Moving Beyond the Framebuffer

Charles D. Estes
University of North Carolina at Chapel Hill
Brooks Computer Science Building, CB 3175
Chapel Hill, NC 27599-3175 USA
cdestes@cs.unc.edu

Ketan Mayer-Patel
University of North Carolina at Chapel Hill
Brooks Computer Science Building, CB 3175
Chapel Hill, NC 27599-3175 USA
kmp@cs.unc.edu

ABSTRACT

This paper explores a new abstraction to replace the framebuffer as the metaphor for a new display interface. Our novel approach aims to provide backward compatibility with applications that require a simple framebuffer, while also providing tremendous channel capacity savings to applications that exploit application level semantics to use the display interface in a more sophisticated way. The goal is to develop a versatile interface that scales from very large displays to small, low-power displays connected over wireless links.

Categories and Subject Descriptors

C.0 [**General**]: Hardware/software interfaces

General Terms

Design, Algorithms, Performance, Experimentation

Keywords

Display interface, framebuffer, scalable display

1. LEGACY OF THE FRAMEBUFFER

Since the rise of raster displays, the framebuffer, as a concept, has been the predominant abstraction for display interfaces. The earliest personal computers either adopted the television for display purposes or integrated a cathode ray tube (CRT) display as part of the computer. The framebuffer as an abstraction was a logical extension of the row-wise scanning of pixels performed in order to generate an analog signal compatible with existing television standards and CRT-based components.

The modular design of the IBM PC and its clones introduced the display adapter as an interface to the display as a peripheral resource. The software drivers for the display adapter formalized the framebuffer as a software construct. The separation of the display as a distinct and somewhat independent component was important because advances in CPU technologies generally outpace advances in display technologies. This allows the CPU to be upgraded or replaced without requiring the purchase of a new display. Likewise, if and when display technologies advance, only the display adapter and monitor need be replaced.

Analog display technologies were slow to evolve because any advance in resolution and/or frame rate required the development and adoption of new signaling and cabling standards for the analog signal driving the display. IBM led the industry through a series of such advances in analog signaling with the development of CGA in 1981, EGA in 1984, VGA in 1987, and XVGA in 1991[9]. These analog display interface standards were so pervasive, that the earliest digital flat panel displays used the same interfaces *despite the fact that these displays were not analog devices.*

2. MODERN DISPLAY CHALLENGES

For the most part, digital display technologies such as liquid crystal displays (LCDs), digital light projectors (DLPs), and plasma displays have completely supplanted CRT-based analog displays. Modern digital display interface standards include Digital Visual Interface (DVI), High-Definition Multimedia Interface (HDMI)[4], and DisplayPort[1][15]. While these protocols directly support digital displays, they all essentially mimic their analog predecessors by continuously packaging the framebuffer and transporting it at the specified refresh rate. Unfortunately, like their analog predecessors, these standards will not be able to keep pace with trends in display technologies and emerging innovations in how displays are used. These challenges include:

- **Increased resolution** - While 1080p is a common native resolution for many displays, this particular spatial format is popular because it matches current high-definition video standards. Higher resolution displays are common in computer applications and there are few technical barriers to prevent the manufacture of digital displays with spatial formats as high as 3840x2400 (WQUXGA)[6].
- **Higher refresh rates** - Many LCD HDTVs today are advertised as being able to support refresh rates of up to 240 Hz.
- **3D television** - The HDMI 1.4a specification allows for 3D over HDMI, but it stops short of full 1080p at 60 Hz support[5]. True stereo 3D television will require a pair of images for each supported viewpoint. One can easily imagine future scenarios with tens of supported viewpoints.

NOSSDAV'11, June 1–3, 2011, Vancouver, British Columbia, Canada.

- **Large-scale displays** - Scalable display walls and jumbotrons currently involve stitching together separate panels driven by individual display adapters[7][10][2]. The jumbotron in Cowboy's Stadium uses HDTV video sources to drive its panels despite the fact that the native format of these displays is more than 5x larger[8]. One of the reasons why such a large display is being under-utilized is the complexity and bandwidth involved in trying to drive it at a resolution that is at the same scale as its physical size. The Walgreen's billboard in Times Square, for example, requires a 48-drive RAID disc streaming data at 3.2 GB/s to drive its 17,000 square foot billboard at a spatial format of 10,000 x 4,000[3].

