



NVIDIA 3D VISION PRO AND STEREOSCOPIC 3D

WP-05482-001_v01 | October 2010

White Paper



DOCUMENT CHANGE HISTORY

WP-05482-001_v01

Version	Date	Authors	Description of Change
01	October 8, 2010	SN, CK, SM	Initial Release

TABLE OF CONTENTS

NVIDIA 3D Vision Pro and Stereoscopic 3D	1
Introduction.....	1
Sources for Data.....	1
Stereo.....	1
Anaglyph Stereo.....	2
Polarized Light.....	2
LCD Shuttered Glasses.....	3
NVIDIA 3D Vision Products.....	3
NVIDIA 3D Vision (Consumer Version).....	4
Hardware.....	4
Software.....	4
How It Works.....	5
NVIDIA 3D Vision Pro (Professional Version).....	7
Hardware.....	7
Implementation Details.....	7
Get Transceiver Information, Change Channel, and Pair Glasses (Example).....	8
Writing Stereo Applications.....	10
OpenGL Implementation.....	10
Configure the Hardware to Support Stereo.....	10
Query and Request Stereo Capability.....	10
Render to Each Eye, and Swap Buffers.....	11
Direct3D Implementation.....	13
References.....	15

LIST OF FIGURES

Figure 1. NVIDIA 3D Vision Kit.....	4
Figure 2. Frustum for Each Eye Is Horizontally Translated Version of Mono Frustum..	6
Figure 3. NVIDIA 3D Vision Pro Kit.....	7

NVIDIA 3D VISION PRO AND STEREOSCOPIC 3D

INTRODUCTION

There is an insatiable need for increasing the realism in how 3D objects are represented to viewers in computer graphics. As the limitations of hardware and software continue to improve, users can now experience unprecedented realistic 3D effects. A technique that is gaining popularity is colloquially called “3D,” but is more correctly described as “stereoscopic 3D” or just “stereo.” This technique for representing objects in 3D is not new, but due to improvements in both hardware and software the technique has gone mainstream from what was once a niche market.

Sources for Data

The origins for 3D data can vary widely, from data obtained from scanners to models created by artists with 3D modeling software. To accurately represent these objects on a 2D computer display, you can use many visual clues to trick the mind into perceiving it as a true 3D object. Simple visual clues such as distant objects appearing smaller, and near objects obscuring far objects are very effective, but augmenting these with stereo can create a truly realistic 3D experience.

Stereo

To achieve a stereoscopic 3D representation, it is necessary to have two copies of the image, one for the left eye and one for the right eye, where the images have a perspective offset relative to each other, which is appropriate for the eye separation of the viewer (inter-ocular distance). This separation tricks the brain into perceiving the

object at a relative depth from the viewer that is not constrained to the plane of the display device.

Many different hardware systems can deliver the eye separation needed to create the effect of 3D stereo. The method used to achieve 3D stereo for a large movie audience is different than for a single viewer interacting with a 3D object on a personal workstation. Likewise, the user experience is much better with active shutter glasses that are exactly synchronized with the displays than inexpensive glasses that filter the images based on color differences.

The following section briefly discusses some of the techniques used to achieve the stereo effect.

Anaglyph Stereo

Anaglyph stereo is the easiest way to achieve 3D stereo. This process involves separating an image into a left and right eye image by a separation in color space. The image is then viewed with glasses that are chromatically opposite—usually red and cyan—to act as filters, and reveals only the left eye image to the left eye, and the right eye image to the right eye. The viewer then fuses the image into a composite image that is perceived in three-dimensional space.

The primary advantage of this technique is that no special display device is required to view anaglyph images, and only inexpensive paper glasses can be used for viewing. This method typically gives the *least satisfying stereoscopic effect*, with poor representation of the full-color spectrum.

Polarized Light

Another method for separation of the eyes is to use polarized light. For this method, the left and right eye images are distinguished by polarizing the light in either a circular or linear manner. As with anaglyph, glasses with passive filters are used to only allow light for the correct eye to pass through.

This method is superior to anaglyph because it can show the complete color spectrum, but requires special projection screens or monitors that preserve the polarity of the light. The implementation can either use two separate projectors, or one projector with a shutter that alternates polarization for each eye.

