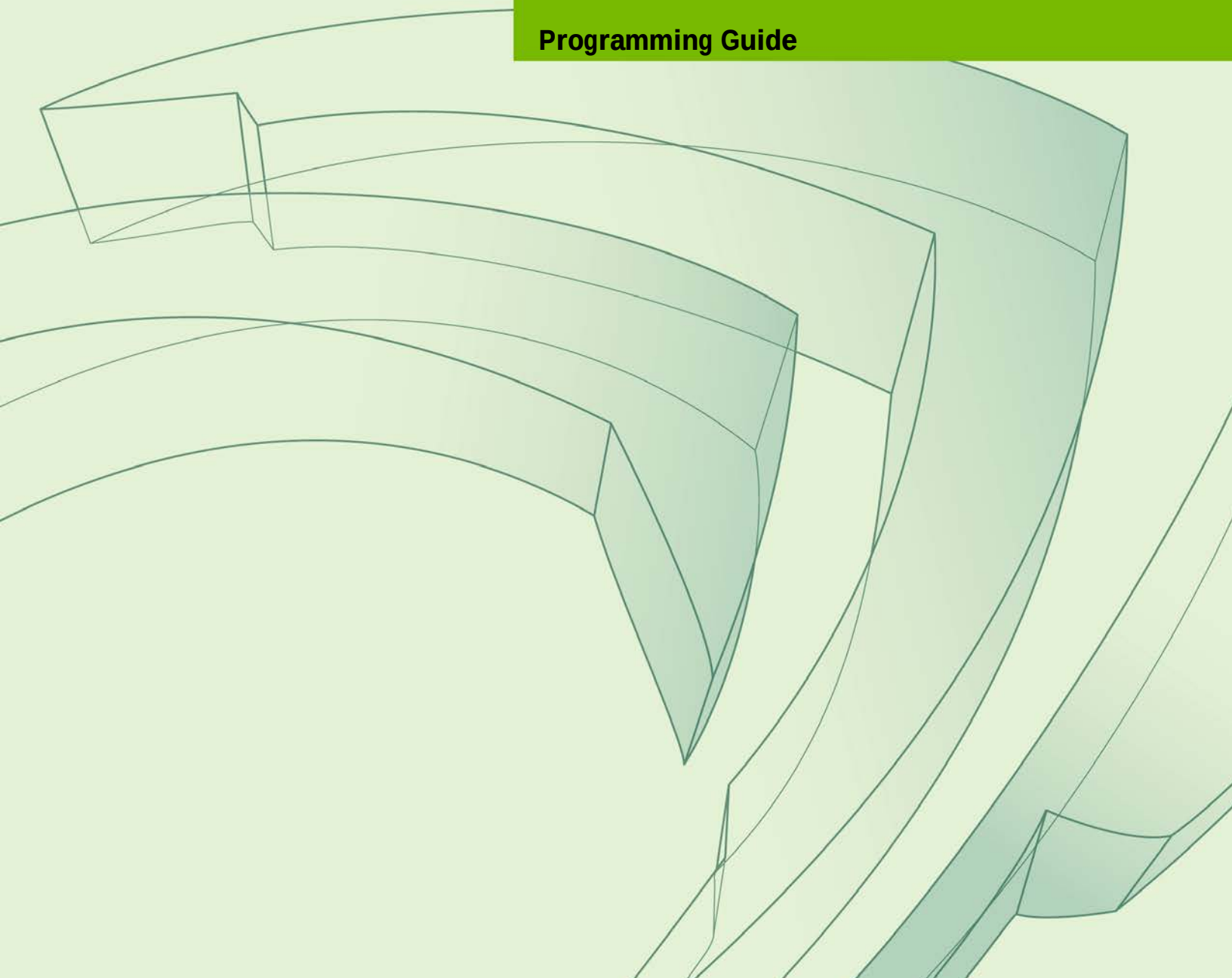
NVIDIA SINGLE PASS STEREO

PG-08038-001_v02 | November 2016

Programming Guide



DOCUMENT CHANGE HISTORY

PG-08038-001_v02

Version	Date	Authors	Description of Change
01	2016-05-19	GK, SB	Initial release
02	2016-11-07	SB	Added documentation for DirectX 12 support

TABLE OF CONTENTS

1 INTRODUCTION TO SINGLE PASS STEREO

The NVAPI programming interfaces for the Single Pass Stereo feature enable rendering of both the left and right eye views in a single draw call.