- **Low-power mobile displays** - Mobile displays face similar challenges on a different scale. A popular current example is HD video playback on tablets. A tablet with a 1280 x 800 pixel display and 16 bits per pixel will require 512 kB per frame. At 24 frames/sec, this requires a channel capacity of 400 Mbps, easily exceeding the capacity of mobile data links. Video compression combats this but at the cost of reduced battery life due to decoding the video at the tablet[11].

- **Remote displays** - Remotely connected display resources must overcome mismatches between the bandwidth required to transmit the framebuffer and the bandwidth available which is subject to fluctuating network conditions and congestion control. A common solution is to employ a remote display application such as Virtual Network Computing (VNC)[13] to avoid transmitting parts of the display that are not changing. This solution essentially requires integrating a computer in the display in order to execute the remote client. Similarly, kiosk displays are often driven by a dedicated collocated computer which is then managed remotely in order to transfer and control content.

In our opinion, the framebuffer as an abstraction for display resources has reached the limits of its scalability. In order to support future advances in spatial formats and refresh rates as well as innovative uses for displays, a new abstraction needs to be considered. We can no longer afford the cost of repeatedly copying out the entire framebuffer tens or hundreds of times per second inherited as the legacy of the framebuffer. This paper presents early work in progress for what such a new abstraction could look like. We start with a clean-slate design derived from first principles that incorporates lessons learned from the framebuffer's legacy. One of the key features of our proposed design is that it allows processes at higher levels of the system such as applications, GUI toolkits, graphics libraries, etc. to leverage high-level semantics in order to most efficiently and flexibly make use of display resources.

The rest of this paper is organized into seven additional sections, in which we explore our search for a new abstraction, outline the design principles that emerged from that search, detail our design, illustrate several use cases for our design, and present a key use case experiment and results.

3. CHOOSING A NEW ABSTRACTION

Consider the exercise of evaluating well-known software abstractions as candidates for a new abstraction for display resources. Several possibilities are shown in Figure 1. First, we note that there is a general tradeoff between channel efficiency and flexibility against implementation complexity. High-level complex abstractions can communicate display updates efficiently for content that matches the data model for which that abstraction was designed for. However, flexibility is decreased such that any content that is outside of the data model becomes very difficult to communicate efficiently.

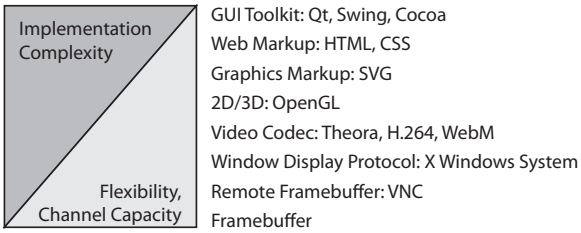

Figure 1: Sample software abstractions.

For instance, it is unreasonable to build a display based on HTTP, webkit, and Javascript for obvious reasons of complexity as well as the implicitly limited flexibility of such a display. While such a display abstraction might provide scalability and throughput savings for web-based applications, any advantage is quickly lost once outside of the abstraction's domain. For example, a high-performance video game.

Window managers based on X[14] represent another abstraction with high implementation complexity due to the need to support features such as z-ordering, blending, focus, and cursors. Furthermore, limited or highly variable bandwidth will severely affect performance.

One reasonably flexible abstraction is the popular remote display application, VNC, which requires that the host machine use a VNC server to monitor framebuffer changes. It sends those encoded changes to a remote VNC client leveraging the Remote Framebuffer (RFB) protocol. While very efficient for coherent display content where changes are concentrated into small areas, such an abstraction will not gracefully handle highly dynamic content such as video and/or gaming. Furthermore, VNC and RFB are still somewhat complex to implement, effectively requiring the display to utilize a microprocessor to support the VNC client.

