Technical Brief

30-Bit Color Technology for NVIDIA® Quadro®

May 29, 2009| TB-04701-001_v01

Document Change History

Version	Date	Responsible	Description of Change
01	May 28, 2009	SV, AP, SM	Initial Release

Table of Contents

List of Figures

List of Tables

30-Bit Color

Introduction

Color precision is critical for professionals in the graphic design, film and scientific visualization communities. Within the film industry color is a critical story telling component; managing color from beginning to end of the pipeline is requirement to correctly expressing the artists' creative vision. Likewise in graphic design, color is used to define a brand or create an emotion; it is crucial that designers be able to correctly visualize these colors and how they interact. Medical and seismic imaging systems often produce raw 12 to 16-bit data that are mapped to color by means of lookup tables. In a system like this the image and graphics processing pipeline may use high-precision floating point data, but the end results will be need to be rendered, or down sampled, for display on 24-bit monitors. This down sampling can lead to color difference, where what is seen on the display does not correctly represent the data underneath. It is critical for all these businesses that the colors and all the subtle transitions between them be correctly mapped and preserved throughout the pipeline.

Recent advances in display and graphics processing unit (GPU) technology are bringing the capability to better represent this critical color information. GPU's like the NVIDIA Quadro® FX line are capable of processing high precision graphics at faster speeds, including 30-bit color. Displays like the HP lp2480zx DreamColor display with its ability to show over a billion shades of colors are setting new standards for profession grade LCD panels. Industry standard display connection like the DisplayPort are making it possible to connect GPU's and displays and reliable move all this critical information. Putting each of these components together provides an easy to use solution for professionals whose livelihood demands the utmost in color quality.

Bit Depth Vs Color Gamut

Concepts of bit depth and color gamut while related should be looked as two separate items. Bit depth, the 30-bit part of "30-bit color" is a reference to how many bits of data are allocated for each color value in a pixel. Color Gamut refers to how much color a display or printer can give. It is possible to have a format with a high bit depth even though it is a low color gamut.

Most professional grade displays available define the color of a pixel by an 8-bit value for each of the red, green and blue components. For each of the Red, Green and Blue values a programmer or artist can select from a range of 0 – 255 (0 being black; 255 full color saturation) giving each pixel a possibility of 16.7 million colors. Now, in a 30-bit display each pixel component is provided 10 bits of color, or 1024 shades resulting in a bit over 1.07 billion shades of color per pixel. As color gamut and bit depth are two separate things, while we have a billion possible shades of color, we do not know what those colors are.

Defining what those colors are is where the color gamut comes in. The color gamut for a display is defined by 5 things:

- **The color of red, green and blue color filters**: The formulation of these filters defines how saturated each color can get.
- **The color and spectra of the backlight**: This light interacts with the filters to create the color that is perceived.
- **The "blackness" of the display of black**: Often overlooked but the performance of a display in the shadow details is often what separates a regular display from a professional display

All of these components can be measured and charted. A common chart for this is a chromaticity diagram, shown in Figure 1. The points on the outside of the triangles are the colors seen at 100% saturation for each primary. The circle in the middle, with the D65 mark is the color of the back light or the color of the display when it is showing 100% white. This diagram has no notion of the black level and a 3D representation is needed for that.

The figure also shows two triangles labeled sRGB [1]and Adobe RGB [2] - both standard color spaces used throughout the photo and imaging world. The sRGB triangle is smaller than Adobe RGB and as such it has a smaller color gamut. The consequence of the differences in color gamut size is that when an application needs colors not inside the color gamut they cannot be reproduced on the display. As an example, an Adobe RGB file with 100% green cannot be shown on a sRGB display. Mapping from one color gamut to another is where color management comes in which is beyond the scope of this paper.

As the color gamut gets bigger, the need for higher bit depth addressability becomes more and more important. Recall that in a 24-bit system the gamut is always divided into 16.7M voxels, as the gamut grows so too do the voxels.

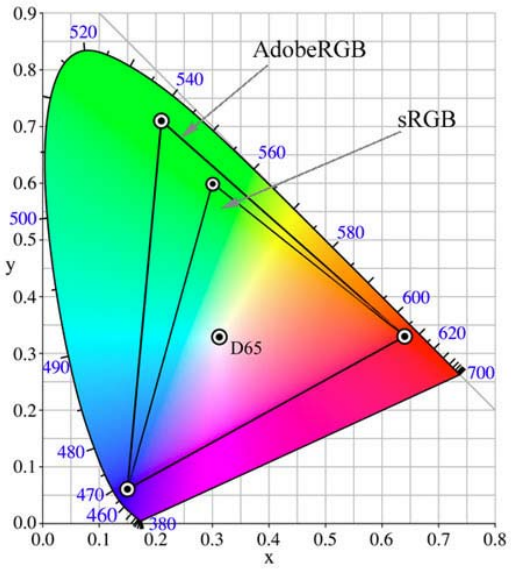

Figure 1. Comparing sRGB with Adobe RGB Color Space

Why 30-Bit Color?

