Proposal for Open Source Graphic Systems v1.3

Written by Brian S. Julin (2004)
Prepared by Nicholas Souchu (2005)

History of modifications

Revision	***Author***	***Page***	***Description***
1.0	N.Souchu	All	copy (formated under OpenOffice.org v1.1.3) of the original proposal of Brian S. Julin
1.1	N.Souchu	P5	Add 5) in layer 1 needs
		Most	Formating
1.2	N.Souchu		
1.3	B.Julin	Most	Cleaned up grammar, clarified some areas, and added verbiage on implications for highly parallel CPU cores.

1.Introduction

The following proposal is not presented as an immediate goal for Open Source graphics systems. The author fully expects that most readers will deem it to be prohibitively difficult to implement, and full of open questions and omissions. It is, rather, presented as look at the general direction towards which all Open Source graphics systems are in fact, slowly evolving. Consider it a work of extrapolative fiction... it may very well be the case that nothing like this is ever realized, but parts of it likely will be in one form or another. The author's objective in presenting it as such is merely to provide a common framework for discussion among Open Source developers.

One of the major problems facing Open Source development of graphics systems is the confusion caused by minced terms. Words like "screen," "display", "device", "viewport" and "mode" have very specific, sometimes rather divergent, meanings in existing solutions and proposals. Often this is the result of functionality being attached to a concept simply because it exists as an opportune code object, without regard to the resulting semantics. As such, their use is avoided where ambiguous, and a new vocabulary is adopted. If nothing else comes of this work, the author hopes this new vocabulary can be used to elucidate discussions on the general topic of graphics systems development. (This new vocabulary appears typographically as underlined text.)

Much of this proposal is, in concept, not very original. Ideas from various Open Source projects and discussion forums that the author has worked with or lurked around for several years are incorporated. Thus there are actually many contributors, both knowing and unknowing, to this proposal -- too many, in fact, to list or even recall. The author must settle for extending his thanks to all those who take the time to apply serious thought to the complex problem which faces Open Source in developing a graphics hardware system appealing to the development community.

The three main sections that follow describe a different “layer” of hardware graphics systems. Each starts with a prelude in a question/answer format. The reader is encouraged to consider each question on his or her own before reading the suggested answer. The answers attempt to justify some judgment calls made by the author, who would appreciate hearing any alternative perspectives. Building upon the issues raised therein, the section continues by describing the purpose of the layer and the concepts and structures necessary to support an implementation of the desired behavior.

2.Copyright and License

Table of contents

3.First Layer

What's needed?

Q. What should the behavior of a system be when a monitor is moved from one port to another?

A. Most users do not expect applications to follow monitors, but rather to be tied to a monitor port. As a practical consideration, moving applications from one GPU to another is extremely complicated, even when the GPUs are of the same make and model. The answer here is clearly that applications remain tied to the monitor port, not to the monitor.

Q. What should be the behavior of the system when a monitor is removed and replaced with a different monitor?

A. Users will want this to be a fairly dumbed-down process, so we should attempt to allow applications to run uninterrupted during this process. Users will expect that display of the application will be restored when the new monitor is detected, and will expect that they can run applications with the monitor detached without suffering a pause. That's just the way computers have always worked in everyday situations and people have come to consider it normal behavior.

From the user's perspective, this seems very simple, and in naive implementations it is, more or less. A mature system takes less for granted, and in so doing deals with complexities that more primitive systems ignore. Specifically we have the following situations to deal with:

Q. What should happen if the new monitor cannot support the refresh rate of the application that was running on the monitor it replaced?

In this case the resolution should be preserved, but the refresh rate should be reduced. Advisory feedback to the application(s) must be delivered. Some applications will simply ignore this as they do not engage in any more complex real-time management than simply waiting for refreshes. Others, like 3D games or video multimedia, may need to adjust internal scheduling parameters. Some may wish to reduce resolution in order to preserve the ergonomic properties of the display. As impact on the application is likely to affect only performance, not impact usability, advisory feedback should be sufficient.

An inferior alternative would be to switch graphics consoles to an application more compatible with the new monitor, disabling display of the incompatible application until user intervention is performed.

Q What if the new monitor does not support the full resolution which the old monitor was providing to the application?

In this case, we should attempt to still display as much of a subarea of the running application as we can. Advisory feedback must be sent to the running application(s). We might consider providing a user with an emergency method to advise the application to pause/resume.

Q. What if the previously running application is just plain incompatible with the new

monitor?

(This could happen if the new monitor places restrictions on other factors such as bitdepth that make displaying the application impossible, or we have to reduce resolution but the graphics chipset is old and cannot display a subarea without affecting the GPU operation, because of interdependency with CRTC registers.)

In this case, a message should be displayed on the monitor indicating that the system is aware of its presence, but that the application must make adjustments before it may resume displaying. Before displaying such a message, however, feedback must be sent to the running application(s). If the application cannot continue to run while the message is displayed, we wait for a certain interval for the application to acknowledge that it has ceased accessing the graphics system. If that acknowledgment does not come in a reasonable amount of time, mandatory feedback which pauses the application is applied, and will not be lifted until the situation is resolved. Then we display the message. (On some systems, this process shares some of the same infrastructure as graphics console switching.)

If it is impossible to determine the characteristics of a new monitor, we fall back to a safe mode and propose a list of modes commonly supported by monitors (VESA modes for example) and the application is notified in the following sequence: first that it is no longer being displayed while the user chooses its mode, second that it is displayed in a different mode/bitdepth. If the user cannot find a correct mode for its application, the system can propose to either: fall back to a different application on this monitor with the current settings (holding the application in “not displayed” status) or kill the application.

Q. What if the application cannot even cope with the removal of the first monitor?

It may be impossible to place the monitor port into a safe mode, and/or to perform monitor detection on the port, without affecting retrace handling for the underlying application (or again in the case of older hardware where there is interdependency between GPU and CRTC/DAC registers, affecting the GPU operation.)