Net2Display is a recent specification from VESA that provides an official standard for remoting displays, competing to replace VNC, Microsoft Remote Desktop Protocol (RDP), Citrix ICA, and others[12]. One of the key differentiators of the Net2Display standard, is that it accommodates dedicated display devices connected over an IP-based link. The Net2Display standard is primarily an architecture that specifies a minimum feature set in order to best encourage interoperability. While this "minimal feature set" may make Net2Display easier to embed in a display, it effectively makes these devices less capable[16].

4. DESIGN PRINCIPLES

In this section, we articulate a number of design principles that a new display abstraction should embody. A primary lesson from the legacy of the framebuffer is that major design shifts are infrequent. Thus, any such design needs to be reasonably simple and highly adaptable in order to ac-

commodate a wide range of applications and future needs. It was to that end that we developed the following set of design principles.

4.1 No Computational State Machine

In considering other possible display interface abstractions, we find that most require some form of a computational state machine, whether it be a general purpose processor to support a VNC-like client or a highly specialized graphics/multimedia processor to support codecs and/or perform higher-order graphics commands. Specifying display hardware that leverages some form of a computational state machine can yield dramatic benefits, but it will consume far more power than a solution that employs simple digital logic. Power consumption may not be a major concern of large home theater displays, but it is a major constraint for a mobile displays using an energy efficient technology like electronic paper (EPD). Thus, the first design principle is that our new abstraction must not require a computational state machine.

4.2 Highly Parallel

While a state machine based design must address scalability by using ever increasing clock rates for the underlying processor, a design based on simpler digital logic scales easily if parallelism is brought to bear. Therefore we identify high parallelism as another key design principle. It should be possible to update every pixel on the display simultaneously via highly parallel digital logic.

4.3 Asynchronous

One consequence of forgoing a computational state machine and adopting high parallelism, is the ability to embrace asynchrony as a design principle. Allowing asynchrony decouples the data rate of the display interface and the refresh rate of the display panel. This is an important consideration because it supports use cases in which quality of service for the connection to the display is either constrained or highly variable. Furthermore, the system can be made more energy efficient by only performing calculations and updating the display when new display data arrives.

4.4 Framebuffer Compatible

The framebuffer abstraction is deeply established and will not be quickly abandoned as a supported abstraction. Its simple elegance and ubiquity in today's software makes it a compelling method for interacting with our new display interface as well. Furthermore, adopting framebuffer compatibility as a design principle better allows for a staged adoption of the new abstraction.

4.5 Progressive Benefit

Recalling the discussion in section 3, there is a general tradeoff between the complexity of an abstraction and the channel capacity required to support it. The more we leverage application knowledge about how display resources are being used, the more compactly we can describe and represent the data required to drive the display. Furthermore, this semantic knowledge can be used to help negotiate appropriate adaptations to fluctuating or constrained channel resources. Thus, another design principle that we espouse is that the proposed abstraction provide a progressive benefit when higher-level application semantics are known.

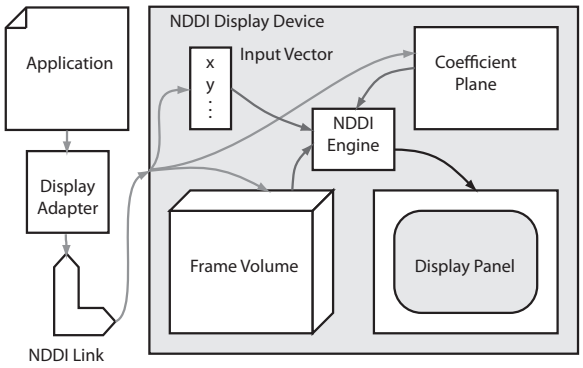

Figure 2: NDDI concept diagram.

5. N-DIMENSIONAL DISPLAY INTERFACE

In this section, we present the design of our proposed display abstraction, the n-Dimensional Display Interface (NDDI). The NDDI is comprised of the following components:

- The Frame Volume
- The Coefficient Plane
- The Input Vector
- The NDDI Engine

These components are illustrated abstractly in Figure 2. The application in this diagram represents an agent using the NDDI Display Device. The Display Adapter represents an interface layer that the application drives much in the same way current systems have a display adapter that provides an interface to the framebuffer. The NDDI Link represents the physical connection between the Display Adapter and the NDDI Device. This physical link and the protocol used to communicate over it are not described in this paper. While an actual realization of NDDI would, of course, require specification of this link, for the exploratory purpose of this paper, it can be thought of abstractly as a wired or wireless connection capable of transmitting and receiving data encapsulated within NDDI commands. The following subsections describe each of the NDDI components in turn.

