10 AND 12-BIT GRAYSCALE TECHNOLOGY

TB-04631-001_v05 | October 2013

Technical Brief

DOCUMENT CHANGE HISTORY

TB-04631-001_v05

Version	Date	Authors	Description of Change
01	April 17, 2009	SV, SM	Initial Release
02	February 9, 2010	SV, SM	Addition of Table 2
03	February 7, 2011	SV, SM	•Updated "System Specification" section •Updated "Supported Connectors" section •Updated Table 3 and Table 4 •Removed "Moving and Spanning Windows Across Displays" section •Removed "Targeting Specific GPUs for Rendering" section •Added "Directed GPU Rendering" section •Updated "Implementation Details" section
04	April 5, 2013	AS, SV, SM	•Updated to include current NVIDIA® Kepler™ products •Updated to include support for OpenGL 10-bit per component pixel formats
05	October 8, 2013	SV, SM	Updated Table 2 with Quadro K600 and Quadro K6000

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

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, expanded display capabilities of the latest NVIDIA Quadro® cards enable tiling together multiple high resolution displays for side-by-side comparisons from a single card.

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. This is done in 2 modes.

- "Pixel Packing" where 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. In recent driers, this pixel packing is done by driver transparent to the application when 10-bit per component pixel formats are used. This greatly simplifies programming in addition to making the application portable across multiple vendor hardware.
- Using 10-bit pixel formats over a VESA® DisplayPort™ output connection. No pixel packing is required as DisplayPort has sufficient bandwidth to transfer 5 MPixels with full 10-bit RGB color channels.

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 Workstation One 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]

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

Figure 2. Application Enhanced Using Multiply Displays[2]

SYSTEM SPECIFICATION

- 10 and 12-bit grayscale is currently supported on
  - Windows XP 32-bit and 64-bit
  - Windows 7 and Windows 8 (Aero should be enabled for best performance)
- Grayscale with 10-bit pixel formats is only supported for OpenGL based applications

Table 1 summarizes the guidelines for specific implementation methods based on the display connector and operating system.

Table 1. Grayscale Implementation Method Based On Display Connector and OS

Operating System	DVI	DisplayPort
Windows 7	OpenGL 10 bpc pixel format	OpenGL 10 bpc pixel format
Windows XP	Application pixel packaging	OpenGL 10 bpc pixel format

[2] Image courtesy of Threepalms, Inc.

SUPPORTED GRAPHICS BOARDS

The boards shown in Table 2 support 10-bit grayscale and are NVIDIA® CUDA® enabled.

Table 2. Quadro Graphics Boards with 10 and 12-Bit Grayscale Support

Mid range – Quadro K2000D, Quadro K2000, Quadro K600, Quadro 2000D, Quadro 600, Quadro 2000 	Recommended for 2D image display and manipulation use cases over multiple displays. No auxiliary power is required.
High end – Quadro K4000, Quadro 4000 	Recommended if the primary usage is to display and compute with 2D grayscale and 3D data.
Ultra high end – Quadro K6000, Quadro K5000, Quadro 6000, Quadro 5000 	Recommended for applications that also require rendering and processing large 3D and 4D geometries and volumes.
Quadro Plex 7000, **Quadro Plex 2200 D2** 	Dedicated desk side visual computing system composed of 2 highest-end Quadro graphics boards with up to 12 GB of total graphics memory. Recommended for advanced visualization and large scale projection and display use cases.

SUPPORTED MONITORS

Table 3. Grayscale Capable Display Panels with Supported Resolution and Pixel Depth

Manufacturer	Panel	Supported Resolutions	Grayscale Depth	Packed Pixel DVI	10-Bit DisplayPort
NDS Surgical Imaging	Dome E2	•1600 × 1200 at 60 Hz •1200 × 1600 at 60 Hz	10 and 12-bit	Yes	No
	Dome E3	•2048 × 1536 at 60 Hz •1536 × 2048 at 60 Hz	10 and 12-bit	Yes	No
	Dome S3, Dome S3c	•2048 × 1536 at 60 Hz •1536 × 2048 at 60 Hz	10 and 12-bit	Yes	No
	Dome E5	•2560 × 2048 at 50 Hz •2048 × 2560 at 50 Hz	10 and 12-bit	Yes	No
	Dome Z10	•4096 × 2560 at 50 Hz •2560 × 4096 at 50 Hz	10 and 12-bit	Yes	Yes
Eizo	GS 520	•2560 × 2048 at 50 Hz •2048 × 2560 at 50 Hz	10-bit	Yes	Yes
	GS 521	•2560 × 2048 at 50 Hz •2048 × 2560 at 50 Hz	10-bit	Yes	No
	GX 530	•2560 × 2048 at 50 Hz •2048 × 2560 at 50 Hz	10-bit	No	Yes
	GX 1030	•2560 × 4096 at 50 Hz •4096 × 2560 at 50 Hz	10-bit	Yes	No
NEC	MD215MG	•2560 × 2048 at 57 Hz •2048 × 2560 at 57 Hz	10-bit	Yes	No
	MD205MG, MD205MG-1	•2560 × 2048 at 57 Hz •2048 × 2560 at 57 Hz	10-bit	Yes	No
	MD211G3	•2560 × 2048 at 57 Hz •2048 × 2560 at 57 Hz	10-bit	Yes	No
	MD213MG	•2048 × 1536 at 60 Hz •1536 × 2048 at 60 Hz	10-bit	Yes	No
	MD21GS-3MP	•2048 × 1536 at 60 Hz •1536 × 2048 at 60 Hz	10-bit	Yes	No
	MD21GS-2MP	•1600 × 1200 at 60 Hz •1200 × 1600 at 60 Hz	10-bit	Yes	No
Wide	IF2105PM	•2560 × 2048 at 50 Hz •2048 × 2560 at 50 Hz	10-bit	Yes	No
Beacon	G32SP	•2048 × 1536 at 60 Hz •1536 × 2048 at 60 Hz	10-bit	Yes	No
	G51SP	•2560 × 2048 at 57 Hz •2048 × 2560 at 57 Hz	10-bit	Yes	No

