The n-Dimensional Display Interface

A More Elastic Narrow Waist for the Display Pipeline

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

The framebuffer is a simple yet powerful abstraction for representing display data. It ably serves as a "narrow waist" for the display pipeline, allowing graphics software and display technology to evolve in parallel. Converging trends such as mobile computing, very large displays, and 3DTV are challenging the inherent inefficiency of practically serializing and encoding the framebuffer tens or hundreds of times per second. In this paper, we present a flexible new software abstraction that can deliver tremendous channel efficiency when application level semantics are exploited. The goal is to develop a versatile interface from this *elastic narrow waist* 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. FRAMEBUFFER AS A NARROW WAIST

The framebuffer has been the predominant abstraction for display data since the rise of raster displays. It was an obvious abstraction, allowing for row-wise scanning of pixels to generate an analog signal compatible with existing television standards and CRT-based components. It was formalized as a software construct by the IBM PC and its clones within a separate display adapter as a means to accommodate the rapid development of display technologies. The modularity of the display adapter allowed computer hardware and software to remain virtually unaffected by the new signaling and cabling standards that were developed for new displays.

MMSys'12, February 22-24, 2012 Chapel Hill, North Carolina, USA.
Copyright 2012 ACM 978-1-4503-1131-1/12/02 ...$10.00.

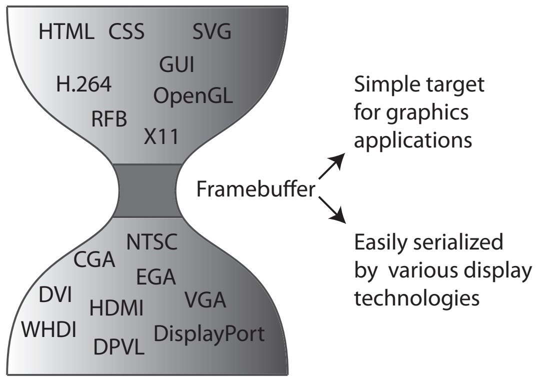

HTML
CSS
SVG
GUI
H.264
OpenGL
RFB
X11
Simple target for graphics applications
Framebuffer
Easily serialized by various display technologies
NTSC
CGA
EGA
DVI
HDMI
VGA
WHDI
DPVL
DisplayPort

Figure 1: Framebuffer as the Narrow Waist.

IBM led the industry through a series of such advances with the development of CGA in 1981, EGA in 1984, VGA in 1987, and XVGA in 1991[9].

The framebuffer has been a vital abstraction, serving as the "narrow waist" (Figure 1) of the display pipeline. It has enabled both computer graphics software and display technology to develop simultaneously and independently of one another. Perhaps its most significant strength is its simplicity, allowing a variety of complex graphical content to be composited and rendered to a simple array of pixels. That simple pixel array is readily sampled at a regular refresh rate, converted, and packaged for any of the myriad display standards.

As a software abstraction, the framebuffer has performed admirably, scaling to meet today's demands of increasingly larger displays through innovations in digital transmission. Digital display technologies such as liquid crystal displays (LCDs), digital light projectors (DLPs), and plasma displays have completely supplanted CRT-based analog displays and have much larger spatial formats. Modern digital display interface standards such as Digital Visual Interface (DVI), High-Definition Multimedia Interface (HDMI)[4], and DisplayPort[1][17] provide a significant advantage in terms of channel capacity. However, they all essentially mimic their analog predecessors by continuously packaging the framebuffer and transporting it at the specified refresh rate.

Despite the success of the framebuffer, it imposes serious restrictions that jeopardize its applicability in the face of new, emerging display challenges. Its inflexibility makes it a bottleneck for large, dynamic displays as well as a source of unnecessary overhead for smaller, less dynamic displays.

A new, more elastic, narrow waist is needed.

This paper presents early work that proposes such a new narrow waist and a display interface designed for it. 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, et cetera to leverage high-level semantics in order to most efficiently and flexibly make use of display resources.

The remainder of this paper is organized into eight additional sections. In Section 2, we enumerate several new modern display challenges that challenge the framebuffer. Sections 3 and 4 discuss our search for a new narrow waist and evaluate a promising new standard. Our search for a new narrow waist formalized a series of design goals that are illustrated in 5. In Section 6, we detail the design for our proposed abstraction, the n-Dimensional Display Interface. We use several example use cases in Section 7, to illustrate our design. Section 8 details a specific use case and the experiments that we developed to evaluate our interface for that use case with regards to channel capacity efficiency. We conclude with closing remarks and a discussion of future use cases, experiments, and cost modeling with a focus on memory and power in Section 9.

2. MODERN DISPLAY CHALLENGES

The simplicity of the framebuffer has made it an ideal narrow waist for past and present display interfaces. However, modern display challenges have strained the framebuffer abstraction prompting the development of creative new extensions to already complicated display interface standards as well as new, high-capacity signaling standards. These challenges bring into question as to whether the benefit of the framebuffer as a narrow waist outweighs its inherent inefficiencies. New trends that illustrate these modern display 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].

