cannot be implemented in hardware in all cases. Having two source images for an operation (source and mask) typically requires the use of the 3D hardware to implement. Most hardware cannot support 8-bit alpha images (which are very common) for source or destination when blending based on alpha values is necessary. Most older hardware 3D engines cannot handle textures with a width or height that are not a power of two number of pixels (NPOT), or are unable to do repeat (wrapping of texture coordinates outside the boundaries) for NPOT textures. Unfortunately, NPOT sources are the norm in 2D operations. Also, the alpha component of source, mask, or destination can be located in a separate buffer from the color data, which will be difficult to implement in hardware. Often the width or height of the source or destination can be larger than the hardware's limits on texture sizes (2048 pixels being common in newer hardware, but as low as 256 on older hardware).

Finally, component alpha will be difficult to accelerate. Component alpha rendering is used most frequently for sub-pixel rendering of anti-aliased fonts, taking advantage of the known order of the red, green, and blue bands in LCD pixels to provide additional screen resolution. When component alpha is used, the mask value is actually a set of four alpha values instead of one. Each source channel is multiplied by the corresponding mask channel to produce the final source color value, and each mask channel is multiplied by the source alpha to produce the final source alpha value. Then, each of these source value and source alpha pairs is blended into the appropriate destination channel using the specified composite operation. Although it should be possible to produce the correct source color values, current hardware can only do the alpha blending stage using a single source alpha value and not the componentized source alpha required. Composite operations that involve the source alpha value will most likely require using a multiple-stage process to accelerate, which may be difficult to implement.

3.2. Implementing Render Acceleration in Kdrive

There are two tasks for implementing Render acceleration. One is to accelerate Composite using existing 2D acceleration hooks whenever possible. The other is to create new hooks for common operations that map well to hardware features. This section covers some of the work on accelerating the most common operations desired by current applications.

Originally, KAA only implemented acceleration for Composite for CopyArea equivalents. This is when the operation is "Src" and there is no repeat flag set, no transform of source coordinates, and no mask. The xcompmgr application uses this operation very frequently to copy windows to the backbuffer of the screen, and to copy the backbuffer to the screen. The acceleration was implemented by using the standard Copy hook that all drivers had to implement.

The first new Composite acceleration hook implemented in Kdrive is called "Blend." Blend is a set of three hooks for a Kdrive driver: PrepareBlend, Blend, and DoneBlend. This mirrors the three hooks each for the Copy and Solid operations already implemented in Kdrive. In the Blend case, there is a source image but no mask image, and both source and destination are located in framebuffer memory. This may be possible to accelerate using only the "front-end scaler" of a card, originally targeted for video scaling, or also with the 3D hardware of cards with a single texture unit. In the PrepareBlend stage, the hardware is set up for the pictures and composite operation that are passed in, and failure can be returned to signal that software fallback is necessary. The Blend call takes the source and destination coordinates and height and width of a rectangle to be blended, with no option to fail. DoneBlend is called after a set of Blends, in case some teardown is necessary. Blend was first implemented for ATI Rage 128 (R128) hardware, but has since been disabled due to problems found using a Render extension test program.

The second new acceleration for Composite is to check for cases where the source is a 1x1 repeating picture and there is no mask, with the "Op" operation being "Src." This can be accelerated using the solid-fill driver hook, which is also required for Kdrive drivers. The source is converted in software into the X server's Pixel type and passed to the Solid hook. In contrast to other hardware acceleration, this hook prefers that the source be in memory rather than the framebuffer, because if it is located in the framebuffer the accelerator has to be idled before the value can be accessed by the CPU for conversion to the destination's pixel format.

The next new acceleration hook provided is a set of PrepareComposite/Composite/DoneComposite calls, where the only thing checked by KAA is that the source, destination, and the optional mask are located in framebuffer memory. The driver is responsible for checking everything else, including handling transform, repeat, and all the various formats. This is useful for newer hardware like the Radeon where almost all commonly-used Composite operations can be implemented in the 3D hardware. The Composite acceleration was first implemented for ATI Radeon 100-series hardware, and subsequently for ATI Rage 128 hardware.

Finally, a new UploadToScratch hook was added that takes two pixmap pointers and makes the second a copy of the first, but with the data located in a scratch area in card memory. This allows temporary migration of a pixmap for the case where a pixmap not being in offscreen memory is all that is preventing acceleration. This occurs frequently in drawing of glyphs, which are not stored in real pixmaps and therefore will not be migrated into offscreen memory normally. The previous allocation to the scratch area is invalid whenever a new UploadToScratch call occurs.