TYPICAL MULTI-DISPLAY CONFIGURATION

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

Case 1. Two 5 MP Grayscale Displays Driven by One Quadro Card

The most commonly used configuration for diagnostic imaging, a single Quadro K4000, Quadro K2000, or Quadro K2000D drives two 5 MP grayscale displays plus a side "worklist" display.

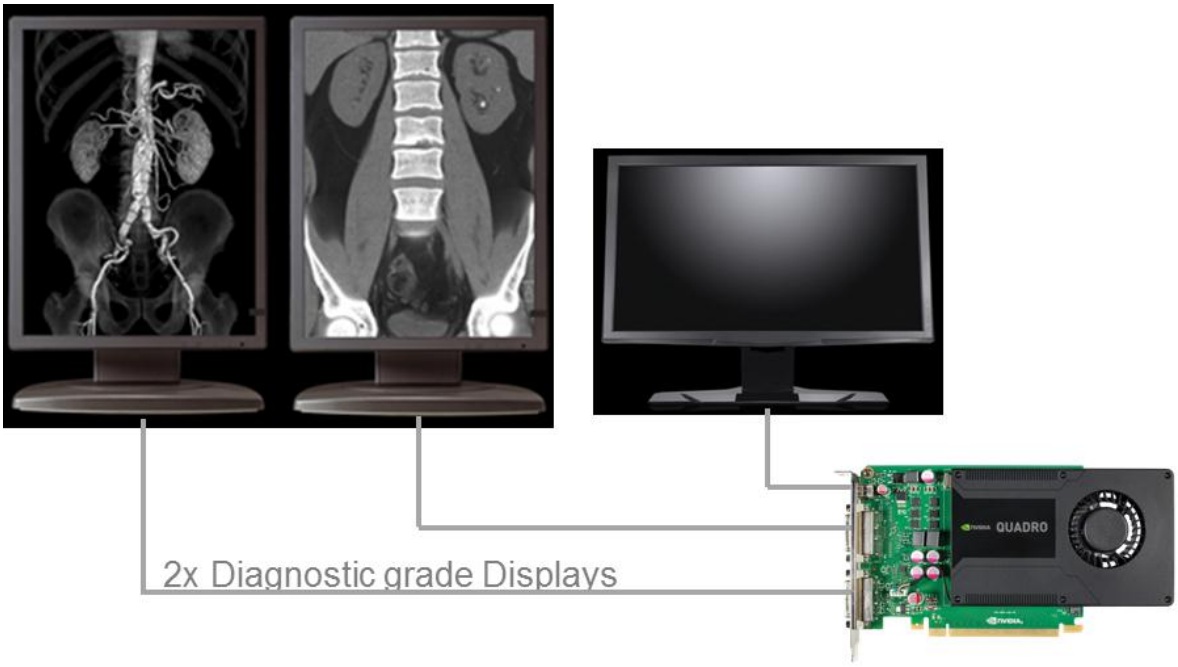

Figure 3. 10 MP Grayscale Configuration

Table 4 shows the two single card configuration options for a 10 MP setup. The customer can choose between the options for either the Quadro K2000 and Quadro K4000 or the Quadro K2000D.

Table 4. Single Card Option for Dual Display Configurations

	Quadro K2000 or Quadro K4000 (If Diagnostic Display Supports DisplayPort)			**Quadro K2000D (If Diagnostic Display Only Supports DVI)**		
Displays	DVI connector	DisplayPort connector	DisplayPort connector	DVI or DisplayPort (through mini-DisplayPort adapter)	Dual-link DVI connector	Dual-link DVI connector
Side Display	●			●		
2-5 MP Diagnostic Display		●			●	
2-5 MP Diagnostic Display			●			●

Case 2. Four 5 MP Grayscale Displays Driven by Two Quadro Cards

In this configuration, a single Quadro K4000, Quadro K2000, or Quadro K2000D can drive two 5 MP grayscale displays plus a side "worklist" display. A second Quadro K4000, Quadro K2000, or Quadro K2000D card can drive the remaining two 5 MP displays.