30-bit color means that each pixel can be built from a bit over a billion shades of color. While there is some value in being able to exactly specify a color, the real value of these extra shades is in the transition from one color to another. The easiest way to see this is in gradient fills, although they will often appear in photographs as well. Figure 2 shows a radial gradient from sRGB (0,90,0) to pure black. Unless viewed on a very high quality display, chances are it has circular bands that run through it. These bands are a result of not having enough shades of color to transition from the mark green to black.

Figure 3 shows a gradient from 0 to 25% sRGB green with increasing bit depth thereby increases the number of shades of color available; the smaller the steps, the smoother the image. Continuing to 10-bit would make the divisions even smaller providing a smooth gradient without resorting to techniques like dithering, or adding some noise into the image to smooth the bands out. While dithering produces a visually smooth image, the pixels no longer correlate to the source data. This matters in mission critical applications like diagnostic imaging where a tumor may only be one or two pixels big.

Figure 2. Radial Banding Observed on a 24-Bit Color Display

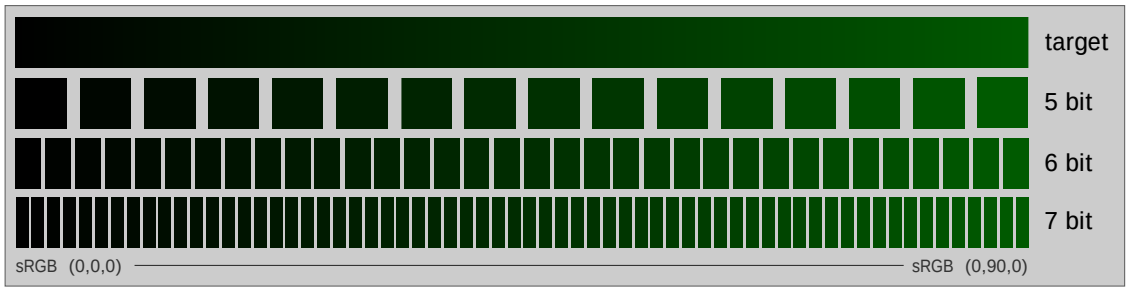

Figure 3. Color Ramp with Increasing Bit Depth

System Specific Information

- 30-bit color is currently supported on the following:
  - Windows XP
  - Windows Vista
  - Windows 7
  - Linux
- OpenGL- windowed and full screen modes
- Direct3D 10 – exposed in full screen only

Supported Graphics Cards

30-bit color is supported on Quadro FX boards (Table 1) that are DisplayPort capable.

Note: 30-bit color is not supported on Quadro NVS boards (including Quadro NVS 450 even though DisplayPort is supported).

Table 1. 30-Bit Capable Cards

GT200GL	G94GL	G96GL
Quadro FX 5800 Quadro FX 4800 Quadro CX Quadro FX 3800	Quadro FX 1800	Quadro FX 5800

Supported Monitors

At least one EDID 1.04 30-bit color monitor must be attached for the driver to automatically switch to 30-bit mode. The tested monitors are HP Dreamcolor LP 2480zx.

Supported Connectors

- **DisplayPort**
  DisplayPort allows a mode with 10 bits per component or 30-bit color. This is the approach explained in this paper.
- **Dual-link DVI**
  DVI will not natively support 30-bit deep color. It is possible to enable 30-bit color over DVI using device specific pixel packing method. In this approach, off screen buffers are used for rendering at a deep color value and a fragment shader is used to implement the monitor-specific pixel packing. Since this approach is display specific it will not be covered by this paper.

Setup Over DisplayPort

30-bit color over display port is significantly easier and all that is required is a 30-bit compatible Quadro FX board. On current operating systems, 8 bits per component or a total of 24-bit color is the default for the desktop and GDI rendering. However, applications can use 30-bit color through OpenGL by requesting the appropriate pixel format (explained in the next section). This approach allows for windowed 30-bit OpenGL buffers that are then composited with the GDI components before scan out. The entire pipeline is done in 30-bit as shown in Figure 4.

When connected to a 30-bit capable monitor, the driver by default switches to 30-bit scan out. On Windows Vista, this mode automatically disables Windows Aero regardless of whether 30-bit application is running. If Aero must be enabled (therefore reverting to 24-bit color rendering), the NVIDIA® Display Control Panel has a "Deep Color for 3D Applications" setting that can be set to “disable” (Figure 5).

