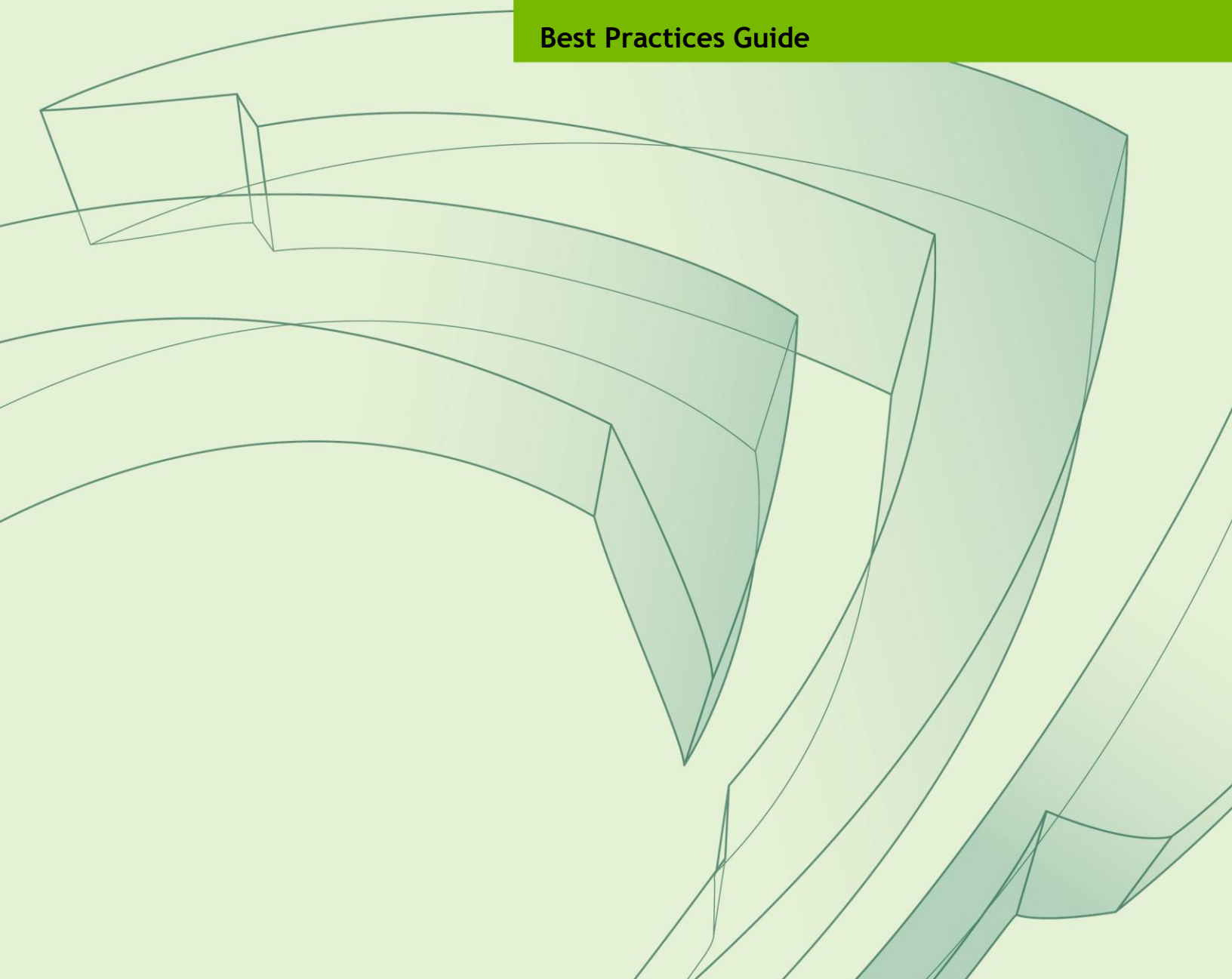
NVIDIA 3D VISION AUTOMATIC

BPG-05394-001_v01 | July 2010

Best Practices Guide



DOCUMENT CHANGE HISTORY

BPG-05394-001_v01

Version	Date	Authors	Description of Change
01	July 13, 2010	John McDonald	Initial release
02	January 3, 2011	John McDonald	Minor edits
03	January 27, 2011	John McDonald	Add information about vPos

TABLE OF CONTENTS

NVIDIA 3D Vision Automatic Best Practices Guide 6
Who Should Read This Guide? 6
Conventions 7
Background Information 8
Stereoscopic Background 8
Stereoizing Content in Games 9
Active Stereoization 9
Passive Stereoization 10
The Existing Conceptual Pipeline 10
How 3D Vision Automatic Fits in 12
Duplicate and Modify 12
Stereoscopic Issues 16
Defeating Driver Heuristics 16
NULL Z-buffer Target 16
Surface Aspect Ratios 16
2D Rendering 17
Rendering Without Separation 17
Specifying a Depth for Stereoization 17
Post Processing 20
Deferred Renderers 22
Scissor Clipping 23
Best Practices 24
The User Interface 24
Out of Screen Effects 25
Wrong is Right 25
Using NVAPI 25
Availability 25
Usage 26
Inclusion in your project 26
Initialization 26
Stereoscopic Functions 27
Retaining User Settings 27
Using nvstereo.h 28
Availability 28
Usage 28
Inclusion in your project 28
What it does 29
How to Use nvstereo.h 29

Stereoscopic QA 30
Testing Suggestions..... 30
User Settings..... 31
Separation 31
Convergence 31

LIST OF FIGURES

Figure 1. Spaces During Normal Render 10
Figure 2. Spaces During Stereoscopic Render 12
Figure 3. Separation and Convergence 14

LIST OF TABLES

Table 1. Determining When to Make a Call 27

NVIDIA 3D VISION AUTOMATIC BEST PRACTICES GUIDE

This Best Practices Guide is intended to help developers deliver the best possible stereoscopic 3D experience to their users. It contains an overview of how the NVIDIA® 3D Vision® Automatic technology fits into existing technology stacks, common problems that arise when using 3D Vision Automatic, and how to easily work around those issues.