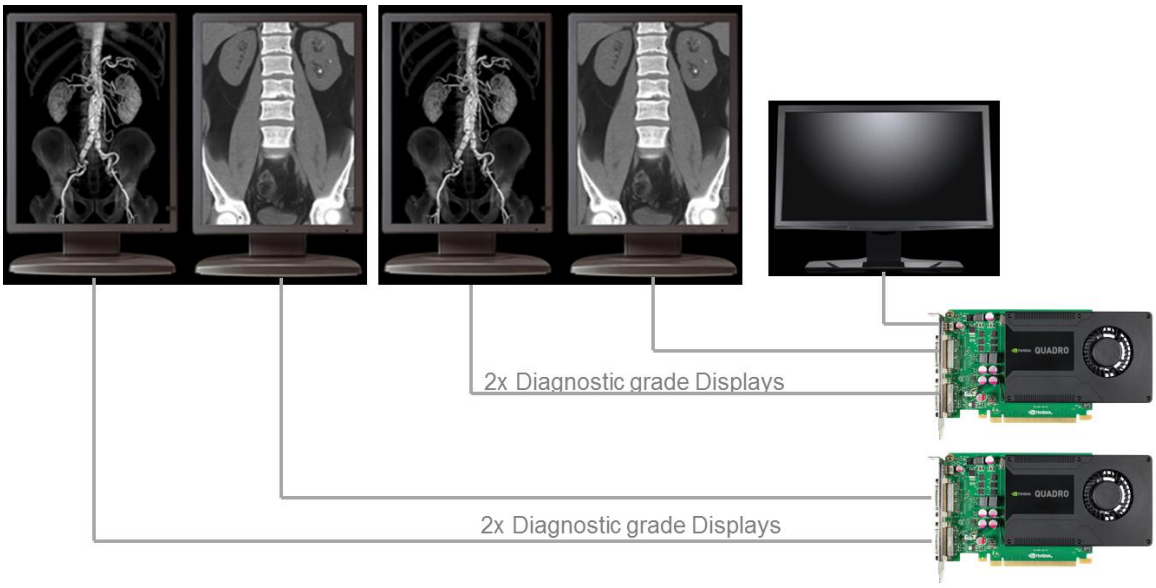

Figure 4. Three GPUs Driving a 20 MP Grayscale Display

Table 5 shows the two graphics cards required configuration for a 20 MP setup. The configuration requires two Quadro K2000, Quadro K2000D, or Quadro K4000 cards.

Table 5. 20 MP Configuration

	Quadro K2000, Quadro K2000D, or Quadro K4000			Quadro K2000, Quadro K2000D, or Quadro K4000		
Displays	DVI connector	DisplayPort or DVI connector	DisplayPort or DVI connector	DVI connector	DisplayPort or DVI connector	DisplayPort or DVI connector
Side Display	●					
2-5 MP Diagnostic Display		●				
2-5 MP Diagnostic Display			●			
2-5 MP Diagnostic Display					●	
2-5 MP Diagnostic Display						●

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 and Adapters

- As displays are increasingly adopting DisplayPort, the native DisplayPort connectors on Quadro cards can be connected directly via DisplayPort cables.
- For standard size DisplayPort connectors to single-link DVI conversion, passive adapters such as Hosiden (P/N TYX1602-010307) and Simula (P/N DJ8028B-1000-10E) are tested and recommended.
- On the Quadro K2000D for latched mini-DisplayPort to standard size DisplayPort conversion, use Bizlink P/N KS30011-B07 or Simula P/N CB802E-4000-10H.
- On the Quadro K2000D for latched mini-DisplayPort to single-link DVI conversion, use Bizlink PN KS30037-C07.

- To support dual-link resolutions from a DisplayPort connector, an active adapter is required. As shown in Figure 8 this dongle includes a built in USB cable connecting to the USB port providing power to the adapter. NVIDIA recommends the Bizlink DisplayPort-to-DVI-D dual-link cable adapter (P/N KS10014-207).
- The Simula and Bizlink adapters can be purchased from NVIDIA's online store at http://store.nvidia.com/ (under the cables category).