5.1 Frame Volume

NDDI expands on the idea of a display with "memory" of its pixels. It does not utilize a fixed, two-dimensional framebuffer matching the format of the display panel. Instead it specifies that a display has a *frame volume*. This frame volume is a very large piece of memory that holds pixel values that can be mapped to the individual pixels on the display panel in a variety of ways.

The frame volume can be configured to any dimensionality. In its simplest representation, it can be configured as a two-dimensional framebuffer that represents the current contents of the display. Another configuration might add a third dimension that represents time, allowing a video stream to buffer on the actual display itself. Exactly how the frame volume is configured is one of the ways the NDDI provides applications driving the display to take advantage of higher-level semantic knowledge. In practice, applications may never use more than three or four dimensions, and so future implementations of NDDI may impose a limit.

5.2 Coefficient Plane

The dimensionality of the Frame Volume most often will not match the display panel, and so the pixel values from the Frame Volume must be mapped to the panel. NDDI accomplishes this through the *coefficient plane*. The coefficient plane is a two-dimensional grid of *coefficient matrices*. This grid matches the dimensions of the display panel. The coefficient matrix at a particular x and y location in the coefficient plane is used in conjunction with the *input vector* to pick a unique value from the Frame Volume in order to display on the panel at the same x and y location.

5.3 Input Vector

The update process is driven by the *input vector*. The input vector is a one-dimensional vector, with the first two values reserved for the x and y position of a pixel. The remaining values are optional and are specified by the application. The x and y values are not driven by the application, but rather by NDDI when it is computing output pixel values for the panel.

5.4 NDDI Engine

The three primary components of NDDI are effectively memory stores. The digital logic that drives the process of updating the display panel resides in the *NDDI engine*. Any time the data in the input vector, coefficient plane, or frame volume changes, the NDDI engine calculates the updates to the *display panel* in parallel. The process begins with the input vector. For each pixel, the NDDI engine 1) sets the x and y value in the input vector, 2) multiplies the input vector by the coefficient matrix at the corresponding x and y location in the coefficient plane to produce a tuple that matches the dimensionality of the frame volume, and 3) finally updates the display panel using the single pixel value from the frame volume addressed by the tuple.

The following illustrates the calculation for a pixel at location $(7, 8)$. The coefficient matrix at that location in in the coefficient plane is multiplied by the input vector with the x and y values set to 7 and 8 to produce the tuple $(8, 8, 2)$. The pixel value at this location in the frame volume is displayed.

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} 7 \\ 8 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 8 \\ 8 \\ 2 \end{bmatrix}$$

The NDDI interface reflects the design principles we identified in section 4. The NDDI mapping mechanism requires only a simple set of matrix and addressing operations allowing it to be implemented directly with digital logic. The NDDI engine computes the value of each display pixel in precisely the same manner which makes the mechanism highly parallel. NDDI is driven exclusively by the contents of its various structures. Thus the NDDI engine only needs to recalculate output when data arrives, making it asynchronous. If necessary, the NDDI structures can be configured in order to mimic a framebuffer directly. This is done by simply configuring the frame volume to be a 2D structure that matches the size of the display and setting the coefficient planes to the identity matrix. Finally, NDDI provides a progressive benefit to applications that are able to employ higher-level semantics about how best to deploy and exploit the NDDI structures. Several use cases that demonstrate how this might be done are described in the next section.

6. USING APPLICATION SEMANTICS

While it is possible to configure an NDDI display as a simple framebuffer, it will not produce throughput savings beyond the benefit of being able to drive the display at framerate other than the fixed refresh rate of a traditional display. In order to realize a more dramatic throughput advantage, the application must configure the NDDI display leveraging higher-level semantics.

