Technical Brief

10 and 12-bit Grayscale Technology for NVIDIA® Quadro®

April 2009| TB-04631-001_v01

Document Change History

Version	Date	Responsible	Description of Change
01	April 17, 2009	SV, SM	Initial Release

Table of Contents

List of Figures

List of Tables

10 and 12-bit Grayscale Technology

Introduction

Advances in sensor technology and image acquisition techniques in the field of radiology are producing high bit depth grayscale images in the range of 12 to 16-bit per pixel. At the same time, the adoption of displays with native support for 10 and 12-bit grayscale is growing. These affordable displays are DICOM[1] conformant to preserve image quality and consistency. Furthermore, tiling together multiple such displays enables side-by-side digital study comparisons driven by a single system.

Standard graphics workstations however are limited to 8-bit grayscale, which provides only 256 possible shades of gray for each pixel sometimes obscuring subtle contrasts in high density images. Radiologists often use window-leveling techniques to identify the region of interest that can quickly become a cumbersome and time-consuming user interaction process.

NVIDIA's 10–bit and 12-bit grayscale technology allows these high quality displays to be driven by standard NVIDIA® Quadro® graphics boards preserving the full grayscale range. By using "pixel packing" the 10-bit or 12-bit grayscale data is transmitted from the Quadro® graphics board to a high grayscale density display using a standard DVI cable. Instead of the standard three 8-bit color components per pixel, the pixel packing allows two 10 or 12-bit pixels to be transmitted, providing higher spatial resolution and grayscale pixel depth as compared to an 8-bit system.

As specialty hardware is not required, NVIDIA's 10-bit grayscale technology is readily available for use with other radiology functions and easy to support amongst a wide range of grayscale panels from various manufacturers. In a preliminary study performed on 10 radiologists using Dome E5 10-bit vs. E5 8-bit displays in conjunction with Three Palms 10-bit, OpenGL accelerated WorkstationOne mammography application, radiologists' performance was statistically significant on the 10-bit enabled display systems, some experiencing triple the read time speedup.

This technical brief describes the NVIDIA grayscale technology, the system requirements and setup. It also aims to guide users through common pitfalls that arise when extending to multi-display and multi graphics processing unit (GPU) environments routinely used in diagnostic imaging and recommends best practices.

Figure 1 shows the latest technology in digital diagnostic display systems, a Quadro card driving a 10 mega-pixel, 10-bit grayscale display. Figure 2 shows a 10-bit enabled mammography application displaying multiple modalities on multiple displays.

Figure 1. 10 MPixel, 10-Bit Diagnostic Mammography Display[1]

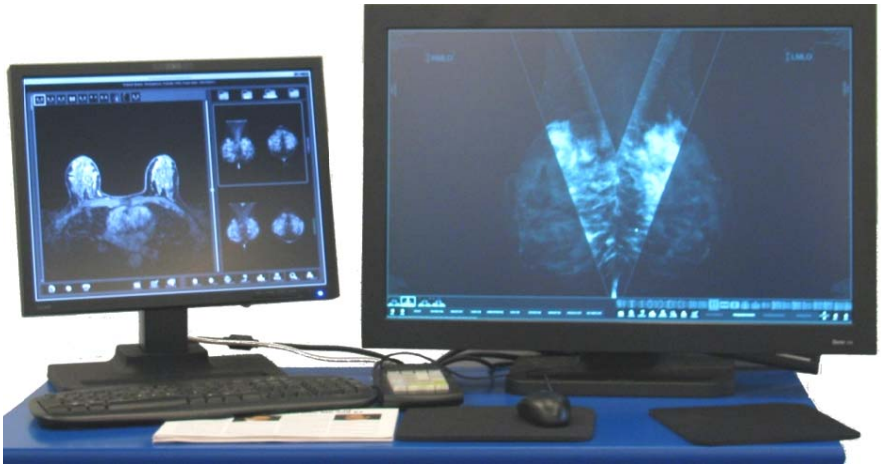

Figure 2. Application Enhanced Using Multiple Displays[2]

[1] Image courtesy of NDS Surgical Imaging, DOME Z10.

[2] Image courtesy of Threepalms, Inc.

System Specific Information

- 10 and 12-bit grayscale currently requires Windows XP.
- Windows Vista support for 10-bit grayscale over DVI is being worked on.
- Grayscale is only supported for OpenGL based applications.

Supported Graphics Boards

10-bit grayscale is supported on Quadro FX graphics boards shown in Table 1. The graphics boards are G80 and higher. The graphics boards are NVIDIA CUDA™ enabled.

Table 1. Graphics Boards with 10 and 12-Bit Grayscale Support

Quadro FX 3800 	Mid-range card with 1 GB of graphics memory. Recommended if the primary usage is to display 2D grayscale images and some 3D data.
Quadro FX 4800 	High-end card with 1.5 GB of graphics memory and 2 DisplayPort outputs. Recommended for applications that also require rendering large 3D.
Quadro FX 5800 	Ultra-high end card with 4 GB of graphics memory. Recommended for applications that also deal with large datasets such as 4D geometries and volumes.
Quadro Plex 2200 D2 	Dedicated deskside visual computing system composed of 2 Quadro FX 5800 graphics boards with a total of 8 GB of graphics memory. Recommended for advanced visualization and large scale projection and display use cases.

Supported Monitors

The monitor should be capable of 10 and 12-bit outputs. We currently support the following displays.

- **NDS Surgical Imaging Dome E5 5MP and Z10 10MP display's [2]**
- **Eizo Radiforce GS520 5MP display[3]** – currently in beta, to be released in the R190 driver.

Supported Connectors

- **Single or Dual-link DVI**

  Although single-link DVI is only capable of transmitting up to HD (1920 × 1200), our grayscale pixel packing mechanism allows 5 MP (2560 × 2048) images to be sent over single-link DVI.