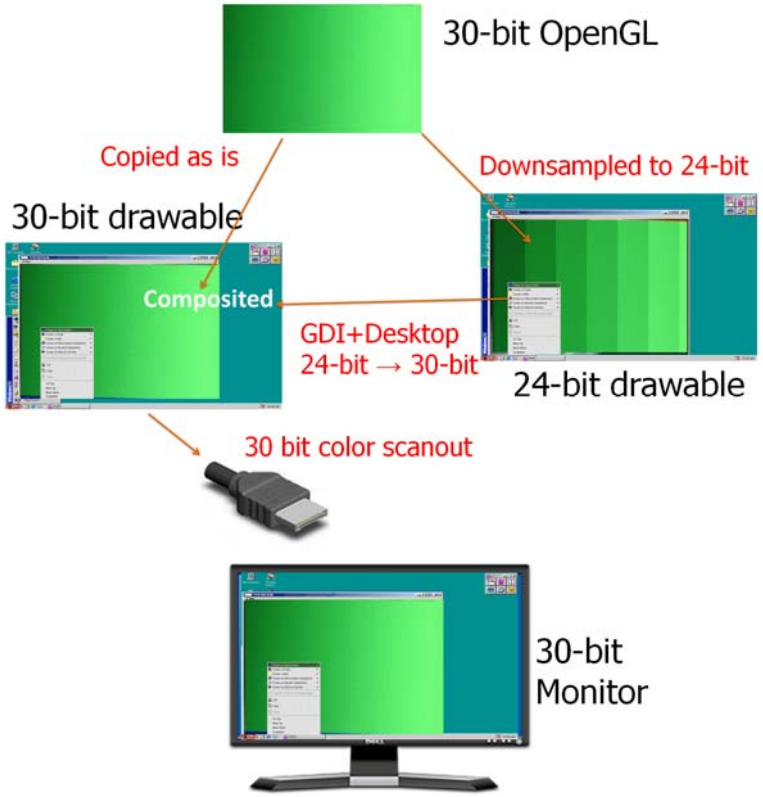

Figure 4. 30-Bit Window within 24-Bit Desktop and the Scan Out Process

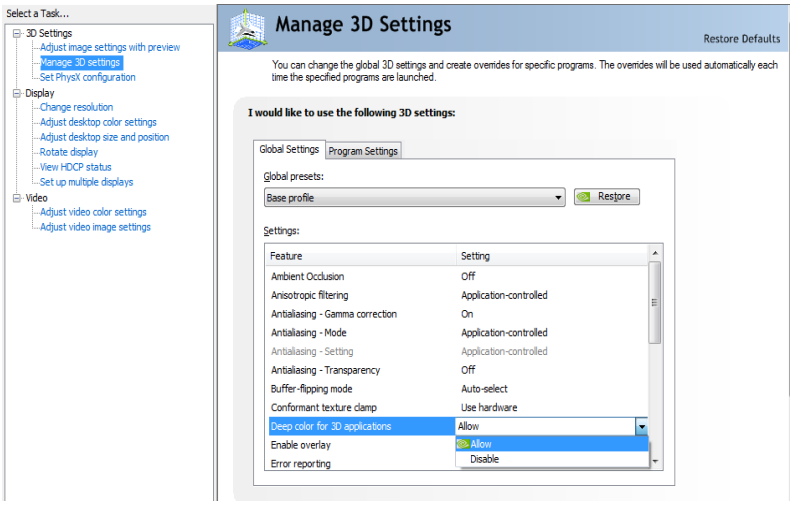

Figure 5. Control Panel Setting on Windows Vista to Disable 30-Bit Color

30-Bit Pixel Format - Win32

On Windows, the displayable 30-bit pixel formats are exported via WGL_ARB_pixelformat extension[3]. To access these pixel formats, the WGL pixelformat 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 code listing below 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);
```

Note: With 30-bit enabled, both active stereo and rotation are NOT possible.

30-Bit Visual on Linux

On Linux, the GLX interface is used to request a 30-bit Config similar to the pixel format on Windows. The desired attributes (10-bit RGB) are set and a matching Config is returned by the `glXChooseFBConfig` as followed. The associated X Visual can then be queried from this Config and subsequently used to create the 30-bit window. It is assumed that the handles to the glx functions are already obtained using `glXGetProcAddress.`

```
Display *dpy = XOpenDisplay(0);
//The desired 30-bit color visual
int attributeList[] = {
    GLX_DRAWABLE_TYPE, GLX_WINDOW_BIT,
    GLX_RENDER_TYPE,   GLX_RGBA_BIT,
    GLX_DOUBLEBUFFER,  True,
    GLX_RED_SIZE,      10,   /* 10bits for R */
    GLX_GREEN_SIZE,    10,   /* 10bits for G */
    GLX_BLUE_SIZE,     10,   /* 10bits for B */
    None
};

int fbcount;
GLXFBConfig *fbc = glXChooseFBConfig(dpy, DefaultScreen(dpy),
                      attributeList, &fbcount);
if (!fbc) {
        //Error Handler
}