6.1 Example: Video Player

A video playback application might configure the frame volume in three dimensions, with the x and y dimensions matching the display panel and the z dimension representing a buffered queue of frames. The buffering of frames would allow the application to handle channel capacity fluctuations when a fixed quality of service is unavailable. Additionally, the application can use a previous frame in the frame volume as the basis for the next frame by utilizing an inexpensive copy command to copy the contents to another z plane and then updating that new frame with only the changed pixels, mimicking the way a video codec would handle P and I frames. The coefficient plane would consist of simple 3 x 3 identity matrices. The display updates would be driven by an input vector with the default x and y values and a third value representing the frame counter, c. The application only needs to update that frame counter to display the next frame.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ c \end{bmatrix} = \begin{bmatrix} x \\ y \\ c \end{bmatrix}$$

6.2 Example: Windowed User Interface

An application representing a computer desktop with a windowed user interface might also configure the frame volume in three dimensions, with the third dimensions instead being used as a cache for various windows. The x and y dimensions of the frame volume would match the largest window dimensions allowed and the pixel values for each window would be stored at the origin of each xy plane in the frame volume. The task of configuring the coefficient plane would not be as simple as with the video player example, because portions of several windows can be displayed simultaneously. In this case, the input vector would instead have a 1 in the third value, and the coefficient matrices would be updated with a w, tx, and ty values to choose and translate the window.

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & w \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ w \end{bmatrix}$$

6.3 Example: Web Tablet

A web tablet device could leverage NDDI to overcome several challenges for tablet computers. Tablets are constrained devices with large displays, diminished power supplies, limited processing power, and low-capacity wireless links. Their large displays bring a higher level of interaction for the user, so manufacturers strive to keep the web and multimedia content downloading, decoding, and rendering quickly. A web tablet using NDDI could eschew a high-power CPU/GPU and instead use an ASIC-based NDDI Engine combined with slow, low-power RAM and a wireless modem to create a thin-client. In a simple configuration, it could arrange the frame

volume in three dimensions with the third representing tabs. The x and y dimensions could be larger than the display, allowing more content outside of viewport to be buffered and then rendered when the user scrolls the viewport. In this configuration, the tabs would be modal, and so they would be chosen and scrolled with b, tx, and ty.

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & b \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ b \end{bmatrix}$$

7. PIXEL BRIDGE EXPERIMENT

Our first proof-of-concept prototype of an NDDI display is a software simulation using a driving application that we call "Pixel Bridge". Pixel Bridge is intended to interface existing legacy applications and as such is an important first step. Pixel Bridge is like VNC in that it monitors framebuffer changes in order to identify areas of the display that require updating. It does not employ higher-level application semantics, but can marshall NDDI resources in a number of different ways in order to be as efficient as possible. In this experiment, we recorded a number of computing sessions and replayed them using Pixel Bridge and measure the amount of throughput required.

7.1 Pixel Bridge Configurations

Pixel Bridge can be configured to operate in five different modes. The first three modes employ progressively more sophisticated configurations of the the NDDI display. The last two modes are calculations that bound the performance.

- **Framebuffer** - Configured as a 2D framebuffer
- **Flat Tiled** - Configured as a 2D, tiled framebuffer
- **Cached Tiled** - Configured as a cache of tiles
- **60 Hz** - Cost of full screen updates at 60Hz
- **Ideal Pixel Latching** - Cost of only changed pixels

7.1.1 *Framebuffer Mode*

When Pixel Bridge is in the *Framebuffer* mode, it configures the frame volume with a dimensionality matching the recorded computing session. The coefficient plane is initialized so that each pixel on the display corresponds to the pixel value in the frame volume at the same x and y location. The input vector only has the default x and y values. For each frame, the entire frame volume is updated. This is similar to the 60 Hz mode, except that the *Framebuffer* mode is only updated at the framerate of the recorded computing session.

7.1.2 *Flat Tiled Mode*

The *Flat Tiled* mode configures the frame volume and initializes the coefficient plane the same way as with the *Framebuffer* mode. However, it logically partitions the frames into tiles. A CRC32 checksum of each tile is computed and compared to the corresponding tile in the frame volume to determine if that tile has changed. Only new tiles are updated.