- **DisplayPort**

  This applies to the Quadro FX 4800 and the Quadro FX 5800 that have DisplayPort outputs. As grayscale monitors currently only support DVI, a DisplayPort-to-single and dual DVI adaptors is needed at the GPU end. The Bizlink dongle (P/N 030-0223-0000) shown in Figure 3 has been tested and is recommended.

Figure 3. DisplayPort to DVI Bizlink Dongle

Grayscale Monitor Settings

When a grayscale compatible monitor is connected to a suitable NVIDIA board, the NVIDIA driver automatically detects it and immediately switches to packed pixel mode. Therefore, there are no control panel settings to enable and disable 10-bit grayscale. The only setting required is to enable the grayscale monitor to display at a maximum resolution of 2560 × 2048. Follow these simple steps.

1. Open the **Display Properties**.
2. Select the **Settings** tab.
3. Click on **Advanced**.
4. Select the **Monitor** tab.
5. Uncheck the **Hide modes that this monitor cannot display** check box.
6. Click **Apply**. The maximum resolution is now set to 2560 × 2048.

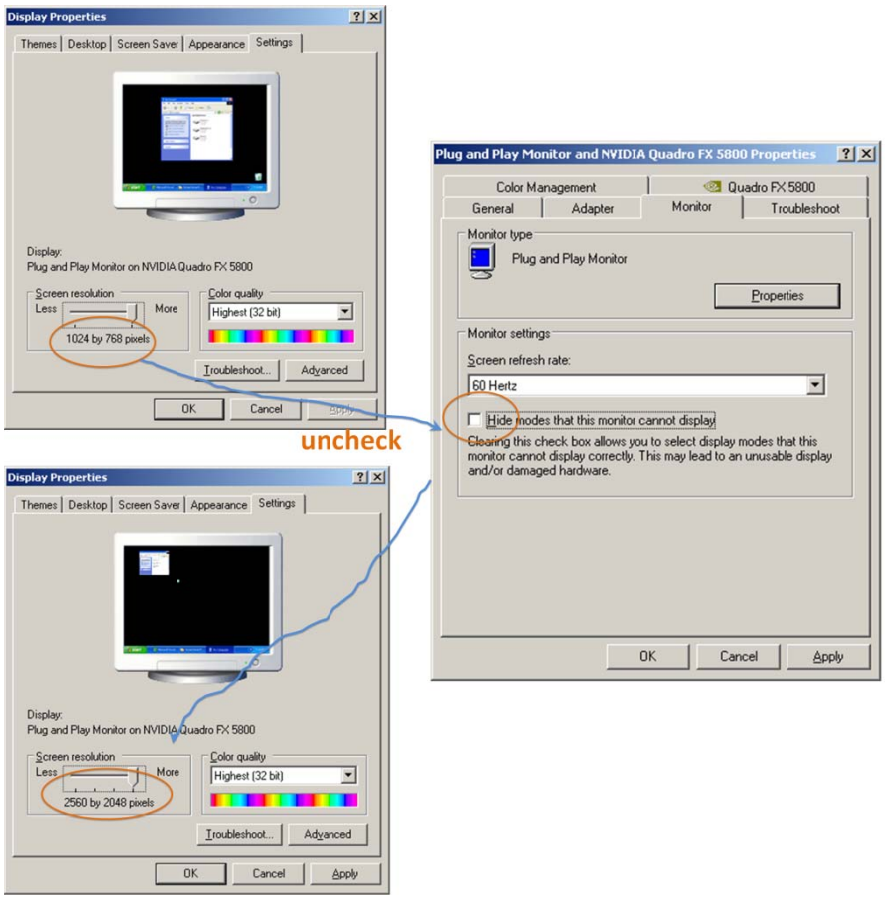

Figure 4. Enable Grayscale Monitor to Display Higher Resolution

Grayscale Implementation

Driver Layer

On grayscale enabled Quadro boards, the driver implements a pixel packing mechanism that is transparent to the desktop and to the application. The 24-bit RGB desktop is first converted to 12-bit grayscale using the NTSC color conversion formula and then two 12-bit gray values are packed into 1 RGB DVI pixel and finally shipped to the monitor. This pixel packing allows displaying of 5 MP gray values just using a single-link DVI (that is normally limited to HD resolution).