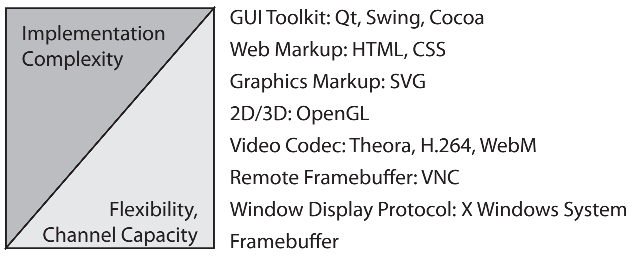
- **Higher refresh rates**
  Many LCD HDTVs today are advertised as being able to support refresh rates of up to 240 Hz.

- **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.

- **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.



Implementation Complexity
Flexibility, Channel Capacity
GUI Toolkit: Qt, Swing, Cocoa
Web Markup: HTML, CSS
Graphics Markup: SVG
2D/3D: OpenGL
Video Codec: Theora, H.264, WebM
Remote Framebuffer: VNC
Window Display Protocol: X Windows System
Framebuffer

Figure 2: Sample software abstractions.

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 narrow waist 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.

3. CHOOSING A NEW NARROW WAIST

Consider the exercise of evaluating well-known software abstractions as candidates for a new narrow waist. Several possibilities are shown and "loosely ranked" in Figure 2. 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. For instance, a display based on HTTP, webkit, and Javascript might provide scalability and throughput savings for web-based applications, but any advantage is quickly lost once outside of the abstraction's domain. For example, a high-performance video game.

Remote windowing systems such as X[14] represent another abstraction with high implementation complexity. Such systems provide strong advantages to the desktop computing paradigm and include a high-level interface to a full-featured environment with operations such as z-ordering, blending, focus, and cursors. However, such a system assumes a certain level of computational resources and connectivity which may be unavailable in the low-power and mobile domains. 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 Video Electronics Standards Association (VESA) that provides an official standard for remoting displays. It is intended 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. Net2Display 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[18].

While these software abstractions are suitable for the specific applications they were designed for, none of them offer the simplicity and broad applicability of the framebuffer, making them poor choices for a new narrow waist. If visualized on Figure 2, an ideal candidate would likely lie just above the framebuffer abstraction to guarantee broad applicability, yet high enough to achieve the critically needed channel efficiency.

4. DIGITAL PACKET VIDEO LINK

With the release of the Digital Packet Video Link (DPVL) standard[15], the Video Electronics Standards Association (VESA) has recognized that traditional display interfaces based solely on the framebuffer abstraction cannot meet future needs. The specification was originally released in 2004, but does not yet have any significant support in the industry. DPVL is the first standard to seriously address the waste involved with updating an entire framebuffer at a given refresh rate. It accomplishes this by sending only the changed rectangles over the wire in packets (Figure 3). By not forgoing the framebuffer altogether, DPVL provides a substantial amount of backward compatibility.

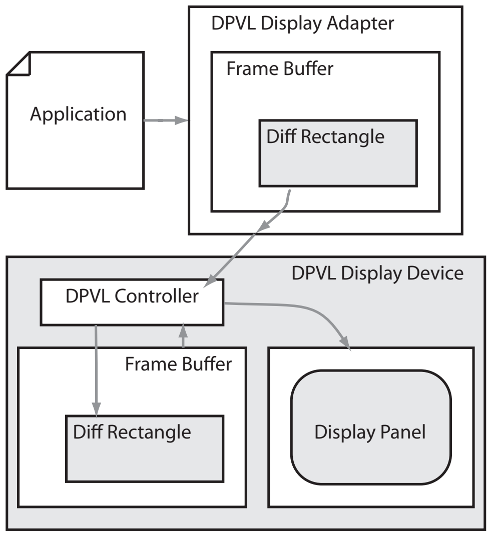


Figure 3: DPVL block diagram.

In addition to automatically detecting regions that require updating, the DPVL display adapter provides a number of high level commands to the application[16]. These include powerful commands like bit-blt, pattern fill, area fill, and scaled video stream. While not a significant departure from the framebuffer abstraction, DPVL is novel in that it acts as the first digital standard to replicate the framebuffer on both the display adapter and the display device thus providing the application a means to update portions of the display only[9].

DPVL is not a strong departure from the framebuffer. Instead it is a promising iteration on the framebuffer concept that provides some level of improved channel efficiency and a little more elasticity to the narrow waist. We argue that forgoing the framebuffer altogether can exceed the benefits of DPVL while providing more flexibility and still maintaining the crucial backward compatibility with the framebuffer.