Figure 5. DisplayPort to Single-Link DVI Adapter (Passive)

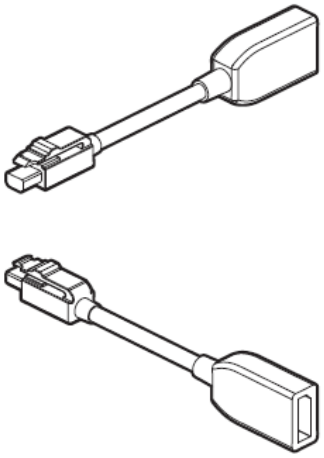

Figure 6. Latched Mini-DisplayPort

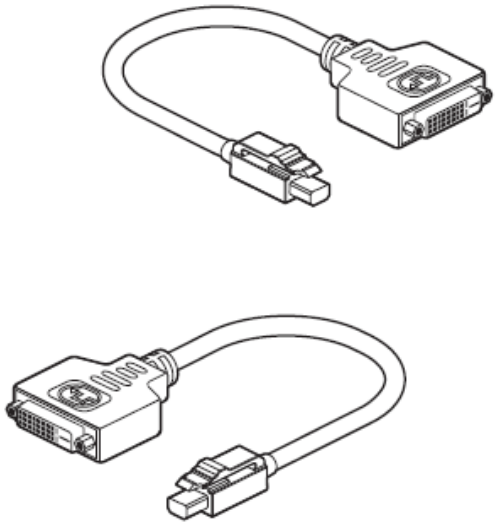

Figure 7. Latched Mini-DisplayPort to Single-Link DVI

Figure 8. DisplayPort to Dual-Link DVI Adapter (Active)

GRAYSCALE MONITOR SETTINGS

When a grayscale compatible monitor is connected to a suitable NVIDIA Quadro solution, 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. On Windows XP, the only setting required is to enable the grayscale monitor to display at its optimal resolution as shown in the following steps for a 5 MP panel with resolution 2560 × 2048.

Note: These steps are not required on Windows 7 and Windows 8.

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.

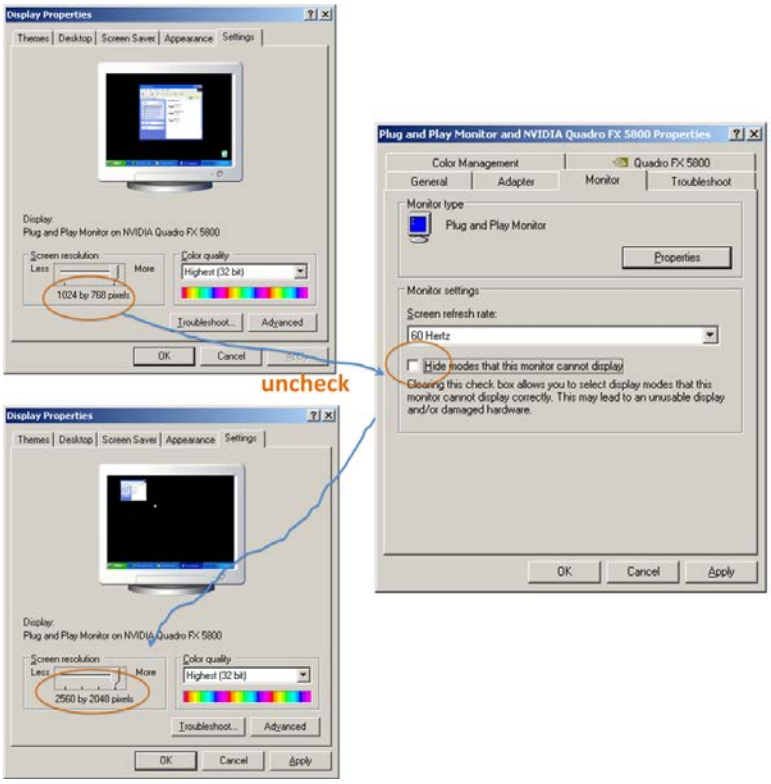

Figure 9. Enable 5 MP Grayscale Monitor to Display Higher Resolution

GRAYSCALE APPLICATION DEVELOPMENT

DVI DRIVER LAYER

On grayscale enabled Quadro solution, 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).