XVisualInfo *vi = glXGetVisualFromFBConfig(dpy, fbc[0]);

//Make sure we have 10bit Red, Green and Blue
int redSize, greenSize, blueSize,
glXGetFBConfigAttrib(dpy, fbc[0], GLX_RED_SIZE, &redSize);
glXGetFBConfigAttrib(dpy, fbc[0], GLX_GREEN_SIZE, &greenSize);
glXGetFBConfigAttrib(dpy, fbc[0], GLX_BLUE_SIZE, &blueSize);
printf("RGB size %d:%d:%d\n",redSize, greenSize, blueSize);

//Create colormap and window here
```

Cross-Platform - GLUT

The GLUT library[4] allows a simple cross-platform way to create a 30-bit color window. The 10 bits per components must be specified at startup before the window creation.

```
glutInitDisplayString("red=10 green=10 blue=10 alpha=2");
glutInitDisplayMode( GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH );
glutCreateWindow("30-bit Demo");
//check to get the no of color bits we have
int redSize, greenSize, blueSize;
glGetIntegerv(GL_RED_BITS,& redSize);
glGetIntegerv(GL_GREEN_BITS,& greenSize);
glGetIntegerv(GL_BLUE_BITS,& blueSize);
printf("RGB size %d:%d:%d\n",redSize, greenSize, blueSize);
```

Off-Screen Rendering

Common HDR operations such as tone-mapping and shadowing are often done in high-precision floating point or 32-bit integer not supported on the window frame buffer. The OpenGL Framebuffer Objects (FBO) extension[5] allows rendering to be diverted from the window's frame buffer (which has limited precision) to off-screen buffers that can be attached to higher-precision integer, float or packed pixel RGB10_A2 textures. The render-to-texture feature of FBO's is used to store the results of a rendering for intermediate calculations. Finally, contents of these off-screen buffers are made visible by a simple blit to the onscreen window (Figure 6). An FBO can be attached to multiple color buffers of various types that can be written to simultaneously. The `framebufferObject` helper class in the attached 30bitdemo example handles the creation, attachment and management of FBO's.

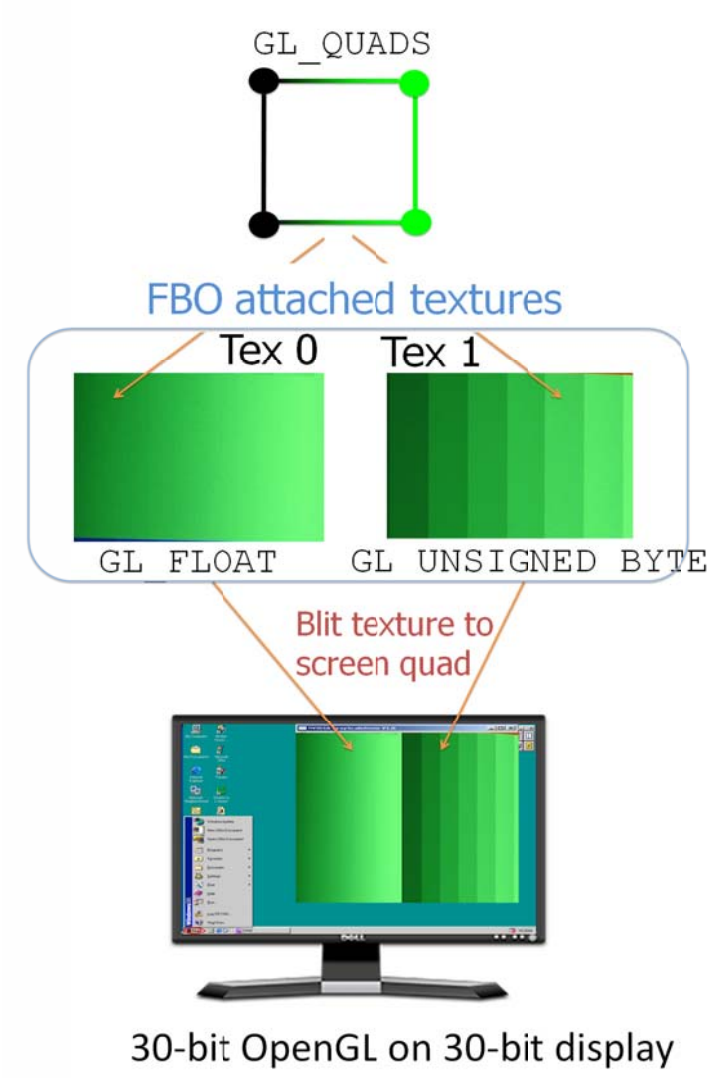

Figure 6. Off-Screen Rendering Using FBO's

The following code snippet shows the two texture formats – unsigned byte and floating point as intermediate rendering targets. The scene is rendered to these 2 textures simultaneously and a second pass blits the 2 textures side by side to an onscreen quad. It is at this point that the data is scaled to the window pixel format specific 30-bit RGB. Full source is given in the accompanying 30bitdemo project. Key 'f' when running 30bitdemo toggles rendering between the dual-viewport FBO mode and single viewport onscreen rendering.