LCD Shuttered Glasses

The technique we are most interested in this paper is the use of LCD shuttered glasses. This method uses glasses that have an LCD for each eye, and the glasses are synchronized to open and close when the corresponding eye image is to be viewed. The right and left eye are presented sequentially on alternating frames, allowing for viewing the full resolution of the image for each eye.

This technique is used for the NVIDIA 3D Vision™ product, and is the *best method to obtain a true 3D stereoscopic effect*. The advantage of this technique is that the image for each eye is displayed at full resolution. Plus, there are no restrictions on the viewing angle to maintain the 3D effect, and the synchronization between the glasses and the display device can be controlled to ensure an excellent 3D stereoscopic effect.

The synchronization signal can be sent to the glasses in a variety of ways, typically over a wire or infrared wireless signal.

The next section describes the NVIDIA 3D Vision product in more detail.

NVIDIA 3D VISION PRODUCTS

Two versions of the NVIDIA 3D Vision product are available: NVIDIA 3D Vision and NVIDIA 3D Vision Pro. Both use the LCD shuttering method for obtaining stereo, but there are several differences between the products, which are described below.

NVIDIA 3D Vision (Consumer Version)

Figure 1 shows the contents of the NVIDIA 3D Vision Kit.



Figure 1. NVIDIA 3D Vision Kit

Hardware

The consumer version of NVIDIA 3D Vision consists of wireless LCD shuttered glasses that receive an infrared signal from an emitter connected to the PC via a USB cable. The glasses are shuttered at 120 Hz frequency, updating each eye 60 times per second for a flicker-free stereoscopic experience. Compatible display devices include CRTs, NVIDIA 3D Vision Ready LCD monitors, and DLP TVs (see <http://www.nvidia.com/object/3d-vision-requirements.html> for a list of supported displays).

Software

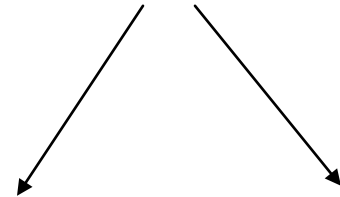
The consumer version of NVIDIA 3D Vision was developed to provide the 3D stereoscopic effect for games that do not already have a stereo mode. This is accomplished by a driver that intercepts Direct3D graphics calls, and creates a left eye view and right eye perspective view with an appropriate offset between them based on the depth information obtained from the 3D scene.

How It Works

3D game data is sent to the stereoscopic driver.



The driver takes the 3D game data and renders each scene twice—once for the left eye and once for the right eye.



A stereoscopic display then shows the left eye view for even frames (0, 2, 4, and so on), and the right eye view for odd frames (1, 3, 5, and so on).



This approach works very well for many games, but an improved experience can be enjoyed if developers pay attention to such things as

- ▶ Ensuring that the user interface text and heads-up-display information is displayed at screen depth so that they appear to be in mono.
- ▶ Taking care to render objects at the appropriate depth. For example, the sky box should be drawn with a valid depth that is at the maximum far distance in the scene.
- ▶ Making sure all the 3D objects have a valid depth relative to the rest of the scene.

Carefully choosing the depth of the objects in the scene is important, particularly for objects that appear to come out of the screen. Objects that appear to come out of the screen are uncomfortable to look at because the viewer's brain must overcome the fact it's seeing something in front of where their eyes are focused. The object should not get close to the edges of the window, where it might get clipped. And, the object should move slowly from inside the screen to out of the screen to give time for the viewer to adjust, using smooth transitions (Figure 2).