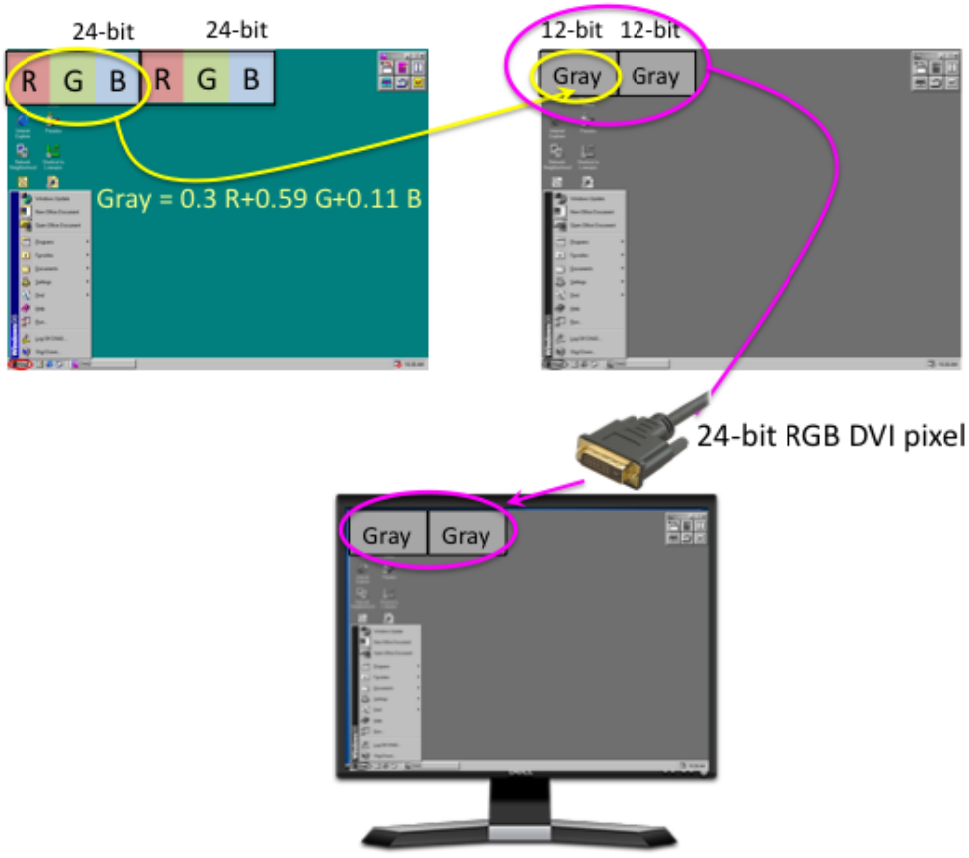

Figure 5. Driver Converts and Packs Desktop from 24-Bit Color to 12-Bit Gray

Application Layer

The 10 and 12-bit grayscale image viewing application is responsible for outputing 24-bit RGB pixels which the driver then converts to 12-bit grayscale values for scanout as described in the previous section.

The application uses a shader that takes in the 12-bit grayscale value from the image and translates it into a 24-bit RGB pixel using a lookup table.The lookup table is generated to find the best RGB pixel with as little as possible differences between the RGB values (preferred is R=G=B) for each grayscale value in the input image. In essence, this process is the inverse of the driver conversion from RGB to grayscale. The end result is that the grayscale image on the desktop looks like a grayscale image on a color monitor.

The integer texture extension, EXT_texture_integer [4] in Shader Model 4 is used to store the incoming grayscale image as a 16-bit unsigned integer without converting to floating point representation saving memory footprint by 2×.

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 2);
glTexImage2D(GL_TEXTURE_2D, 0, GL_ALPHA16UI_EXT, width, height, 0,
GL_ALPHA_INTEGER_EXT , GL_UNSIGNED_SHORT, TextureStorage);
```

The lookup table mapping the grayscale image to 24-bit RGB values is stored as 1D texture. The lookup table dimensions should exactly match the bit depth of the grayscale values expected in incoming image so that no filtering and interpolation operations will be performed thus preserving image precision and fidelity. Changes to contrast, brightness and window level of the image are easily done by changing the lookup table resulting in a 1D texture download without any change to the sourceimage.

```
glBindTexture(GL_TEXTURE_1D, lutTexId);
glTexImage1D(GL_TEXTURE_1D, 0, 4, lutWidth, 0, GL_RGBA,
       GL UNSIGNED BYTE, Table );
```

At run time, the applicaton draws a quad that is texture mapped with the grayscale image. In the rasterization stage, the fragment shader is invoked for each grayvalue which then does a dependant texture fetch into the 1D LUT texture. The complete source is found in `GrayScaleDemo.cpp`.