5. DESIGN GOALS

From the exercise of analyzing the strengths of the framebuffer and of exploring other software abstractions we have distilled a key set of design principles. Our goal is to use these principles to define an abstraction that serves as a new narrow waist from which to design a new display interface. These design principles are:

- **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 deployment of the new abstraction.

- **Data-Driven**
  The data-driven nature of the framebuffer is key to its strength as a narrow waist. It is simply an array of pixel values, easily modeled and represented in

software and quickly serialized into a variety of standard display signals. Fundamentally, a display built on a framebuffer interface can be viewed as a memory-mapped output device allowing an application to simply write its array of pixel values to a data region in order for them to be displayed. Our new abstraction should be similarly data-driven.

- **Progressive Benefit**
  Acknowledging the general tradeoff between complexity and channel efficiency, the abstraction should support a progressive benefit allowing applications that utilize higher-level application semantics to realize a greater channel efficiency. Furthermore, this semantic knowledge can be used to help negotiate appropriate adaptations to fluctuating or constrained channel resources.

- **Highly Parallel**
  It should be possible to update many pixels on the display simultaneously via highly parallel digital logic.

- **Asynchronous**
  The abstraction should support asynchrony and should not force applications to comply to a particular refresh rate.

These design principles work in concert to ensure that the abstraction fulfills the critical role of a narrow waist. Ensuring framebuffer compatibility and a data-driven design allows for ease of building display adapters that provide support for legacy applications and existing hardware. Furthermore, the progressive benefit will allow for crucial flexibility, satisfying the needs of a range of applications and display technologies.

While our discussion of modern display challenges has largely involved the channel capacity requirements of emerging use cases, the implementation of these use cases must also be considered. A design that is data-driven, highly parallel, and asynchronous can easily scale to meet the demands of incredibly large displays such as jumbotrons where the pixel data is not highly volatile. Furthermore, these design principles ensure that an embedded mobile display can be implemented with simple digital logic instead of a CPU or GPU that consumes valuable power. Additionally, an asynchronous interface can accommodate low-capacity wireless data links with variable quality of service.

6. N-DIMENSIONAL DISPLAY INTERFACE

In this section, we present our proposed narrow waist abstraction and display interface, 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 4. 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 data. The following subsections describe each of the NDDI components in turn.

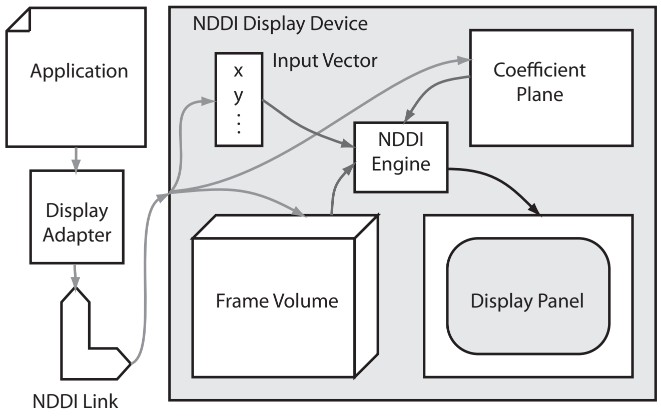


Figure 4: NDDI concept diagram.

6.1 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, for example, 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.

6.2 Coefficient Plane

The dimensionality of the Frame Volume most often will not match the display panel. 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.

6.3 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. The application defines the optional values based on the desired configuration of the coefficient plane. Furthermore, it can choose to either leave the values static or drive them in order to enact "global" changes through the resulting multiplication with each the coefficient matrices.

6.4 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 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 5. 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 Section 7.

6.5 NDDI Commands

Applications transfer and modify the contents of the NDDI memory stores via a number of high-level NDDI commands. These include:

Configure Configures an NDDI display by specifying the dimensionality $(xmax, ymax, zmax, ...)$ of the frame volume and the size of the input vector. The coefficient plane is configured based on this information as well as the dimensions of the attached display panel.

QueryDisplay Used to query the NDDI display for critical parameters (i.e. Panel dimensions, memory capacity).

PutPixel Copies a pixel to the specified location in the frame volume.

CopyPixels Copies an array of pixels to the frame volume, using the dimensions of the specified region as strides within the buffer.

CopyPixelStrip Copies a one dimensional array of pixels along a particular dimension in the frame volume.

FillPixel Fills a region of the frame volume with a specified pixel.

UpdateInputVector Used to update the input vector for configurations of the NDDI display that extend beyond the default x and y values.

PutCoefficientMatrix Used to copy a coefficient matrix into the specified location of the coefficient plane. The input matrix can be masked in order to preserve some of the values in the destination.

FillCoefficientMatrix Used to fill the specified coefficient matrix into a range of locations in the coefficient plane. The input matrix can be masked.

UpdatePanel An optional synchronization mechanism to signal when the NDDI engine should update the display panel.