```
CFBO fbo; GLuint fboTextures[2];
unsigned int imgWidth = 2048, imgHeight = 2048;
void oglInit() {
    glGenTextures(2, &fboTextures);
    // offscreen floating pt texture
    glBindTexture(GL_TEXTURE_2D, fboTextures[0]);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F_ARB, imgWidth, imgHeight,
    0, GL_RGBA, GL_HALF_FLOAT_ARB, 0 );
    // offscreen unsigned byte texture
    glBindTexture(GL_TEXTURE_2D, fboTextures[1]);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, imgWidth, imgHeight, 0,
        GL_RGBA, GL_UNSIGNED_BYTE , 0);
    fbo.AttachTexture(GL_TEXTURE_2D, fboTextures[0],
        GL_COLOR_ATTACHMENT0_EXT);
    fbo.AttachTexture(GL_TEXTURE_2D, fboTextures[1],
    GL_COLOR_ATTACHMENT1_EXT);
    fbo.IsValid();// Validate the FBO after attaching textures
    CFBO::Disable(); //Disable FBO rendering for now
}

//1:1 texel->onscreen quad blit
void drawQuads() {
    glBegin(GL_QUADS);
      glTexCoord2f(0.0, 0.0); glVertex2f(0.0, 0.0);
      glTexCoord2f(0.0, imgHeight); glVertex2f(0.0, 1.0);
      glTexCoord2f(imgWidth, imgHeight); glVertex2f(1.0, 1.0);
      glTexCoord2f(imgWidth, 0.0); glVertex2f(1.0, 0.0);
    glEnd();
}

void oglDraw (int winWidth, int winHeight) {
    fbo.bind(); // Draw to offscreen buffer
    // 1st PASS - Render to the two offscreen textures at once
    GLenum buffers[]={GL_COLOR_ATTACHMENT0_EXT,GL_COLOR_ATTACHMENT1_EXT};
    glDrawBuffers(2, buffers);
    // TODO – All app drawing here including viewport/projection/camera
    fbo.unbind();// Revert to onscreen draw

    // 2nd PASS - Blit offscreen texture to onscreen window
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glEnable(GL_TEXTURE_RECTANGLE_NV);
    // Create left viewport, BLIT the float texture to screen
    glBindTexture(GL_TEXTURE_RECTANGLE_NV, fboTextures[0]);
    glViewport(0, 0, winWidth/2, winHeight);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, 0.5, 0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    drawQuads();
    // Create right viewport, BLIT the unsigned byte texture to screen
    glBindTexture(GL_TEXTURE_RECTANGLE_NV, fboTextures[1]);
    glViewport(winWidth/2, 0, winWidth/2, winHeight);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, 0.5, 0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    drawQuads();
}
```

Rendering for 30-Bit

This section deals with 30-bit rendering modes for geometry, pixels and textures. Running the example 30bitdemo by default will display 2 identical windows. One rendered in 24-bit and another in 30-bit. The space bar switches between the 4 rendering modes as explained in the following section. Key 'f' toggles between dual-viewport off-screen and regular onscreen rendering explained in the previous section.

Shaded Geometry

Any existing shaded OpenGL drawing should work as is on a 30-bit system. Application color values for vertices, lights etc will be interpolated and shaded using high precision and only at the end, scaled to 10-bit per RGB component when the pixel is written. A good test to check 30-bit is to simply draw a shaded quad with existing color interpolated between the vertices. This same quad on an 8-bit pixel format will show banding while the 10-bit shows smooth transitions (Figure 7). The 30bitdemo on startup shows this mode. Maximizing the window or downscaling the color values will show more prominent banding on the 24-bit.