1.1 DIRECTX VERSIONS SUPPORTED

The Single Pass Stereo feature is supported on DirectX 11 and DirectX 12.

1.2 DRIVER VERSIONS SUPPORTED

The Single Pass Stereo feature for DirectX 11 is available starting with Release 367.

The Single Pass Stereo feature for DirectX 12 is available starting with Release 375.

2 DIRECTX 11 RENDERING WITH SINGLE PASS STEREO

Use the NVAPI programming interfaces for the Single Pass Stereo feature to enable rendering of both the left and right eye views in a single draw call as explained in the sections that follow.

2.1 QUERYING THE FEATURE CAPABILITY

To query the Single Pass Stereo feature capability, call the `NvAPI_D3D_QuerySinglePassStereoSupport` NVAPI.

```
NV_QUERY_SINGLE_PASS_STEREO_SUPPORT_PARAMS Stereo = {0};

Stereo.version = NV_QUERY_SINGLE_PASS_STEREO_SUPPORT_PARAMS_VER;

NvAPI_D3D_QuerySinglePassStereoSupport (pDevice, &Stereo);

// Stereo.bSinglePassStereoSupported will be TRUE if supported
```

2.2 CREATING AN HLSL SHADER PIPELINE BY USING NVAPI

You can create an HLSL shader pipeline by using NVAPI in one of the following ways:

- Using NVAPI functions to create all the world space shaders in the pipeline that are used in Single Pass Stereo rendering. For details, see “Creating All the World Space Shaders in the Pipeline” on page 3.
- Using NVAPI functions to create **only** the last world space shader in the pipeline. If you create the shader pipeline this way, you must set a flag when calling NVAPI functions as explained in “Creating Only the Last World Space Shader in the Pipeline” on page 7.

2.2.1 Defining the X Component for an HLSL Shader

An HLSL shader must compute the x component of the vertex position for both eye views in a single pass. Therefore, you must define the x component of the view position of each eye as follows:

- For the left eye, define the x component of the view position in the regular .x component of the `SV_POSITION` variable.
- For the right eye, define the x component of the view position in a custom semantic variable, for example, `NV_X_RIGHT`.

The string name of the custom semantic variable is programmable. To associate the Single Pass Stereo custom semantic with the x position of the right eye view, you must use the NVAPI create shader functions to create the shaders.

2.2.2 Creating All the World Space Shaders in the Pipeline

If you use NVAPI to create all the world shaders in pipeline, you must create each of the following world space shaders, which are used in Single Pass Stereo rendering:

- Vertex shader
- Hull shader
- Domain shader
- Geometry shader

2.2.2.1 Creating the Vertex World Space Shader

The following example creates a Vertex shader to compute the vertex positions for both the left eye view and the right-eye view.

```
cbuffer ConstantBuffer : register( b0 )
{
    matrix World;
    matrix View;
```

```
    matrix ViewRight;
    matrix Projection;
}

struct VSOutput
{
     float4 Pos : SV_POSITION;
     float4 Color : COLOR;
     float2 Tex : TEXCOORD0;

     float4 X_Right : NV_X_RIGHT; // Custom Semantic for Single Pass
Stereo
     uint4 ViewportMask : NV_VIEWPORT_MASK;
}

VSOutput VSMain (float4 Pos : POSITION,
                 float4 Color : COLOR,
                 float2 Tex : TEXCOORD0)
{
    VSOutput output = (VSOutput)0;
    float PosRight  = 0.0f;

    // Compute the vertex positions
    output.Pos = mul(mul(mul(Pos, World), View), Projection);
    PosRight   = mul(mul(mul(Pos, World), ViewRight), Projection);
    ...
    X_Right = PosRight.x; // Store in Custom Semantic for Single Pass
Stereo
    ...
    output.ViewportMask = 0x00000001;
}
```

Note: You must declare the custom semantic for Single Pass Stereo as `float4`.