7.1.3 *Cached Tiled Mode*

The final NDDI mode is the *Cached Tiled* mode. This mode still logically partitions the frames into tiles, but it configures the frame volume and coefficient plane differently

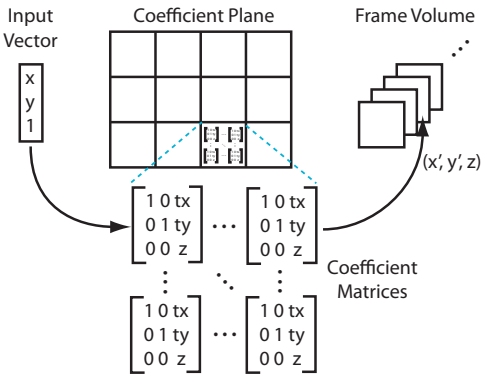

Figure 3: Cached tile mode NDDI configuration.

(see figure 3). The frame volume is configured in three dimensions, forming a cache of tiles. The x and y dimensions match the tile dimensions. The z dimension is set to a value representing the size of the cache. The coefficient plane is then logically partitioned into tiles. The coefficient matrix for each pixel in that logical tile maps to the pixel in the frame volume with the matching x and y coordinate and a z value for the particular tile in the cache.

The *Cached Tiled* mode shares the same advantages as the Flat Tiled mode, but adds the ability to leverage the cache to easily duplicate a tile at multiple locations on the display and to re-use older tiles in the cache.

7.1.4 *60 Hz and Ideal Pixel Latching Modes*

These non-NDDI modes are not implemented in Pixel Bridge as rendering modes, but rather are derived directly from the frame content and frame rate. The 60 Hz mode considers the number of frames and framerate of the original recorded session. It then calculates how many bytes are updated to the display if the display is updated at a constant 60 Hz rate. This should represent a worst-case performance boundary for Pixel Bridge. The Ideal Pixel Latching mode calculates only the number of bytes required to communicate just those pixels that change in value from one frame to the next. This mode completely disregards the obvious need to address pixel locations when updating a pixel value, and in doing so it serves as a theoretical, best-case boundary for Pixel Bridge.

7.2 Experiment Design

The following computing sessions were recorded and used as test vectors for the experiment:

- **office** - Document Editing
- **browser** - Web Browsing
- **eclipse** - Application Development using Eclipse
- **presentation** - Viewing a Presentation
- **video** - H.264 Video Playback

The first four sessions were recorded at 1280x1024 and 10 Hz. The relatively low framerate is justified because these computing activities do not produce highly dynamic changes to the display. The last session is an H.264 video playback

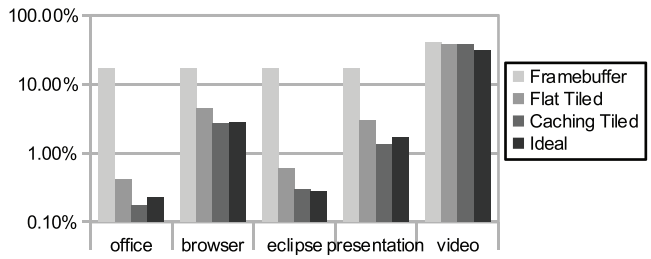

Figure 4: Preliminary Pixel Bridge Results.

at 656 x 352 and 23.98 Hz. Each of the five recorded sessions was tested with all five modes. Tile size was calculated dynamically, resulting in forty square tiles along the longest dimension. Each experiment tracked the number of bytes transmitted over the NDDI link without compression.

7.3 Results and Analysis

The results are shown in Figure 4, with each data point representing the ratio of the bytes transferred "over the wire" for that mode versus the 60 Hz mode. Recall that the 60 Hz mode represents the current status quo of transferring the entire framebuffer at a constant refresh rate. For the first four recorded sessions (i.e., not including video playback), Pixel Bridge demonstrates progressively better performance between *Framebuffer* mode to *Flat Tile* mode to *Cached Tiled* mode. Each of these modes employs NDDI resources in an increasingly sophisticated manner which is reflected in the improved performance. The *Framebuffer* mode illustrated the advantage of updating the display at a framerate that matches the source content instead a fixed refresh rate. The *Flat Tile* mode brought the advantage of only updating the changed regions of the display. The *Cached Tiled* mode showed the gains of filling the display with identical tiles as well as using previously cached tiles allowing it to outperform the *Ideal Pixel Latching* mode for some tests.