While this document can be read serially, most technology stacks will not encounter all of the problems described herein. Instead, it is suggested that developers first read *Background Information* on page 8 to enable a common vocabulary then skip to the section that describes the specific problem or problems that have been encountered.

WHO SHOULD READ THIS GUIDE?

This guide is intended for programmers who have a basic familiarity with 3D rendering technology. You most likely already have an existing technology stack including a rendering engine, and are interested in pairing your stack with 3D Vision to provide a richer experience for your users.

CONVENTIONS

As this guide is aimed primarily at game developers, Direct3D conventions are used throughout. To be specific, we use:

- Left-handed coordinate systems
- Left-handed matrix operations
- Left-handed winding

To apply the contents of this document to OpenGL, one would need to flip the appropriate handedness, particularly of matrix operations and expected results from computations involving depth. At this time, 3D Vision Automatic is not available for OpenGL.

BACKGROUND INFORMATION

This chapter briefly covers the basics of all stereoscopic technologies then delves into the math behind the existing 3D conceptual pipeline. It extends the pipeline to encompass stereoscopic spaces and covers the basic operations 3D Vision Automatic performs on your behalf.

STEREOSCOPIC BACKGROUND

Delivering a compelling 3D image on a 2D screen has been the dream of content producers for nearly 160 years, beginning with anaglyph technology originally developed in the 1850s.

Although the technology has matured significantly, the basic concepts have remained largely the same.

Stereoscopic technology works by presenting each eye with a slightly different image, prepared as though the viewer was standing at a particular location. This can be done using many techniques.

Anaglyph, as already mentioned, was the first such technique. This is the traditional red/blue glasses with images that have been separated and desaturated. Originally developed in the 1850s, this technique saw fairly widespread adoption in Hollywood in the 1950s. This is also the technology used by 3D Vision Discover as a preview for what 3D Vision can deliver with a true shutter glass setup.

Polarization systems, as seen in some movie theaters, use multiple projectors with polarizing filters to display two images overlaid directly on top of one another. The viewer then wears passive glasses with matching polarized lenses—each eye gets one of the two images at exactly the same time.

Active systems, such as the shutter glasses sold by NVIDIA, use a set of lenses that alternatively become opaque and then transparent at a refresh rate that matches that of the user's display. By displaying only with monitors capable of using a native 120 Hz input signal, NVIDIA is able to deliver a smooth 60 Hz to each eye. This significantly reduces eyestrain over previous generations of shutter technology.

STEREOIZING CONTENT IN GAMES

There are two basic techniques to stereoized[1] content for games: *active* stereoization and *passive* stereoization.

Active Stereoization

In active stereoization, a game utilizes two different cameras to build and render distinct scenes to each eye. At the end of a single render update, the game engine tells the API which render target corresponds to which eye and moves on to building the next frame(s).

This poses significant challenges to developers that are developing in existing engines. One obvious obstacle is that many game engines rely on there being a single camera, while this technique requires two. In addition, the engine itself has to make decisions about which components of the scene are eye dependent and which components are not. Shadow maps are a common example of buffers that should be built only once. The engine additionally has to offer options to the user to manage the strength and complexity of the stereoscopic effect.

Active stereoization incurs runtime costs as well. For example, most titles already budget their draw call count to get the maximum fidelity possible while maintaining playable frame rates. Active stereoization will result in substantially more draw calls-up to twice as many—which can result in an application becoming severely CPU limited.

Finally, the effort to add support for active stereoization can range from a week or so to several months. The actual development time required is dependent upon the technology stack in question.

[1] Pseudo stereo effects or *Stereoizers* create a stereo signal from a source signal which is possibly (but not necessarily) a mono signal.

Passive Stereoization

Passive stereoization attempts to mitigate these shortcomings and reduce development effort without compromising on the quality of the 3D effect. The application continues to render the 3D scene as normal. The 3D display driver watches as rendering calls go by and builds stereoscopic information from the calls made to Direct3D.

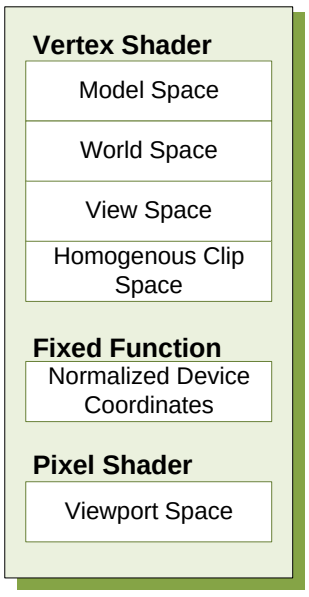
Using heuristics, the stereoscopic driver decides which objects need to be rendered per-eye and which do not, building the full left and right eye image in a manner that is transparent to the developer.

