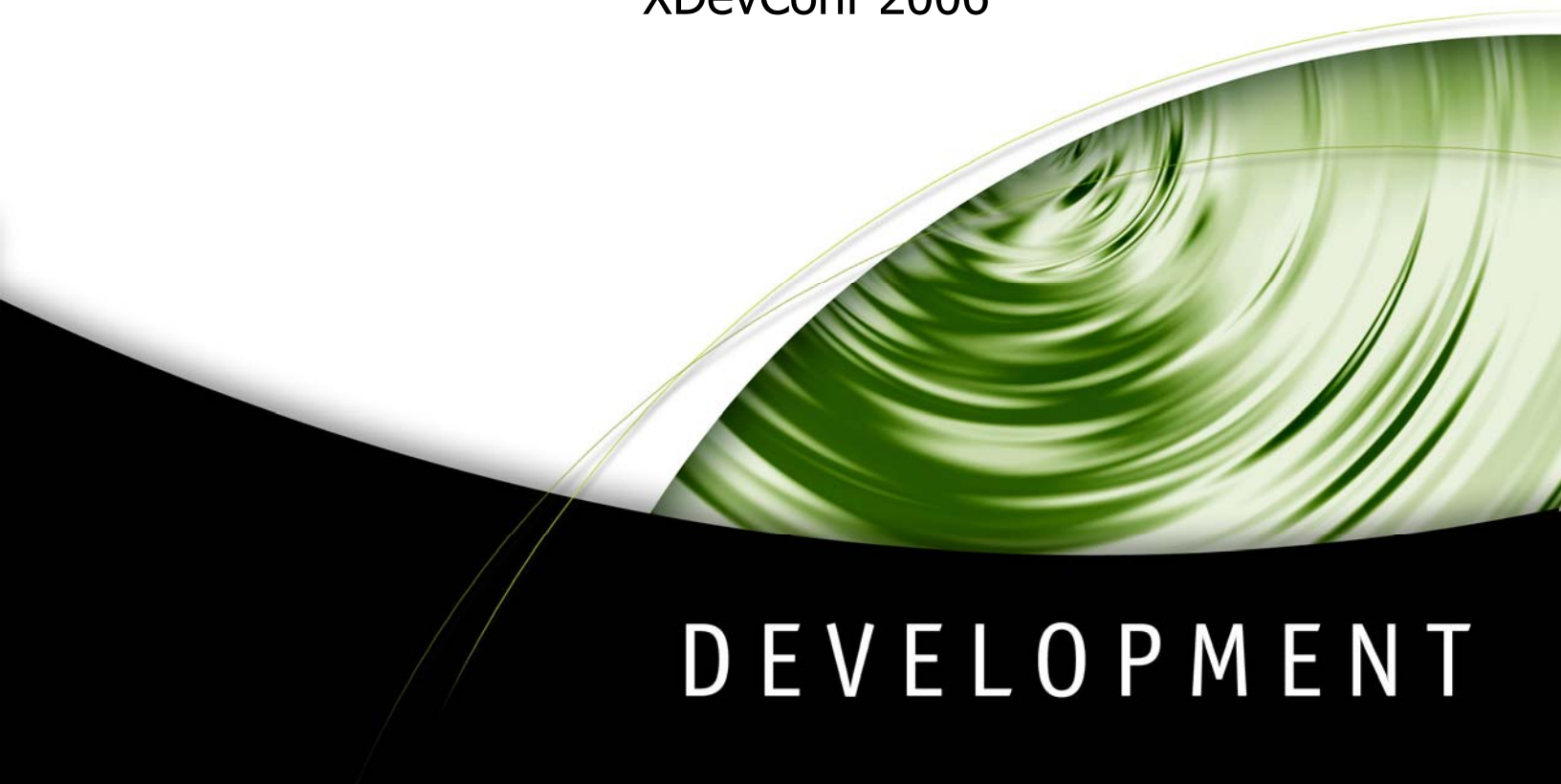

Whitepaper

# An Overview of the NVIDIA UNIX Graphics Driver

Andy Ritger

NVIDIA Corporation

XDevConf 2006



# Abstract

The NVIDIA UNIX Graphics Driver provides a wide spectrum of features and capabilities to users of NVIDIA hardware on Linux, Solaris, and FreeBSD. This paper provides an overview of the NVIDIA UNIX Graphics Driver, its components, features, and how it and how it coordinates direct-rendering OpenGL. Issues of ABI and API are discussed and relevant suggestions are made. Lastly, we describe how direct-rendering OpenGL might interact with a composited X desktop.

Andy Ritger
aritger@nvidia.com

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050

February 8, 2006

# An Overview of the NVIDIA UNIX Graphics Driver

The function of a hardware graphics driver is to provide interfaces by which programs that require computer rendering may leverage processing power of specially designed graphics processing units (GPUs). Rendering performed on dedicated graphics hardware is typically significantly faster than rendering operations performed on a computer's general purpose Central Processing Unit (CPU). Fast rendering is desirable for real-time simulations, gaming, computer-aided design, cinematic production, and to drive dramatic and compelling user interfaces.

The NVIDIA UNIX Graphics Driver provides a wide spectrum of features and capabilities to users of NVIDIA hardware on Linux, Solaris, and FreeBSD. Here we provide an overview of the NVIDIA UNIX Graphics Driver, its components, features, and how it coordinates direct-rendering OpenGL. We discuss various engineering challenges, current ABI and API issues, and resolutions to these problems. One of the newer compelling features of windowing systems is support for a composited desktop environment. We describe how direct-rendering OpenGL might interact with a composited X desktop.

## The NVIDIA UNIX Graphics Driver

The code base for the NVIDIA graphics driver leverages the majority of the source code across all the operating systems we support. Everything that is specific to a particular operating or windowing system is abstracted behind interface layers. In other words, the core OpenGL driver source code is used on Microsoft Windows, Linux, Solaris, FreeBSD, and Mac OS X. Similarly, most of the code for the NVIDIA kernel module is common across all of those operating systems.

NVIDIA's graphics drivers support all of its GPUs dating back to GeForce 2 MX [1]. Built-in hardware compatibility coupled with software to manage GPU differences allow us to maintain a single driver that supports several generations of GPUs.