We proceed as above, advising and then halting the application, before we begin probing for new monitors. Note the time constraints here need to be chosen such that the application gets as much time to react as possible, but without giving the user too much of an opportunity to damage the new monitor by exposing it to an incompatible signal. 1 to 2 seconds seems a reasonable default.

Layer 1 Summary

Layer 1 is concerned with initial and runtime hardware (re)configuration and resource inventory. It provides the higher layers with notification of reconfiguration events, handles ancillary communications with monitors, and brokers all requests for access to resources such as VRAM and GART mappings. (The last point was not addressed in the above Q&A.)

Author's Note: In this section, I intentionally skip the whole "bus abstraction" discussion as far as enumerating devices and how such enumeration is presented to the user or system administrator. I of course have some opinions on this, but the discussion for some reason always seems to become a distraction. Besides, it represents a policy decision that has implications far outside the graphics driver system. Normally the below would also include such material, but suffice to say, whatever mechanism or structure is decided upon to handle initialization and hotplug, and whether this happens in user or kernel space: chipset-specific code must be located, that code must examine the characteristics of the hardware, it must create instances of the below objects as appropriate, and any information needed by the user or system administrator must be made accessible to userspace.

Overview of layer 1 objects

Valet

A "valet" object represents a type of storage or aperture that is made available for assignment to the kernel or application, examples being VRAM, MMIO aperture, and GART. Valets are responsible for keeping track of the amount and organization of each resource available, fielding requests for allocation, and programming underlying hardware to tie together aperture(s) and storage into working address spaces.

Sink

A "sink" object represents a monitor port on a graphics device -- not the monitor itself, the port it plugs into. (In the case of laptop LCDs the sink is still there, it's just not physically accessible to the user.) There is one sink per port, whether or not the port shares any resources with other ports. Sinks have many responsibilities described in more detail below.

Clock

A "clock" is an optional slave object of a sink which is simply a way to modularize support for common clock chipsets that are found on video hardware with different driver codebases.

Monitor

A "monitor" object represents a physical monitor (CRT, LCD, etc.), and that is meant precisely. If you take one physical monitor off of a sink, and plug another one in, the second physical monitor is not represented by the same monitor instance. In this respect they are handled similarly to a simple USB device -- they can be moved from sink to sink and, when feasible, the instance data can be made to persist and follow the physical device. The monitor object's primary responsibility is to prevent the physical monitor's electronics from being overdriven by the sink.

Prism

A "prism" object represents the circuitry that converts framebuffer data into signals that can be output through a sink to a monitor. This includes RAMDACS and ratiometric expanders. The prism object's primary responsibilities are to protect these electronics from being overdriven, and to provide service to any sinks to which a signal path can be configured.

Lut

A "lut" is a lookup table contained in hardware, usually for palette or gamma-mapping. Instances of lut objects are used in various places in the proposed design. On level 1, however, a lut object is a simple accessory object to a sink and represents a final translation, usually performed in the RAMDAC, of framebuffer pixel values into output levels.

Description of valet operation

The valets on their own represent a separable half of layer 1 functionality. Many valets will be nothing more than thin wrappers around existing OS facilities, and only serve to unify the calling conventions of these services.

An operating system will have several valets each serving different types of system resources, and a few more valets serving different types of resources from each piece of graphics display hardware. The below list enumerates several types of valet instance that will be found on many systems. Not all of the valets in the below list are mandatory -- if the developers of an OS decide not to offer one or more of the below valets, that is their prerogative. Some valets, however, are essential to support certain graphics hardware in certain modes of operation. The below list makes note of these cases.

1) The "**fragmented system RAM**" valet can be used to allocate pages of physical system RAM that are not necessarily consecutive. As requesting such an allocation generates not only the pages themselves but also a list used to keep track of them, this valet is usually used in combination with another valet which absorbs the list to create a consecutive mapping. The type of RAM allocated may also be influenced by what valets are used in combination with this one, for example, using this valet in combination with the "PCI GART" valet would cause the allocation to be taken only from areas of RAM that can be made accessible to the PCI bus for bus-mastered transfers.

2) The "**contiguous bus DMA RAM**" valets are used to allocate physically contiguous blocks of RAM that are visible from a given system peripheral bus. There would be one for each DMA-capable peripheral bus present in hardware. These valets must be provided if it is desired to support cards that have DMA functionality but which have no address remapping facilities like PCI-GART or AGP-

GART. Sometimes single pages are sufficient for these purposes. As such, even when an OS does not support allocation of contiguous pages in these ranges, it may elect to define a valet which serves only single pages.

3) The "**motherboard GART aperture**" valet provides a contiguous mapping for RAM acquired from another valet into the physical address space. This valet is required if support of high performance operation of AGP cards is desired.

4) The "**process virtual address space**" valet provides a contiguous mapping for RAM or VRAM acquired from another valet into the virtual address space of a given userspace process. This valet is required.

5) The "**execqd virtual address space**" valet provides a contiguous mapping for RAM or VRAM acquired from another valet into the virtual address space of the process and/or kernel routine where GPU command buffers are serialized for output to the GPU. (The term “execqd” will be explained in the third section.) This valet is pretty much required.

6) The "**cache control**" valets are used to apply special flags, such as MTRR ranges, to given areas of storage. The use of write-through can greatly reduce coherency issues and the workarounds needed to address them, so these valets are strongly encouraged.

7) The "**GPU VRAM**" valets provide allocation of on-board video RAM storage for a specific graphics hardware component; there is one for each separate component. These must be provided in order to access any memory on a given piece of graphics hardware, so they are pretty essential.

8) The "**GPU MMIO aperture**" valets are companions to each of the "GPU VRAM" valets and provide mapping of VRAM into the physical address space such that it is visible to post-translated CPU accesses. This may include MMIO registers, or a separate "GPU MMIO aperture" valet may exist for such registers. (The two might then be used in combination in situations where the register API can be moved around inside the GPU's main MMIO aperture.)