```
void oglDraw() {
    glViewport(0, 0, gWinWidth, gWinHeight);
    glMatrixMode(GL_PROJECTION);
    gluOrtho2D(-1.0, 1.0, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glShadeModel(GL_SMOOTH); // Enable Smooth Shading
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_QUADS);
        glColor3f(0.0,0.0,0.0); glVertex2f(-1.0,-1.0);
        glColor3f(0.0,0.0,0.0); glVertex3f(-1.0,1.0);
        glColor3f(0.0,0.5,0.0); glVertex3f(1.0,1.0);
        glColor3f(0.0,0.5,0.0); glVertex3f(1.0,-1.0);
    glEnd();
    // Swap Buffers
}
```

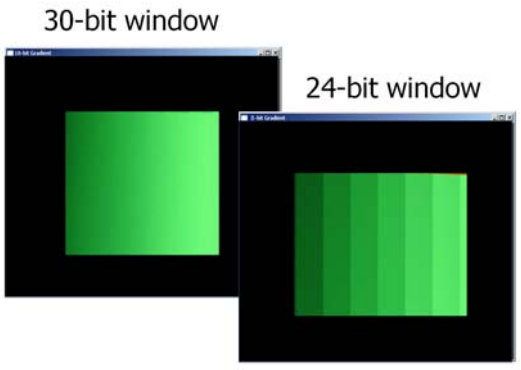

Figure 7. Comparing 24-Bit (Banded) and 30-Bit (Smooth)

Higher Bit Depth Pixels

For data types higher than 8-bit (for example, short or int) it is also possible and often simpler to do the pixel and texture operations as is on unsigned short/int data as shown in the following code sample. In this case, the color values in the application are scaled to the window bit depth at the display stage by the graphics. No code changes are required from an application programming perspective. However, the drawback is that when only 10-bits per component are needed for display, 16 and 32-bits per component are stored blowing up memory footprint. For example, a RGBA unsigned int texel requires 16bytes or 4X more than the corresponding 4 byte RGB16_A2 format.

```
unsigned int imgWidth, imgHeight;
unsigned short* data_short = new unsigned short[imgWidth*imgHeight*4];
//TODO - read from file or generate values to data
//TODO – Standard OpenGL texture initialization goes here
glPixelStorei(GL_UNPACK_ALIGNMENT, 4); //4 components per pixel
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16, imgWidth,  imgHeight, 0,
        GL_RGBA,GL_UNSIGNED_SHORT, data_short);
//Make sure we got the right RGBA16 internal format
GLint internalFormat;
glGetTexLevelParameteriv(GL_TEXTURE_2D,0, GL_TEXTURE_INTERNAL_FORMAT,
        &internalFormat);
assert(internalFormat == GL_RGBA16);
```

Packed Pixel Format

The OpenGL Packed Pixel extension[6] addresses the memory overhead introduced earlier with higher bit depth pixels. A packed pixel is represented entirely by 1 unsigned integer. 10 bits are allocated for each of the RGB components and 2 alpha bits (referred to as RGB10_A2). The following code snippet shows the 2D unsigned short image allocated earlier packed into the RGB10_A2 format. Note how the alignment here is set to 1 instead of 4 in the previous example as the packed pixel is considered as a 1-component texture.

```
//typedef here
glEnum packedType =  GL_UNSIGNED_INT_10_10_10_2; //or
GL_UNSIGNED_INT_2_10_10_10_REV
glEnum packedFormat =  GL_RGBA; //GL_RGB/GL_BGRA/GL_BGR

//the 4 components will be packed in 1 unsigned int
unsigned int* packedData = new unsigned int[imgWidth*imgHeight];
for (unsigned int y=0; y < imgHeight; y++)
    for (unsigned int x=0; x < imgWidth; x++) {
    uint alpha = data[offset*4+3]&0x3;
    blue = data[offset*4+2]&0x3FF;
    green = data[offset*4+1]&0x3FF;
    red = data[offset*4]&0x3FF;
    if (packedType == GL_UNSIGNED_INT_10_10_10_2) {
        if (packedFormat == GL_RGBA) //R10G10B10A2
        packedData[offset] = (red<<22 | green<<12 | blue<<2 | alpha);
        else  //glformat is GL_BGRA, packed in B10G10R10A2
        packedData[offset] = (blue<<22 | green<<12 |red<<2 | alpha);
    }
    else if (packedType == GL_UNSIGNED_INT_2_10_10_10_REV) {
        if (packedFormat == GL_RGBA) //A2B10G10R10
        packedData[offset] = (alpha<<30 | red<<22 | green<<12 | blue<<2);
        else  //glformat = GL_BGRA, packed as A2R10G10B10
        packedData[offset] = (alpha<<30 | red<<22 | green<<12 | blue<<2);
    }
}
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

The populated packed pixel data can now be used as the source to a 2D texture. The internal format is queried after the texture download to verify that the requested format is supported by the hardware. When drawing, this texture is mapped 1:1 onto an onscreen quad.

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB10_A2, imgWidth, imgHeight, 0,
        packedFormat , packedType, packedData);
//confirm we have packed pixel internal format
glGetTexLevelParameteriv(GL_TEXTURE_2D,0, GL_TEXTURE_INTERNAL_FORMAT,
        &internalFormat);
assert(internalFormat == GL_RGB10_A2);
```

OpenEXR File Format

OpenEXR [7], developed by ILM, is a set of C++ libraries which implement a high-dynamic range (HDR) image file format for computer imaging. It supports 3 pixel types -- 16-bit floating-point, 32-bit floating-point, and 32-bit integer. Its 16-bit FP called "half" is compatible with NVIDIA's 16-bit FP format and supported natively on the Quadro FX hardware. Reading a EXR image and mapping to OpenGL textures is made simple with the open source ILM library as shown below. The ILM library and header files along with sample EXR images are included with the accompanying code.