3.3. Offscreen Memory Management

Properly managing offscreen memory is a critical feature for an X server, and Render operations that read from the destination are making the problem more visible. This is because video cards typically have very slow framebuffer read speeds, making software fallbacks that result in the CPU reading from card memory very expensive. A test on the Rage 128 showed a 23% slowdown when writing to framebuffer compared to system memory, versus a 99% slowdown when reading from the framebuffer instead of memory, as seen in Table 1.

	read	***write***
R128 system	531MB/sec	247MB/sec
R128 AGP	14.4MB/sec	443MB/sec
R128 FB	5.11MB/sec	192MB/sec
Trident system	228MB/sec	160MB/sec
Trident FB	9.74MB/sec	15.9MB/sec

Table 1: Read and write speed to 512KB blocks of system, AGP, and framebuffer memory on a 700 Mhz Pentium 3 with ATI Rage 128 Mobility M4 and a 300Mhz Pentium 2 with Trident Cyber 9525/DVD.

The original Kdrive offscreen memory management system was very simple. A score is kept that marks whether a pixmap should be in framebuffer or system memory. Pixmaps with a size larger than a certain value (some heuristic) are moved into offscreen memory when allocated. When a copy operation occurs, the source pixmap has its score increased if the destination is in offscreen memory, or decreased if it is not. When an unaccelerated Composite operation occurs, the source and mask have their scores decreased, and when an accelerated Composite operation occurs their scores are increased. Other operations performed in software, such as core protocol text rendering, do not modify the score. If the score reaches the *move in* threshold, it is moved into offscreen memory if possible, replacing other pixmaps as necessary to do so. The replacement process starts at the beginning of offscreen memory and continues until enough space is freed (approximately -- it does not start moving out pixmaps until it locates a sufficient stretch without hitting locked offscreen areas). If the score goes below the *move out* threshold, the pixmap is moved back from offscreen memory to system memory. The scores are clamped to a minimum and maximum to make the migration more responsive to changing usage of a pixmap. The score values can be seen in Table 2, and it is not clear from CVS logs if the values are tuned or simply based on estimates.

Score name	***Value***
KAA_PIXMAP_SCORE_MAX	20
KAA_PIXMAP_SCORE_MOVE_IN	10
KAA_PIXMAP_SCORE_INITIAL	0
KAA_PIXMAP_SCORE_MOVE_OUT	-10
KAA_PIXMAP_SCORE_MIN	-20

Table 2: KAA pixmap migration thresholds.

This initial memory management system worked well enough for the initial goal of getting hardware acceleration between pixmaps and to the screen, but it has limitations. It does not migrate destination pixmaps that are being software-rendered towards system memory, which includes some significant core operations like text rendering and PutImage. It also has problems with thrashing. As soon as more pixmaps should be offscreen than there is sufficient memory for, choosing which pixmaps to keep in memory becomes an issue. First, when a pixmap is replaced by another allocation, it will not be moved back into offscreen memory until its score goes below the *move in* threshold and back above it again. If the *move in* process is changed to move any pixmap back in that has a score above the *move in* threshold, thrashing occurs when the first pixmaps in offscreen memory are chosen to be replaced each time. This state can be seen after anywhere from seconds to a few minutes of typical desktop usage.

To fix this, several things were changed. The first improvement was to modify the pixmap migration score to express whether there is more overall need for that pixmap to be in framebuffer or system memory, rather than how often it is used as a source image in a screen to screen copy to framebuffer versus system memory. This means that acceleration operations like Copy and the various solid operations were made to migrate both source and destination toward framebuffer. Also, the KdCheck* software fallbacks that are not called as a result of a failure of one of the acceleration operations were made to migrate the destination toward system memory.

The next change was to keep a new score in the structure for each allocated offscreen area. Unlike the migration score kept in the pixmap private structure, this one is meant to represent how important it is for a particular offscreen area to stay there. This new score is increased by a constant value whenever a pixmap gets its migration score increased. Periodically the scores of each offscreen area are reduced by a fraction so that the accumulated scores of offscreen areas decay over time. When a new area is to be allocated, the set of offscreen areas with the lowest total score is chosen to be replaced, rather than simply the first set of areas found. This system is not optimal, because a pixmap that gets replaced loses all of its accumulated score. Also, the score is not a very accurate representation of the value of a particular offscreen area being offscreen. However, it is sufficient to reduce the thrashing problem that existed.

The final change was to add a “dirty” flag to the pixmap private structure. Whenever the pixmap is modified, the dirty flag is marked, and it is cleared when the pixmap is moved into or out of offscreen memory. If the dirty flag is not set when a pixmap is to be moved out of offscreen memory, the expensive process of reading the pixmap from the framebuffer is skipped.