9) The "**GPU GART**" valets provide a peripheral-side mapping of contiguous or fragmented system physical RAM addresses into the GPU address space, though there may be GPU-specific restrictions on the use of GPU addresses so created. This valet is necessary in order to support high performance operation of PCI-Express cards, or the high-end of PCI-based graphics cards.

Inside level 1, valets are passed as arguments to a set of functions that resemble object methods. These methods would include attempting to allocate storage/aperture in a form closely resembling most OS page and mapping allocators, temporarily deactivating and reactivating storage/aperture to allow overlapping allocations to time-share a resource, and query functions about remaining storage and allocation restrictions. These functions are not intended to be used outside of level 1.

(A level 2 object called a "lot" is used as a bidirectional communications mechanism between layers 1 and 2, and is created simply by requesting a storage allocation from a group of valets simultaneously --

how the storage and apertures are tied together is inferred from the list of valets used, and thus this level of detail is secreted in level 1.)

Description of sink and prism operation

The second separable half of layer 1 centers primarily around the "sink" object, which ties all the other objects together. The sink object, along with a global table called the "signal switching table", implements the back end of the layer 2 API which allows layer 2 access to all of the following functionality:

1) hotplug support as it relates to monitor insertion/removal

2) GPU and monitor power saving system support

3) ancillary monitor communications

4) video signal routing

5) retrace feedback when not supported through accelerator

6) mode setting including ratiometric expansion

One of the most complicated, but essential, structures of this proposed design is the signal switching table. This table is populated based on information discovered about the hardware by bus detection and probing, and defines the relationships present between individual prisms, clocks, luts and sinks.

Note that it is possible for a prism to be capable of driving multiple sinks at once, for example when using a laptop with an external monitor and displaying the same image on both the LCD and the monitor. It is also possible for a sink to choose between the signal generated by more than one prism, for example a TV-out port that can choose between framebuffer and hardware decoder. The association table fully defines which prisms may be routed out of which sinks, and the restrictions surrounding doing so.

Since many chipsets interface to DDC, palette data, clock chips, and other facilities through the RAMDAC, the association table must also note whether exclusive access to the prism is needed by a sink in order to perform:

A) detection of monitor removal/insertion

B) inquiry of monitors (e.g. via DDC)

C) programming of clock objects

D) programming of lut objects

E) power savings functions

F) ancillary monitor communications

This is to prevent use of such functionality from interfering with normal operation of the prism when it is in use on another sink. On more recent setups (flat panel) monitor setup has started to become detached from the prism, or at least made independently accessible, so this problem is mitigated.

Description of monitor operation

In addition to the signal routing table, Layer 1 maintains information about attached monitors and administrative policies relating to monitors. This service is supplied through three main constructs:

1) **The monitor pool**:

The monitor pool is intended to provide policy control by the OS, and persistence of state for reattached monitors, when desired.

The monitor pool is a list of monitor instances. Monitor instances in this pool are considered not to be attached to any sink. They can be added to the pool in one of several ways:

A) Defaults pre-installed for kernel boot.

B) Installed by the user/OS.

C) Orphaned instances returned from a sink after physical monitors are removed.

Certain attributes of a monitor instance pertain to how they may be used when they are located in the monitor pool. The monitor may be compared with probed data in a manner similar to that used, for example, by input-linux to locate matches for (re)attached monitors. The relevant attributes are:

A) Whether the instance can be garbage collected, and an associated TTL value.

B) Whether the instance is an orphan, a template, or a blacklist.

C) What attributes of the instance should be used to determine if it matches a newly detected monitor (including don't-cares and inverses)

D) A "preferred" sink may be specified.

When a new monitor is instantiated by a sink, the monitor pool may be searched for a best match. This is done according to the following ordered set of rules:

A) Blacklist pool entries preferring other sinks are ignored.

B) Other blacklist pool entries are tried. If any of them match, the monitor is considered unsuitable for use.

C) Orphans that prefer this sink are searched. If any match is found, it is removed from the pool and reused.

D) Templates preferring this sink are tried. If a match is found, a new instance is created by copying the template.

E) Other orphans are tried. If a match is found it is removed from the pool and reused. The preferred sink is changed to the current sink.

F) Templates not preferring any sink are tried.

G) Templates preferring other sinks are ignored.

2) **The timing cache**:

A global list of monitor timings is used to cache results, which are truly a menial detail not worth explaining in depth. It serves only to speed up the process of finding a suitable mode.

3) **The monitor communications routine**:

The monitor communications routine is a periodic or IRQ driven daemon, tasklet, or whatever fits the bill best. It iterates across a list of callbacks registered by sinks. These callbacks handle the following tasks:

A) Detecting monitor removal/insertion

B) Performing inquiry on new monitors

C) Communicating power savings requests to the monitors while they are in use.

D) Performing any other communications with the monitors.

The monitor communications routine must be able to identify situations, via the signal routing table, where multiple sinks cannot probe or otherwise communicate simultaneously due to shared hardware issues. It is responsible for time-slicing the sink communications traffic and/or blocking the use of a sink while a contending sink is in use. When a communications task will affect an application, the monitor communications routine must signal up to the higher layers and wait for them to perform any necessary housekeeping before proceeding.

The monitor communications routine may elect to play games with using the sinks as a source of IRQs, or not -- it may suffice to use a simple periodic task or process. Sinks must be able to communicate their real-time needs and capabilities to the monitor communications routine.

Since any of the above tasks may be performed while an application is using the sink, solutions which allow the communications to be interleaved with application access are to be preferred (for example, though DCC keep-alive might appear nice for detecting monitor removals, a simple query of the sense pins register is often less entangling.) When this is not possible, the monitor communications routine is responsible for providing feedback to the higher layers, which will in turn pause the application's graphics access if necessary, allowing the monitor communications routine to proceed.

Each sink object is endowed with a monitor communications policy which is used by the monitor communications routine to determine the actions performed by a sink to detect monitor insertion and removal. It is user configurable, but is initialized to conservative defaults.

Settings contained in the monitor communications policy include:

1) For each possible detection method: pin sense, DDC, DDC2, etc.