```
#include <ImfRgbaFile.h>
#include <ImfArray.h>
#include <half.h>

//Load EXR file now
Imf::Rgba * pixelBuffer;
try
{
    Imf::RgbaInputFile in("test.exr");
    Imath::Box2i win = in.dataWindow();
    Imath::V2i dim(win.max.x - win.min.x + 1, win.max.y - win.min.y + 1);
    pixelBuffer = new Imf::Rgba[dim.x * dim.y];
    int dx = win.min.x;
    int dy = win.min.y;
    in.setFrameBuffer(pixelBuffer - dx - dy * dim.x, 1, dim.x);
    in.readPixels(win.min.y, win.max.y);
    imgWidth = dim.x; imgHeight = dim.y;
}
catch(Iex::BaseExc & e) {
    std::cerr << e.what() << std::endl;
    // Handle exception.
}
glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0, GL_RGBA16F_ARB, imgWidth,
        imgHeight, 0,GL_RGBA,GL_HALF_FLOAT_ARB, pixelBuffer);
glGetTexLevelParameteriv(GL_TEXTURE_RECTANGLE_NV,0,
        GL_TEXTURE_INTERNAL_FORMAT, &internalFormat);
assert(internalFormat == GL_RGBA16F_ARB);
```

Multi-Display Configurations

In the case when a single GPU drives multiple displays, as long as there is one 30-bit compatible display active, 30-bit color is automatically enabled on all displays. At scan out, the driver automatically downgrades the 30-bit output to 24-bit for regular displays.

Multi-GPU Compatibility

It is possible to combine the 30-bit color capable Quadro boards (**Error! Reference source not found.**) with each other to drive more than 2 displays. In this case, for a valid 30-bit pixel format to be obtained, the primary display MUST be attached to the GPU also driving the 30-bit panel. The ultra high-end cards like the Quadro FX 5800 requires the full 2 auxiliary power inputs and are typically used with lower-end Quadro cards that do not have any auxiliary power requirements (for example, Quadro FX 1800).

Multi-Display Setup

To enable multi-display from the desktop, open the Display Properties→Settings Tab and check the "Extend my Windows desktop onto this monitor" checkbox for each display as shown in Figure 8.

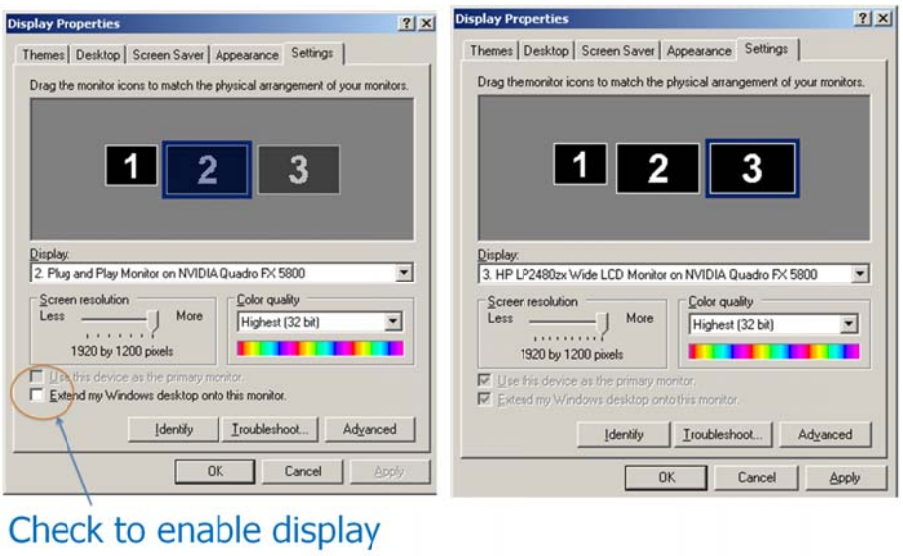

Figure 8. Display Properties Before and After all the Displays are Enabled

Programming for Multi-Display

For an application using multiple displays it is often useful to programmatically find out their attributes and capabilities. This section and the following sections show code samples to demonstrate that in progressive detail. Full source is found in Check30bit project within the accompanying source. The `CDisplayWin` structure defined in `CDisplayWin.[h|cpp]` encapsulates the attributes of each display with an attached window and the `displayWinList` is a container for all DisplayWin's.

```
class CDisplayWin {
    HWND      hWin; // handle to display window
    HDC       winDC; // DC of display window
    HGLRC     winRC; // GL resource context from winDC
    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      color30bit; //Is this a 30-bit display
public:
    //Full public function definitions are found in CDisplayWin.h
}
#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];
```

The following code 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 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);
    }//if attached to desktop
    displayCount++;
} //while(enumdisplay);
```

Running this enumeration code on our 3 display example (shown in Figure 8) prints out the following.