NVIDIA's marketing uses the term "Unified Driver Architecture" to refer both to one driver for all NVIDIA GPUs, and one driver code base to support all of our operating systems.

The NVIDIA UNIX Graphics Driver is provided for:

- Linux-x86
- Linux-x86_64
- FreeBSD-x86
- Solaris-x64/x86

The UNIX operating systems and architectures we support are based primarily on the interests of our customers and end-users.

The NVIDIA UNIX Graphics Driver consists of the following components:

- A kernel module (nvidia.ko): responsible for all resource management, DMA setup, allocation of system memory and video memory, servicing of interrupts, thermal monitoring, GPU clock management, and all things related to managing of the GPU. All other NVIDIA driver components are viewed as clients of the kernel module's resource management.

  The kernel module consists of a binary-only portion and a kernel interface layer (aka "shim"). The source code to the shim is distributed with the driver and is the only portion of the kernel module that directly accesses OS kernel interfaces or even includes OS kernel header files. In the case of Linux and FreeBSD, this shim must be compiled specifically for the version and configuration of the user's kernel. In the case of Solaris, there is a kernel driver ABI, so one shim works across all kernel versions.
- An implementation of OpenGL (libGL.so): this library provides the API entry points for OpenGL and GLX function calls. It is linked to at run-time by OpenGL applications. The OpenGL library either performs direct-rendering to the GPU (through a DMA command buffer allocated on its behalf by the NVIDIA kernel module), or sends GLX protocol requests to a server-side GLX implementation.
- An X.Org loadable driver module (nvidia_drv.so): responsible for all X rendering, modesetting, configurability through the NV-CONTROL X extension, GLX visual/FBConfig management, and buffer allocation on behalf of OpenGL clients (both direct and indirect). As with libGL.so, the NVIDIA X driver submits its rendering requests to the GPU through a DMA command buffer allocated by the NVIDIA kernel module.
- An X.Org loadable module (libglx.so): provides an implementation of the GLX X extension. The libglx.so and libGL.so libraries share a common core rendering library for GPU programming to hardware-accelerate OpenGL indirect-rendering.
- Assorted utilities licensed under the GPL:
  - nvidia-installer: utility for installing, upgrading, and uninstalling the NVIDIA graphics driver.