A) Whether or not to use this method

B) Whether to search the orphans in the monitor pool when detection is successful

C) Whether to remove orphans from the monitor pool after use or just use them as a template to copy from

D) Whether to consider the monitor usable if no match is found among the orphans.

E) Various method-specific parameters (timeout intervals, etc.)

F) Whether to orphan monitors to the monitor pool after they are removed.

2) Parameters controlling how aggressively the system tries to preserve the ability of an application to remain running throughout a monitor swap, including such things as whether ergonomics are a higher priority than application transparency.

3) A list of alternate graphics applications which may be given control of the graphics hardware automatically if the current application cannot be accommodated by a new monitor. This list is set based on system security policies or explicit user preferences, and is relayed via layer 2.

4) A flag that forces re-detection/inquiry when a policy is changed. For example, this allows a monitor which was configured with a conservative default policy to be reconfigured through more advanced means.

A very conservative monitor communications policy and monitor pool will be enacted as a default. This will cause detection to use only simple pin sense, and will only instantiate a copy of a default monitor template with extremely strict, hard-coded, timing values that are deemed to be the safest possible. (Note that if fossil-ware like EGA, 8514, and MDA is to be supported, different templates are needed.)

(A less conservative policy/pool, which is advised for use by the operating system post-boot when operating with no information about expected/supported monitors, is to allow DDC detection, but not to ramp up DDC to full bit rate at first -- use a known safe video mode and only enter high bit rate after it is determined that the monitor can tolerate it. However, administrators are welcome to customize this behavior at their leisure/peril, which may be desired for quick response with KVM switches, for example.)

4.Second Layer

What's needed?

Q. Should an application's framebuffer VRAM be preserved when it is "switched away" or "loses focus"?

A. In an ideal world, yes. In reality, this would use far too many resources to be practical, and too few applications really need such support, since they can either redraw or skip frames. As a compromise, this design proposes that it be possible to lock an application's framebuffer into VRAM when enough resources are available to do so. However, support for automatic shadowing of framebuffer contents into RAM, while not obstructed by the proposal, is not advised. Those rare applications that must preserve computationally expensive framebuffer content should handle their own back-buffer system.

Q. Can and should an application that is running in the background, not displayed on any monitors, still be allowed to access the GPU and run as if it were displayed?

A. Depending on the hardware, it is possible to run multiple graphics applications through the GPU even if only one of them is displayed. Primarily, the hardware would have to have a clean separation between the CRTC and the GPU such that the programming of the CRTC does not affect the operation of the GPU (most newer hardware does.)

Even when MMIO and/or GART hardware has limitations on the number of regions it can map into physical memory space, multiple applications could be given non-intersecting slices of the overall mapping in their virtual address space, thus ensuring that they cannot trivially access each other's data areas.

This should only be done if all of the applications can fit all their resources into the VRAM and GART at the same time. There is really no benefit for allowing storage areas to be shared via time-slicing because restoring the contents of framebuffer, texture cache, etc, is not an operation that should be done at every context switch. Furthermore, reprogramming GTLBs to allow GART address space to be shared between two private RAM areas is also not an operation well suited for context switches.

When considering multiple GPU contexts, this is almost identical to the case where multiple “direct accelerated access” applications utilize the GPU in a cooperative suite like X11. The only difference is that the number of registers which must be restored when switching from one GPU context to another will be greater, as they may need to alter the base definitions of the location and layout of the framebuffer, not just 3-D drawing engine registers.

The proposed design takes this option into account and implementation of this scenario is provided for by the ability to simultaneously commit graphics resources to different applications. Whether support for running background applications should be implemented on a given OS, however, is a decision best left to the OS developers.

It is the position of this paper that even if the OS developers decide against this, there is no benefit to be gained by crippling the proposed design; the general structure should be preserved and no shortcuts taken in this respect such that the structure is the same as it will be on other operating systems which do decide to implement such support, for code sharing purposes.

Q. How do we treat multi-headed cards -- like one desktop with a few monitors, or as separate framebuffers capable of running an application on each monitor?

A. Both. By separating the management of buffer space from the management of sequencers/monitors, the proposed design adopts a new conceptual model that allows for more flexible control of multi-monitor cards. It is not necessary to decide exclusively on one or the other paradigm for multiple monitor use, only to consider the effect on any “virtual console” system provided by the OS, since changing which application is displayed on a multi-display desktop must provide notification to all affected applications. (The OS console system is considered merely a user, albeit a special one, of the proposed design, and the needs of all flavors of console systems are accommodated.)

Layer 2 Summary

Layer 2 ties layer 1 components together into a simpler model more resembling today's conventions. It hides most of the complexity surrounding monitors in layer 1. It also defines the method by which layer 1 valets create simpler, more usable, objects. Graphics applications that do not use any acceleration could conceivably use layer 2 concepts exclusively, as they are sufficient to define a simple framebuffer and set a video mode.

Overview of Layer 2 objects

Lot

A "lot" is some quantity of storage or address space allocated using the "valet" objects from level 1. It hides all unwanted details about system resource allocation.

Spot

A "spot" is a subarea of a lot. This is not necessarily a contiguous area, as a spot may have a strided memory layout. A spot also serves as a indication (from the application to the graphics subsystem) as to the intended use of the underlying area -- e.g. as framebuffer data, or texture data, or overlay data, or GPU commands. In most cases, this data is merely informational unless layer 3 "proofreading" (described in the next section) is in use, though it could otherwise be potentially useful in resolving contention between cooperative applications in a suite.

Slide

A "slide" is just a spot that has additional information as to higher level access alignment restrictions. For example, a spot can tell you where textures can be sourced from, but it cannot tell you the fact that certain kinds of textures must start on a 32-byte boundary or that they have size restrictions. This extra information is mostly advisory in the case of textures, since card drivers should know this information already, but it could be useful to non-driver-specific graphics memory management code in the application. Slides are also used by "lens" objects for tasks such as panning/wrapping of the main framebuffer by the console system.