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

DeviceName    = \\.\DISPLAY2
DeviceString  = NVIDIA Quadro FX 5800
Position/Size = (1280, 0), 1920x1200

DeviceName    = \\.\DISPLAY3
DeviceString  = NVIDIA Quadro FX 5800
Position/Size = (3200, 0), 1920x1200
```

Note that the enumeration shown in this section abstracts special hardware capabilities of the displays such as 30-bit color capability and the display connector used (DisplayPort or DVI). The next section addresses that.

Mixing 30-Bit Color with 24-Bit Color Displays

The previous section demonstrated how to get the general characteristics of a display such as extents etc, but more specific properties of monitors will decide how to layout the application. For example, user interface and launching elements can be placed on the regular 24-bit color LCD's while the HDR images can be rendered to the 30-bit color displays.

For a display to be 30-bit color compatible, it must be connected via DisplayPort to the graphics card and support 10-bit per component. This information is provided by the NVIDIA NVAPI [8] – an SDK that gives low level direct access to NVIDIA GPUs and drivers on all windows platforms. The following example enumerates the attached displays and its associated GPU using NVAPI similar to the WinGDI example earlier. The complete source with the error checking is found in `Check30Bit.cpp` in the accompanying SDK.

```
// 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 display port info for this display
    NV_DISPLAY_PORT_INFO curDPInfo = {0};
    curDPInfo.version = NV_DISPLAY_PORT_INFO_VER;
    nvapiStatus = NvAPI_GetDisplayPortInfo(hDisplay[i], curDisplayId,
                      &curDPInfo);
    nvapiCheckError(nvapiStatus);

    //if GPU & monitor both support 30-bit color AND they are connected
    by display port, set the color30bit flag
    if (curDPInfo.isDp &&(curDPInfo.bpc == NV_DP_BPC_10))
        displayWinList[i].color30bit = true;
    else
        displayWinList[i].color30bit = 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 30-bit window to a 24-bit color display so that the image quality is preserved. In the event-handling code for a window move and resize, all the displays that the current window spans are queried to check for 30-bit color 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;
       GetWindowRect(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 30-bit color display
               if (!displayWinList[i]. color30bit) {
               //do something eg prevent spanning
               printf("WM_MOVE : Window is now spanning a
                      NON-30bit color display\n");
               }
           }
           //Send a draw message to all the windows
           InvalidateRect(displayWinList[i].hWin, NULL, FALSE);
           UpdateWindow(displayWinList[i].hWin);
       } //end of for
        break;
       case WM_MOVE:
       RECT rect;
       //Repeat as above for WM_SIZE

   }
}
```

References

[1] IEC sRGB official specification - IEC 61966-2-1:1999

[2] AdobeRGB (1998) specification - http://www.adobe.com/digitalimag/pdfs/AdobeRGB1998.pdf

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

[4] GLUT library - http://www.opengl.org/resources/libraries/glut/

[5] OpenGL FBO extension - http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object

[6] OpenGL Packed Pixel extension - http://www.opengl.org/registry/specs/EXT/packed_pixels.txt

[7] Kainz, F. 2004. The OpenEXR image file format. In GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics, R. Fernando, Ed. Pearson Higher Education, ch. 26, 425--444

[8] NVIDIA NVAPI – www.nvapi.com

[9] 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 source code is divided into 2 separate projects listed as follows. The intent is for these components to be mixed and matched according to the user application requirements.

- **30bitdemo.sln**
  - **30bitdemo.[cpp|h]** – Main test program that opens a 30-bit and 24-bit window and allows user to cycle between different rendering methods explained in the paper as well as onscreen and off-screen rendering.
  - **framebufferObject.[cpp|h]** – Class that encapsulates all attributes and operations related to a Framebuffer Object.
- **Check30bit.sln**
  - **CDisplayWin.[cpp|h] –** Class CDisplayWin that encapsulates all attributes of an attached display such name, extents, driving GPU, 30-bit compatibility etc.
  - **Check30Bit.cpp** – Main program that enumerates all attached GPU's and display's using Win GDI API and uses NVIDIA NVAPI to check the displays that are 30-bit color compatible.

Notice

Macrovision Compliance Statement

NVIDIA Products that are Macrovision enabled can only be sold or distributed to buyers with a valid and existing authorization from Macrovision to purchase and incorporate the device into buyer's products.

Macrovision copy protection technology is protected by U.S. patent numbers 5,583,936; 6,516,132; 6,836,549; and 7,050,698 and other intellectual property rights. The use of Macrovision's copy protection technology in the device must be authorized by Macrovision and is intended for home and other limited pay-per-view uses only, unless otherwise authorized in writing by Macrovision. Reverse engineering or disassembly is prohibited

Trademarks

Copyright