```
#extension GL_EXT_gpu_shader4 : enable // for unsigned int support
uniform usampler2D texUnit0; // Gray Image is in tex unit 0
uniform sampler1D  texUnit1; // Lookup Table Texture in tex unit 1
void main(void)
{
 vec2  TexCoord  = vec2(gl_TexCoord[0]);
 //texture fetch of unsigned ints placed in alpha channel
 uvec4 GrayIndex = uvec4(texture2D(texUnit0, TexCoord));
 //low 12 bits taken only ;
 float GrayFloat = float(float(GrayIndex.a) / 4096.0);
 //fetch right grayscale value out of table
 vec4  Gray      = vec4(texture1D(texUnit1, GrayFloat));
 // write data to the framebuffer
 gl_FragColor  = Gray.rgba;
}
```

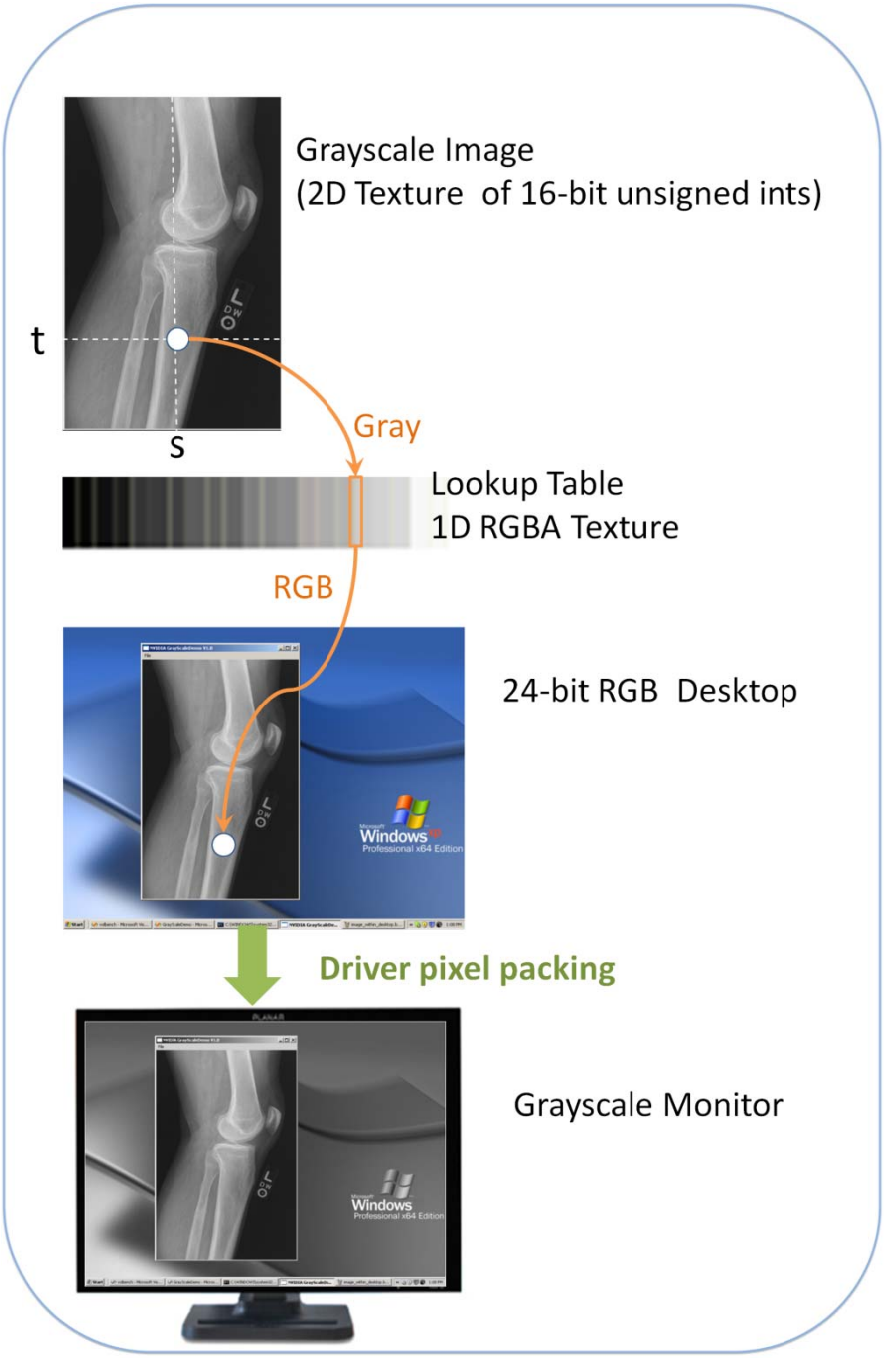

Figure 6. Application Level Texture Setup for 10 and 12-Bit Grayscale Display

Multi-Display Configurations

Diagnostic imaging commonly requires multiple displays for side by side modality comparisons. Multi-display configurations are becoming more practical with systems capable of supporting multiple graphics boards that in turn drive multiple displays. A single Quadro board can drive a maximum of 2 displays. Depending on the available PCI slots within a system, multiple cards can be used to drive several displays. These multiple displays can be a mix of regular color LCD panels and specialty grayscale monitors. This section explains the issues that arise from such a heterogeneous configuration and programming pointers to address them. The full source code for the examples is found in the accompanying Grayscale10-bit SDK

Multi-GPU Compatibility

Grayscale capable Quadro boards can be mixed with other Quadro boards that can drive one or many side displays as shown in Table 2. These "Side Display GPU's" may not yield the grayscale effect but the system will be compatible. Mixing of GPU's is only guaranteed to work if the GPU's are G80 and later.

Note: The mixing of older cards (pre G80) is not supported in grayscale configurations.

Table 2. Multi-GPU Compatibility

Side Display GPU \ Grayscale GPU	Quadro FX 3800	Quadro FX 4800	Quadro FX 5800	Quadro Plex
Quadro NVS 290	✓	✓	✓	✓
Quadro FX 1800	✓	✓	✓	✓
Quadro FX 3800	✓	✓	X	X
Quadro FX 4800	✓	✓	X	X
Quadro FX 5800	X	X	X	✓

Note: These are theoretical compatibilities. In practice, the physical system attributes such as availability of PCI slots and their placements will determine the final working set of cards from Table 2. The Quadro FX 5800 requires the full 2 auxiliary power inputs and therefore is only used with lower-end Quadro cards that do not have any auxiliary power requirements.

Multiple Display Setup

To enable multi-display from the desktop follow these simple steps.

1. Open the **Display Properties.**
2. Select the **Settings** tab.
3. Check the **Extend my Windows desktop onto this monitor** checkbox for each display as shown in 7.

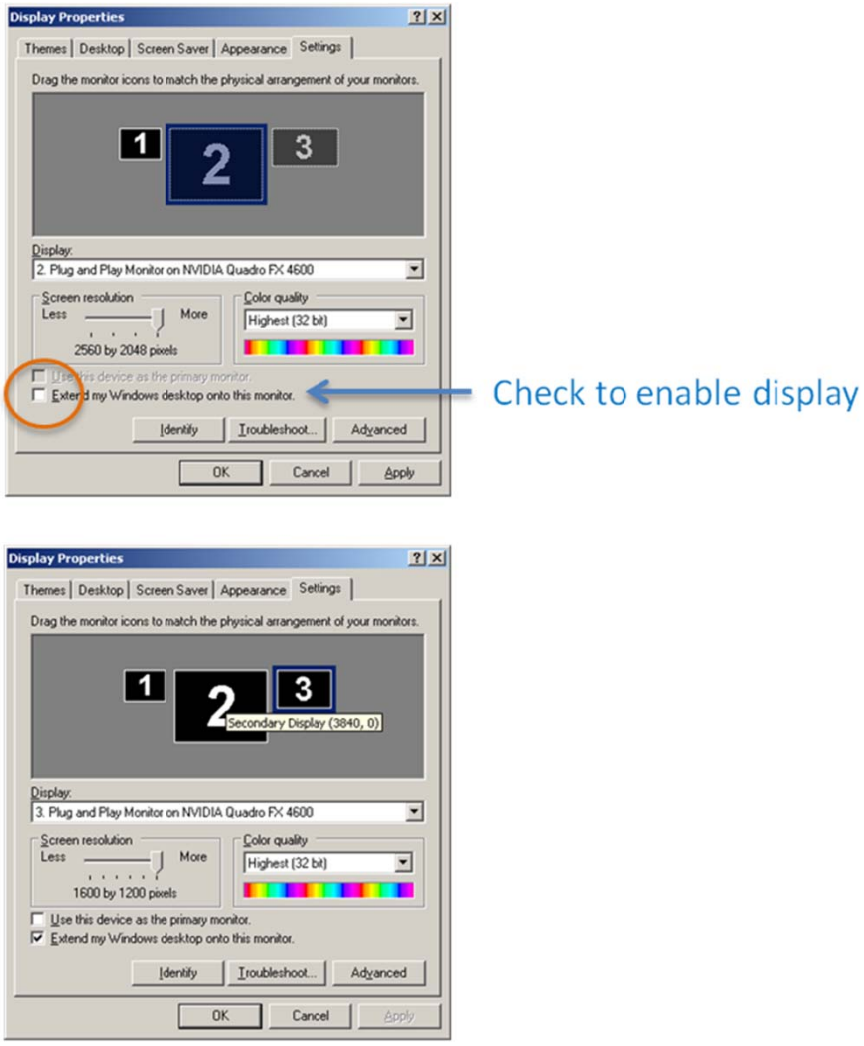

Figure 7. Display Properties Before and After Displays are Enabled

For an application using multiple GPU's and displays it is often useful to programmatically find out their attributes and capabilities. This section and the following ones show code samples to demonstrate that in progressive detail. Following are some data structures used throughout the document examples. The `CDisplayWin` structure defined in `CDisplayWin.[h|cpp]`encapsulates the attributes of each display and the `displayWinList` is a container for all displays. Accessor functions have been omitted to aid readability.