- nvidia-settings: gtk-based configuration tool that communicates with the NVIDIA X driver via the NV-CONTROL X extension to dynamically configure various parameters of the NVIDIA driver.
- nvidia-xconfig: commandline tool for manipulating xorg.conf files.

These components are packaged together and distributed as the NVIDIA UNIX Graphics Driver.

# Features Provided by the NVIDIA UNIX Graphics Driver

The NVIDIA UNIX Graphics Driver provides numerous features.

- Hardware-accelerated direct and indirect OpenGL/GLX rendering.
- Support for TwinView: one X screen with multiple display devices (CRTs, DFPs, TVs) scanning out arbitrary portions of the X screen. There are several advantages to TwinView over traditional Xinerama (e.g., it is one root window in video memory -- all rendering spans all display devices "for free"). However, it breaks the "one display device per X screen" model and introduces some grey areas (e.g., what is the DPI when in TwinView? How to deal with different non-rectangular layouts?). All of NVIDIA's modern GPUs support multiple display devices from one GPU, and we find TwinView to be a very popular feature. Other drivers have similar support and call this "MergedFB".
- Support for multiple X screens on one GPU: this is an alternative to TwinView, in which multiple root windows can be allocated on the same GPU, and each display device can scan out a different one. Some people prefer this alternative because it alleviates some of the ill-defined corner cases with TwinView. Also, multiple X screens allow one to advertise different capabilities on each X screen (e.g., Workstation Overlays on one, Quad-Buffered Stereo on another, etc.).
- Support for direct-rendered OpenGL with Xinerama: OpenGL rendering spans and migrates between physical X screens as one drags the windows around the Xinerama root window. This is especially popular with high-end workstation customers driving power walls or CAVE labs.
- Configurability: the NVIDIA driver has a large number of options that can be controlled through X configuration file options. Many more options can be controlled dynamically through the NV-CONTROL X extension. NV-CONTROL clients, such as nvidia-settings, can query and set NV-CONTROL attributes to affect the behavior of the X driver and OpenGL.
- Quad-Buffered Stereo: another popular feature with workstation customers. OpenGL applications can separately render images for left and right eyes. This allows the driver to flip between left and right eye images on subsequent vertical refreshes. Stereo goggles are

synced with the driver's stereo flipping, so the user's left eye only sees the left image, and the user's right eye only sees the right image.