Note: You must declare the custom semantic for Viewport Mask as `uint4`.

After the shader is compiled to Direct3D shader bytecode, use the `NvAPI_D3D11_CreateVertexShaderEx` NVAPI to create the shader on the Direct3D 11 device.

```
ID3D11VertexShader *pVertexShader = NULL;
NvAPI_D3D11_CREATE_VERTEX_SHADER_EX VSExArgs = {0};

// Create Vertex Shader with NVAPI
VSExArgs.version = NVAPI_D3D11_CREATEVERTEXSHADEREX_VERSION;
VSExArgs.NumCustomSemantics = 2;
```

```
VSExArgs.pCustomSemantics = (NV_CUSTOM_SEMANTIC*)malloc
    ((sizeof(NV_CUSTOM_SEMANTIC)*VSExArgs.NumCustomSemantics);
memset(VSExArgs.pCustomSemantics, 0,
    (sizeof(NV_CUSTOM_SEMANTIC))*VSExArgs.NumCustomSemantics);

VSExArgs.pCustomSemantics[0].version = NV_CUSTOM_SEMANTIC_VERSION;
VSExArgs.pCustomSemantics[0].NVCustomSemanticType =
    NV_X_RIGHT_SEMANTIC;
strcpy_s(&(VSExArgs.pCustomSemantics[0].NVCustomSemanti
    cNameString[0]), NVAPI_LONG_STRING_MAX, "NV_X_RIGHT");
VSExArgs.pCustomSemantics[1].version = NV_CUSTOM_SEMANTIC_VERSION;
VSExArgs.pCustomSemantics[1].NVCustomSemanticType =
   NV_VIEWPORT_MASK_SEMANTIC;
strcpy_s(&(VSExArgs.pCustomSemantics[1].NVCustomSemanticNameString[0]),
   NVAPI_LONG_STRING_MAX, "NV_VIEWPORT_MASK");


NvAPI_D3D11_CreateVertexShaderEx(pDevice, pBytecode, BytecodeSize,
                                 NULL, &VSExArgs, &pVertexShader);
...
free(VSExArgs.pCustomSemantics);
```

2.2.2.2 Creating the Hull World Space Shader

If the application uses a Hull shader during Single Pass Stereo rendering, create the shader by using NVAPI with the same usage of custom semantic for Single Pass Stereo.

After the Hull shader is compiled to Direct3D shader bytecode, use `NvAPI_D3D11_CreateHullShaderEx` NVAPI to create the shader on the Direct3D 11 device.

```
ID3D11HullShader *pHullShader = NULL;

NvAPI_D3D11_CREATE_HULL_SHADER_EX HSExArgs = {0};

// Create Hull Shader with NVAPI
HSExArgs.version = NVAPI_D3D11_CREATEHULLSHADEREX_VERSION;
HSExArgs.NumCustomSemantics = 2;
HSExArgs.pCustomSemantics = (NV_CUSTOM_SEMANTIC*) malloc
     ((sizeof(NV_CUSTOM_SEMANTIC))* HSExArgs.NumCustomSemantics);
memset(HSExArgs.pCustomSemantics, 0, (sizeof(NV_CUSTOM_SEMANTIC))*
     HSExArgs.NumCustomSemantics);

HSExArgs.pCustomSemantics[0].version = NV_CUSTOM_SEMANTIC_VERSION;
HSExArgs.pCustomSemantics[0].NVCustomSemanticType =
     NV_X_RIGHT_SEMANTIC;
strcpy_s(&(HSExArgs.pCustomSemantics[0].NVCustomSemanticNameString[0]),
     NVAPI_LONG_STRING_MAX, "NV_X_RIGHT");
HSExArgs.pCustomSemantics[1].version = NV_CUSTOM_SEMANTIC_VERSION;
HSExArgs.pCustomSemantics[1].NVCustomSemanticType =
```

```
   NV_VIEWPORT_MASK_SEMANTIC;
strcpy_s(&(HSExArgs.pCustomSemantics[1].NVCustomSemanticNameString[0]),
   NVAPI_LONG_STRING_MAX, "NV_VIEWPORT_MASK");

NvAPI_D3D11_CreateHullShaderEx(pDevice, pBytecode, BytecodeSize,
     NULL, &HSExArgs, &pHullShader);
...
free(HSExArgs.pCustomSemantics);
```

2.2.2.3 Creating the Domain World Space Shader

If the application uses a Domain shader during Single Pass Stereo rendering, create the shader by using NVAPI with the same usage of custom semantic for Single Pass Stereo.