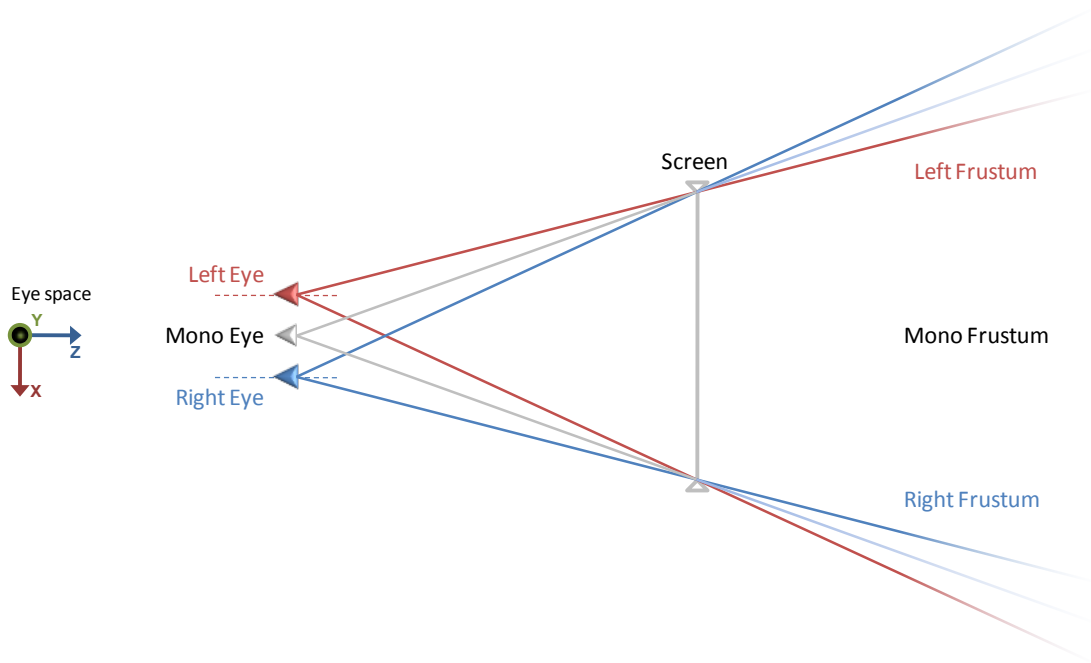


Figure 2. Frustum for Each Eye Is Horizontally Translated Version of Mono Frustum

NVIDIA 3D Vision Pro (Professional Version)

Hardware

Figure 3 shows the 3D Vision Pro glasses and hub.



Figure 3. NVIDIA 3D Vision Pro Kit

Instead of using an infrared signal from the emitter to the glasses to signal the transition between left and right eyes, the signal for NVIDIA 3D Vision Pro uses a radio frequency (RF) signal. The advantage of using RF is that the communication is bidirectional and glasses can be paired with an emitter. This scenario enables the use of multiple emitters from different computers in the same proximity without interference. It also allows multiple emitters to be used from one computer to cover a larger physical space. And since the glasses can communicate with the hub, additional capabilities are possible, as will be described in the next section.

When connected to a high-end NVIDIA[®] Quadro[®] graphics board, NVIDIA 3D Vision Pro has a wired connection to the 3-pin mini-DIN stereo connector on the board. This connection guarantees synchronization between the GPU and the hub regardless of the operating system or the CPU load on the system, and results in a flicker-free stereoscopic 3D experience.

Implementation Details

Using NVIDIA's NvAPI interface, users can control many aspects of the NVIDIA 3D Vision Pro product. Here are some examples of how to use this interface.

Get Transceiver Information, Change Channel, and Pair Glasses (Example)

```

NvAPI_Status CreateContext( Nv3DVPContextHandle * pContext )
{
    assert( NULL != pContext );
    Nv3DVPContextHandle contextPre( *pContext );
    NvAPI_Status status = NvAPI_3DVP_CreateContext( pContext );
    return status;
}

bool EnumTransceivers( const Nv3DVPContextHandle context,
                      TransceiverHandles * pTransceivers )
{
    assert( NULL != pTransceivers );
    NvAPI_Status status;
    Nv3DVPTransceiverHandle transceiver( NULL );
    pTransceivers->clear();
    do
    {
        Nv3DVPTransceiverHandle transceiverPre( transceiver );
        status = NvAPI_3DVP_EnumTransceiver( context,
                                             &transceiver );

        if ( NVAPI_OK == status )
        {
            pTransceivers->push_back( transceiver );
        }
    }
    while ( NVAPI_OK == status );
    return ( NVAPI_END_ENUMERATION == status );
}

NvAPI_Status OpenTransceiver( const Nv3DVPContextHandle context,
                              const Nv3DVPTransceiverHandle
                              transceiver,
                              const NVAPI_3DVP_ACCESS access )
{
    NvAPI_Status status = NvAPI_3DVP_OpenTransceiver( context,
                                                       transceiver, access );
    return status;
}

Nv3DVPContextHandle context( NULL );
TransceiverHandles transceivers;
NVAPI_3DVP_TRANSCEIVER_INFO info;

// (0) Initialize NvAPI