In effect, the stereoscopic driver does what an application developer would do, but it does so once for all titles. The result is a robust stereoscopic solution that requires significantly less effort from an application developer's standpoint.

Passive Stereoization, and specifically 3D Vision Automatic, is available on Windows Vista and Windows 7 for Direct3D 9, 10 and 11.

THE EXISTING CONCEPTUAL PIPELINE

On its way through the pipeline, a primitive is transformed through many different coordinate systems, or spaces. Understanding issues with stereoscopic is greatly simplified with a solid understanding of the existing pipeline, and how stereoscopic modifies that pipeline. A reasonable facsimile of the spaces that a primitive travels through during display is shown in Figure 1.



Geometry is usually authored in *model* space. In this space, geometry is rooted at a local origin, which is commonly a location near the base of the model. This allows the model to be easily placed in the *world* as you might place a chess piece on the board.

To place models in the world, they will be transformed by the world transform. World space is application-defined, and serves as a useful mechanism for relating objects with each other.

Figure 1. Spaces During Normal Render

Once in world space, models are further transformed by the view transform, to relocate the model into eye space. Eye space has many names, among them view space and camera space. (For the purposes of this document, we will consistently refer to this as *eye space* to avoid confusion with viewport space). In a left-handed coordinate system, eye space has its origin at (0, 0, 0), **X** increasing to the right, **Y** increasing upwards and **Z** increasing into the screen. The canonical View matrix is as follows:

```
Z = normalize(At - Eye)
X = normalize(Up x Z)
Y = normalize(Z x X)
⎛   X·x      Y·x      Z·x      0 ⎞
⎜   X·y      Y·y      Z·y      0 ⎟
⎜   X·z      Y·z      Z·z      0 ⎟
⎝-X·Eye   -Y·Eye   -Z·Eye     1 ⎠
```

After eye space, the model is further transformed by the projection matrix to clip or projected space. This transform typically serves three purposes. First, it scales the **X** and **Y** position of the object by the appropriate aspect ratio of the display window. Second, it reinterprets eye-space **Z** from [**Zn, Zf**] to [**0, Zeye**] and stores this in the **Z** coordinate of the vertex. Finally, the projection matrix stores eye-space **Z** in the **W** coordinate of the vertex. A typical left-handed projection matrix is given below.

$$\begin{pmatrix} \mathrm{Cot}(\mathrm{fov}_x) & 0 & 0 & 0 \\ 0 & \mathrm{Cot}(\mathrm{fov}_y) & 0 & 0 \\ 0 & 0 & \frac{Z_f}{Z_f - Z_n} & 1 \\ 0 & 0 & -Z_n * \frac{Z_f}{Z_f - Z_n} & 0 \end{pmatrix}$$

Note that while they are unique spaces, the *model*, *view* and *projection matrix* are often concatenated together to create the Model-View-Projection matrix. Vertices are transformed directly from model space to homogenous clip space.

Homogenous clip space is the last space developers typically think about, as it is the last space they can manipulate from the vertex shader. However, the trip through the conceptual pipeline is not complete. Next, vertices are clipped against a cube of size **W**. The vertex coordinates **X, Y** and **Z** are divided by **W**, and **1/W** is placed in the W coordinate for later perusal. This step, referred to as the perspective divide, is performed in fixed function hardware. The resulting vertex is in normalized device coordinates.

Vertices are then grouped into primitives according to ordering or an index buffer, and are rasterized. The viewport transform is applied, creating fragments in viewport space.

The pixel shader gets a chance to modify the output color, after which the final result is written to the render target for later usage or display to the user.

HOW 3D VISION AUTOMATIC FITS IN

3D Vision Automatic modifies the existing conceptual pipeline by splitting the post clip space pipeline into left- and right-eye spaces. The stereoscopic conceptual pipeline is shown in Figure 2. The following sections cover how the existing conceptual pipeline is modified to create the stereoscopic pipeline.