After the Domain shader is compiled to Direct3D shader bytecode, use the `NvAPI_D3D11_CreateDomainShaderEx` NVAPI to create the shader on the Direct3D 11 device.

```
ID3D11DomainShader *pDomainShader = NULL;
NvAPI_D3D11_CREATE_DOMAIN_SHADER_EX DSExArgs = {0};

// create domain shader with NVAPI
DSExArgs.version = NVAPI_D3D11_CREATEDOMAINSHADEREX_VERSION;
DSExArgs.NumCustomSemantics = 1;
DSExArgs.pCustomSemantics = (NV_CUSTOM_SEMANTIC*) malloc
     ((sizeof(NV_CUSTOM_SEMANTIC))* DSExArgs.NumCustomSemantics);
memset(DSExArgs.pCustomSemantics, 0, (sizeof(NV_CUSTOM_SEMANTIC))*
     DSExArgs.NumCustomSemantics);

DSExArgs.pCustomSemantics[0].version = NV_CUSTOM_SEMANTIC_VERSION;
DSExArgs.pCustomSemantics[0].NVCustomSemanticType =
     NV_X_RIGHT_SEMANTIC;
strcpy_s(&(DSExArgs.pCustomSemantics[0].NVCustomSemanticNameString[0]),
     NVAPI_LONG_STRING_MAX, "NV_X_RIGHT");
DSExArgs.pCustomSemantics[1].version = NV_CUSTOM_SEMANTIC_VERSION;
DSExArgs.pCustomSemantics[1].NVCustomSemanticType =
   NV_VIEWPORT_MASK_SEMANTIC;
strcpy_s(&(DSExArgs.pCustomSemantics[1].NVCustomSemanticNameString[0]),
   NVAPI_LONG_STRING_MAX, "NV_VIEWPORT_MASK");

NvAPI_D3D11_CreateDomainShaderEx(pDevice, pBytecode, BytecodeSize,
     NULL, &DSExArgs, &pDomainShader);

...
free(DSExArgs.pCustomSemantics);
```

2.2.2.4 Creating the Geometry World Space Shader

If the application uses a Geometry shader during Single Pass Stereo rendering, create the shader by using NVAPI with the same usage of custom semantic for Single Pass Stereo.

After the shader is compiled to Direct3D shader bytecode, use the `NvAPI_D3D11_CreateGeometryShaderEx_2` NVAPI to create the shader on the Direct3D 11 device.

```
ID3D11GeometryShader *pGeometryShader = NULL;
NvAPI_D3D11_CREATE_GEOMETRY_SHADER_EX GSExArgs = {0};

// create geometry shader with NVAPI
GSExArgs.version = NVAPI_D3D11_CREATEGEOMETRYSHADEREX_2_VERSION;
GSExArgs.NumCustomSemantics = 1;
GSExArgs.pCustomSemantics = (NV_CUSTOM_SEMANTIC*) malloc
  ((sizeof(NV_CUSTOM_SEMANTIC))*GSExArgs.NumCustomSemantics);
memset(GSExArgs.pCustomSemantics, 0,
  (sizeof(NV_CUSTOM_SEMANTIC))*GSExArgs.NumCustomSemantics);

GSExArgs.pCustomSemantics[0].version = NV_CUSTOM_SEMANTIC_VERSION;
GSExArgs.pCustomSemantics[0].NVCustomSemanticType =
   NV_X_RIGHT_SEMANTIC;
strcpy_s(&(GSExArgs.pCustomSemantics[0].NVCustomSemanticNameString[0]),
   NVAPI_LONG_STRING_MAX, "NV_X_RIGHT");
GSExArgs.pCustomSemantics[1].version = NV_CUSTOM_SEMANTIC_VERSION;
GSExArgs.pCustomSemantics[1].NVCustomSemanticType =
   NV_VIEWPORT_MASK_SEMANTIC;
strcpy_s(&(GSExArgs.pCustomSemantics[1].NVCustomSemanticNameString[0]),
   NVAPI_LONG_STRING_MAX, "NV_VIEWPORT_MASK");

GSExArgs.UseViewportMask = false;

NvAPI_D3D11_CreateGeometryShaderEx_2(pDevice, pBytecode, BytecodeSize,
                                     NULL, &GSExArgs, &pGeometryShader);
...
free(GSExArgs.pCustomSemantics);
```