- RGB/CI Overlays: another popular workstation feature (utilized by applications like Maya[2]) is rendering to drawables in the overlay plane, which does not damage rendering already present in the main plane. This does not cause expose events to drawables in the main plane. Typically, user interface elements are drawn into the overlay while a complicated scene is rendered to the main plane. The user-interface elements can be changed without requiring re-rendering of the main plane's scene.
- FrameLock: NVIDIA offers several high-end workstation boards (Quadro G-Sync, and the older Quadro FX 3000G) to allow locking multiple displays together (on different GPUs, possibly in different systems), such that their Vertical Refreshes occur simultaneously. This requires use of the NV-CONTROL extension to configure the FrameLock hardware. Then the GLX_NV_swap_group extension can be used by applications to request that applications running on different systems in this FrameLock group have their swaps locked together. This is a compelling feature for powerwalls and CAVEs.
- SDI: another set of high-end specialty workstation boards, NVIDIA's SDI products allow users to output GPU data over SDI (Serial Digital Interface) a video format commonly used in the broadcast industry. SDI can be 8, 10, or 12-bit per component, and requires special repacking of pixels by the NVIDIA driver. There is also a GLX_NV_video_out extension to allow applications to send their pbuffers directly to the SDI output.
- SLI: this feature allows multiple GPUs to be joined together to drive a single X screen. SLI can be configured such that each GPU renders horizontal swaths of each frame (Split Frame Rendering), or each GPU renders alternate frames (Alternate Frame Rendering), or each GPU renders a differently sampled version of the scene, and the results are filtered together for improved AntiAliasing (SLIAA).
- XVideo, XvMC: The NVIDIA X driver provides various Xv adaptors and an XvMC implementation for hardware-accelerated mpeg decode.

The above features require special support from the X driver beyond the traditional X rendering and modesetting responsibilities of an X.Org loadable X driver module.

# How a Direct-Rendering Client Interacts with the X Window System

In traditional X rendering, X clients send protocol to the X server to request a rendering operation, and the X server performs that rendering on behalf of the X client. In addition to network transparency, another benefit of that model is synchronization, all rendering requests are serialized through the X server. Rendering commands are submitted to the hardware in the order that the X server processes the X protocol requests.

The GLX X extension (the interface between X windows and OpenGL), operates using the same paradigm: a GLX client sends GLX protocol requests to an X server that advertises the GLX extension, and the X server will perform the OpenGL rendering on behalf of the client.

However, common OpenGL usage patterns suffer from the inter-process communication overhead of sending GLX protocol to the X server. Often, OpenGL applications will manage very large quantities of data, such as textures, or will make many API calls (e.g., glVertex3f() for each vertex in a scene with millions of vertices per frame). In these cases, transferring large quantities of data or making many protocol requests is suboptimal.

When the GLX client is running on the same system as the X server, it is desirable for the OpenGL client-side library to optimize rendering by submitting rendering commands directly to the GPU. This is called OpenGL direct-rendering, in contrast to indirect-rendering where GLX protocol is sent to the X server and the X server submits the rendering commands to the GPU [3].

It is worth noting that the idea of direct-rendering OpenGL is nothing new. Direct-rendering OpenGL has its roots in the UNIX workstations of the 1990s. See [4], [5], and [6] for more historical perspective.

Additionally, it is worth distinguishing between OpenGL hardware-acceleration and direct-rendering. Hardware-acceleration refers to using the GPU to perform some or all of the OpenGL rendering pipeline, while direct-rendering is as described above: the OpenGL client library bypasses the X server and submits rendering commands directly to the GPU. NVIDIA's UNIX Graphics Driver supports hardware-acceleration of both indirect and direct OpenGL rendering.

An OpenGL direct-rendering implementation must coordinate closely with the X driver and server-side GLX implementation in the X server for several reasons: the direct-rendering client must be told where to render and certain synchronization must be performed to ensure that the direct-rendering client only renders where it is supposed to when it is safe to do so.

## Data Propagation

Data describing a drawable's geometry, clips, and other attributes must be propagated from the X server to the OpenGL direct-rendering client.

The ValidateTree(), ClipNotify(), and PostValidateTree() pScreen procs are wrapped so that we can track changes to drawables to which OpenGL is rendering. These changes are processed and fed (in the NVIDIA case, through shared memory) to the OpenGL direct-rendering clients.

Because OpenGL direct-rendering clients are running asynchronously to the X server they cannot be notified directly of window change events. Instead OpenGL clients will look for drawable updates when performing operations that must be done with an up-to-date view of the window system. If the drawable data in the shared memory segment is newer than the data OpenGL has, OpenGL will retrieve this new data and update its state accordingly.