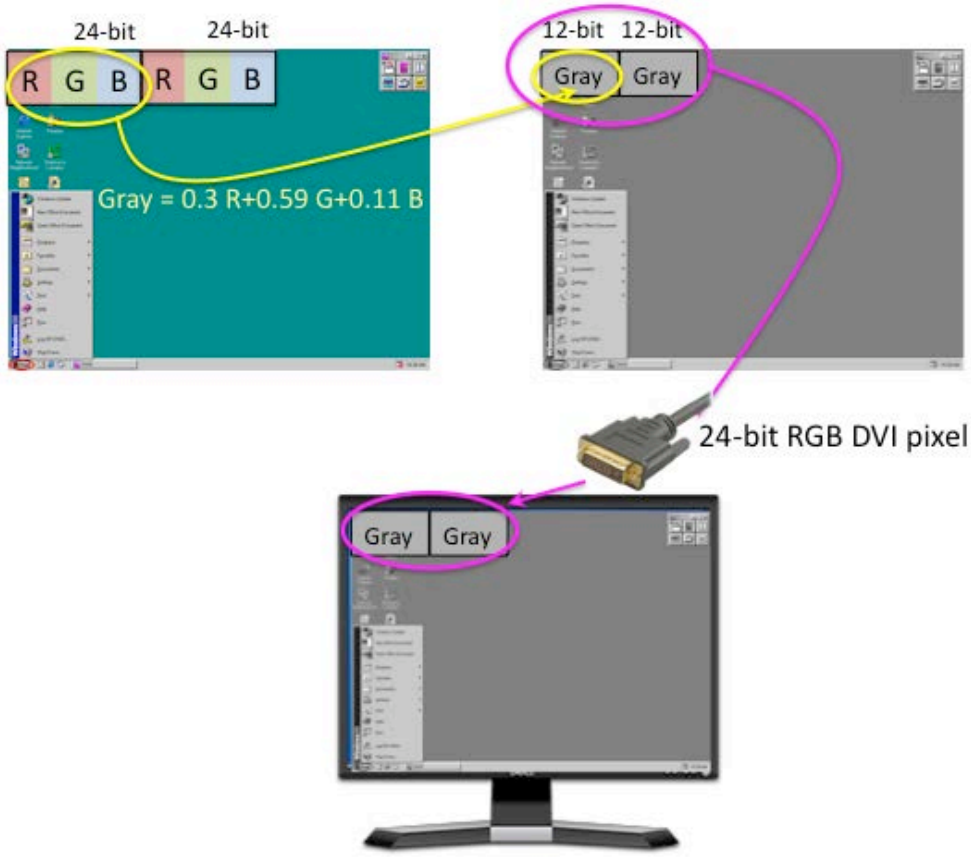

Figure 10. Driver Converts and Packs Desktop from 24-Bit Color to 12-Bit Gray

OLDER METHOD FOR DVI APPLICATION LEVEL PIXEL PACKING

This is a legacy method mostly used for applications on Windows XP using DVI displays, but will work with Windows 7 and Windows 8 if the same code base is to be shared.

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 source image.

```
#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;
}
```