2.2.3 Creating Only the Last World Space Shader in the Pipeline

You can use NVAPI to create **only** the last world space shader in the Single Pass Stereo rendering pipeline. In this situation, you must use the regular DirectX API to create the remaining shaders in the pipeline:

- To create the Vertex shader, use `CreateVertexShader`.
- To create the Pixel shader, use `CreatePixelShader`.

To create the last world space shader in the pipeline, use the appropriate NVAPI for the type of world space shader as shown in the following table.

Last World Space Shader	NVAPI
Vertex	NvAPI_D3D11_CreateVertexShaderEx
Domain	NvAPI_D3D11_CreateDomainShaderEx
Geometry	NvAPI_D3D11_CreateGeometryShaderEx_2

For example, the Single Pass Stereo rendering pipeline might contain a Vertex shader, Geometry shader, and Pixel shader in that order. In this situation, create only the Geometry shader by using the NVAPI.

When you use the NVAPI to create a last world space shader, set `UseSpecificShaderExt` to `TRUE`.

The following example uses the NVAPI to create a Geometry last world shader.

```
ID3D11GeometryShader *pGeometryShader = NULL;
NvAPI_D3D11_CREATE_GEOMETRY_SHADER_EX GSExArgs = {0};

// create geometry shader with NVAPI
GSExArgs.version = NVAPI_D3D11_CREATEGEOMETRYSHADEREX_2_VERSION;
GSExArgs.NumCustomSemantics = 2;
GSExArgs.pCustomSemantics = (NV_CUSTOM_SEMANTIC*) malloc
  ((sizeof(NV_CUSTOM_SEMANTIC))*GSExArgs.NumCustomSemantics);
memset(GSExArgs.pCustomSemantics, 0,
  (sizeof(NV_CUSTOM_SEMANTIC))*GSExArgs.NumCustomSemantics);
GSExArgs.pCustomSemantics[0].version = NV_CUSTOM_SEMANTIC_VERSION;
GSExArgs.pCustomSemantics[0].NVCustomSemanticType =
   NV_X_RIGHT_SEMANTIC;
strcpy_s(&(GSExArgs.pCustomSemantics[0].NVCustomSemanticNameString[0]),
   NVAPI_LONG_STRING_MAX, "NV_X_RIGHT");
GSExArgs.pCustomSemantics[1].version = NV_CUSTOM_SEMANTIC_VERSION;
GSExArgs.pCustomSemantics[1].NVCustomSemanticType =
   NV_VIEWPORT_MASK_SEMANTIC;
strcpy_s(&(GSExArgs.pCustomSemantics[1].NVCustomSemanticNameString[0]),
   NVAPI_LONG_STRING_MAX, "NV_VIEWPORT_MASK");

GSExArgs.UseViewportMask = false;
...
GSExArgs.UseSpecificShaderExt = true;

NvAPI_D3D11_CreateGeometryShaderEx_2(pDevice, pBytecode, BytecodeSize,
                                     NULL, &GSExArgs, &pGeometryShader)
...
free(GSExArgs.pCustomSemantics);
```

2.2.4 Using Single Pass Stereo with FastGS

Fast Geometry (FastGS) shader enables you to write Geometry shaders with certain restrictions to make them run more efficiently on the GPU.

Note: FastGS is not supported with the optimization disabled (/Od) HLSL compilation option.

