A key based framestore for the Desk Area Network

Ian Pratt

April 1995

Abstract

This paper describes the two concepts behind the design of the Desk Area Network Framestore, namely key based pixel protection, and virtual windows. The former enforces window clipping in hardware, dispensing with the need for a server on the datapath of graphics operations. 'Virtual windows' is a technique that allows digital video coming from devices such as the network interface, video grabber or even CPU, to be 'piped' direct to the framestore, rather than having to go via main memory as in conventional systems.

1 Background

There is a continuing quest to build multimedia workstations, that can display high quality video and audio, while performing more conventional computational tasks simultaneously. Displaying high quality video requires a high bandwidth path through to the frame buffer, something that has challenged the capabilities of many I/O busses, leading to the acceptance of higher performance busses such as PCI. Just being able to get the bits across the bus is not the whole story. In many systems all graphical operations must pass through a graphics server, the only process authorised to access the frame buffer. Its job is to provide a clean abstraction of the display hardware, and enforce clipping to ensure drawing operations can not stray from the visible part of the target window. Although very suitable for displaying computer generated graphics, these servers can impair the performance of high frame rate video display.

2 Displaying video through a graphics server

Traditionally, graphical applications have performed drawing operations by making calls to a server, such as the MIT X Server, or the Microsoft Windows API. The drawing operations can often be expressed quite compactly, in the form of commands to draw lines, rectangles, print text etc. The server is responsible for performing drawing operations on behalf of the client, clipping the operation to constrain it to affect only those pixels of the clients window that are 'visible'. Clipping is usually performed by checking the bounding box of the operation against one or more rectangular regions that describe the exposed part of the client window. Drawing operations that require clipping are broken down into a number of smaller operations that do not.

On a conventional memory mapped framestore, the server will then perform the drawing operation by rendering into the frame buffer memory. Some 'smart' frame buffers are equipped with specialist graphics processors. These typically accept 2 or 3D drawing commands, and render them to the local frame buffer. This can reduce the required bandwidth between the CPU and the video device.

Although the idea of having all drawing commands performed by a trusted server on behalf of clients seems attractive for conventional applications, a number of problems come to light when attempts are made to display full motion video. The foremost problem is bandwidth. Whereas the drawing operations used by most applications can be described compactly, video requires that the actual data to appear in the window be communicated to the server. The server must then clip and transfer the data to the frame buffer. 2D graphics accelerator chips can not help to reduce the bandwidth in this instance.

The MIT X Server has shared memory extensions to enable the data to be sent from client to server without extra data copying. The fact that the data is being sent to a server rather than being rendered directly effectively introduces a data copy anyway; The client had the pixel value it wanted to write to the frame buffer, but had to store it to memory so that the server could read it back then perform the write on it's behalf. This data copying operation also results in the displacement of data that could more usefully be stored in the processor's memory caches.

The per operation overhead of invoking the server can be quite significant. When rendering MPEG video, for instance, it is necessary to perform large numbers of small rectangular updates. Using the standard 'put rectangle' commands would result in a large percentage of CPU time being spent communicating the command to the server. The X Servers allows specialised rendering code to be incorporated into the standard server by means of 'extension libraries'. In the above MPEG rendering example, a server extension library could be written with special calls to reduce some of the overhead. The problem with extension libraries is that they have to be linked with server, and thus can not be dynamically loaded by application programs. This makes it very difficult for applications with 'special needs' to do their own thing.

Multimedia applications may wish to reserve the resources they require in advance, which may enable them to perform in a more predictable manner. If they are producing graphical output, they need to consider what resources are required to get the data to the screen. If all graphics operations in the system are performed by a server, the server should provide both accounting and scheduling to attempt to provide clients with a pre-negotiated quality of service. Neither Microsoft's Win32S or MIT's X Server provide any support for such features. Without them, it is quite possible for a single client to saturate the server with requests, resulting in unpredictable performance for other clients. Since the time spent by the server on behalf of a client is never accounted to the client, the operating system can be led to see a very distorted view of the client's resource use.

Some applications, particularly those performing animation, may need to perform drawing operations in some synchronous manner, e.g. relative to the raster scan cycle. Although the operating system may be able to meet such scheduling requirements, building such features into a server further complicates matters.

Although convenient for traditional non-realtime graphics applications, it seems that multiplexing drawing commands from multiple clients through a single server makes providing QoS guarantees difficult.