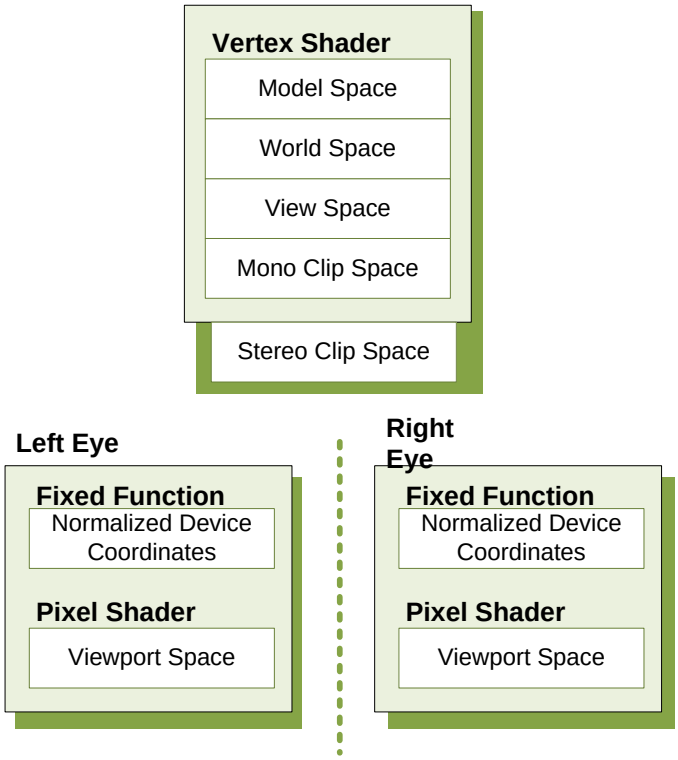


Figure 2. Spaces During Stereoscopic Render

Duplicate and Modify

Conceptually, the 3D Vision Automatic can be thought of doing two tasks: duplication and modification. As can be seen in the stereoscopic pipeline in Figure 1 2, viewport space has been split into a left- and right-eye viewport space. To support this, the driver does several things on the application's behalf:

- Duplicate render targets based on stereoscopic heuristics
- Modify vertex shaders to perform mono-to-stereoscopic Clip space transformation
- Replace individual draw calls with two draw calls, one per each eye

Duplicate Render Targets

One of the most obvious tasks that 3D Vision Automatic performs on an application's behalf is to duplicate the primary render target used. This allows the driver to build a render target to present each eye individually. Additionally, other render targets may be duplicated based on heuristic analysis at creation time. The driver handles all of the mapping for the developer, so that when the developer asks to bind a target, the appropriate per-eye target is bound. Likewise, if the developer uses render-to-texture, stereoization may be performed. If bound for reading, the proper left or right variant of the texture will be used.

Vertex Shader Modification

While render target replication is necessary for correct stereoscopic display, it is not sufficient. 3D Vision Automatic also monitors vertex shader creation, and adds a footer to each shader. By using a footer, the driver is able to operate in clip space, which has several unique properties. Among them is that clip space is oriented the same way for all applications, regardless of local concerns. It is also unit-less. Because it is directly before the perspective divide, clip space has the unique property that scaling the x, y, z and w coordinates of a vertex by a scalar factor affects the apparent stereoscopic depth without altering the rasterized location or z-buffer depth of the resultant fragments. Given that the shader has placed the position result in a variable called **PsInput**, the equation for the footer looks as follows:

$$PsInput.x \mathrel{+}= Separation * (PsInput_w - Convergence)$$

Convergence is set to a relatively low value by default, and can be modified by users using advanced global hotkeys that can be enabled or disabled in the NVIDIA Control Panel. Values less than the convergence value will experience negative separation, and appear to the user as *out of screen* effects. When $\mathbf{PsInput_w}$ is equal to Convergence, no separation is present. This is ideal for many HUD elements. Finally, objects further than Convergence depth will experience normal parallax effects. The effect becomes more pronounced the further the object is located from the Convergence plane.

Note: Current Convergence values can be read using the function call **NvAPI_Stereo_GetConvergence**, and set with **NvAPI_StereoSetConvergence.**

Separation is more easily controlled either by hotkeys or more commonly via the wheel on the back of the IR emitter that ships with GeForce 3D Vision. Users will adjust the magnitude or Separation, while the current eye being rendered (left or right) will modify the sign of separation—positive for the left eye and negative for the right eye.

Note: Current Separation can be computed as the result of multiplying the results from calls to **NvAPI_Stereo_GetSeparation** and **NvAPI_Stereo_GetEyeSeparation**, and dividing by 100.

Neither Separation nor Convergence should ever be set to negative values.

Figure 3 shows separation and convergence as they conceptually apply to stereo.

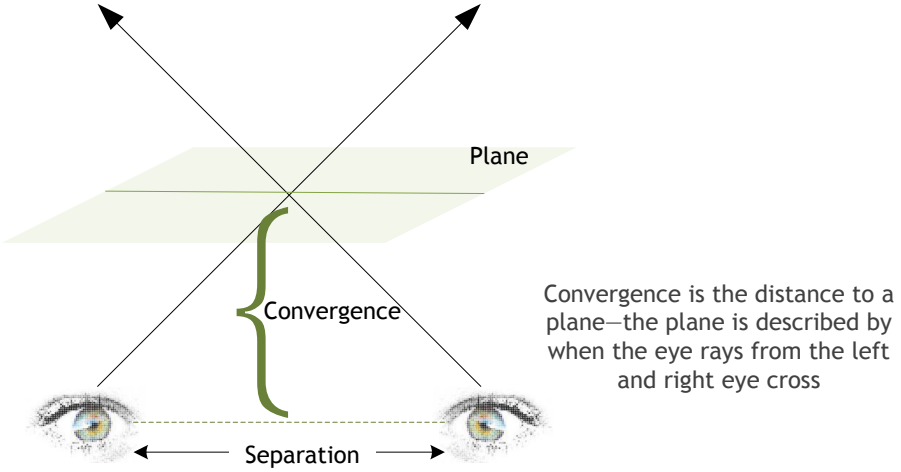


Figure 3. Separation and Convergence

Draw Call Substitution

When running in 3D Vision Automatic mode, application issued draw calls are substituted for two separate draw calls—one for the left eye and one for the right eye. The following pseudo-code could be thought of executing:

```
HRESULT NVDisplayDriver::draw()
{
  if (StereoActive) {
    VShader = GetStereoShader(curShader);
    VShaderConstants["Separation"] = curSeparation;
    VShaderConstants["Convergence"] = curConvergence;
    SetBackBuffer(GetStereoBuffer(curBackBuffer, LEFT_EYE));
    reallyDraw();

    VShaderConstants["Separation"] = -VShaderConstants["Separation"];
    SetBackBuffer(GetStereoBuffer(curBackBuffer, RIGHT_EYE));
    reallyDraw();
  } else {
    // Normal draw call stuff
    reallyDraw();
  }
}
```

Although this code is naturally a gross simplification, it is conceptually accurate.