You can use FastGS with the Single Pass Stereo feature when creating all the world space shaders with NVAPI or when creating only some of the world space shaders with NVAPI.

2.2.4.1 Using FastGS when Creating All the World Space Shaders with NVAPI

The following example HLSL code shows how to use Single Pass Stereo with FastGS.

```
struct VSOutput
{
     float4 Pos : SV_POSITION;
     float4 Color : COLOR;
     float2 Tex : TEXCOORD0;

     float4 X_Right : NV_X_RIGHT; // Single Pass Stereo
}

struct GSOutput
{
    VSOutput Passthrough;

    uint4 ViewportMask : NV_VIEWPORT_MASK;
};

VSOutput VSMain (float4 position : POSITION,
                 float4 color : COLOR,
                 float2 Tex : TEXCOORD0)
{
     VSOutput OutVtx;

     OutVtx.Pos = position;
     ...
     ...
     OutVtx.X_Right = (...); // right eye X value computation
     ...
     ...
     return OutVtx;
}

[maxvertexcount(1)]
void FastGSMain(triangle VSOutput In[3],
```

```
                inout TriangleStream<GSOutput>
                 TriStream)
{
    GSOutput output;

    output.Passthrough = In[0];
    output.ViewportMask= 0x00000001;

    TriStream.Append(output);
}
```

To create the Geometry shader, use the `NvAPI_D3D11_CreateGeometryShaderEx_2` NVAPI with the fast GS option as shown in the following example.

```
ID3D11GeometryShader *pGeometryShader = NULL;

// create geometry shader with NVAPI
NvAPI_D3D11_CREATE_GEOMETRY_SHADER_EX GSExArgs = {0};
...
// specify custom semantics
...
GSExArgs.ForceFastGS = true;
...
GSExArgs.UseViewportMask = false;

NvAPI_D3D11_CreateGeometryShaderEx_2(pDevice, pBytecode, BytecodeSize,
                                     NULL, &GSExArgs, &pGeometryShader)
```

2.2.4.2 Using FastGS when Creating Only Some of the World Space Shaders with NVAPI

You might want to use FastGS when creating only some of the world space shaders with NVAPI. In this situation, you cannot create only the last world space shader in the pipeline by using NVAPI. Instead, you must use NVAPI to create both of the following world space shaders:

- The FastGS Geometry shader
- The world space shader immediately before the FastGS Geometry shader in the Single Pass Stereo rendering pipeline

You can then use the regular DirectX API to create the remaining world space shaders in the pipeline.

For example:

- The Single Pass Stereo rendering pipeline might contain the Vertex shader and the FastGS Geometry shader in that order.

  In this situation, create the Vertex shader **and** the FastGS Geometry shader by using the NVAPI for creating the last world space shader.

- The Single Pass Stereo rendering pipeline might contain the Vertex shader, Hull shader, Domain shader, and the FastGS Geometry shader in that order.

  In this situation, create **only** the Domain shader and the FastGS Geometry shader by using the NVAPI for creating the last world space shader.

  Use the regular DirectX API to create the Vertex shader and Hull Shader.

For information about how to create a shader by using the NVAPI for creating the last world space shader, see "Creating Only the Last World Space Shader in the Pipeline" on page 7.

2.3 GENERATING THE LEFT-EYE AND RIGHT-EYE VIEWS

Generate the left-eye view and right view with Single Pass Stereo by using one of the following methods:

- **Generating both views on a render target array.** In this method, the view for each eye occupies one element in the render target array. The left-eye view occupies the first element in the array and the right-eye view occupies the second element in the array.
- **Generating side-by-side views on a single render target.** In this method, the two views are generated side-by-side directly on a single render target.