```
class CDisplayWin {
    HWND     hWin; // handle to display window
    HDC      winDC; // DC of display window
    RECT     rect; // rectangle limits of display
    bool     primary; //Is this the primary display
    char     displayName[128]; //name of this display
    char     gpuName[128]; //name of associated GPU
    bool     grayScale; //Is this a grayscale display
public:
    bool     spans(RECT r);//If incoming rect r spans this display

}
#define MAX_NUM_GPUS 4
int displayCount = 0; //number of active displays
//list of displays, each gpu can attach to max 2 displays
CDisplayWin displayWinList[MAX_NUM_GPUS*2];
```

Following is a simple example using the Windows GDI to enumerate the attached displays, gets their extents and also check if the display is set as primary. The following code can be easily modified to include unattached displays.

```
DISPLAY_DEVICE  dispDevice;
DWORD displayCount = 0;
memset((void *)&dispDevice, 0, sizeof(DISPLAY_DEVICE));
dispDevice.cb = sizeof(DISPLAY_DEVICE);
// loop through the displays and print out state
while (EnumDisplayDevices(NULL,displayCount,&dispDevice,0)) {
if (dispDevice.StateFlags & DISPLAY_DEVICE_ATTACHED_TO_DESKTOP) {
    printf("DeviceName    = %s\n", dispDevice.DeviceName);
    printf("DeviceString  = %s\n",dispDevice.DeviceString);
    if (dispDevice.StateFlags &DISPLAY_DEVICE_PRIMARY_DEVICE)
        printf("\tPRIMARY DISPLAY\n");
    DEVMODE devMode;
    memset((void *)&devMode, 0, sizeof(devMode));
    devMode.dmSize = sizeof(devMode);
    EnumDisplaySettings(dispDevice.DeviceName, ENUM_CURRENT_SETTINGS,
                                &devMode);
    printf("\tPosition/Size = (%d, %d), %dx%d\n",
    devMode.dmPosition.x,      devMode.dmPosition.y,devMode.dmPelsWidth,
    devMode.dmPelsHeight);
    HWND hWin =
    createWindow(GetModuleHandle(NULL),devMode.dmPosition.x+50,
    devMode.dmPosition.y+50, devMode.dmPelsWidth-50,
    devMode.dmPelsHeight-50);
    if (hWin) { //got a window
        HDC winDC = GetDC(hWin);
        // TODO - set pixel format, create OpenGL context
    }
    else
        printf("Error creating window \n");
    }//if attached to desktop
    displayCount++;
} //while(enumdisplay);
```

Running this enumeration code on our 3 display example (shown in Figure 7) prints out the following.

```
DeviceName    = \\.\DISPLAY1
DeviceString  = NVIDIA Quadro FX 1800
PRIMARY DISPLAY
Position/Size = (0, 0), 1280x1024

DeviceName    = \\.\DISPLAY2
DeviceString  = NVIDIA Quadro FX 4800
Position/Size = (1280, 0), 2560x2048

DeviceName    = \\.\DISPLAY3
DeviceString  = NVIDIA Quadro FX 4800
Position/Size = (3840, 0), 1600x1200
```

Note: The enumeration shown in this section abstracts special hardware capabilities of the displays such as grayscale or color capability. For such physical display details, we need access to the Extended display identification data (EDID)-the data structure provided by the computer display to the graphics card. This is described in the next section.

Mixing Grayscale and Color Displays

The previous section demonstrated how to get the general characteristics of a display such as extent etc, but more specific properties of monitors will decide how to layout our application. For example, user interface and launching elements are normally placed on the regular color LCD's while the radiological images will be rendered to the grayscale displays. A display is defined to be grayscale compatible if both the monitor and the GPU attached are grayscale enabled. To determine if a monitor is grayscale we parse its EDID to get the model name and compare it with the list of enabled monitors. This EDID is provided by the NVIDIA NVAPI [5] – an SDK that gives low level direct access to NVIDIA GPUs and drivers on all windows platforms. The following example shows enumerating the attached displays and its associated panel and GPU string. Refer to the complete source in `CheckGrayscale.cpp` for error checking functions and the `isGrayscaleGPU` and `isGrayscaleMonitor` string parsing functions.