Lens

A "lens" represents the ability to take the contents of a spot or slide and send them to a given combination of prisms/sinks. A lens defines a subarea of a spot or slide, within the restrictions imposed by the underlying object, and controls any expansion, transformation, or peripheral-side positioning performed on the data contained in this subarea. Thus, a lens is the primary type of object used to manipulate the CRTC offset, turn on pixel doubling, position cursor or video overlays, tweak video scaling/sub-sampling, etc.

Special lens objects exist that can initiate power saving states -- feel free to call these "lenscaps" if you cannot resist doing so.

Description of lot operation

A lot object is obtained from layer 1 valets by specifying the size of the required resource. The attributes of the returned storage are inferred by the combination of valets specified. The lot object must be able to provide enough information about the underlying storage to allow the application and/or the higher layers to utilize the storage, but other details about the storage are opaque.

A lot object need not necessarily represent a permanent allocation of resources. For certain usages, like framebuffer, it may be desirable to designate a set of lots of which only a limited number may be active at a time, thus sharing limited resources. A lot provides for temporary deactivation of its claim on underlying resources. In a security sensitive environment, this includes disposing of any data contained in the resource before turning it over to other lots which share the resource. For maximum flexibility, a lot also provides a relocation mechanism – through which the actual address of underlying resources may be reassigned when it is reactivated. (Usually this will only happen to the GPU-side address.)

The mapping of which lots may be simultaneously active is contained in a "lot link table" which also contains details to handle relocation. Since such a complex system may not be desirable, it is not required in this proposal that lots actually be dynamically relocatable or even capable of sharing resources, so this table is optional (access to the table is not done directly, but through a functional interface centered around the lot object.) However, in the interest of code sharing it is encouraged that, when used from the application or from the higher layers, lots be treated as though these features are available, and checked for activation failures, status changes, or relocation before use.

Description of spot/slide operation

The owner of a lot is allowed to divide it up into spots and slides as they deem fit. In most cases, the lower layers do not even need to know about the intended usage of the subareas of lots -- they either have no implications for security, or security mechanisms can deduce whether security is maintained simply by examining the commands sent to the GPU. Most of the information needed for security is provided by the fact that sensitive usage areas like GPU command buffers require different storage configurations (e.g., must be mapped through the exeqd valet) and as such, are separated into different lots.

However, such advisory information from the application can make security related code significantly less complex. Whether to implement advanced security features is a decision left to OS developers, but for code sharing purposes, it is strongly encouraged that spots and slides be “requested” through an access API by the application and/or higher layers. On systems which do not require this information, the API should simply ignore the proferred advice and signal success back to the caller. Thus spot/slide instances are maintained on the application or higher layer level, and may or may not be shadowed at layer 2 depending on the supported features of the given OS.

Description of lens operation

Lenses are used to tie together layer 1 objects into a working display. This can be as simple as selecting a single sink to handle a fixed framebuffer and presenting it's lut to the application for manipulation. However, lenses may also combine a larger set of layer 1 objects into a more sophisticated system.

For example, a lens might control two sinks, one representing the LCD of a laptop, and the other representing an attached CRT. The attached CRT usually supports many more resolutions than the LCD. The lens defines the behavior desired when an application tries to change the resolution. This is a policy decision -- the LCD could be blanked when large resolutions are in use, or the application could be told that the only resolutions supported are those supported by both the CRT and the LCD.

Multiple lenses may also share certain attributes in order to link some behaviors of independent sinks. For example, two lenses might bond the lut objects of their sinks together causing palettes or gamma-maps on both lenses to behave as one.

As such, lens objects are registered into a "lens link table", similar to the signal routing table on layer 1, which defines the relationship, if any, of each lens to the others, and assigns prisms,sinks, spots and slides to lenses.

It should be mentioned that the "spot", "slide", and "lens" objects are sufficiently abstract to extend their usage into the application area for other facilities such as controlling the properties of video overlays. However, this proposal is not concerned with such activities past the point where the application requires privilege elevation. Though the concepts and possibly even the object constructs themselves may be of use for these purposes, it is not suggested in this proposal that any kernel code or superuser daemon be required to actually use them for such purposes, with the possible exception of virtual console systems utilizing a hardware cursor feature.

5.Third Layer

What's needed?

Q. How do multiple applications using the GPU accelerator work together without interference?

A. There are three ways to do achieve this result, and this proposal allows OS and driver developers to decide which of the methods are worth the effort to support. This decision should not me made lightly, as it has a major impact on the requirements imposed on the much larger user application codebase.

The first way is to only allow one application (like an X server) to talk directly to the GPU, and for all other applications to use a protocol (like X11) to request drawing operations. However, the introduction of an abstract API at the "command pipe" level obviously presents obstacles that prevent applications from realizing the full potential of underlying hardware, and introduces latency and extra resource utilization into the system.

The second way is to allow only one application to access the GPU at a time, but to allow the applications to be sent a signal causing them to pause. When resumed, the application is sent a second signal which informs it that it is necessary to restore any settings which it is counting on the GPU to retain. This method is suitable only for environments where these switches between applications do not happen very frequently (for example, a virtual console system), as the latency incurred in the handshake between applications and the graphics system would severely hurt performance were the handshakes to occur too often. It also requires applications to be capable of fully restoring the settings of GPU registers.

The third method is to "virtualize" the GPU such that several GPUs are emulated in software. In this solution the graphics system must automatically save and restore state when different virtual GPUs are used. Several suggestions have been made as to how this system would work. It has been suggested that each process be given a virtual GPU "context", and also that each thread be given a virtual GPU "context.” This design asserts that tying the GPU context to either the process or thread is not necessary and creates unwanted entanglement with the operating system. Moreover, even a single-threaded process may have a need for multiple virtual GPUs. In this case, the simplest solution also happens to be the most flexible solution: when a graphics application needs to send commands to the GPU, it specifies which virtual GPU the commands belong to. This only requires of applications that they hold on to GPU context identifiers, and use a specific mechanism to introduce commands for processing -- a much easier modification to preexisting code than either of the above.