Remembering it All

NVIDIA creates a stereoscopic profile for each and every title that goes through our QA process. These profiles contain settings to control and guide heuristics as well as specifying reasonable default values for separation and convergence. The stereoscopic profile also contains information that can be used to control the footer that is applied to vertex shaders.

STEREOSCOPIC ISSUES

DEFEATING DRIVER HEURISTICS

As described in Chapter 1.3, 3D Vision Automatic uses driver heuristics to decide which draw calls need to be stereoized and which ones should not be. In this section, some of the common problem heuristics are described, and their specific behaviors outlined.

NULL Z-buffer Target

The most common stereoscopic heuristic that applications run into is the state of the Z-buffer. When the Z-buffer is set to NULL, the 3D Vision Automatic driver uses this as a cue that the rendering being actively performed is a blit-with-shader, and disables the stereoscopic portion accordingly.

If you wish to avoid rendering to the Z-buffer while performing an operation, but still need that operation to be stereoized, set the Z-test function to ALWAYS and Z-write-enable to FALSE, while leaving a Z-target bound.

Surface Aspect Ratios

In Direct3D9, all Render Targets (surfaces created with IDirect3DDevice9::CreateRenderTarget), regardless of aspect ratio, are stereoized.

By default, non-square surfaces that are equal to or larger than the back buffer are stereoized. Non-square surfaces smaller than the backbuffer are not stereoized by default.

Square surfaces are (by default) not stereoized. The expected usage for these surfaces is typically projected lights or shadow buffers, and therefore they are not eye-dependent.

In order to apply these heuristics to a Direct3D9 title, applications should create the desired renderable surface with IDirect3DDevice9::CreateTexture, taking care to set the D3DUSAGE_RENDERTARGET bit of the usage parameter.

2D RENDERING

2D Rendering is typically the area of the rendering engine that requires the most care to get right for a successful stereoscopic title.

Rendering Without Separation

To render an object without separation, at the same screen-space position in the left and right eye, the best approach is to render these objects at convergence depth. This depth can be retrieved from NVAPI by calling **NvAPI_Stereo_GetConvergenceDepth**.

If the W coordinate of the output position from the vertex shader is at this depth, no separation will occur between each eye-the triangles will be at the same screen space position in each eye. See *Specifying a Depth for Stereoization* on this page for suggestions of how to modify the vertex shader output position.

While there are other methods available if your title has a custom 3D Vision Automatic profile, this method is the most robust.

Specifying a Depth for Stereoization

As explained in *The Existing Conceptual Pipeline* on page 10, controlling the **W** coordinate output from the vertex shader is the key to controlling the apparent depth of rendered objects. There are several methods to modify this value; the appropriate method for your application depends on your current pipeline and requirements. None of the methods described here should affect rendering when running in non-stereoscopic modes, and can be left enabled at all times for *free*.

Vertex Shader Constant (preferred)

The preferred method for modifying this depth is to pipe a constant into the vertex shader, then scale the entire output position of the vertex shader to this value.

For example, you might have a vertex shader that looks like this:

```
float4x4 gMatHudWVP;

struct VsOutput
{
```

```
    float4 Position   : SV_POSITION;
    float4 TexCoord0  : TEXCOORD0;
};

VsOutput RenderHudVS(float4 pos : POSITION,
                     float2 texCoord0 : TEXCOORD)
{
    VsOutput Out;

    Out.Position = mul(pos, gMatHudWVP);
    Out.TexCoord0 = texCoord0;

    return Out;
}
```

In this method, you would first pipe down an additional constant, then modify the final position by this coordinate. This is shown in the following code snippet:

```
float4x4 gMatHudWVP;
float gApparentDepth; // Depth at which to render the object

struct VsOutput
{
    float4 Position   : SV_POSITION;
    float4 TexCoord0  : TEXCOORD0;
};

VsOutput RenderHudVS(float4 pos : POSITION,
                     float2 texCoord0 : TEXCOORD)
{
    VsOutput Out;

    Out.Position = mul(pos, gMatHudWVP);
    Out.TexCoord0 = texCoord0;
    if (Out.Position.w != 0.0f && gApparentDepth > 0.0f) {
        Out.Position *= (gApparentDepth / Out.Position.w);
    }

    return Out;
}
```

Modify the Input Coordinate

Another approach to this problem, if your application is passing down a 4-component coordinate, is to scale the entire input coordinate by the desired depth. For example, if you were going to render a vertex at (1,2,3,1), you but you wanted it to render at an apparent eye-depth of 20, you could instead render at the position (20,40,60,20). This

would produce the exact same screen space position, but would yield correct apparent depth in stereo.

Modify the Transform

The final approach to specify an apparent depth to the vertex shader is to modify the transform appropriately. As with modifying the input coordinate (as described in *18Modify the Input Coordinate*), a scaling is all that is necessary. Apply a final scaling transform to your matrix that scales **X**, **Y**, **Z** and W by the desired apparent depth.