```
// Declare array of displays and associated grayscale flag
NvDisplayHandle hDisplay[NVAPI_MAX_DISPLAYS] = {0};
NvU32 displayCount = 0;
// Enumerate all the display handles
for(int i=0,nvapiStatus=NVAPI_OK; nvapiStatus == NVAPI_OK; i++) {
    nvapiStatus = NvAPI_EnumNvidiaDisplayHandle(i, &hDisplay[i]);
    if (nvapiStatus == NVAPI_OK) displayCount++;
}
printf("No of displays = %u\n",displayCount);

//Loop through each display to check if its grayscale compatible
for(unsigned int i=0; i<displayCount; i++) {
    //Get the GPU that drives this display
    NvPhysicalGpuHandle hGPU[NVAPI_MAX_PHYSICAL_GPUS] = {0};
    NvU32 gpuCount = 0;
    nvapiStatus =
    NvAPI_GetPhysicalGPUsFromDisplay(hDisplay[i],hGPU,&gpuCount);
    nvapiCheckError(nvapiStatus);
```

```
    //Get the GPU's name as a string
    NvAPI_ShortString gpuName;
    NvAPI_GPU_GetFullName (hGPU[0], gpuName);
    printf("Display %d, GPU %s",i,gpuName);
    nvapiCheckError(nvapiStatus);

    //Get the display ID for subsequent EDID call
    NvU32 id;
    nvapiStatus = NvAPI_GetAssociatedDisplayOutputId(hDisplay[i],&id);
    nvapiCheckError(nvapiStatus);

    //Get the EDID for this display
    NV_EDID curDisplayEdid = {0};
    curDisplayEdid.version = NV_EDID_VER;
    nvapiStatus = NvAPI_GPU_GetEDID(hGPU[0],id,&curDisplayEdid);
    nvapiCheckError(nvapiStatus);

    //Check if the GPU & monitor both support grayscale
    //and set the grayFlags table
    if (isGrayscaleGPU(gpuName)&& \\
        isGrayscaleMonitor(curDisplayEdid.EDID_Data,NV_EDID_DATA_SIZE))
        displayWinList[i].grayScale = true;
    else
        displayWinList[i].grayScale = false;
}
```

Moving and Spanning Windows Across Displays

Many applications allow users to freely move windows across multiple displays. It may be desirable for some applications to prevent spanning a grayscale image window to a color display. In the event-handling code for a window move and resize, we query all the displays that the current window spans to check for grayscale compatibility. The following code snippet refers to the data structures populated in the previous examples to do the runtime query.

```
LONG WINAPI winProc(HWND hWin, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg) {
        case WM_SIZE:
        RECT rect;
        GetClientRect(hWin, &rect);
        for (int i=0;i<displayCount;i++) {
           //check if the window spans this display
           if (displayWinList [i].spans(rect)) {
                //Now check this is grayscale compatible display
                if (!displayWinList[i].grayScale) {
                //do something eg prevent spanning
                }
            }
        } //end of for
         break;
        case WM_MOVE:
        RECT rect;
        //Repeat as above for WM_SIZE

    }
}
```

Targeting Specific GPUs for Rendering

The default behavior is for OpenGL commands to be sent to all GPUs. While this works for many applications, it makes runtime graphics capability checking and handling more complicated. Therefore, it is desirable to limit grayscale rendering to the GPUs that are capable of grayscale output. In this case, when the window moves to a display connected to a GPU where grayscale is not enabled, no screen refresh or drawing happens. In fact, some applications prevent window movements at all, minimizing user interaction to increase efficiency. To target specific GPUs for rendering, we use the WGL NV Affinity extension [6] available for Windows on Quadro professional cards.

The GPU Affinity for a window is defined by an affinity mask that contains a list of GPUs responsible for the window drawing. This extension also introduces the concept of an affinity DC which is simply a device context embedded with the affinity mask. When an OpenGL context is created from this DC it inherits the DC's affinity mask that is immutable. For on-screen drawing, when this affinity context is associated with a window DC, any OpenGL calls made with this context current will be sent to the GPUs specified in the affinity mask.

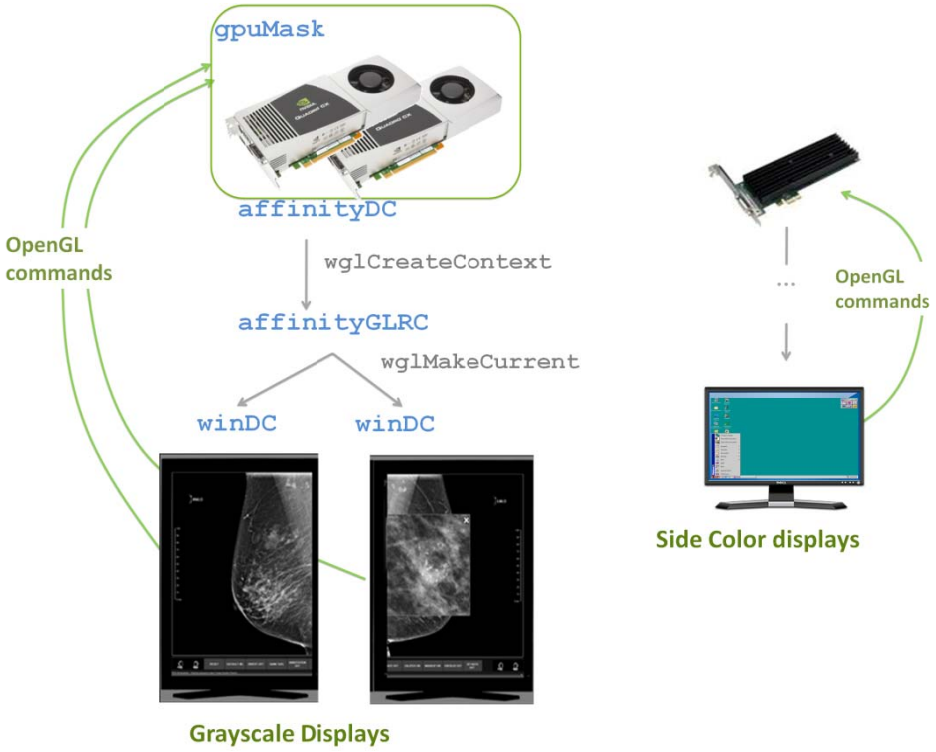

Figure 8. Using Affinity Extension to Target Specific GPUs for OpenGL Rendering

We introduce another class, `CAffGPU` to encapsulate all the attributes for an Affinity GPU and the `affGPUList` which is a collection of `CAffGPU's`.