3 Client rendering

Allowing clients to access the frame buffer directly solves many of the problems identified; The operating system remains solely in charge of scheduling, and the data copy inherent in using a server can be eliminated. Client rendering is not necessarily a panacea however. This section explores the problems.

Given that clients are now responsible for rendering their own graphics operations, there is the potential for a lot of code duplication amongst clients as they each implement their own rendering routines for lines, circles, rectangles etc. This is of course easily solved by the use of a shared library implementing some standard set of drawing commands. It would be quite possible to implement all of the X protocol calls like this. By having a standard shared library, clients that do not wish to be aware of the underlying hardware they are running on can be abstracted from it. Those that have special requirements can implement them accordingly.

Despite clients drawing on their own behalf, a server is still required for management purposes. Clients will still need to make calls to a server to request new windows, colourmap entries etc., and the server will need to inform the client of exposure events, keyboard and mouse movements etc. The job of the server becomes more akin to that of a Window Manager, as opposed to a graphics server; It is there for control purposes rather than appearing on the datapath of every graphics operation.

Allowing client programs access to the frame buffer requires that the frame buffer be mapped into their virtual address space. This opens up a number of security issues. User programs should not be able to perform any operation that could 'crash' the system. Allowing a user process access to the frame buffer memory does not usually violate this, but depending on the video hardware, there may be hardware registers with 'dangerous' functions. The protection offered by the memory management system should be used to ensure that any such registers are not within reach of any user processes.

On conventional frame buffer hardware, client rendering makes clipping the responsibility of the client. The client will be provided with it's visibility information by the window manager, but it must be trusted to draw only to parts of the screen it has been allocated. As such, a renegade program could repeatedly scribble over the entire frame buffer area, making the job of 'killing' it from the console very difficult.

A side issue is that of privacy between clients within the same frame buffer. Allowing clients to have access to the entire frame store potentially means that one program could 'steal' information from another's window. This sort of thing is easily protected against in server based systems, but most allow it due to the number of programs that utilise such snooping activities for honest purposes e.g. 'screen dump' utilities.

One solution to the problem of clients scribbling over each other is to implement clipping in the frame buffer hardware. This could use similar methods to those employed in the X Server, pre-processing a drawing command to break it down into wholly visible segments. Although the most efficient method in software, this may not be the best hardware implementation. An alternative technique would be to maintain a bitmap indicating which clients are allowed to access which pixels. This 'per pixel' as opposed to 'per operation' approach can be particularly beneficial when many small operations are required, such as the MPEG video case discussed previously.

The next section describes an approach for 'per pixel' hardware protection of a frame buffer that allows client rendering in a secure way.

4 Key based pixel protection

The Desk Area Network (DAN) framestore provides the necessary hardware protection to allow client rendering in a safe manner. It does this by maintaining a small tag along with every pixel that makes up the frame buffer. The tag contains a 'client number' which indicates which client owns the pixel, and is thus able to manipulate it. Processes wishing to use the frame buffer are allocated client numbers by the windowing system, which is also responsible for updating pixel's tag information. Whenever a process writes to the frame buffer, the tag associated with the target pixel is read, and a comparison performed with the process' client number. If the two match, the write is performed as normal, otherwise the write is silently thrown away. No protection exception is generated, as processes will frequently attempt to write to pixels they don't own as they attempt to draw in a partly obscured window.

In a similar manner, a process that attempts to read from the frame buffer will have its client number compared with the pixel's tag. In the event of a mismatch, some 'undefined' data is returned, e.g. all zeros.

For the pixel protection to be secure, it should not be possible for a process to forge a different client number. Similarly, un-privileged processes should not be able to update the tag information. This can be achieved by using the memory management unit, and a minor modification to the operating system. Processes that wish to use the frame buffer can make calls to the windowing system to 'open' windows in the normal way. The server will then allocate them a client number and write it into a special register on the frame buffer. It will also update the pixel tag information to give the client ownership of the pixels in its window. Both of these operations require accessing the graphics card through physical addresses that are not in the memory map of client processes. Only the specially privileged window manager can update them. The set of physical addresses that select the graphic card's frame buffer can then be mapped into the virtual address space of all the processes that wish to access the frame buffer. This mapping can be set up as a result of the 'open window' call. The client number register within the graphics device needs to be updated whenever a process that uses the frame store with a different client number is scheduled. This is can be done by modifying the operating system context switch code.