7. 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 a 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 to leverage higher-level semantics.

7.1 Example: Video Player

A video playback application might configure the frame volume in three dimensions, with the x and y dimensions matching the video frame 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. Each coefficient matrix is based on a 4 x 4 translation affine transform matrix without the final row. Setting the tx and ty of matrix independently allows for features such as rudimentary scaling without fixed point support by replicating or omitting pixels periodically. The display updates would be driven by an input vector with the default x and y values, a third value representing the frame counter, c, and a fourth value for the affine transform. The application

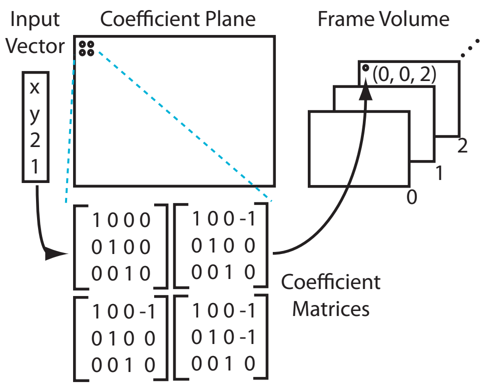


Figure 5: Video player with 4x scaling.

only needs to update that frame counter to display the next frame.

Figure 5 illustrates an example, with a display larger than the source video by a factor of two in each dimension. The coefficient plane matches the dimensions of the display panel, and therefore the samples from the source video must be duplicated. For the first group of four pixels, this is accomplished by setting tx and ty such that all four coefficients still pick the upper left pixel of the frame volume for the frame where $c = 0$.

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

7.2 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 dimension 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 n, tx, and ty values to choose and translate the window.

Figure 6 depicts an example with two windows. The first window is in the foreground and is rendered into the plane of the frame volume with $n = 0$. The window is anchored at location $(6, 6)$ of the display, and so the tx and ty is set to translate the pixels. The second window is rendered into the framebuffer at $n = 1$ at an offset of $(9, 2)$.

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

7.3 Example: Web Tablet

A web tablet device could leverage NDDI to overcome several challenges for tablet computers. Tablets are constrained

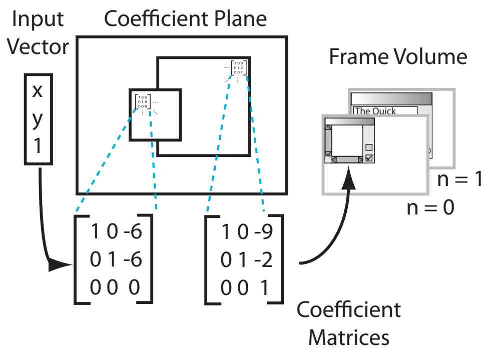


Figure 6: Windowed user interface.

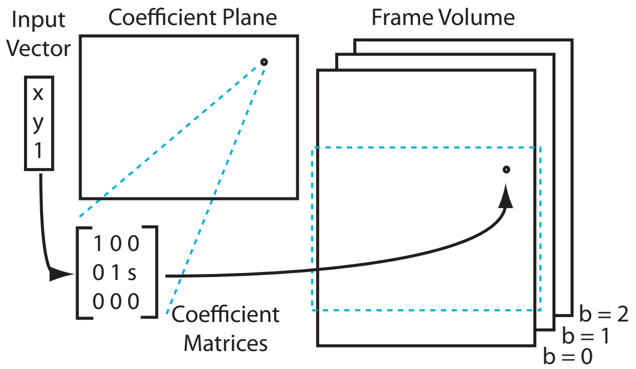


Figure 7: Web tablet with view port.

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 the viewport to be buffered and then rendered when the user scrolls. In this configuration, the tabs would be modal, and so they would be chosen and scrolled with b, tx, and ty.

Figure 7 shows a configuration with three tabs. The example coefficient matrix is choosing a pixel from the tab with $b = 0$ and a view port that is scrolled down by s.

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

7.4 Example: GUI Toolkit

Graphical user interface (GUI) toolkits allow application developers to quickly build the visual elements of a user interface while maintaining a consistent look and feel. These toolkits maintain a catalog of raster art that is used to render the various user interface components. The frame volume of an NDDI display is well suited as a storage for such a cata-

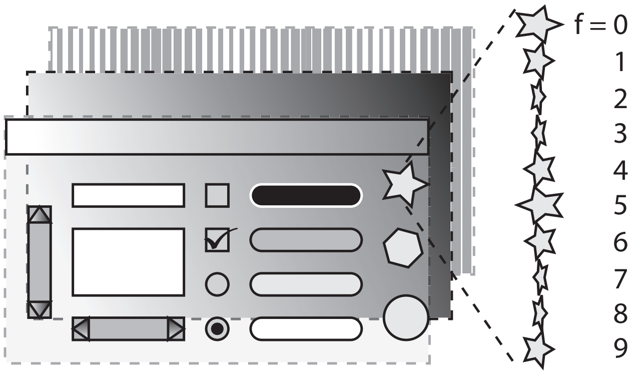


Figure 8: Framebuffer visualization for GUI.