```
class CAffGPU {
    HDC     affinityDC; // Device Context of affinity gpu
    HGLRC   affinityGLRC; // OpenGL Resource Context
public:
    init(HGPUNV* pGPU, int num); //List of GPU handles in the mask
    ~ CAffGPU();
};
unsigned int gpuCount = 0;
CAffGPU affGPUList[MAX_NUM_GPUS]
```

To encapsulate the displays attached to affinity GPUs, the class `CAffDisplayWin` is extended from the existing `CDisplayWin` class to include a pointer to the `CAffinityGPU` instance that is responsible for its rendering.

```
class CDisplayWin {
    ...
    CAffGPU* pAffinityGPU; //The list of GPU's responsible for rendering
this window
    ...
};
```

The following `CAffinityGPU::init` initialization function shows the affinity DC and OpenGL resource context are created for an affinity GPU instance with just one physical GPU specified in the mask. Of course, multiple physical GPUs can be associated with one affinity GPU using this affinity mask for more complex rendering topologies.

```
//Get affinity DC and RC for this GPU.
//In the case below one GPU is associated with
void CAffinityGPU::init(HGPUNV* hGpu, int num) {
    // Assume just 1 GPU in the list for simplicity
    HGPUNV  gpuMask[2];
    gpuMask[0] = *hGpu;
    gpuMask[1] = NULL;
    //Create affinity-DC
    if (!(affinityDC = wglCreateAffinityDCNV(gpuMask)))
        ERR_MSG("Unable to create GPU affinity DC");
    //Set the pixel format for the affinity-DC
    setPixelFormat(affinityDC);
    //Create affinity-context from affinity-DC
    if (!(affinityGLRC = wglCreateContext(gpuDC)))
        ERR_MSG("Unable to create GPU affinity RC");
}
```

Handles for all the system GPUs are enumerated by the following `wglEnumGpusNV` call. This example also shows another way of enumerating the display devices using the `wglEnumGpuDevicesNV` and GPU_DEVICE structure that resemble closely the windows GDI `enumDisplayDevices` and DISPLAY_DEVICE introduced earlier.

```
HGPUNV curNVGPU;
//Get a list of GPU's
while ((gpuCount < MAX_NUM_GPUS) && wglEnumGpusNV(gpuCount, &curNVGPU)) {
    unsigned int curDisplay = 0; //displays per current GPU
    GPU_DEVICE  gpuDevice;
    gpuDevice.cb = sizeof(gpuDevice);
    affGPUList[gpuCount].init(&curNVGPU,1);
    //loop through displays devices for this GPU
    while (wglEnumGpuDevicesNV(curNVGPU, curDisplay, &gpuDevice)) {
        displayWinList[displayCount].setGPUName(gpuDevice.DeviceString);
        displayWinList[displayCount].setDisplayName(gpuDevice.DeviceName)
        displayWinList[displayCount].setRect(gpuDevice.rect);
        if ((gpuDevice.Flags & DISPLAY_DEVICE_PRIMARY_DEVICE))
            displayWinList[displayCount].primary = true;
        curDisplay++;
        displayCount++;
    } //end of enumerating displays
    gpuCount++;
} //end of enumerating gpu's
```

At run time, the GPU resource context must be made current to the window DC before any OpenGL calls are made. This way, rendering only happens to the subrectangles of the windows that overlaps parts of the desktops that are displayed by the GPUs in the affinity mask of the resource context.