The results of the H264 video clip are quite contrary to the first four. Using a trivial, static tile size proved to be a poor approach to partitioning the screen. Furthermore, the frame content is so dynamic, that the caching was barely beneficial. The NDDI overhead of updating the coefficient plane negated most of the benefit from the cached tiling. This result was somewhat expected, since the *Cached Tiled* mode was conceived with the computing pixel bridge use cases in mind. Furthermore, full motion video represents highly dynamic content which does not match well with the title cache abstraction of Pixel Bridge. However, a more video-specific application that marshaled NDDI resources in a manner more in line with how video is compressed and represented may be more effective. This early experiment is meant only to serve as a proof-of-concept for the NDDI architecture.

8. CONCLUSIONS AND FUTURE WORK

Although the move to digital display interface standards was a significant step in developing higher capacity channels, it was largely an iteration on the same framebuffer abstraction for interfacing with a display. NDDI splits the framebuffer concept between application and display and uses a dramatically different approach to organizing that memory. Our initial Pixel Bridge experiment represents a "base case", bridging pixels from a computing session to an NDDI display. It showed increased savings as NDDI resources were marshaled in increasingly sophisticated ways despite not leveraging any application-level semantics. Our ongoing work will refine the Pixel Bridge experiment as well as explore new use cases in order to test our hypothesis that greater performance gains can be realized when application-level semantics are brought to bear.

9. REFERENCES

[1] Digital visual interface dvi revision 1.0. Digital Display Working Group, April 1999.

[2] S. Eilemann, M. Makhinya, and R. Pajarola. Equalizer: A scalable parallel rendering framework. *Visualization and Computer Graphics, IEEE Transactions on*, 15(3):436 –452, 2009.

[3] The insane hardware driving the world's biggest led billboard. website. http://gizmodo.com/#!5096475/the-insane-hardware-driving-the-worlds-biggest-led-billboard.

[4] High-definition multimedia interface specification version 1.3a. HDMI Founders, November 2006.

[5] High-definition multimedia interface specification version 1.4a extraction of 3d portion. HDMI Founders, March 2010.

[6] Ibm introduces world's highest-resolution computer monitor. Press Release, June 2001. http://www-03.ibm.com/press/us/en/pressrelease/1180.wss.

[7] B. Jeong, L. Renambot, R. Jagodic, R. Singh, J. Aguilera, A. Johnson, and J. Leigh. High-performance dynamic graphics streaming for scalable adaptive graphics environment. *ACMIEEE SC 2006 Conference SC06*, (November):24–24, 2006.

[8] Mitsubishi electric diamond vision is dallas cowboys' choice for new stadium. Press Release, April 2008. http://www.businesswire.com/news/home/20080416005327/en.

[9] R. L. Myers. *Display Interfaces: Fundamentals and Standards*. Series in Display Technology. Wiley, 2002.

[10] Nirnimesh, P. Harish, and P. Narayanan. Garuda: A scalable tiled display wall using commodity pcs. *Visualization and Computer Graphics, IEEE Transactions on*, 13(5):864 –877, 2007.

[11] Nvidia tegra 2 specifications. website. http://www.nvidia.com/object/tegra-2.html.

[12] K. Ocheltree, S. Millman, D. Hobbs, M. McDonnell, J. Nieh, and R. Baratto. Net2display: A proposed vesa standard for remoting displays and i/o devices over networks. In *Proceedings of the 2006 Americas Display Engineering and Applications Conference*, Atlanta, GA, October 2006. ADEAC.

[13] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[14] R. W. Scheifler and J. Gettys. The x window system. *ACM Trans Graph*, 5(2):79–109, 1986.

[15] Vesa displayport: Version 1.1. Video Electronics Standards Association, April 2007.

[16] Vesa net2display remoting standard: Version 1. Video Electronics Standards Association, October 2009.