After the perspective divide, you will get the same rasterized position and the same value in the Z-buffer, but with a different amount of stereoscopic offsetting.

HUD in the World

A common effect in games is to render HUD elements in the world. Some examples of this are:

- Floating Character Names
- Waypoint Markers
- Targeting Reticules

In all of these cases, stereoscopic can provide additional information to your user over normal HUD layouts. Specifically, the depth cues given by stereoscopic can indicate whether the object being rendered is in front of or behind potential obstructions.

Consider Drawing at Apparent Depth

While these objects are technically part of the HUD, they will feel more connected to the object or location they represent in the world if they have a matching amount of separation. The solution to this is to draw the element at an apparent depth value using one of the methods described in *Specifying a Depth for Stereoization* on page 17.

Accurate screen depth can be computed using the following equation

$$ScreenDepth = normalize(Camera_{Fwd}) \cdot (Object_{WorldPos} - Camera_{WorldPos})$$

However, NVIDIA has found that this approximation works without causing eyestrain:

$$ScreenDepth = length(Object_{WorldPos} - Camera_{WorldPos})$$

Also note that for floating nameplates (for example), it's usually acceptable to draw the nameplate at the depth of the object the nameplate is logically attached to.

Crosshairs

Crosshairs fall into the category of objects described in *Wrong is Right* on page 25. They are actually a part of the HUD, but they just look wrong when drawn at screen depth.

Instead, NVIDIA recommends that you determine the depth of the object drawn under the hotspot of the crosshair, and draw at that apparent depth. NVIDIA refers to this type of crosshair as a *Laser Sight*.

Laser Sight Crosshairs appear to be on top of the object they're over, which is typically the desired effect. In general, bounding box depth is sufficient for specifying apparent depth. An application can perform a ray cast from the camera location through the crosshair, and compute the length of the returned vector. This length can be used as the apparent depth to render the crosshair at using any of the techniques covered in *Specifying a Depth for Stereoization* on page 17.

Mouse Cursor

The mouse cursor can be thought of exactly like a crosshair that can move around the screen. The guidance for mouse cursors is unsurprisingly identical to that for crosshairs. See *Crosshairs* on this page for more information.

POST PROCESSING

A common case in post processing is to unproject from window space to world space, perform some calculations and write a value out.

This will pose a problem for stereoscopic rendering because of the hidden mono-to-stereoscopic clip space transformation. To fix this, the mono-to-stereoscopic transformation will also need to be inverted. A potential example of existing unprojection code might look like this:

```
float4x4 WVPInv; // World-View-Projection Inverse

// Viewport Transform Inverse. Stored as
// x = 1 / (ResolutionX / 2)
// y = 1 / (ResolutionY / 2)
// z = -offsetX (usually -1)
// w = -offsetY (usually -1)
float4 VportXformInv;

float4 ScreenToClip(float2 ScreenPos, float EyeDepth) {
  float4 ClipPos = float4(ScreenPos.xy * VportXformInv.xy
                                       + VportXformInv.zw,
                          0,
                          EyeDepth);
```

```
  // Move the coordinates to the appropriate distance
  // for the depth specified.
  ClipPos.xy *= EyeDepth;

  // Screen and clip space are inverted in the Y direction
  // from each other.
  ClipPos.y = -ClipPos.y;

  return ClipPos;
}

float4 ScreenToWorld(float2 ScreenPos, float EyeDepth) {
  float4 ClipPos = ScreenToClip(ScreenPos, EyeDepth);

  return float4(mul(WVPInv, ClipPos).xyz, 1.0f);
}
```

The usual method of fixing this is to use a specially crafted stereoscopic texture, which has eye-specific parameters, and use these results to invert the stereoscopic transform. For example:

```
float4x4 WVPInv; // World-View-Projection Inverse

// Viewport Transform Inverse. Stored as
// x = 1 / (ResolutionX / 2)
// y = 1 / (ResolutionY / 2)
// z = -offsetX (usually -1)
// w = -offsetY (usually -1)
float4 VportXformInv;

// pixel(0.0).x = finalSeparation
// pixel(0,0).y = convergence
Texture2d StereoParmsTexture;

float4 StereoToMonoClipSpace(float4 StereoClipPos)
{
  float4 MonoClipPos = StereoClipPos;
  float2 StereoParms = tex2D(StereoParmsTexture, 0.0625).xy;
  MonoClipPos.x -= StereoParms.x * (MonoClipPos.w - StereoParms.y);

  return MonoClipPos;
}

float4 ScreenToClip(float2 ScreenPos, float EyeDepth) {
  float4 ClipPos = float4(ScreenPos.xy * VportXformInv.xy
                                       + VportXformInv.zw,
                          0,
                          EyeDepth);
  // Move the coordinates to the appropriate distance
  // for the depth specified.
  ClipPos.xy *= EyeDepth;
```

```
  // Screen and clip space are inverted in the Y direction
  // from each other.
  ClipPos.y = -ClipPos.y;

  return ClipPos;
}

float4 ScreenToWorld(float2 ScreenPos, float EyeDepth) {
  float4 StereoClipPos = ScreenToClip(ScreenPos, EyeDepth);
  float4 MonoClipPos = StereoToMonoClipSpace(StereoClipPos);

  return float4(mul(WVPInv, MonoClipPos).xyz, 1.0f);
}
```

The specifics of how to build the **StereoParmsTexture** are available in the **`nvstereo.h`** header file, described in *Using nvstereo.h* on page 28.