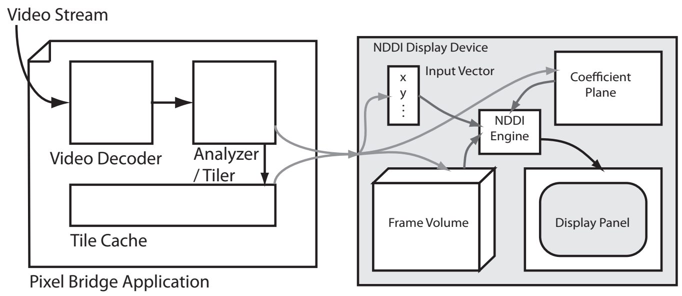


Figure 9: Pixel Bridge Application Diagram.

log. In the simple example illustrated in Figure 8, the frame volume is configured in four dimensions, x, y, z, and f. The x and y dimensions are at least as large as the dimensions of the display panel. The xy planes can be used to hold large images for desktop wallpaper, small images to construct GUI components, or sprites. In the illustration there are three xy planes. Two of the planes hold wallpaper images and one holds a collection of scrollbars, text fields, text boxes, a menu bar, several buttons with different states, and three individual sprites. Although it is not shown in the illustration, these planes can also be used to store font glyphs, animated wallpapers, or the results of rendering operations in the form of canvases.

The task of compositing an application's output for the display is now a matter of filling the coefficient matrices of the coefficient plane so that they choose and translate pixels from a particular xy plane. This effectively builds those components from the catalog of art already stored in the frame volume. Small updates resulting from user actions such as hovering over a button and changing its appearance are accomplished by setting just a few coefficients. Sprites require even fewer bytes to trigger their animations by taking advantage of the fourth dimension, f. Their individual animation frames are laid out in the f dimension as shown by the ten frames of the rotating star sprite in Figure 8. The animation is driven with the input vector, globally updating each sprite.

The equation below illustrates the format of the input vector and coefficient matrices. The input vector consists of default x and y values, a constant identity value for affine transforms, and the f value representing the animation frame number. Much like the other NDDI examples presented earlier, the coefficient matrix is based on a two dimensional affine transform, where tx and ty translate pixels values from one region of an xy plane to the particular location of the display. The z value in the coefficient matrix selects the particular xy plane to use. The last value in the coefficient matrix, s, can be thought of as a boolean flag indicating whether this pixel belongs to a sprite (or animated wallpaper). This prevents the f value in the input vector from affecting a region that is not a sprite. Any region of an xy plane that does not hold a sprite can be thought of as a sprite with only one "logical" frame. Setting that s value in the coefficient matrix to $false$ (0) ensures that only the pixels in that first logical frame are used. If the coefficient matrix is choosing a sprite, then the s value will be $true$ (1).

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

This example offers considerable promise for channel capacity savings. With the support of canvas xy planes, the application has a choice to either use art from the catalog in the frame volume, or render items to a canvas directly and then using regions. In our current design NDDI does not yet support a number of important operations such as alpha-blending that would be beneficial in this use case. With an alpha-blended copy, the artwork in a component from one xy plane can be composited to a canvas xy plane. Future research will define memory copy and move commands that will allow true compositing within the frame volume.

8. 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 (Figure 9), measuring the amount of throughput required. Pixel Bridge employs a video decoder to decode each frame of the computing session and passes the frame data to an analyzer based on the chosen configuration. Some configurations tile the frame and either update the entire NDDI display or only the changed tiles.

8.1 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 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

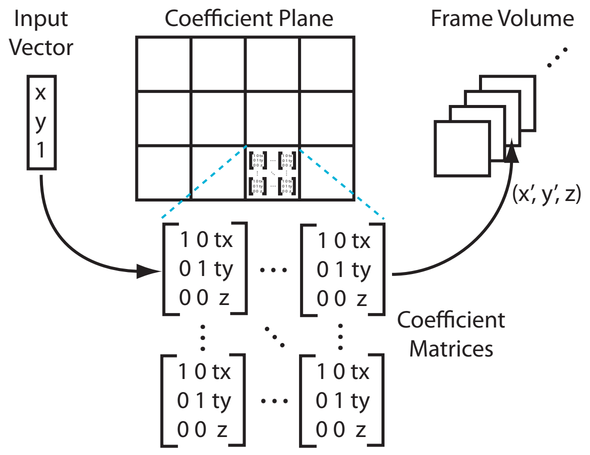


Figure 10: Cached tile mode NDDI configuration.