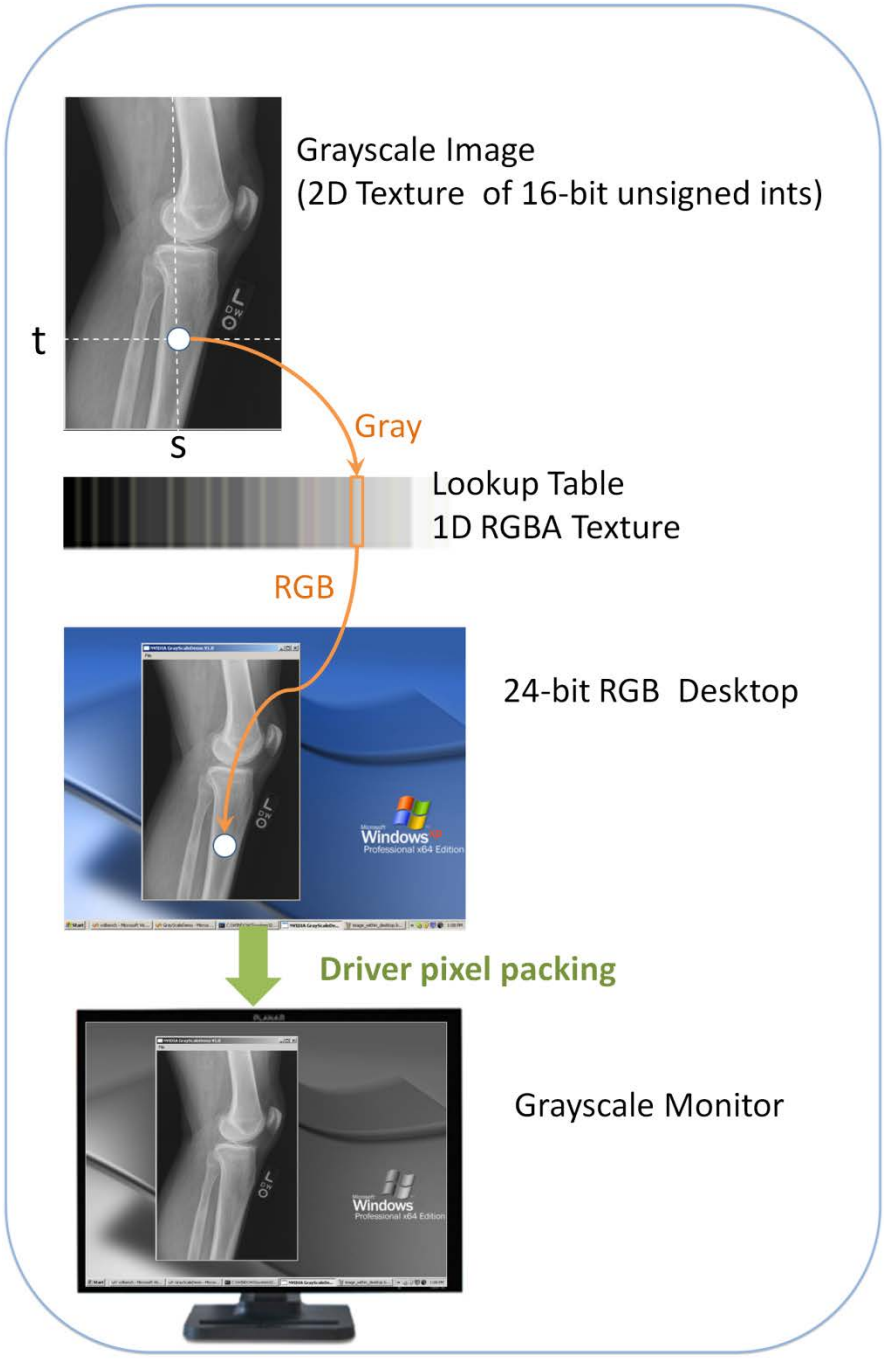

Figure 11. Application Level Texture Setup for 10 and 12-Bit Grayscale Display

OPENGL 10-BIT PIXEL FORMAT FOR DVI AND DISPLAYPORT ON WINDOWS 7

A significant drawback of the pixel packing mechanism explained in the previous section is that it requires more programming work by the application developer and is vendor dependent. With the latest drivers, the implementation is significantly made easier with the use of OpenGL native 10-bit pixel format. This is the recommended path for the following:

- Windows 7 based applications – supported for monitors requiring DVI pixel packing and ones that support grayscale over DisplayPort.
- Windows XP – only enabled on panels with DisplayPort support.

In this case, the application doesn't need to manage the lookup into the grayscale table in the shader as this is done seamlessly under the hood by the driver (for DVI). The application is responsible for creating the input texture using the integer texture extension as explained in the previous section and the driver does the necessary conversions to make the data visible on screen.

Creating a 10 bpc OpenGL Window

On Windows, the displayable 30-bit pixel formats are exported via `WGL_ARB_pixelformat` extension [5]. To access these pixel formats, the WGL pixel format functions are used instead of their GDI equivalents. A dummy OpenGL window must be created in order to get a handle to these WGL functions.

```
// Create a dummy window to query the WGL_ARB_pixelformats
HWND dummyWin  = CreateDummyGLWindow(szClassName, "Dummy", FALSE);
if (dummyWin == NULL) {
        // TODO - Error Handling here
}
HDC dummyDC = GetDC(dummyWin);
// TODO – Set Pixel Format
HGLRC dummyRC = (HGLRC) wglCreateContext (dummyDC);
// Set the OpenGL context current
wglMakeCurrent(dummyDC, dummyRC);

// Find the 30-bit color ARB pixelformat
wglGetExtensionsString = (PFNWGLGETEXTENSIONSSTRINGARBPROC)
        wglGetProcAddress("wglGetExtensionsStringARB");
if (wglGetExtensionsString == NULL) {
        // TODO - Error Handling and Cleanup here
}
const char *szWglExtensions = wglGetExtensionsString(dummyDC);
if (strstr(szWglExtensions, " WGL_ARB_pixel_format ") == NULL) {
        // TODO - Error Handling and Cleanup here
}
wglGetPixelFormatAttribiv = (PFNWGLGETPIXELFORMATATTRIBIVARBPROC)
        wglGetProcAddress("wglGetPixelFormatAttribivARB");
wglGetPixelFormatAttribfv = (PFNWGLGETPIXELFORMATATTRIBFVARBPROC)
        wglGetProcAddress("wglGetPixelFormatAttribfvARB");
```

```
wglChoosePixelFormat = (PFNWGLCHOOSEPIXELFORMATARBPROC)
       wglGetProcAddress("wglChoosePixelFormatARB");
if ((wglGetPixelFormatAttribfv == NULL) || (wglGetPixelFormatAttribiv ==   NULL) ||
(wglChoosePixelFormat == NULL))
       // TODO - Error Handling and Cleanup here
```

The 10 bits per component is specified in the desired attribute list before calling the `wglChoosePixelFormat` which returns the matching pixel formats. The following code listing also checks the RGB color depth after the call to make sure that a 30-bit color pixel format is in place.

```
int attribsDesired[] = {
    WGL_DRAW_TO_WINDOW_ARB, 1,
    WGL_ACCELERATION_ARB, WGL_FULL_ACCELERATION_ARB,
    WGL_RED_BITS_ARB, 10,
    WGL_GREEN_BITS_ARB, 10,
    WGL_BLUE_BITS_ARB, 10,
    WGL_ALPHA_BITS_ARB, 2,
    WGL_DOUBLE_BUFFER_ARB, 1,
    0,0
};

UINT nMatchingFormats;
int index = 0;
if (!wglChoosePixelFormat(dummyDC, attribsDesired, NULL, 1, &index, &nMatchingFormats)) {
    printf("ERROR: wglChoosePixelFormat failed!\n");
    goto cleanup;
}

if (nMatchingFormats == 0) {
    printf("ERROR: No 10bpc WGL_ARB_pixel_formats found!\n");
    goto cleanup;
}

// Double-check that the format is really 10bpc
int redBits;
int alphaBits;
int uWglPfmtAttributeName = WGL_RED_BITS_ARB;
wglGetPixelFormatAttribiv(dummyDC, index, 0, 1, &uWglPfmtAttributeName, &redBits);
uWglPfmtAttributeName = WGL_ALPHA_BITS_ARB;
wglGetPixelFormatAttribiv(dummyDC, index, 0, 1, &uWglPfmtAttributeName, &alphaBits);

printf("pixelformat chosen, index %d  red bits: %d  alpha bits: %d", index, redBits,
alphaBits);
```