```

```

NvAPI_initialize();

// (1) Create a context
status = CreateContext( &context );

// (2) Enumerate all the transceivers
Status = EnumTransceivers( context, &transceivers);

// (3) Open transceiver.begin() for read/write access
if ( NvAPI_OK != OpenTransceiver( context, *transceivers.begin(),
                                  NVAPI_3DVP_ACC_READWRITE ) )
{
    //Error: cleanup with NvAPI_3DVP_DestroyContext(context)
}

// (4) Get transceiver information
status = NvAPI_3DVP_GetTransceiverInfo( context,
                                       *transceivers.begin(),
                                       &info );

printf(" version      = 0x%x\n", info.version );
printf(" hwFeatures   = 0x%x\n", info.hwFeatures );
printf(" fwRevA       = 0x%x\n", info.fwRevA );
printf(" fwRevB       = 0x%x\n", info.fwRevB );
printf(" fwRevC       = 0x%x\n", info.fwRevC );
printf(" rfAddress    = %d.%d.%d.%d.%d\n", info.rfAddress.a0,
                                       info.rfAddress.a1, info.rfAddress.a2, info.rfAddress.a3,
                                       info.rfAddress.a4 );
printf(" channelCount = %d\n", info.channelCount );

// (5) Set transceiver mode to NVAPI_3DVP_TM_LOW_RANGE
status = NvAPI_3DVP_SetTransceiverMode( context,
                                       *transceivers.begin(),
                                       NVAPI_3DVP_TM_LOW_RANGE );

// (6) Set channel to 1
status = NvAPI_3DVP_SetTransceiverChannel( context,
                                       *transceivers.begin(), 1 );

// (7) Pair glasses
NvU32 timeOut=4;
status = NvAPI_3DVP_PairGlasses( context, *transceivers.begin(),
                                 timeOut );

Sleep( timeout * 1000 );

// (8) Close Transceivers
for ( TransceiverHandles::const_iterator iter(
      transceivers.begin() ); transceivers.end() != iter; ++iter )
{
    if ( NvAPI_OK != NvAPI_3DVP_CloseTransceiver( context, *iter )
    )

```

```

    {
        printf("// ERROR: failed to close transceiver\n" );
    }
}

```

WRITING STEREO APPLICATIONS

Applications may use either the OpenGL or Direct3D API to implement stereoscopic 3D. The next two sections show implementations for both.

OpenGL Implementation

To get the full benefit of the stereoscopic experience, developers need to change an application to render to what is commonly referred to as “quad-buffered stereo.” This has historically only been possible using the OpenGL API, but with NVIDIA’s NvAPI interface, Direct3D developers can now also create quad-buffered stereo applications.

Following, these steps are described in more detail:

- ▶ Configure the hardware to support stereo.
- ▶ Query and request a stereo capability.
- ▶ Create an asymmetric viewing frustum.
- ▶ Render to separate left and right eye buffers with an eye offset translation between.
- ▶ Call `SwapBuffers` to swap both the left and the right eye buffer.

Configure the Hardware to Support Stereo

Go to the NVIDIA Control Panel and make these menu selections:

- ▶ 3D settings → Stereo Enable → On
- ▶ 3D settings → Stereo – Display mode → On-board DIN connector

These steps configure the environment to offer stereo pixel formats. Without changing from the default (Off), applications cannot obtain a stereo pixel format descriptor.

Query and Request Stereo Capability

An application has to show its intention to render in a stereo manner because the default configuration does not include stereo capabilities. On Windows, this is done by requesting a pixel format descriptor with the `PFD_STEREO` bit mask for the call to `DescribePixelFormat` and `ChoosePixelFormat`:

```

PixelFormat = DescribePixelFormat(hdc, iPixelFormat,
                                sizeof(PIXELFORMATDESCRIPTOR), &pfd);
if ((pfd.dwFlags & PFD_STEREO) == 0)
    // no stereo pixel formats available
    StereoIsAvailable = FALSE;
else
    StereoIsAvailable = TRUE;
if (StereoIsAvailable)
{
    ZeroMemory(&pfd, sizeof(PIXELFORMATDESCRIPTOR));
    pfd.nSize          = sizeof(PIXELFORMATDESCRIPTOR);
    pfd.nVersion       = 1;
    pfd.dwFlags        = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
                        PFD_DOUBLEBUFFER | PFD_STEREO;
    pfd.iPixelFormat   = PFD_TYPE_RGBA;
    pfd.cColorBits     = 24;

    iPixelFormat = ChoosePixelFormat(hdc, &pfd);
    if (iPixelFormat != 0)
    {
        if (SetPixelFormat(hdc, iPixelFormat, &pfd))
        {
            hglrc = wglCreateContext(hdc);
            if (hglrc != NULL)
            {
                if (wglMakeCurrent(hdc, hglrc))
                {
                    ...
                }
            }
        }
    }
}