```
case WM_PAINT:
        //Use the affinitv context for this window
   CAffinityGPU::~CAffinityGPU() {
       if (gpuRC)
           wglDeleteContext(gpuRC);
       if (gpuDC)
           wglDeleteDCNV(gpuDC);
   }
```

At application shutdown, the Affinity DC must be deleted

Note: When the affinity GL context is used, it is not recommended to create another OpenGL context from the Windows DC. Doing so may lead to unexpected behavior when querying OpenGL attributes using glGetString

Typical Multi-Display Configurations

We examine the commonly used multi-display setups that mix grayscale monitors and color panels and their underlying GPU configuration.

Case 1. 2 5 MP Grayscale Displays Driven by 1 GPU

The most commonly used configuration for diagnostic imaging, a high-end Quadro GPU drives 2 5 MP grayscale displays. One or two side displays are driven by a low-end Quadro NVS card (if there are no PCI 16× slots available) or another Quadro FX card.

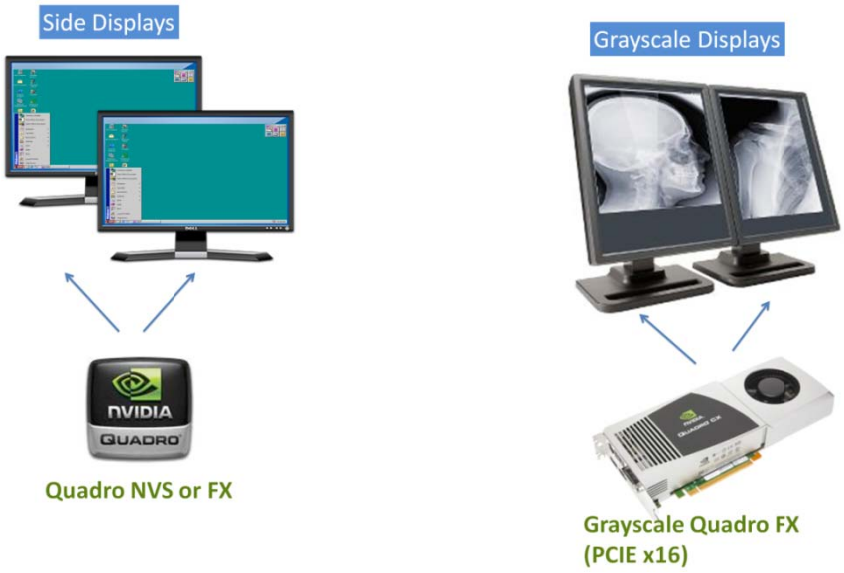

Figure 9. 10 MP Grayscale Display Configuration

Table 3. Characteristics for 10 MP Setup

Total Resolution	10 MP	5120 x 2048 (landscape) or 4096 x 2560 (portrait)
Side Display (Primary)	Quadro NVS 290	1 PCI 1x slot; good for system with only 2 PCI x16 slots
	Quadro FX 1800 Quadro FX 3800	1 PCI 16x; recommended for systems with 3 PCI 16x slots
	Quadro FX 4800	2 PCI 16x; high-end systems with 4 PCI 16x slots
Grayscale Display	Grayscale GPUs (Table 2)	2 PCI 16x slot

Case 2. 4 5 MP Grayscale Displays Driven by 2 GPUs

Two high-end Quadro GPUs drive 4 5 MP grayscale displays. This configuration assumes that the system has at least 4 PCI 16x slots. One or two side displays are driven by a low-end Quadro NVS card.

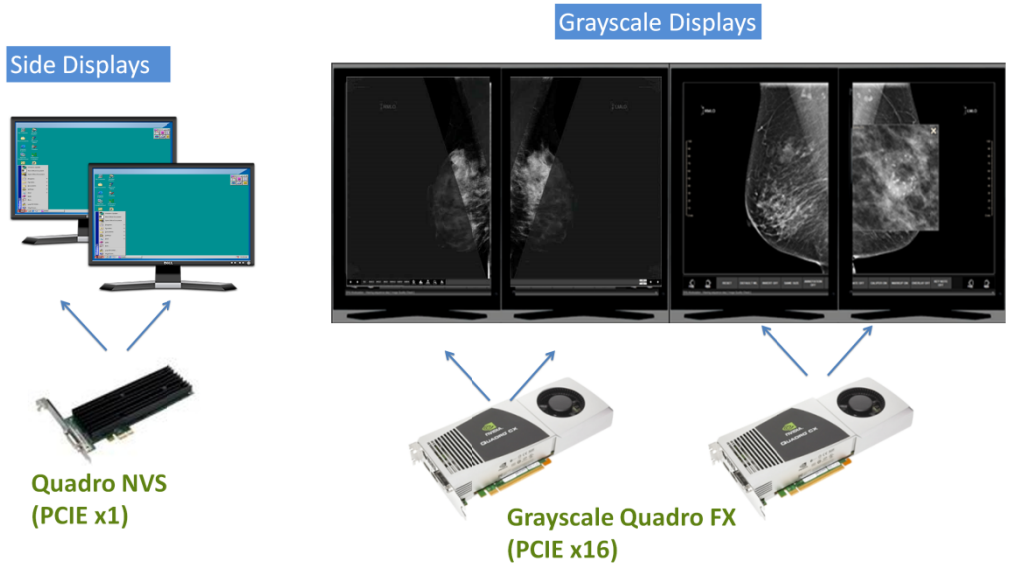

Figure 10. 3 GPUs Driving a 20 MP Grayscale Display

Table 4. Characteristics for the 20 MP Setup

Total Resolution	20 MP	10, 240 x 2048 (landscape) or 8192 x 2560 (portrait)
Side Display GPU (Primary Display)	Quadro NVS 290	1 PCIE x1 slot
Grayscale Display GPU 1	Grayscale GPUs (Table 2)	2 PCIE x16 slot
Grayscale Display GPU 2	Grayscale GPU' (Table 2)	2 PCIE x16 slot

References

[1] Digital Imaging and Communications in Medicine (DICOM)- Part 14 grayscale standard display function. http://medical.nema.org

[2] NDS Dome E5 Display
http://www.ndssi.com/products/dome/ex-grayscale/e5.html

[3] Eizo Radiforce GS520 Display
http://www.radiforce.com/en/products/mono-gs520-dm.html

[4] Integer Texture Extension
http://www.opengl.org/registry/specs/EXT/texture_integer.txt

[5] NVIDIA NVAPI – www.nvapi.com

[6] GPU Affinity Specification
http://developer.download.nvidia.com/opengl/specs/WGL_nv_gpu_affinity.txt

[7] Ian Williams, HD is now 8MP &HDR, Slides from NVISION 2008.
http://www.nvidia.com/content/nvision2008/tech_presentations/Professional_Visualization/NVISION08-8MP_HDR.pdf

Implementation Details

The following source code is divided into 3 separate projects. The intent is for these components to be mixed and matched according to the user application requirements.

- **GrayscaleDemo.sln**
  - **GrayscaleDemo.[cpp|h]** – An example demo application that does the various texture setups and allows the user to choose a grayscale image for display.
- **CheckGrayscale.sln**
  - **CDisplayWin.[cpp|h]** – Class CDisplayWin that encapsulates all attributes of an attached display such name, extents, driving GPU, etc.
  - **CheckGrayscale.cpp** – Main program that enumerates all attached GPUs and displays using Win GDI API and uses NVIDIA NVAPI to check the displays that are grayscale compatible.
- **MultiGPUAffinity.sln**
  - **CAffGPU.[cpp|h]** – Class CAffGPU that encapsulates an affinity GPU with its attributes such as the DC, OpenGL context, etc.
  - **CAffDisplayWin.[cpp|h]** – Class CAffDisplayWin that extends CDisplayWin to include affinity specific information.
  - **MultiGPUAffinity.cpp** – Main program that enumerate all GPUs creates the affinity data structures and does the event handling.