Q. In what format should an application send commands to the graphics system?

A. There is a big clash between two paradigms involved in this question, which has resulted in years of vacillation and discord in the Open Source community.

One paradigm is that of presenting applications with an abstracted API for graphics commands. In this school of thought, applications should send commands in a standard format regardless of the hardware being used.

Another paradigm is that of presenting applications with virtual access to the raw hardware. This implies that applications will supply driver code for the hardware and pre-format the commands in a form native to the hardware.

The first paradigm is criticized highly for creating "yet another API" – a middleman where none is needed, since application libraries like OpenGL already provide a standardized interface for common graphics tasks. The second is criticized because it allows the application too much latitude, requires a larger codebase, and makes it difficult to virtualize the hardware for multiple applications.

Most often a hybrid system is invented as a compromise. For example, as of this writing, in Linux DRM raw graphics commands are used for drawing primitives, but vector tables are loaded through an abstracted API.

This design asserts that the second paradigm should be applied wherever possible, and the first should be avoided, for the following reasons:

> 1) Using the native hardware graphics language is what many existing codebases do, either directly or through a userspace driver library like OpenGL, and as such translating code to use any new API hinders the process of porting that code to the new graphics system. While the first paradigm does not require application-side driver code, the fact is that that code is already written, and in fact, the likely result of creating a new API is that a pseudo-driver for that API will have to be written for popular graphics libraries, and those libraries will continue to maintain application-side drivers anyway, so there will still be massive levels of code duplication.
>
> 2) Driver-side decoding of native graphics commands sent by the application, when necessary, is rarely more difficult than creating native graphics commands from parameterized abstract APIs, and for some purposes can actually be more optimal. The difficulty and required code size of virtualization of raw format command FIFOs has been somewhat exaggerated.
>
> 3) Mixing the two paradigms in a hybrid system results in situations where synchronization between API-based functionality and raw command functionality is necessary. Such synchronization often must occur across a less than optimal boundary such as a context switch, and could even tangle up with task scheduling under some models.
>
> 4) Abstraction, while a tried and true technique for many other purposes, often leads to orphaning of hardware and hardware features which do not fit the model used by the abstractor. In this case it is misapplied.
>
> 5) If required, extra functionality that engages non-graphics system facilities can just as easily be accessed from within a native GPU command stream simply by using locally-significant extensions to the native graphics command language – a simple escape code in the native command stream can be followed by OS-local coded commands. There is no absolute need for an all-inclusive API in order to utilize such facilities.

As such, this proposal suggests that all communications from the application to the GPU, other than that first used to initialize access to GPU resources and to de-initialize them after use, be done through command buffers containing a locally extended version of the native command language which encapsulates any special requests that require interaction with other system facilities. The only exceptions made are for bidirectional communications, which are only really needed for synchronization mechanisms in common usage.

Q. Should all graphics commands be sent to the GPU by the kernel, directly by the application, or by a userspace daemon?

A. Allowing individual applications free access to sensitive GPU registers creates prohibitive obstacles to extending the graphics system gracefully into the properly virtualized form that is so badly needed on modern operating systems, and is in general a threat to overall system stability and security. There are good arguments for and against both of the latter two options.

A keystone argument against a centralized userspace daemon is that of the latency incurred in communications between applications and the daemon. However, this argument is significantly diminished in some OS environments by optimizations in adaptive and real-time scheduling.

Another argument is that for some functionality like handling IRQs, kernel-side code is essential. However, a hybrid system where only those aspects that must be handled kernel-side are so located is certainly feasible.

A compelling argument against placing functionality entirely in the kernel is that it drastically increases the quantity of graphics driver code which must be implemented kernel-side, and that constitutes a lot of very complex code to be developed and maintained in an environment that many developers are afraid to even touch.

The proposed design recognizes that this decision must be made on a case-by-case basis by developers of the OS. As such, it does not demand either approach. The design refers to an exeqd "process" but this is not an implication that the routines performed are necessarily performed in userspace, it is simply a name for the ongoing job of serializing and dispatching GPU commands.

Certainly, some level of kernel support is needed in order to expose GPU facilities and to share them between the application and exeqd. As such, the decision on what functionality to place on which side of the user/kernel boundary can affect performance and system flexibility, most notably:

1) For what purposes various types of RAM and address spaces (for examine, high RAM) can be used, or the complexity and latency involved in using a given type of RAM or address space for any given purpose. Such considerations need also include the real-time requirements of TLB implementations on both the host architecture and inside the GPU.

2) The overall system latency and response to real-time events such as low-water marks and raster retrace.

3) The latency incurred during powersaving events, context and console switches, and administrative operations, as resources are reconfigured while the GPU is in use, and the complexity of the locking systems involved.

4) Incurred codebase complexity when highly entangled functionality is split across the boundary.

The astute reader will notice that we referred to two "separable halves" of layer 1, which can be taken as a hint that all the monitor related code would be an easy candidate for placement in userspace, whereas the "valets" are more likely to be implemented in kernel space. This proposal does not expect the division of kernel and userspace code to necessarily occur along the prescribed boundary between levels 1, 2 and 3.

Summary of layer 3

Layer 3 is responsible for the delivery of packets of commands to the GPU. Virtualization of the GPU can be accomplished at this layer through context tracking and scheduled serialization of packets from multiple sources. Layer 3 may also optionally support a "proofreader" that can inspect all commands before they are allowed to enter the GPU, which can either simply warn about suspicious commands for debugging purposes, or can render moot any commands that threaten to compromise system security or stability.

Overview of layer 3 objects

Exeq

An "exeq" is a buffer used to store GPU commands. These commands can be generated by an application, by objects in the lower layers, or internally inside the exeqd "process". An exeq need not be stored in lot accessible to the GPU, but if not then the contents must be copied into a second exeq owned by a lot that is so accessible before being sent to the GPU.