Note that this solution can utilize the same code path for both stereoscopic and non-stereoscopic cases—by specifying a 0 value for convergence and separation, the result of the computation will match the inputs. This is useful in reducing the code complexity of dealing with stereoscopic being disabled or enabled, and for whether the underlying hardware does or does not support stereo.

One additional difficulty in post processing comes from correctly determining the screen position in the first place. Often, games will perform the complete transform into screen space directly into the vertex shader, storing the results in the output vertex (typically in a texture coordinate). Unfortunately, this also misses the stereo transform, resulting in an incorrect position. This can be easily rectified by using the VPOS (SV_Position in Direct3D10 and higher) built-in attribute in the pixel shader to lookup the current pixel's screen position. The other alternative would be to perform the stereo transform manually in the vertex shader, before storing the output screen position.

Alternatively, you can make use of the freely available **`nvstereo.h`** header to handle building and updating this texture for you in a performance friendly manner. See *Using nvstereo.h* on page 28 for more details.

DEFERRED RENDERERS

Deferred Renderers suffer the unprojection problem described in Post Processing, but to a more extreme degree because unprojection is even more common in deferred renderers.

There are two main approaches to fixing this problem. The first solution is exactly the same as described in Post Processing on page 20.

The second solution is to simply skip unprojection altogether by storing the world-space position directly in the G-buffer. This solution, while trivial, is not usually practical due to space and bandwidth constraints.

SCISSOR CLIPPING

Special care should be taken when using the scissor rectangle for anything that may be stereoized. Currently there is no way to indicate a scissor rectangle for each eye. As a result, objects drawn with a scissor rectangle will likely be clipped incorrectly for both eyes. There is currently no workaround for this issue.

BEST PRACTICES

THE USER INTERFACE

The User Interface provides the most latitude for polish with stereo. While the following are general rules of thumb, they are not hard and fast rules.

- When rendering full screen menus (main menu, for example) or a pre-recorded, non-stereoized movie, render at Screen Depth.
- When rendering UI over the world, for example as part of a HUD or a popup menu, render at or just beyond Screen Depth.
- Ensure the mouse separation matches the separation of what's underneath it. This will cause the mouse to appear to render without separation, which is usually desired.
- Don't include stereoscopic options in-game. Users can use the button on the IR emitter to enable/disable stereo, and can use the wheel on the back to control separation.
- If your title has a very wide range of acceptable depth values, consider setting Convergence dynamically based on the depth of the scene. A common behavior for applications is to have one value of convergence per cutscene, and a different value for normal gameplay.
- Separation can be modified by the application, but this is generally considered a user setting and should be left alone.

OUT OF SCREEN EFFECTS

Out of screen effects are very cool, but they can also be very fatiguing to users. The brain does everything it can to reject the image being in front of the monitor. For that reason, NVIDIA recommends you use these effects very sparingly. Following these guidelines will also help:

- Try to have objects move in depth from being in-screen to out-of-screen. This gives the brain time to adapt and accept the object.
- Avoid having out of screen objects clip with the edges of the screen.
- Avoid allowing user control of the camera during out-of-screen effects as much as possible.

For these reasons, NVIDIA has found that out of screen effects work best during cut-scenes.

WRONG IS RIGHT

A subtle aspect of rendering in stereoscopic is that what is correct is not always right. Sometimes it is better to reduce eye strain than to be physically correct. This is particularly true for users that play with high values for separation.

One example of this is with very strong, very tight specular highlights. To be physically accurate, the application would need to be aware of the actual view vector for each eye, and then compute the specular highlights accordingly. In testing, NVIDIA has found that using a unified camera that matches the specular highlight in both eyes reduces eyestrain and feels better. As an added bonus, this method requires no additional effort on the part of the developer.

USING NVAPI

NVAPI provides you the ability to query and manipulate the way that 3D Vision Automatic works on behalf of your application; along with allowing you access to important internal variables.

AVAILABILITY

NVAPI is available for Windows in both 32- and 64-bit flavors.

Calls are guaranteed to be safe (in that they will not crash) even when made without NVIDIA drivers installed. For example if a user has another vendor's hardware installed, calls will simply return an error code instead of crashing or misbehaving.

Despite this, it is good practice to check the return codes, particularly if NVAPI is being relied on for application-critical functionality. Doing so allows a fallback path to be selected.

The public version of NVAPI is available at:
http://developer.nvidia.com/object/nvapi.html.

In order to provide the best possible stereoscopic experience for developers, NVIDIA has released all of the stereoscopic functions in NVAPI in the public version. However, there is an NDA version of NVAPI that may be available for your project with the appropriate paperwork in place. Contact NVIDIA Developer Support for more information.

USAGE

Using NVAPI in your project is a straightforward process, and is described in this section.

Inclusion in your project

NVAPI should be statically linked with your PC builds. The included static libraries are shims that will handle dynamic loading of the nvapi.dll (which is shipped as part of the NVIDIA driver). The shim will handle several cases for you automatically, for example when NVAPI is unavailable or when you've linked against a newer version of NVAPI than your user has installed on their system.