Note that OpenGL rendering, like all rendering in X, is clipped to the pixels within the visible portion of the window to which rendering is targeted.

## Synchronization

In NVIDIA's OpenGL driver model, each direct-rendering OpenGL client fills its command buffer asynchronously to the X driver and any other OpenGL clients. When those rendering commands are ready to be submitted to the GPU, the OpenGL library locks out the X driver from changing any relevant windows, checks that OpenGL is up-to-date with the drawable data in the shared memory segment, and finally submits the rendering commands to the GPU.

Appropriate synchronization must be performed to ensure the integrity of the data getting propagated through shared memory (e.g., OpenGL cannot read the data from shared memory while X is half-way through updating it). Additionally, synchronization must be performed to ensure correct ordering of GPU rendering commands between the NVIDIA X driver and OpenGL.

A traditional GPU has a single command buffer which all graphics drivers (X, OpenGL, etc) must share. This sharing requires synchronization to ensure that different drivers do not trash each other's state and that all writes to the command buffer are atomic.

In contrast, NVIDIA GPUs accept input from multiple command buffers (each graphics driver gets its own command buffer). The hardware context switches between command buffers and virtualizes the rendering resources. This eliminates the software problem of synchronizing a single command buffer, but the tradeoff is the need for the drivers to coordinate to ensure correct ordering of GPU commands when necessary.

For example: consider the case of two OpenGL clients, each rendering to separate windows, the order that the GPU processes each OpenGL's command buffer is not critical (the order that the two OpenGL clients' rendering becomes visible does not affect correctness). However, in the case of a window move, the ordering between the X server's commands to move the window contents and OpenGL's rendering commands does matter, so that the correct OpenGL rendering lands properly in the new window position. If this is not handled properly, OpenGL rendering could scribble on pixels no longer within the bounds of the visible portion of the window.

NVIDIA GPUs have various means to perform inter-command buffer synchronization. The details of those means will not be described here, and is an implementation detail that should be transparent to the rest of the system. However, the concept is important, because it is relevant to cases in which one client's rendering is read by another client. Examples of this include: direct-rendering OpenGL clients rendering to redirected windows, or X rendering to pixmaps that are used as textures through the proposed GLX_EXT_texture_from_pixmap GLX extension.

Whether the OpenGL direct-rendering client is rendering to an on-screen window, a pixmap, pbuffer, or redirected X window, the same principles of data propagation and synchronization apply.

# Interactions between Rendering and Scanout

When a double-buffered OpenGL application has completed rendering to its backbuffer, it calls glXSwapBuffers() to copy the content from the backbuffer to the visible frontbuffer. With NVIDIA hardware, this can be accomplished by "blitting" or by "flipping". While a blit is effectively a memcpy, flipping is a technique in which the scanout hardware in the GPU is pointed at a different region of video memory. To flip, we scanout from the backbuffer, making it the new logical frontbuffer, and what was the front buffer, is now the backbuffer. Flipping is more efficient than blitting, so it is preferable to do so whenever possible.

It is worth noting that current NVIDIA GPUs can only scanout contiguous regions of video memory. This means that we can only flip the entire desktop, not just windows within that desktop. Thus, to support flipping of windowed applications, we use unified buffers and keep all of the desktop content the same between buffers, such that only the OpenGL content within the window is different between buffers. There are clearly performance tradeoffs to be made here.

While performance may be the only motivation for flipping with basic OpenGL SwapBuffers, it is a necessity for other functionality. Quad-Buffered Stereo, for example, requires that the GPU flip between left and right images on every Vertical Refresh.