The previously described method is by no means the only one. For instance, if there are enough physical address bits to spare, the client number register on the graphics card can be eliminated. Each client could have the frame buffer region mapped into a separate physical address range. The graphics device will be able to implicitly decode the client number by looking at the upper physical address bits.

Nothing has been said so far about the size of the tag associated with each pixel. The tag needs to be big enough to hold sufficient client numbers such that all non completely obscured windows can be allocated different numbers. Different people use window systems in different ways; some have many overlapping randomly placed windows, others prefer a neat 'tiled' layout. A rough survey of X windows and Microsoft Windows users by the author found that no-one had more than 15 applications visible on a single screen simultaneously. Storing a single byte for the tag information allows up to 255 windows to be visible on screen together, satisfying even the most untidy window users. One of the possible client numbers is withheld to enable the window manager to perform window decoration, pop up menus, draw icons etc. Note that the tag width doesn't limit the number of windows that can be 'open', only the number that can be simultaneously visible, either completely or partially.

The single byte of tag data per pixel can be stored in DRAM. The cost of this is small in comparison to the cost of 24bits per pixel of VRAM found in most high performance framestores. The VRAM costs in such a framestore would be at least six times the cost of the tag DRAM.

The pixel update operation in a key based framestore can be easily pipelined such that the read of the tag information for the current pixel occurs concurrently with the write operation for the previous pixel. As a result, throughput to the frame buffer is unaffected by the protection lookup.

In the presence of client rendering, the main responsibility of the graphics server is to manage the pixel tag space as windows open, close, resize and move. The server should still send 'window exposure change' events to clients, as clients may use these to optimise their behaviour. For instance, if a client knows it is completely occluded e.g. iconified, it can stop generating updates.

The server is also responsible for allocating client numbers. Since these are a finite hardware resource, it should be fairly frugal with regards to handing them out. In general, a client number should only be allocated for 'top level' windows, not ones that form panes within other windows. For example, some graphical toolkits attempt to open a new window for every button widget they create. Since these buttons lie inside the client's main window, they should be allocated the same client number. As a result, these windows will not be clipped against their parent windows, which may require modifications to some standard toolkit libraries to force them to do their own clipping.

There are occasions where it is reasonable for panes to have their own keys. Typically, these would be for areas of the screen where the update rate is very high, and a large benefit could be had by having the clipping performed in hardware rather than software. Video windows would benefit from this in particular.

When a window is completely occluded, the server can reclaim the window's client number. The window will then be assigned a client number that is common to all currently invisible windows, preventing it from doing further drawing.

One side effect of having per pixel protection is that the rectangular constraint on the shape of windows is lifted. Windows can now be arbitrarily shaped, for example, ring shaped. As yet, no particular use has been found for such windows, but it makes for some cool demos.

In summary, key based pixel protection enables Xserver like drawing functions to be securely implemented in a shared library without the overhead of copying and context switches introduced by a server. Resource accounting is also made easier, to the benefit of soft real time scheduling.

5 Desk Area Network video architecture

In order to describe 'Virtual Windows' a brief description of the Desk Area Network (DAN) is necessary. The DAN was designed and built as a workstation architecture to support multimedia. It consists of a collection of heterogeneous devices such as network interfaces, video capture devices, framestores and CPUs interconnected by a high speed virtual circuit cell switch. The design was motivated by a number of issues. Firstly, the electrical constraints of multi-tapped busses mean that they do not scale well in either throughput or number of ports; Switches provide a solution, and for the number of ports required in a workstation, can be simple crossbar types. The switch is used to send small fixed sized cells along virtual circuits between devices. Flow control is used to guarantee reliable delivery of cells, enabling simple protocols to be used for communication between DAN devices. The small cell size allows frequent opportunity for preemption by higher priority streams.

The virtual circuits between devices are controlled by the operating system. When a device receives cells on a particular virtual circuit, it can index on the virtual circuit identifier to discover any necessary state information, such as access privileges. This provides a convenient mechanism to allow a single device to have multiple communications simultaneously active.

Conventional workstation architecture is geared towards getting data between I/O devices and main memory. When used for live media streams, main memory often acts as a temporary buffer for transferring data from input to output devices. A key observation behind the DAN concept was that that a multimedia workstation might benefit from having I/O devices that are able to communicate data directly, bypassing main memory when it is not necessary.