```

Render to Each Eye, and Swap Buffers

Once a pixel format descriptor has been obtained with the **PFD_STEREO** capability, it is necessary to change the rendering slightly. For most stereo applications, all the rendering needs to be done twice, once for each eye. For the correct stereo projection, a left and right frustum is created with an offset from each other along the X axis by a simple translation along the X axis. This method is preferred over using a 'toe-in' method which occurs when using **gluPerspective** where both frustums point at the same focal point. This method incorrectly introduces vertical parallax between the left and right frustums.

The parameters to **glFrustum** should be modified to create asymmetric frustums for each eye for the correct stereo projection. The frustum asymmetry should be the same amount as the eye offset used, except since the parameters to **glFrustum** are taken at the near clipping distance and the frustum asymmetry adjustment is done at the convergence distance, the **FrustumAsymmetry** value needs to be scaled by the ratio of the near clipping distance to the convergence distance.

Then, instead of just rendering the scene once, it is necessary to render twice—once to each eye—using `glTranslate` to offset the rendering between the eyes by the desired eye offset.

This eye offset value should allow for user modification, but a good starting point is to use a total offset of 7% (3.5% for each eye) * $(X_{max} - X_{min})$, where the `Xmax` and `Xmin` values are the limits of the scene on the horizontal axis at the depth where it is desired to have zero parallax (the convergence distance).

Now, when beginning the rendering, instead of choosing the `glDrawBuffer(GL_BACK)` as is done when doing Mono rendering, you need to specify the back buffer for each eye:

```
GLfloat eyeOffset = Xrange*0.035f ; // Initial stereo separation
FrustumAsymmetry = eyeOffset*Znear/Convergence;
// Provide mechanism for user adjustments
eyeOffset *= Xrange*UserOffsetAdjustment;
FrustumAsymmetry *= UserFrustumAdjustment;
// Select back left buffer
glDrawBuffer(GL_BACK_LEFT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// Setup the frustum for the left eye
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

glFrustum(Xmin + FrustumAsymmetry,
          Xmax + FrustumAsymmetry,
          bottom, top, nearVal, farVal);
glTranslatef(eyeOffset, 0.0f, 0.0f);
// Setup the transformation matrix for the object.
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

<Rendering calls>
// Select back right buffer
glDrawBuffer(GL_BACK_RIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Setup the frustum for the right eye.
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(Xmin - FrustumAsymmetry,
          Xmax - FrustumAsymmetry,
          bottom, top, nearVal, farVal);
glTranslatef(-eyeOffset, 0.0f, 0.0f);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
<Rendering calls>
SwapBuffers(hdc);
```


Direct3D Implementation

Using NVIDIA's NvAPI interface, it is now possible to render quad-buffered stereo as a Direct3D application. The Direct3D application uses the NVIDIA 3D Vision stereo driver in a new mode that allows separate rendering for each eye.

For setup, the NVIDIA 3D Vision software needs to be installed and enabled.

Then, similar to the OpenGL implementation, the application needs to be changed prior to initial rendering, and then some changes need to be made to the rendering code.

The NVIDIA 3D Vision driver needs to be set into the **NVAPI_STEREO_DRVIER_MODE_DIRECT6** mode in order to be able to use the back buffer for left and right eye rendering. The application then needs to set up the viewport with half the width for the right eye and the other half for the left eye.