Include **`nvapi.h`** from your header files as normal, and then link against the appropriate 32- or 64-bit flavor of the library depending on your build configuration.

Note: At this time NVAPI should not be dynamically loaded, and does not currently support dynamic unloading.

Initialization

To initialize NVAPI, call the function **`NvAPI_Initialize`**. You can determine whether NVAPI was correctly initialized or not by checking the return code, although

you may still safely make calls to other NVAPI functions even if the library failed to load.

Stereoscopic Functions

Stereoscopic functions are categorized in two axes: *Getters* and *Setters*, and functions that take a Stereoscopic Handle and those that do not. The way that a function is categorized determines when it is valid to make calls against that function. Note that making a call when it is invalid to do so may lead to undefined behavior. For simplicity's sake, Table 1 provides the matrix that is used to determine when you may make a call:

Table 1. Determining When to Make a Call

	Getter	Setter
StereoHandle Parameter	After device creation	After device creation
No StereoHandle Parameter	Anytime	Before device creation

At the time of this writing, this is the list of functions that must be called **prior** to Direct3D device creation:

- **`NvAPI_Stereo_CreateConfigurationProfileRegistryKey`**
- **`NvAPI_Stereo_DeleteConfigurationProfileRegistryKey`**
- **`NvAPI_Stereo_SetConfigurationProfileValue`**
- **`NvAPI_Stereo_DeleteConfigurationProfileValue`**
- **`NvAPI_Stereo_Disable`**
- **`NvAPI_Stereo_Enable`**

Retaining User Settings

In order for an application to retain user settings from run to run (for example the separation and convergence values), an application needs only to call **`NvAPI_Stereo_CreateConfigurationProfileRegistryKey`** prior to device creation.

USING NVSTEREO.H

In order to simplify correct usage of NVAPI with stereo, NVIDIA has written a header file **`nvstereo.h`**, which can be freely used in your application. This header will work with Direct3D 9 or 10.

AVAILABILITY

`nvstereo.h` is available as part of the NVIDIA SDK 11.

USAGE

Inclusion in your project

The **`nvstereo.h`** header is a simple header library, and needs only to be included from your source code to be used. However, it does have a dependency on the public NVAPI. You can follow the instructions in Section 4.2.1 to setup NVAPI with your project.

What it does

`nvstereo.h` manages several important aspects for you.

- Manages user settings for your application
- Manages the StereoHandle for your application
- Updates an app-provided texture with per-eye constant data.

Although the specific data stored in the texture can be modified by the application, by default the information is stored as:

- Eye-specific separation stored in pixel(0, 0).r
- Covergence stored in pixel(0, 0).g
- Unit Vector identifying the current eye stored in pixel(0, 0).b
  - Left eye is -1
  - Right eye is 1

How to Use nvstereo.h

Detailed usage instructions for **`nvstereo.h`** are included in the header itself to ease integration. However, the general steps are:

1. Create appropriate **ParamTextureManager** prior to device creation.
2. Call **UpdateStereoTexture** once per frame, at the beginning of the frame. Call even while the device is lost.
3. Delete the **ParamTextureManager** after the device has been destroyed.

STEREOSCOPIC QA

As with any unique code path, 3D Vision Automatic does require a bit of attention during the QA process. Following these tips and steps will minimize the cost of stereoscopic QA while providing maximum coverage for your application.

TESTING SUGGESTIONS

NVIDIA suggests that developers use anaglyph mode as often as possible, both in daily development and for QA purposes. This will reduce the resource contention and specialty hardware requirements for your testing efforts without compromising on coverage.

3D Vision Discover mode (Red/Blue glasses and a default driver install) uses the same code paths as 3D Vision Automatic, except for the final step. In 3D Vision Automatic mode, the final step is to present the image to each eye, while in 3D Vision Discover mode the final step is to composite the two images into a single Red/Blue image pair.

3D Vision Discover Mode can be enabled by running the Stereoscopic Wizard from the Stereoscopic 3D section of the NVIDIA Control Panel.

USER SETTINGS

The two primary settings that users can adjust for a title are Separation and Convergence. Adjustments of these settings should be considered an important part of the testing matrix.

Separation

Separation can be adjusted using the hotkeys (**Ctrl-F3**, **Ctrl-F4** by default), or more easily using the wheel on the back of the IR emitter cube. Separation should be tested from the lowest setting to the highest.

In general, rendering should work for the entire gamut of separation values. In particular, fullscreen effects shouldn't work at low separation but fail at high separation.

Convergence

Convergence adjustment is considered an advanced feature, and must be enabled from the control panel. This can be done in the NVIDIA control panel, by selecting **Stereoscopic 3D** ➔**Set up stereoscopic 3D** ➔**Set Keyboard Shortcuts**. In the **Set Keyboard Shortcuts** dialog box, check **Enable advanced in-game settings**. The default keys to adjust convergence are **Ctrl-F5** and **Ctrl-F6**.

Adjusting convergence will move the convergence plane closer or further from the user. Most engines can tolerate some adjustment to this value, but objects will tend to separate somewhat severely at high values of convergence. This is normal and expected.

As with Separation, full screen and post-processing effects should continue to work regardless of the value used for convergence, even after objects have diverged beyond the brain’s ability to rectify the image.

Notice

HDMI

ROVI Compliance Statement

OpenCL

Trademarks

Copyright