Flipping is an example of the grey area between rendering and scanout. It is a common misconception that rendering and scanout are orthogonal. In an ideal world this would be true. In practice, there are many areas of overlap:

- The rendering driver needs to control what portion of video memory is scanned out in order to support flipping.
- Syncing OpenGL rendering to Vertical Refresh requires coordination between scanout and rendering.
- Whether a video memory allocation will be scanned may impact how that video memory is configured and formatted.
- Filtering for anti-aliasing can in some cases be performed by scanout hardware for better performance; these situations require coordination between scanout and rendering.
- SLI (SFR, AFR, SLIAA) introduces complex interactions between rendering and scanout.
- Frame delivery for video is very time-sensitive, and precise control in the driver of when to display the frame is crucial; this is often accomplished with flipping.

# Video Memory

Though modern graphics cards usually are packaged with large quantities of video memory, there are caveats: not all video memory may be accessible by the CPU. We can quickly exhaust limited BAR1 space allocated by the SBIOS, so we typically only map a smaller

portion of video memory. Ideally, we would map buffers in video memory individually and map those buffers on demand, to conserve BAR1 space.

Some newer NVIDIA GPUs support rendering to system memory over the PCI-E bus. Marketing has branded this "TurboCache". This memory is CPU mappable, but GPU rendering is slower because it has to travel across the PCI-E bus.

The layout of video memory may not be linear. The organization of bits within video memory is optimized for cache use for both rendering to and texturing from video memory. A video memory buffer that is setup for hardware rendering/texturing may require expensive work to transition such that the buffer can be mapped linearly and rendered to by the CPU.

When transitioning from GPU rendering to CPU rendering, synchronization must be performed such that all GPU rendering is complete before the CPU can read from that memory. Thrashing between CPU and GPU rendering is undesirable due to the overhead of this synchronization.

There can be many different attributes to the memory used by the graphics system. Selecting the optimal placement of data in the correct memory space with the correct memory layout is non-trivial and can be best performed when the driver has knowledge of how that data is going to be used.

# ABI Compatibility and API Compatibility

NVIDIA provides a single X driver that our customers expect to work with any version of the XFree86 or X.Org X servers they choose to use. Thus, we depend on ABI compatibility and dynamic loading of symbols that were added in newer X servers. For over 5 years, XFree86 and X.Org have not broken ABI compatibility, with only a few minor exceptions.

We understand that there are reasons that the ABI may need to be broken (such as making the maximum number of X Screens dynamic, eliminating the MAXSCREENS definition) and from our perspective that is fine. NVIDIA can update its X driver or provide multiple copies of the X driver built for different ABI versions as necessary.

Here are a few suggestions for ABI compatibility, though:

- Breaking ABI compatibility is not only painful for 3rd party vendors such as NVIDIA, but also anyone trying to distribute a driver separately from the X server tree. It was our understanding that with the Modular X tree, separate distribution of X drivers would become more prevalent. If that is the case, then breaking ABI compatibility will be painful for OS distributors and all driver maintainers. Therefore, we recommend breaking the ABI infrequently and only when absolutely necessary.
- In many cases, it may be preferable to add a new entry point and deprecate the old one rather than simply changing the old entry point. This gives driver maintainers an opportunity to phase in the new support.
- Be sure to change the ABI version number appropriately when altering the ABI.

- Please make it possible at both run time and install time for a 3rd party vendor to query the ABI version.
- Please try to minimize how often the ABI is broken, and thus minimize the number of incompatible driver versions that need to be maintained by driver maintainers. If there are several ABI breakages pending, it may be beneficial to put them in the tree at roughly the same time.
- If the ABI is going to be broken or extended anyway, it is worth designing a new API that properly solves the problem. Several cases that come to mind are glyph management and the Xv interface.

# OpenGL and the Damage and Composite X Extensions

The Damage and Composite model works well for compositing windows that were rendered to by X (see [7] for background on the Damage and Composite X extensions). However, it introduces new challenges for direct-rendering clients such as OpenGL and XvMC. See above for a detailed description of how a direct-rendering client fits into the X window system.

Direct-rendering clients will need to know when their drawable has been redirected. They will need to notify the X server when they have damaged the drawable, so that the X server can then notify the composite manager who can then request that drawable be composited back into the desktop.