The cell switch is used to communicate packets between devices. A number of data formats have been defined to support various services, such as audio, video and cache line traffic. These formats have been designed such that the packets are easily encapsulated by a DAN network interface node to enable them to be sent to a remote workstation. The network interface in the remote DAN will be able to remove the encapsulation, and send the video directly to the frame buffer, or CPU node if further processing is required.

The basic data type for video on the DAN is a packet containing new pixel values for a rectangular area within a window. The format of the bits making up the pixels is dependent on the frame buffer type. Formats are defined for 8 bit pseudo colour and for 16 and 24 bit true colour framestores. Conversion between different formats could be achieved by 'pipeing' the data through a CPU node, or perhaps a specialist conversion device. This format is clearly optimised for video, but is also used for updating the framestore as a result of standard graphics operations. For this purpose there are a number of additions to the protocol.

6 Virtual windows

A DAN framestore may have many virtual circuits connected to it. It can tell the source of any pixel updates by looking at the virtual circuit identifier on which it arrived. This mechanism is used to identify clients, necessary for implementing the key based pixel protection described previously.

Every virtual circuit has a rectangular region of the screen associated with it, known as a virtual window. The x,y co-ordinates of the updates contained in the incoming video streams are relative to the start of the virtual window. The DAN framestore converts these relative values into screen co-ordinates by adding on the x,y offset of the virtual window. The pixel updates are then clipped by matching the client number associated with the virtual circuit against the tag stored alongside each frame buffer pixel. This allows the visible region of a window to be arbitrarily shaped, even though the region associated with the virtual circuit is rectangular.

Using the framestore to translate window relative co-ordinates into screen co-ordinates means that the video stream doesn't contain state referring to a particular destination. This allows a single video stream to be multicast to several destinations, some of which may be across a LAN.

As well as adding in the x,y offset associated with the virtual circuit, the framestore may check that the rectangular update does not exceed the rectangular region defined for the virtual circuit. If these extra checks are made, 'client numbers' as used by the key based pixel protection scheme may be re-used between non overlapping virtual windows.

Using a combination of virtual windows and per-pixel clipping, the DAN framestore is directly render video updates without processor intervention. This allows appropriately formated video coming from the network, disk or video capture interfaces to be rendered to the framestore without passing through main memory or the CPU.

7 Architecture of the DAN Framestore

The initial framestore implementation, known as the DFS, was built to show 'proof of concept' rather than achieve ultimate performance. The design has a 2MB VRAM framestore that can be configured to yield a number of screen size and pixel depths. The favoured resolution was 1024x864 with 15bit 'true colour' pixels. The 32 avaliable shades of red, green and blue provide perfectly acceptable live video quality, and are reasonable for smoothly shaded computer rendered images.

In a separate memory bank from the pixel framestore, tags are stored for every pixel. The memory used to store the tags is actually VRAM, rather than the DRAM that is strictly necessary. This was done because in this simple design, bits are stolen from the tag field to be used for pixel overlay purposes. This mechanism enables a three colour software cursor to overlay the 'real' pixels underneath. The tag RAM is also used for another purpose. The Brooktree RAMDAC that was used in the design is capable of switching between true colour and pseudo colour modes on a per-pixel basis under the control of an input pin. This pin may optionally be connected to one of the tag bits, enabling true colour and pseudo colour windows to exist on a single screen. One might argue that pseudo colour windows are made obsolete by a true colour display, but many applications that do not wish to display images can use an 8bit pseudo colour display perfectly adequately; In fact many have the notion that all displays are 8bit ingrained deep within them. The mechanism described above allows such applications to use pseudo colour windows alongside other applications using 15bit true colour.

One area to be investigated by building DFS was what format pixel update data should take, and what kinds of operations the device should be able to perform. This suggested that software rather than a hardware state machine should form the control logic. At the heart of the design is an ARM 710 CPU, which is used rather like a micro-coded DMA engine.

The ARM instruction set contains 'store multiple register' (STM) instructions, that will store the contents of a selected number of registers to sequentially incrementing locations from a base address register. In the DFS, STM instructions to a special physical address region do not actually result in the storing of registers at all; Instead, they result in the DFS hardware moving pixels from the DAN interconnect FIFOs to the frame buffer, using the addresses that the ARM's STM instruction is generating. While transferring the pixels, the hardware clips them by comparing the current client number (found from the virtual circuit identifier on which they arrived) to the number stored in the destination pixel's tag. Using the ARM to just generate the addresses rather than loading the pixels into the registers and storing them back out again yields double the performance by halving the bus traffic. Clipping by comparing pixel tags with client numbers is performed by one of four EPLDs. The other three control the DAN interconnect, VRAM timing and address decoding. In a commercial design, the ARM and EPLDs could easily be fabricated in a single ASIC.