- **60 Hz** - Cost of full screen updates at 60Hz
- **Ideal Pixel Latching** - Cost of only changed pixels

8.1.1 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.

8.1.2 Flat Tiled Mode

The *Flat Tiled* mode configures the frame volume and initializes the coefficient plane 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.

8.1.3 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 (see Figure 10). 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.

8.1.4 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 driven at a constant 60 Hz rate. This should represent a worst-case performance bound 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 bound for Pixel Bridge.

8.2 Experiment Design

The experiment consists of recorded computing sessions as well as full-motion video. The computing sessions recorded the entire desktop of a Mac OS X computer at a resolution of 1920x1200 at 25+ fps. They were resized and transcoded to produce three different desktop resolutions. All of the sessions except for the *desktop* session were recorded using the respective applications in a maximized window. The resized computing session resolutions are small (960x600), medium (1440x900), and large (1920x1200), all at a constant 24 fps.

We recorded the sessions with recording software on the same host computer. Attempts to record using a lossless format did not produce a sufficiently high framerate due to disk IO limitations, and so the sessions were recorded using Apple Intermediate Codec (AIC) due to its streaming efficiency. The use of AIC for capture and then the use of H.264 for the resized, resampled, and transcoded videos does lead to a noticeable impact on visual quality. However, we believe that any artifacts produced can only have a negative effect on our results, making them more conservative. Future experiments may replace our methods of convenience and instead utilize an external recording device in order to reduce encoding errors. We expect doing so will further improve the results.

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

- **desktop** - Switching windowed applications on desktop
- **digg** - Web browsing on Digg.com
- **document** - Editing a document in OpenOffice
- **email** - Reading and writing emails
- **flickr** - Viewing photos on Flickr.com
- **maps** - Navigating a map on Google Maps
- **npr** - Web browsing on NPR.com
- **presentation** - Viewing a full-screen PDF presentation
- **slideshow** - Viewing a slideshow on Google Picasa Web
- **xcode** - Application development using Xcode
- **youtube** - Watching videos on YouTube.com

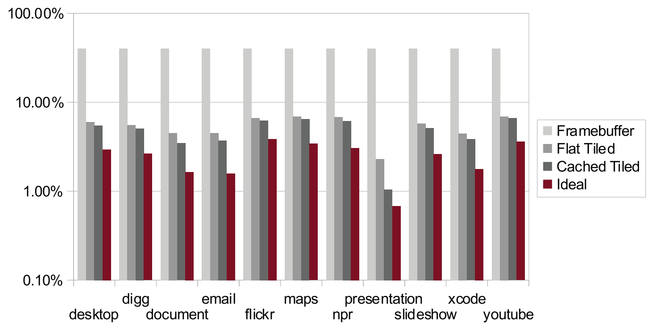


Figure 11: Small Session Results(960x600).

The full motion video clips are a combination of media obtained from a DVD and downloaded from the Internet[1]. The Bourne Identity DVD clips are 720x362. The Captain America Trailer and Limitless Clip are 1280x544 and 1920x816 respectively for their 720p and 1080p variations. The full motion video clips are all at a constant 23.9 fps.

The following full motion video clips were used as test vectors for the experiment:

- **bourne-dialog** - Clip of Character Dialog from Bourne Identity
- **bourne-moderate** - Clip of Moderate Activity from Bourne Identity
- **bourne-action** - Clip of Heavy Action from Bourne Identity
- **captain-720** - Official Captain America 720p Trailer
- **captain-1080** - Official Captain America 1080p Trailer
- **limitless-720** - Clip from Limitless at 720p
- **limitless-1080** - Clip from Limitless at 1080p

Each of the full motion videos and all three sizes of the eleven recorded sessions were tested with all five of the Pixel Bridge modes. For the tiled modes, the tile size was calculated to result in forty square tiles along the longest dimension (e.g. A 640x480 video would use 16x16 tiles since 640/40 = 16). For the Cached Tile mode, a cache of 10,000 tiles was used. The initial run of each full motion video processed 800 frames. A second run was performed which played 600 frames forward, then played the previous 200 frame in reverse, then played the next 400 frame forward. The runs for the computing sessions ran to completion and their length varied based on the duration of the recordings. Each experiment tracked the number of bytes transmitted over the NDDI link without additional compression.

8.3 Results and Analysis

The results for the recorded computing sessions are shown in Figures 11, 12, and 13, with each data point representing the ratio of the bytes transferred "over the wire" for that mode versus the 60 Hz mode. Please note that the y-axis on these first three figures use a log scale. Recall that the 60 Hz mode represents the current status quo of transferring the entire framebuffer at a constant refresh rate suitable for modern LCD monitors. For these recorded sessions, Pixel Bridge demonstrates progressively better performance from Framebuffer mode to Flat Tiled 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 illustrates the advantage of updating the display at a framerate that matches the source content (24 Hz) instead a fixed refresh rate (60 Hz) yielding a ratio of 40%. The Flat Tiled mode brings the advantage of only updating the changed regions of the display. This proves to be the simplest yet most substantial gain, with nearly every result below 10%. The Cached Tiled mode shows the gains of filling the display with identical tiles as well as using previously cached tiles.