The projection matrix also needs to be modified to take into account the eye separation and convergence parameters. The rendering needs to be done once for each eye, and then after both eyes are complete, a single **present()** call will swap both eyes.

```
// initialize NvAPI and enable stereo
NvAPI_initialize();

// enable direct mode
NvAPI_Stereo_SetDriverMode(NVAPI_STEREO_DRVIER_MODE_DIRECT);

// Check for availability of Stereo capability
NvU8 isStereoEnabled;
NvAPI_Status status = NvAPI_Stereo_IsEnabled(&isStereoEnabled);

// Stereo status report an error
if ( status != NVAPI_OK)
{
    // 3D Vision driver is not installed on the system
    MessageBoxA(NULL, "Stereo is not available\nMake sure the
        stereo driver is installed correctly", "Stereo not
        available", MB_OK|MB_SETFOREGROUND|MB_TOPMOST);
}

// Stereo is available but not enabled, let's enable it
else if(NVAPI_OK == status && !isStereoEnabled)
{
    MessageBoxA(NULL, "Stereo is available but not enabled\nLet's
        enable it", "Stereo not enabled",
        MB_OK|MB_SETFOREGROUND|MB_TOPMOST);
    status = NvAPI_Stereo_Enable();
}
}
```

```

// create D3D device
ID3D10Device* pDevice;
D3D10CreateDevice( ..., &pDevice );

// create the StereoHandle
StereoHandle stereoHandle;
NvAPI_Stereo_CreateHandleFromIUnknown( pDevice, &stereoHandle );

// At this point Device is ready for stereo in direct mode until
// it's destroyed `

// NV_STEREO_ACTIVE_EYE enum
typedef enum {
    NVAPI_STEREO_EYE_RIGHT,
    NVAPI_STEREO_EYE_LEFT
} NV_STEREO_ACTIVE_EYE;

// Select the current side active
NVAPI_INTERFACE NvAPI_Stereo_SetActiveEye( StereoHandle
    stereoHandle, NV_STEREO_ACTIVE_EYE eye);

```

The viewport is split between the right and left eye:

```

g_LeftSideViewport.Width      = g_DXGISwapChainDesc.BufferDesc.Width/2;
g_LeftSideViewport.Height     = g_DXGISwapChainDesc.BufferDesc.Height/2;
g_LeftSideViewport.MinDepth   = 0.0f;
g_LeftSideViewport.MaxDepth   = 1.0f;
g_LeftSideViewport.TopLeftX   = 0;
g_LeftSideViewport.TopLeftY   = g_LeftSideViewport.Height/2;

g_RightSideViewport.Width     = g_DXGISwapChainDesc.BufferDesc.Width/2;
g_RightSideViewport.Height    = g_DXGISwapChainDesc.BufferDesc.Height/2;
g_RightSideViewport.MinDepth  = 0.0f;
g_RightSideViewport.MaxDepth  = 1.0f;
g_RightSideViewport.TopLeftX  = g_RightSideViewport.Width;
g_RightSideViewport.TopLeftY  = g_RightSideViewport.Height/2;

```

The projection matrix needs to be modified to take into account the stereo parameters.

The **NvAPI_Stereo_GetSeparation** call returns the eye separation normalized by the physical screen width, and **NvAPI_Stereo_GetConvergence** returns the current convergence value. These can be used to calculate a new projection matrix:

```

NvAPI_Stereo_GetSeparation(g_StereoHandle, &sep );
status = NvAPI_Stereo_GetConvergence(g_StereoHandle, &conv );

D3DXMATRIX mCurrentProj = g_Projection;
float CurrentSeparation = g_EyeSeparation * g_Separation;
if (CurrentEye == NVAPI_STEREO_EYE_RIGHT)
{
    mCurrentProj._31 += CurrentSeparation;
    mCurrentProj._41 -= CurrentSeparation*g_Convergence;
}
else
{
    mCurrentProj._31 -= CurrentSeparation;
    mCurrentProj._41 += CurrentSeparation*g_Convergence;
}
}
SetMatrix(..., mCurrentProj);

< Render >

NvAPI_Stereo_SetActiveEye(g_StereoHandle, NVAPI_STEREO_EYE_LEFT);

SetViewport(&g_LeftSideViewport);

//Present both left and right back buffers
IDXGISwapChain::Present();

```

REFERENCES

- ▶ NvAPI: <http://developer.nvidia.com/object/nvapi.html>
- ▶ NVIDIA 3D Vision: <http://www.nvidia.com/object/3d-vision-main.html>

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication of otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA, the NVIDIA logo, NVIDIA 3D Vision, and Quadro are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2010 NVIDIA Corporation. All rights reserved.