MULTI-DISPLAY CONFIGURATIONS WITH KEPLER

Diagnostic imaging commonly requires multiple displays for side by side modality comparisons. Multi-display configurations are becoming easier to manage with new Quadro boards like the Quadro K5000, Quadro K4000, and Quadro K2000 - which can drive up to 4 simultaneous displays. Depending on the available PCI slots within a system, multiple cards can be used to drive more than 4 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.

MULTIPLE DISPLAY SETUP

For Windows XP, the multi-display capability has to be enabled explicitly as follows. 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 Figure 8.

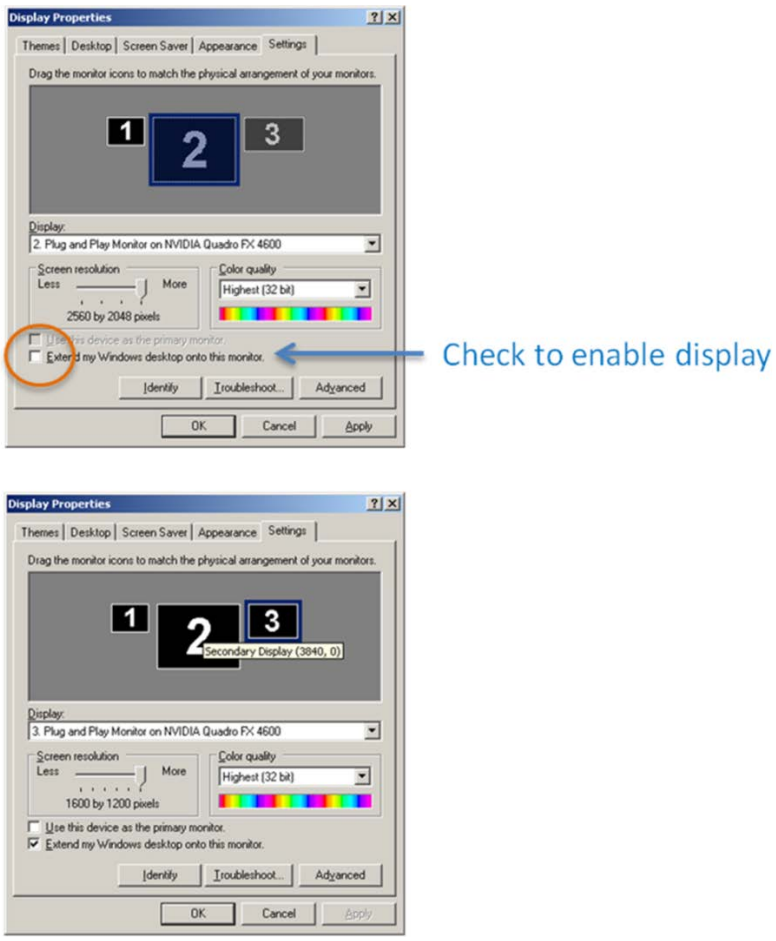

Figure 12. Display Properties Before and After Displays are Enabled

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. Accessing 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 12), prints out the following:

```
DeviceName    = \\.\DISPLAY1
DeviceString  = NVIDIA Quadro K2000
PRIMARY DISPLAY
Position/Size = (0, 0), 1280x1024

DeviceName    = \\.\DISPLAY2
DeviceString  = NVIDIA Quadro K2000
Position/Size = (1280, 0), 2560x2048

DeviceName    = \\.\DISPLAY3
DeviceString  = NVIDIA Quadro K2000
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 to 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 LCDs 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;
}
```

APPENDIX

MULTI-GPU COMPATIBILITY FOR PRE-KEPLER CARDS

Grayscale capable Quadro boards can be mixed with other Quadro boards that can drive one or many side displays as shown in Table 6. 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 of the same generation.