One failed experiment was based on the difference between the framebuffer and system memory speeds on the Rage 128. That experiment was to throw out clean offscreen areas when the migration score is decreased (when the pixmap is about to be used for software rendering), along with the usual move out process of pixmaps whose score goes below the threshold. However, the process of moving pixmaps in and out appears to outweigh the advantage of avoiding some software rendering to the framebuffer.

3.4. Issues With Direct Rendering in Kdrive

The Composite extension creates several challenges for implementing the DRI. The existing DRI drivers which we would like to use are designed only for a static front/back buffer and a front buffer which is actually visible onscreen. With the Composite extension, however, rendering may need to target a dynamically allocated buffer located in offscreen memory. Therefore the kernel's knowledge of the front/back buffer location (if it has any) must be per graphics context rather than per-device. Also, for a direct rendering context, the server must be notified in some way when the client updates its front buffer, so that damage can be computed.

3.5. Implementation of OpenGL in Kdrive

At the time of this writing, the work to implement hardware-accelerated OpenGL in Kdrive is incomplete. The first step was to bring in a software implementation of GLX. The GLX code was taken from XFree86 (actually DRI CVS, a separate repository for OpenGL development in XFree86), and the server build was modified to use a CVS checkout of Mesa (rather than including a copy of Mesa source code in the server tree as in XFree86). At this time, it is limited to only functioning when Composite is disabled, but it allows the XFree86 libGL to function.

Next, the server-side component of the DRI was brought in from XFree86 and the ATI driver was modified to initialize the DRM and submit its 2D commands through DMA. All that remains to get direct rendering in Kdrive on par with XFree86 appears to be debugging the ATI driver's initialization of the DRM.

3.6. XVideo

Implementing XVideo with the overlay scaler has several limitations. One limitation is that there is usually only one overlay scaler port, so only one video can be handled at once. It also prevents capturing screenshots, because the screenshot will include the color key instead of the scaled, converted video. Finally, the presence of the Composite Extension means that blending may occur over the video window's contents, so that the color key will not match and a blended color key will appear instead of the blended video.

The solution is to use the 3D hardware or front-end scaler to do the conversion and scaling of YUV data for XVideo. The Rage 128 and Mach64 have front-end scalers. The ATI Radeon and Rage 128, Matrox Gx00-series, and 3dfx Voodoo3+ should all support YUV data as textures (though the 3dfx older than the Voodoo4 may not be useful due to limitations on texture sizes). Video using the front-end scaler was implemented on the Rage 128. However, one problem with doing video scaling using the texture units is that there are fewer controls of the output. The overlay scaler typically has controls for brightness and saturation, while the Rage 128 front-end scaler and texture capabilities appear to only have a temperature (whitepoint) control and no brightness/saturation. Lack of brightness or saturation control may be possible to work around by combining the video with a secondary texture.

4. Results

The Render acceleration is by far the most measurable acceleration implemented as a result of this work. Even without support for all the operations necessary, the Composite acceleration on Radeon 100-series hardware tripled the speed of non-subpixel antialiased text rendering, according to x11perf's aa24text test (2x533Mhz Celeron, Radeon 7500). The Rage 128 Composite implementation showed the largest improvement, with over a fivefold improvement in non-subpixel antialiased text rendering, as seen in Table 3. It also greatly improved the perceived speed of using xcompmgr, which frequently uses Composite Over operations with a 1x1 repeating mask picture.

	aa24text	***rgb24text***
R128 software	14200/sec	11900/sec
R128 hardware	78500/sec	2550/sec

Table 3: x11perf results for antialiased text (aa24text) and subpixel-rendered antialiased text (rgb24text) on a 700Mhz Pentium 3 with Rage 128 Mobility M4, before and after hardware acceleration of Composite.

However, at the same time as non-subpixel text rendering speed increased, a near fivefold decrease in subpixel rendering speed was seen. This is because at least some of the operations necessary, particularly component alpha blending, are not supported by the Rage 128 Composite implementation, so the effort that goes into migrating the pixmaps (including UploadToScratch) to make them possible to accelerate is wasted. This suggests more offscreen memory management work is needed, to avoid that extra effort when it is not necessary.

5. Future Work

At this point, it needs to be decided if Kdrive is an appropriate server architecture for use as a desktop server. In particular, it lacks drivers for most hardware and lacks the support for control of video modes that XFree86 offers, though it now offers better 2D acceleration for Render operations and improved offscreen memory management. This section will describe future work to be done on 2D acceleration based on the new work in the previous section, and the

OpenGL work necessary to make Kdrive usable as a desktop server.

5.1. Render Acceleration

Even after adding the catch-all Composite hook, hooks for simpler operations such as Blend remain useful because they can be tailored to common hardware features. These hooks free driver authors from having to figure out the exact set of conditions under which they can accelerate, and from needing to manually migrate and pull out the data.