The results of the full motion video clips are shown in Figures 14 and 15. They are quite contrary to the results for the recorded computing sessions. The same 40% ratio is achieved with the Framebuffer mode. The Flat Tiled mode only brings mild savings for the Captain America trailer, which was filled with multiple short and dramatic clips. The Bourne Identity clips prove to be too small for the Cached Tiled mode to provide a benefit. It actually produces results worse than the Framebuffer mode. The NDDI overhead of updating the coefficient plane negates any benefit from the cached tiling.

Cached Tiled mode requires a careful balance of cache hits and misses in order to overcome the NDDI overhead. Specifically, there are three outcomes when attempting to update a tile: *unchanged*, *cache hit*, and *cache miss*. The following equations represent the cost (C) in bytes of each mode based on the total pixels (P_{total}), pixels per tile (P_{tile}), number of frames (F), and statistics of changed tiles ($T_{changed}$), unchanged tiles ($T_{unchanged}$), cache hits (T_{hit}), and cache misses (T_{miss}). The constants represent the number of bytes per pixel (4), the addressing overhead for updating an entire two dimensional frame volume (16), the addressing overhead of updating a flat tile in a two dimensional frame volume (16), the addressing overhead of updating a tile in the cache of a three dimensional frame volume (24), and the cost of filling a region of the coefficient plane with a single coefficient (28).

$$C_{framebuffer} = F \cdot (P_{total} \cdot 4 + 16) \tag{1}$$

$$C_{flat} = T_{changed} \cdot (P_{tile} \cdot 4 + 16) \tag{2}$$

$$C_{cached} = T_{hit} \cdot 28 + T_{miss} \cdot (P_{tile} \cdot 4 + 16 + 28) \tag{3}$$

In order for the Cached Tiled mode to outperform the Flat Tiled mode, the cache hit rate must be substantial enough to overcome the 28 byte penalty for updating the coefficient matrices of each changed pixel. In an attempt to raise the cache hit, we introduced a novel change to the tile matching algorithm for determining cache hits. Instead of computing exact checksums of the tiles being matched, we only compute checksums of n significant bits per channel. It is

[1]All videos were obtained legally and their use for academic research is permissible.

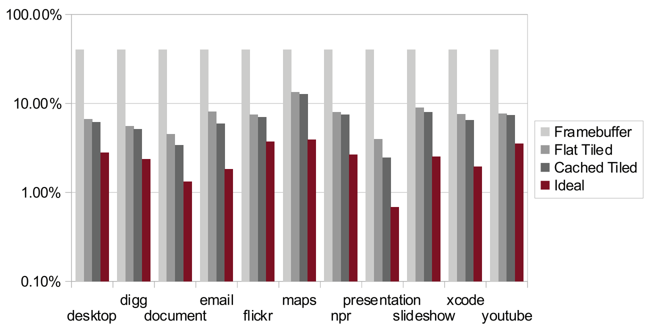


Figure 12: Medium Session Results (1440x900).

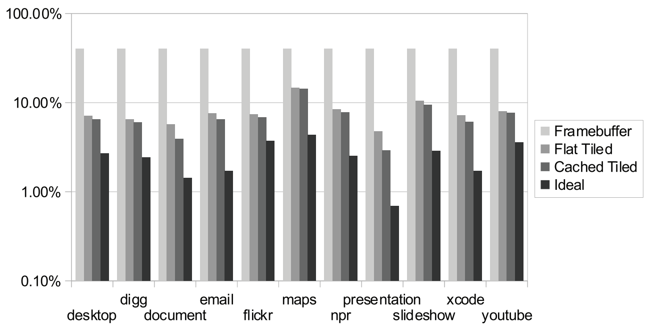


Figure 13: Large Session Results(1920x1200).

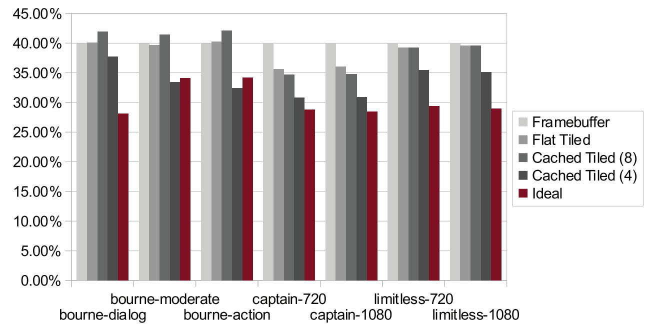


Figure 14: Video Results.