There are also synchronization issues between the graphics channel performing the direct-rendering and the graphics channel used to perform the compositing.

These topics have been discussed in more detail in this email thread:

```
http://lists.freedesktop.org/archives/xorg/2004-May/000607.html
```

To the best of our knowledge, as of this writing, no vendor has shipped a direct-rendering OpenGL implementation that can interoperate with the Damage and Composite extensions. The closest has probably been Xgl [8], though in that case all OpenGL rendering is indirect.

These are not impossible problems to solve, but vendors need to solve these problems within their drivers. NVIDIA intends to provide an OpenGL implementation that can interoperate with Damage and Composite.

The overhead will be noticeable for a direct-rendering OpenGL implementation to synchronize with the X server so that OpenGL content can be properly composited. Most severely impacted will be applications with a high frame rate, as well as applications that render to the front buffer. There will also be increased latency between when OpenGL rendering is performed, and when that rendering finally becomes visible when it is composited into the desktop.

One of the most important things about the Damage and Composite model is that it provides the flexibility to disable the composite manager so that windows are no longer redirected, and no longer require the extra step of being composited into the desktop. This is very important for many workstation users who require full performance OpenGL, and value this full performance over the special effects possible through Damage and Composite. It is our hope that composite managers provide a way to dynamically disable compositing so that users can make the tradeoffs themselves between full performance OpenGL and a composited desktop with a modest performance impact.

Additionally, it is not immediately clear how features like Quad-Buffered Stereo or Workstation Overlays interoperate with Damage and Composite. It is possible that NVIDIA will not be able to support stereo flipping when the stereo drawable has been redirected to off-screen memory. Workstation Overlays with Damage and Composite would likely require additional support in the X.org X server, if not also in the composite manager. Users of Quad-Buffered Stereo or Workstation Overlays may need to disable the composite manager (if not completely disable the Composite extension) when they require these special workstation features. More investigation is needed in this area.

# Conclusion

The NVIDIA UNIX Graphics Driver provides a wide spectrum of features and capabilities to users of NVIDIA hardware on Linux, Solaris, and FreeBSD. This paper provided an overview of the NVIDIA UNIX Graphics Driver, its components, its features, and how it coordinates direct-rendering OpenGL. Issues of ABI and API were discussed and relevant suggestions were made. Lastly, we described how direct-rendering OpenGL might interact with a composited X desktop.

# End Notes


[1] The NVIDIA unified graphics driver supports all NVIDIA GPUs, though we have phased out support for our oldest GPUs, due to maintenance and testing costs. The oldest GPUs are instead supported through a separate dedicated "legacy" driver branch, which does not pick up new features, and only receives fixes for critical bugs that affect those legacy GPUs.

[2] Autodesk. Online Resources: Maya, February 2006. http://www.alias.com/glb/eng/products-services/family_details.jsp?familyId=3900009

[3] There can be other direct-rendering clients besides OpenGL, such as XvMC. This paper focuses on OpenGL, but the same direct/indirect-rendering model applies to any direct-rendering client.

[4] Mark J. Kilgard, Simon Hui, Allen A. Leinwand, and Dave Spalding. *Multi-rendering for OpenGL and PEX*. X Technical Conference, January 25, 1994.

[5] Mark J. Kilgard, David Blythe, and Deanna Hohn. *System Support for OpenGL Direct Rendering*. Graphics Interface 1995.

[6] Mark J. Kilgard. *The Silicon Graphics X Server, IRIX 4.0.5*. The X Journal, SIGS Publications, January 1993.

[7] Andy Ritger. *Using The Existing XFree86/X.Org Loadable Driver Framework To Achieve a Composited X Desktop*. XDevConf 2006.

[8] David Reveman. Online Resources: Xgl, February 2006. http://www.freedesktop.org/wiki/Software_2fXgl

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com