There is at least one more Render acceleration hook that could be useful. One common operation is to do a Composite with an ARGB source and a 1x1 repeating mask to blend opaque images over the destination with a constant alpha value. This could be done on hardware with only one texture unit or with a front-end scaler by putting the constant mask value in the hardware's primitive color registers instead of using a secondary texture map. KAA would synchronize the accelerator if necessary, pull the value from the mask picture, and pass it, along with the same arguments as Blend, to a new BlendMask hook.

5.2. Offscreen Memory Management

There are still many things that could be done to improve offscreen memory management. There are circumstances where a pixmap should be marked for migration out of framebuffer which are not being covered. However, the current system for measuring whether a pixmap should be in framebuffer is very crude, being based on the number of software versus hardware operations, rather than any measurement of the cost of those operations.

Also, there are serious weaknesses in how migration is being dealt with for operations that are going to fail (such as component-alpha compositing, or unsupported composite operations). Currently, we do not migrate operations failing in the Setup stage away from the framebuffer, which is a major speed penalty. If we do migrate failing operations away, we'll see flip-flopping of the migration. This is because for repeated rendering of the same failing operation, the pixmap will always moved toward the framebuffer until it is migrated, at which point the Prepare will start failing and it will start to be moved back toward system memory. The solution will most likely require adding a fourth hook that performs all of the checking of the operation before any migration towards framebuffer happens and changing the Setup hook to be non-failing.

The excellent write speeds to AGP memory suggest that it might be a good choice to use for the scratch area for the UploadToScratch hook instead of framebuffer when available, if the penalty for the card to read from AGP instead of local memory is not too heavy.

If accessing AGP memory from the card does not include much overhead, it may be valuable to set up hardware with limited offscreen memory to have a large piece of AGP memory addressable and use that as an extension of the card's local memory.

Another area to research would be working on a method to calculate an optimal working set of pixmaps to be located in offscreen memory, based on how often they are dirtied, how often they get read from and written to, size, and other factors. This could be a significant improvement over the current system of simply moving in whatever is necessary for the current operation. Designing a complete solution to this may be hard, and benchmarking varying implementations is difficult due to the lack of a standard general desktop usage benchmark to be run that would stress offscreen memory management.

5.3. OpenGL

The next step in developing quality OpenGL support is to fix the problems with Composite in the GLX implementation. When that is completed, the major piece of work is to convert the software GLX implementation to use a software-rendering DRI driver. This would basically be a wrapper around the same sort of software GLX implementation, but with a normal DRI driver interface. Once that is completed, it should be straightforward to replace the software-rendering DRI driver with a hardware-rendering DRI driver, thanks to work that has been done in Mesa CVS to produce DRI drivers that do not rely on the X protocol. The drivers should soon be ready for rendering to buffers other than statically allocated back/depth buffers because of work being done to support pbuffers in the DRI and Mesa projects.

5.4. XVideo

Despite being an improvement visually, the current XVideo acceleration is still a large consumer of CPU time when playing movies. The CPU usage may be attributable to the copying of data to the framebuffer, in which case using AGP memory when available may help.

Also, most overlay scaler implementations of XVideo allow control of saturation and brightness, while the current R128 XVideo does not offer these controls. More work needs to be done to see if these controls can be made for the textured video, either using the same scaler setup as for the overlay scaler, or using texturing features.

6. Availability

All work described here is available under the MIT/X11 license, and instructions for getting the code are available at:

http://www.freedesktop.org/~anholt/freenix2004/

7. Acknowledgments

Many thanks to Keith Packard for his guidance in understanding the Kdrive architecture and patiently explaining Render. Thanks also to Anders Carlsson for help in implementing parts of the KAA and Rage 128 Blend work. Thanks to Keith Packard, Carl Worth, and Deborah Anholt for many rounds of editing of this paper. Finally, many thanks to LinuxFund for sponsoring my work on Kdrive.

Bibliography

[1] Robert W Scheifler and James Gettys, *X Window System 3d ed.*, Digital Press. 1992.

[2] Keith Packard, *Design and Implementation of the X Rendering Extension*, FREENIX Track, 2001 Usenix Annual Technical Conference. http://keithp.com/~keithp/talks/usenix2001/xrender/, (2001)

[3] Keith Packard, *A New Rendering Model for X*, FREENIX Track, 2000 Usenix Annual Technical Conference. 279-284 (2000)

[4] Keith Packard, *The X Rendering Extension*, The XFree86 Project, Inc.

[5] Mason Woo, Jackie Neider, Tom Davis, & Dave Shriener, *OpenGL Programming Guide 3rd ed.*, (1999)

[6] Mark Vojkovich and Marc Aurele La France, *XAA.HOWTO*, The XFree86 Project, Inc.