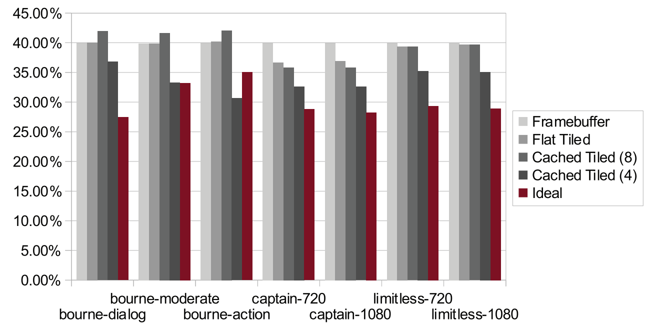


Figure 15: Video Rewind Results.

Video	Hit % (8)	Hit % (4)	PSNR (4)
bourne-dialog	0.01%	1.20%	48.73
bourne-moderate	0.23%	2.57%	40.83
bourne-action	0.36%	9.43%	38.93
captain-720	2.42%	2.20%	38.06
captain-1080	3.07%	2.02%	36.71
limitless-720	0.27%	1.75%	46.93
limitless-1080	0.16%	1.63%	46.66

Table 1: Cached Tiled Mode Statistics.

important to note that this does not imply that the colors of the displayed tiles are quantized. If a match is made using the n-bit checksum for a tile in the cache, then that full-color tile from cache will be displayed. Figures 14 and 15 also include the "Cached Tiled (8)" and "Cached Tiled (4)" data representing the use all of eight bits per channel and then just four bits per channel to determine tile matches. With the relaxed matching, we are able to outperform Flat Tiled mode and in some cases outperform the Ideal Pixel Latching mode. Table 1 shows the cache hit percentage for "Cached Tiled (8)", the improved cache hit percentage for "Cached Tiled (4)", and the corresponding peak signal to noise ration (PSNR) for "Cached Tiled (4)" using only the simple 800 frame forward playback test runs. Subjectively, the output was similar to standard definition web video with the addition of noticeable patterns when a single tile from the cache was reused for several other tiles in a frame.

Full motion video represents highly dynamic content which does not match well with the title cache abstraction of Pixel Bridge. Through the innovative tile matching, we were able to achieve results with acceptable PSNR and visual quality similar to standard definition streaming Internet video. 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.

9. CONCLUSIONS AND FUTURE WORK

The framebuffer abstraction has ably served as the narrow waist of the display pipeline from the earliest analog display technologies through to today's digital standards. With the aid of the higher channel capacity of these new standards, the framebuffer has kept pace with the increasing demands of evolving display technology despite the inherent inefficiency of repeatedly updating the entire frame. DPVL is the first standard to challenge the framebuffer by effectively splitting it between the display adapter and the display device. However, DPVL is far from a disruptive technology, and is still only a mild evolution of the framebuffer. We argue that the vital backward compatibility with the framebuffer can still be achieved even while using a divergent abstraction. NDDI provides framebuffer compatibility while also allowing for a stronger benefit for applications that more progressively utilize the NDDI resources. This tradeoff between complexity and channel capacity savings is the fundamental strength of NDDI making it an ideal candidate for a new, more elastic narrow waist.

Our initial Pixel Bridge experiment represents a "base case", bridging pixels from a computing session to an NDDI display. It demonstrated progressive benefit 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. The examples explored in this paper are ideal candidates for future research, especially the Video Player. Full-motion video was only marginally improved with our Pixel Bridge use case, and stands to benefit tremendously if key areas of the video decoding pipeline leverage NDDI. This will undoubtably lead to new features for NDDI, but great care will be taken to ensure that such enhancements do not violate the integrity of our design goals. Instead of simply building an advanced video codec into NDDI, we will opt for smaller enhancements such as support for fixed-point numbers allowing for sub-pixel motion vectors. Support for alternate color spaces is another such enhancement.

Other uses cases will likely drive NDDI in new directions. The GUI toolkit example illustrated the need for common primitives such as memory copy, move, and alpha blending. Furthermore it was the first example that configured the frame volume with a fourth dimension. While we have not introduced a limit on the dimensionality to the frame volume, allowing it to grow unbounded will present severe engineering challenges. Instead, we hope to introduce some form of virtual memory management that allow for higher dimensionality without allocating memory until it is initialized by the application. However, any such feature will need to adhere to our data driven, parallel, and asynchronous design and will be evaluated against an expanded cost model that captures statistics on the frequency, size, and regions for each memory access. We ultimately intend to take NDDI beyond simulation and build a hardware prototype, implementing much of the NDDI engine in an FPGA. The cost model will aid in evaluating varying candidate memory architectures for feasibility and power efficiency.

10. 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 digital packet video link standard: Version 1. Video Electronics Standards Association, April 2004.

[16] Vesa dpvl software interface standard: Version 1. Video Electronics Standards Association, February 2006.

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

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