Exeq context

An "exeq context" object is the “handle” that virtualizes access to the GPU, making it appear to the entity that employs it as though it has exclusive access to the GPU, with two exceptions. First, the virtualization does not extend into the area of memory management, which is effectively virtualized on a per-process basis by the layer 2 lot objects. Many exeq contexts can share the same memory layout. Secondly, hardware (a.k.a. "shadow") context features are reserved for internal use by the graphics system, to make this virtualization more efficient (emulation of these may be considered as an option.)

Control block

A "control block" is a buffer used to pass status information back and forth between the exeqd "process" and the application regarding a set of exeq objects. This includes applying various flags to the exeq objects which affect their behavior, assigning exeq objects to exeq contexts, and an advisory latch scheme for synchronization between the exeqd "process" and the application.

Description of exeq operation

The exeq object is chosen usually to be the native architecture page size. For large operations that must be atomic in respect to the GPU, multiple exeqs may be linked together.

The format of the data contained in an exeq is that of the native graphics command language associated with the underlying hardware, possibly with a few extensions provided to access GPU facilities which are not included in the command set, or possibly even system facilities useful in conjunction with the GPU. Most GPU command formats leave ample room for such special commands to be embedded as otherwise invalid or moot commands.

The process of extending a native GPU command set should not be undertaken lightly, both due to the time required to parse the exeq, and due to the coherency issues raised by mixing GPU commands with other functionality. (On the latter point, their presence may force an exeq to be subdivided, or possibly even force a full idle-down of the GPU, to ensure serialization of the extended commands with the truly native commands.) Since the control block format is also extensible (not to mention more standardized in format than the native command stream) a driver should consider that as an alternate avenue for some functionality. An example of a good exeq extension command is one that deals with display synchronization issues (e.g. tagging a set of commands to take place immediately after vertical raster retrace) since a GPU idle may be required in this case anyway.

When multiple applications or threads share an exeq, they must take care to perform their own scheduling/locking to serialize the use of the exeq. There is no locking service offered by the graphics system, only advisory latching.

Description of control block operation

A "control block" contains descriptors for a set of exeq objects, and these descriptors can be seen by both the application owning the exeq objects and the exeqd "process.” It is a two-way communication line that provides fast updates on the status of exeq objects, and as such a fast mechanism like a shared page of system RAM is recommended for this use. On SMP systems, write-through caching can also be of help in the control block to reduce latency.

The core functionality of the control block is to implement a latch system that provides advisory synchronization. As a lowest common denominator, this can be implemented in the most trivial manner through a simple two-owner latch -- before data in each control block datum is considered valid for use by the exeqd “process” the application must write a 1 into a designated bit in that datum. Before the application writes into a datum in the control block, it must read a 0 from that bit. The application never writes a 0 into designated bits and the exeqd "process" never writes a 1 into these bits. As such, even in SMP environments with caching enabled, this advisory locking scheme ensures the integrity of data in the control block as long as no two of these bits occupy the same atomically modifiable datum. Of course, more efficient schemes are possible, and the OS is free to employ them. However, in order to preserve compatibility with this lowest common denominator solution, a bit of each atomically modifiable datum in the control block should be reserved for this use.

Each control block contains descriptors of a certain fixed size, chosen by the driver author. Standardization of the actual format of parts of these descriptors is certainly possible, and likely desirable, but is something that should be evolved with time and experience. As such, this design suggests that we only specify the behavior of one datum (actually only a few bits) in the descriptor, and leaves the rest up to the driver authors for now.

The first datum in each descriptor is used as the primary "trigger" -- it is the first datum that is inspected to find a status change in the buffer (in the designated bit), only after which are the remaining data examined. Some of the possible data sent from the application to the exeqd "process" are detailed in the following list, some items of which could be candidates for immediate standardization.

1) The "trigger" indicating that the exeq should be sent to the GPU

2) The exeq context in which this exeq should be run

3) The number or length of commands contained in the exeq

4) A starting offset into the exeq at which commands start

5) The index of the next descriptor in a GPU-atomic exeq chain

6) The level of content inspection to perform:

   A) whether to emit debugging information/stats (and at what level)

   B) whether to implement security mooting

   C) whether the exeq contains extended commands that must be interpreted

   D) whether/how the exeq alters context, for GPU context tracking

7) Sequence numbering to guarantee ordering of multiple exeqs

When multiple applications or threads share a control block, they must take care to perform their own scheduling/locking to serialize the use of descriptors. There is no locking service offered by the graphics system, other than the advisory exeqd latching system.

Description of exeq context operation

Virtualization of the GPU, and thus the exeq context object, is optional, however, without such virtualization the sharing of the GPU by multiple applications can only be achieved through cooperative application-side mechanisms outside the scope of this proposal. (The “option” of implementing a non-virtualized system is presented in order to allow dedicated-use embedded systems to use a simple codebase while still maintaining compatibility with the overall design.)

An exeq context is created for each virtual instance of a GPU. In order to properly virtualize a GPU, it is necessary to ensure that the state of all registers in the GPU are stored/restored when an exeq from a different exeq context than the last one used is sent to the GPU. There are many ways to do this. Which ones are feasible depends on characteristics of the GPU hardware, and as such, the method used is chosen by the author of the driver code for a given chipset.

Firstly, some GPUs support multiple contexts through the use of "shadow" registers or through saving/restoring state in VRAM. While this feature can be used to speed up the process of saving and restoring context, it is sometimes the case that the number of hardware supported "shadow" contexts is limited, or that some register subsets are not handled by this mechanism. As such these hardware features will often only be useful as an optimization for one of the subsequent techniques.

A trivial way to approach this problem is to idle the GPU before an exeq context switch and read its register contents into the outgoing exeq context object, then restore the register contents from the incoming exeq context and restart the delivery of exeq objects to the GPU. Though the proposed design allows for this simple approach, there are two problems with it:

> 1) GPUs may contain write-only registers which cannot be read back out, making it impossible to save and restore some values in such a fashion.
>
> 2) Idling the GPU causes an introduction of latency into the command stream, and will limit performance in situations where exeq contexts are switching back and forth rapidly.