Table 6. Multi-GPU Compatibility

Side Display Quadro GPU \ Grayscale GPU	Quadro 2000D Quadro 2000 Quadro FX 1800	Quadro 5000 Quadro 4000 Quadro FX 4800 Quadro FX 3800 Quadro FX 4600 Quadro FX 3700	Quadro 6000 Quadro FX 5800 Quadro FX 5600	Quadro Plex 7000 Quadro Plex D2
Quadro 600 Quadro NVS 450 Quadro NVS 420 Quadro NVS 300 Quadro NVS 310	✓	✓	✓	✓
Quadro 2000D Quadro 2000 Quadro FX 1800	✓	✓	✓	✓
Quadro 5000 Quadro 4000 Quadro FX 4800 Quadro FX 3800 Quadro FX 4600 Quadro FX 3700		✓	X	✓
Quadro 6000 Quadro FX 5800 Quadro FX 5600			X	✓

Notes: These are theoretical compatibilities assuming the availability of 2 auxiliary power inputs. 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 6. The Quadro FX 5800 and Quadro FX 6000 require the full 2 auxiliary power inputs and therefore only used with lower-end Quadro cards that do not have any auxiliary power requirements.
The mixing of older pre-G80 cards is not supported in grayscale configurations.

DIRECTED GPU RENDERING

In a multi-GPU setup, the default behavior is for OpenGL commands to be sent to all GPUs. While this works for many applications, performance is gated by the capabilities of the lowest-end card. In a typical grayscale setup, the side display with GUI elements is normally connected to a lower-end Quadro while the grayscale panels are connected to a higher-end Quadro card. It is desirable to limit grayscale rendering to the GPUs that are driving the grayscale panels and not involve the side GPU at all in the render process. Previous approaches required programmatically selecting the GPU using OpenGL extensions which can quickly become an additional programming burden for a radiology developer. The newer Quadro drivers have a feature called "Directed Rendering" that allows the user to target the GPU for rendering and decouple it from

the display GPU. This is done via the NVIDIA Control Panel as shown in Figure 13 or programmatically using NVAPI. When the render GPU and the display GPU are different, the driver transparently uses a fast transfer path on Quadro cards to split the rendered images onscreen. By default, the driver will pick the biggest GPU for render.

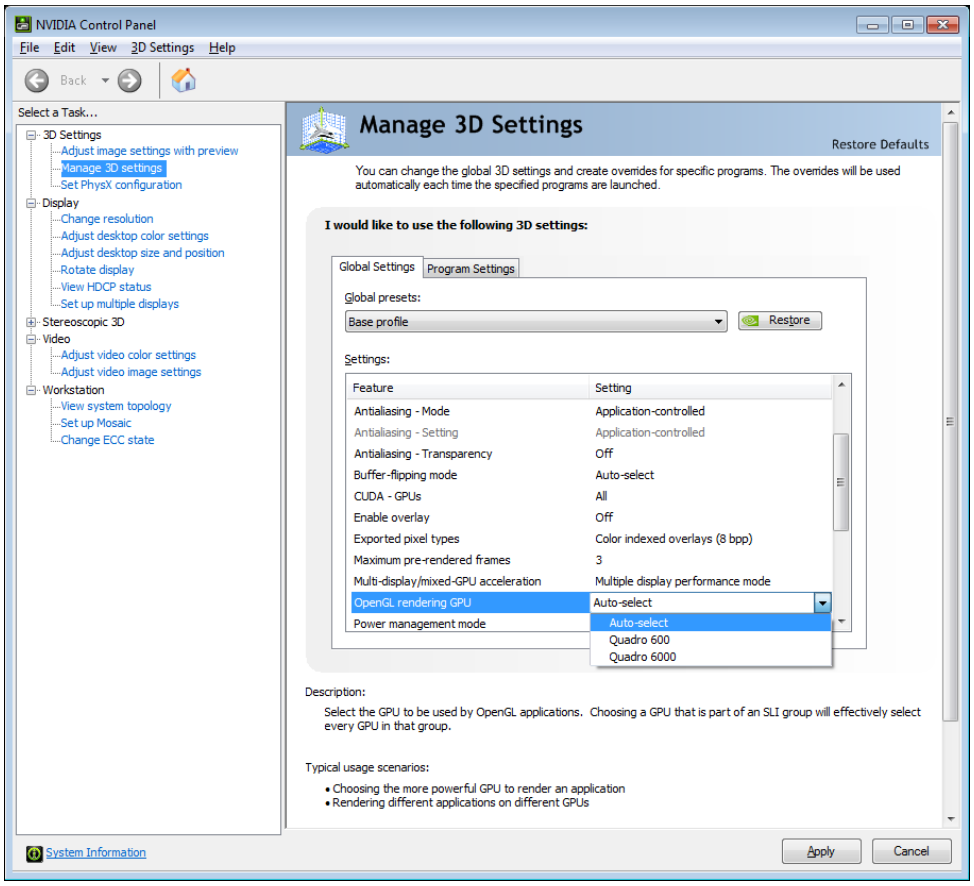

Figure 13. Setting Render GPU from NVIDIA Control Panel

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] WGL_ARB_pixelformat extension
http://www.opengl.org/registry/specs/ARB/wgl_pixel_format.txt

[6] NVIDIA NVAPI – www.nvapi.com

[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 accompanying 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.