The ARM contains a write buffer, which allows the processor core to race ahead of the memory bus, calculating the addresses for the next STM instruction while the current one completes. The net result is that the ARM is nearly able to saturate the memory bus, resulting in a constant stream of pixel updates.

The ARM instruction stream is downloaded over the DAN interconnect into spare VRAM at boot time. The code quickly becomes resident in the ARM's instruction cache, consuming no bus cycles that could more useful be used for transferring pixels.

The datapath of the DFS is 32bits wide, enabling two true colour pixels to be clipped and written to the framestore every clock cycle (80ns). This enables a peak update performance of 25Mpixels/s (400Mb/s). In practise, the peak is not sustainable, as the ARM needs to process the header of each cell to determine to which stream it belongs. When cells from different streams arrive in an interleaved fashion, the ARM needs to context switch between the different streams. This is simply a matter of saving and restoring a few registers into the on chip cache, but does cost some cycles. In an ASIC implementation, the context switch time could be significantly reduced.

The DFS is basically an output device, though a small back channel into the interconnect does exist. This is mainly used to allow other devices to probe status information, and check that the framestore is still 'alive'. It is possible to use the back channel to read back pixel data from the frame buffer. This use has proved largely unnecessary, as many of the operations that required reading from the frame buffer in traditional systems are handled differently on the DAN. Applications that frequently require to read back their pixel values are better off keeping a shadow copy of the window in local DRAM. This is frequently the case with conventional video cards anyway, where read latency is often large. Having a shadow window also makes dealing with expose events trivial.

8 Experimental results to date

The DFS has been configured in an 8 port DAN system, along with two 'DAN Camera2' video capture cards, and a 100Mb/s ATM LAN interface. During testing, the framestore has been bombarded with pixel streams from up to 6 simultaneous live video sources. The streams used were:

- Two 384x288 16bit colour video streams at 13fps sourced from DAN Cameras.
- One 384x288 8bit monochrome video streams at 12fps sourced from a DAN Camera.
- One 384x288 16 bit colour stream at 13fps from an AVA-200 over the 100Mb/s DAN ATM interface.
- One 192x144 16 bit colour stream at 12fps from an AVA-200 over the 100Mb/s DAN ATM interface.
- One 192x144 16 bit colour stream at 12fps from a DAN Camera.

Logic analyser traces show that while the DFS is dealing with a sequence of cells belonging to a single stream it is able to almost saturate it's pixel bus. This means it is able to sink video at a rate far greater than the 108Mb/s interconnect fabric can deliver it at. When cells from different streams arrive interleaved (something the fabric's round-robin arbiter actually ensures), the DFS needs to save and restore state to switch between the streams. The current ARM code that implements this is somewhat inefficient, taking over microsecond. The DFS can thus not quite keep up with the full fabric rate. This results in 'back pressure' to the sources through the fabric's flow control mechanism. In the 6 stream experiment described above this resulted in a throughput of around 90Mb/s, 86% of the line rate. It is hoped that this figure can be improved by some more Richard Black assembler.

9 Further work

There are plans to build a simple windowing system for the DFS, and to provide a client library that supports some X like drawing operations. This will enable the experiments to assess the benefits of client rendering.

Work also needs to be done to find the set of rendering primitives the display hardware should perform. The DFS currently only supports 'put rectangle of pixels', but there are plans to implement two others:

- 'move rectangle of pixels'. This would be useful for scrolling text windows, and also for moving windows around the screen. Care has to be taken to ensure that pixel data that does not belong to the client is not copied into a window that does.

- ‘put rectangle of pixels with mask’. For 16 bit true-colour modes, the unused top bit could be used as a mask bit to indicate whether the pixel should be updated or not, modulo the normal clipping. This would allow updates to be sent for specific pixels without affecting surrounding ones. This may make the implementation of the X drawing commands easier, and may obviate the need for a shadow copy of the window in some cases.

Use of the DAN Framestore as an ATM LAN ‘network video device’ should be investigated. This could be achieved by placing a single port controller and a DFS in a mini DAN unit, and putting it out on the network for other machines to connect and send updates to. The virtual circuits on which the DAN receives pixel updates are now unreliable. The impact on the update protocol needs to be considered.