A second method is to inspect all commands before they are sent to the GPU, and calculate changes to the register values from the inspected commands, storing them in the exeq context. This offers the advantage of eliminating the step of reading register values back out of the GPU. However, this method also has some serious disadvantages:

> 1) The work of inspecting the commands requires processing resources which can significantly slow down the overall performance of the system.
>
> 2) The creation and maintenance of the code necessary to inspect the commands can represent a fairly large workload for the driver author.

It is the position of this proposal that the second method, once mitigating techniques are applied, is superior. The subtler methods are discussed towards the end – more brutally, the first factor above can be mitigated over time by teaching applications to separate context-altering commands into different exeqs, and use the advisory system described above in the control block objects to cause only those exeqs to be thoroughly inspected. That would not be a huge chore for many applications developers since setup of less frequently altered registers is quite often tucked away in discrete areas of the application code.

(It is also possible, though cretinous, for the chipset driver author to simply demand of the application that some of the more commonly altered context be considered local to a given exeq, and unconditionally restored by the application using commands at the beginning of each exeq. This would be a good way for a partially complete driver to be alpha-tested with a custom application/library, for example.)

Furthermore, when security mooting or debug proofreading is turned on, the second factor is mitigated since the process of inspecting commands is performed unconditionally on all exeqs. When combined with proofreading/mooting, the cost of context tracking is insignificant.

The two methods above can also be combined in a hybrid approach. For example, this would allow the second method to be used to handle write-only registers, while other registers are read back out of the GPU as per the first method.

A mature, optimized, driver might elect to divide the register state into sections or perform a differential comparison of register states to limit the amount of restoration work needed. In addition, since entire subsets of registers can be made moot in hardware by switches in other registers, the driver could elect to restore the content of these subsets only when those switches are turned on. Finally, when default register values assigned at the initialization of a GPU are factored in, untouched register sets might be ignored by the driver. The resulting "lazy context" system could reduce the overhead of exeq context switching considerably.

Another optimization is for the exeq context to store register state in a preformatted exeq which can be dumped directly into the GPU. Such an exeq could be stored in VRAM for fast access by the GPU. Preformatted exeqs containing commands to read the values are usually possible, and in some cases the GPU can even store the results in VRAM directly.

When taken together, the above methods offer the potential to retain decent graphics performance even while performing emulation of a fully virtualized GPU. The tradeoff is code complexity versus performance, which is a good tradeoff to have because it lends itself to gradual development of optimized code while full functionality is available from the outset. Or to put it in more famous words, we can "first make it work, then make it work well."

Description of the exeqd "process"

As stated above, this design does not mean to imply that the exeqd "process" is necessarily a userspace daemon, which is it has been carefully referred to this way up to this point. With that clear, it will just be referred to as the exeqd in this section.

The exeqd handles the actual delivery of GPU commands contained in exeqs from the application to the GPU. Depending on the flags assigned (via the control block) to the exeq, it may also delete or alter the commands for security or debugging purposes, or may perform extra functionality not available in the hardware's native command set.

The exeqd distinguishes between three kinds of exeqs:

1) The first kind of exeq is a "command request exeq" which contains GPU commands written by the application. These exeqs must be mapped into the process virtual address space of the application, as the process must be able to write commands into them. They will also be mapped into exeqd's virtual address space, as exeqd may need to copy or inspect the buffers.

2) The second kind of exeq is a "command dispatch buffer" and contains finalized GPU commands to be sent to the GPU. These exeqs must be mapped into exeqd's address space, as exeqd must be able to write commands into them. When the GPU is being fed by DMA or AGP, these buffers must be mapped into appropriate hardware address spaces as well.

3) The third kind of exeq is an "internal command request exeq" and is generated from within exeqd

itself, or by objects in layer1 or layer2 to inject asynchronous events into the GPU command stream (including, most importantly, GPU context switches.) These exeqs are used, rather than direct access to GPU hardware registers, because doing so relieves several tricky synchronization and coherency problems that would occur were access to be performed directly on GPU registers through another route. It also makes exeqd a central choke point where almost all access to the GPU(s) must pass, which can make debugging much easier by keeping stray IO primitives out of the rest of the code.

The proposed design allows for, but does not require, OS developers the option of implementing a 0-copy mechanism. If the OS can be configured to do so with a net performance gain (avoiding much TLB and GTLB work) then command request exeqs may be sent directly to the GPU, bypassing a copy into a command dispatch exeq. This can only happen if:

1) These command request exeqs are also mapped into the appropriate hardware address spaces the same way as command dispatch exeqs are.

...and either...

2a) Security censoring of command request exeqs is not enabled or is used in a merely advisory role.

...or...

2b) A mechanism for allowing exeqd to prevent applications from writing to a command request exeq between the start of inspection and the end of GPU execution can be efficiently implemented.

It is the position of this paper that even if OS developers decide against this, there is no benefit to be gained by crippling the proposed design; The general convention of treating "triggered" exeqs as non-writable and non-readable until exeqd flags them as finished should be observed by application code, even though the system may be purely advisory. No shortcuts in userspace code should be taken, such that the structure is the same as it will be on other operating systems which do decide to implement such support, for code sharing purposes.

As exeqd lumps together callback functions registered from multiple drivers, it will mostly be a simple scheduler and a collection of convenience functions helping those drivers to perform their own exeq management. As the format of control block descriptors becomes more standardized, more functionality can be extracted from the drivers and provided as part of exeqd.

Author's note: Although this proposal remains flexible as to the prospect of placing exeqd in userspace, OS developers are encouraged to consider the future direction of graphics computing, specifically with regards to highly parallel architectures such as the upcoming Cell Broadband Engine. One could surmise that systems where a dedicated mainboard CPU core handles graphics, or where the GPU supports enough of a general instruction set to itself be the host processor for exeqd, will become more prevalent in the future. As such, strong consideration should be given to kernel-space implementation.

6.APPENDIX