2.3.1 Generating Both Views on a Render Target Array

The following example creates a render target array in which the right-eye view is generated on the second element of the array.

```
ID3D11RenderTargetView* pRenderTargetView      = NULL;
ID3D11Texture2D*        pTempRenderTargets     = NULL;
D3D11_VIEWPORT          Viewports[1];

...
```

```
D3D11_TEXTURE2D_DESC tempRTDesc;
ZeroMemory(&tempRTDesc, sizeof(D3D11_TEXTURE2D_DESC));

tempRTDesc.Width              = ...; // total screen width / 2
tempRTDesc.Height             = ...; // total screen height
tempRTDesc.MipLevels          = 1;
tempRTDesc.ArraySize          = 2;
tempRTDesc.Format             = DXGI_FORMAT_R8G8B8A8_UNORM;
tempRTDesc.SampleDesc.Count   = 1;
tempRTDesc.Usage              = D3D11_USAGE_DEFAULT;
tempRTDesc.BindFlags          = D3D11_BIND_RENDER_TARGET;
tempRTDesc.CPUAccessFlags     = 0;
tempRTDesc.MiscFlags          = 0;

...

pDevice->CreateTexture2D(&tempRTDesc, NULL, &pTempRenderTargets);

D3D11_RENDER_TARGET_VIEW_DESC RTVDesc;
ZeroMemory(&RTVDesc, sizeof(D3D11_RENDER_TARGET_VIEW_DESC));
RTVDesc.Format         = tempRTDesc.Format;
RTVDesc.ViewDimension  = D3D11_RTV_DIMENSION_TEXTURE2DARRAY;
RTVDesc.Texture2DArray.ArraySize         = 2;
RTVDesc.Texture2DArray.MipSlice          = 0;
RTVDesc.Texture2DArray.FirstArraySlice   = 0;

pDevice->CreateRenderTargetView(pTempRenderTargets, &RTVDesc,
     &pRenderTargetView);

...

pDeviceContext->OMSetRenderTargets(1, &pRenderTargetView, ..);

...

ZeroMemory(&Viewports, sizeof(D3D11_VIEWPORT)*_countof(Viewports));
Viewports[0].Width     = ...; // total screen width / 2
Viewports[0].Height    = ...; // total screen height
Viewports[0].MinDepth  = 0.0f;
Viewports[0].MaxDepth  = 1.0f;
Viewports[0].TopLeftX  = 0;
Viewports[0].TopLeftY  = 0;

pDeviceContext->RSSetViewports(1, Viewports);
...
NvAPI_D3D_SetSinglePassStereoMode (pDeviceOrContext, 2, 1, false);
```

If you generate a render target array, you must specify the viewport mask in the HLSL shader as 0x00000001 in a custom semantic variable, for example, NV_VIEWPORT_MASK.

The following example sets the viewport mask to 0x00000001 for a Geometry shader.

```
struct GSOutput
{
     float4 Pos : SV_POSITION;
     float4 Color : COLOR;
     float2 Tex : TEXCOORD0;

     float4 X_Right : NV_X_RIGHT; // Single Pass Stereo
     uint4 ViewportMask : NV_VIEWPORT_MASK;
}

[maxvertexcount(3)]
void GSMain(triangle VSOutput Input[3],
            inout TriangleStream<GSOutput> TriStream)
{
     GSOutput OutVtx;

     for (int v = 0; v < 3; ++v)
     {
         OutVtx.Pos    = Input[v].Pos;
         OutVtx.Color  = Input[v].Color;
         OutVtx.Tex    = Input[v].Tex;
         OutVtx.X_Right = Input[v].X_Right;

         OutVtx.ViewportMask = 0x00000001;
         ...
         TriStream.Append(OutVtx);
     }
}
```

2.3.2 Generating Side-by-Side Views on a Single Render Target

If you generate side-by-side views on a single render target, you must create one viewport for each eye view.

The following example creates a single render target with two viewports.

```
ID3D11Texture2D*        pSingleRenderTarget          = NULL;
ID3D11RenderTargetView* pSingleRenderTargetView      = NULL;
D3D11_VIEWPORT          Viewports[2];

D3D11_TEXTURE2D_DESC singleRTDesc;
ZeroMemory(&singleRTDesc, sizeof(D3D11_TEXTURE2D_DESC));
pBackBuffer->GetDesc(&singleRTDesc);
pDevice->CreateTexture2D(&singleRTDesc, NULL, &pSingleRenderTarget);

...

pDevice->CreateRenderTargetView(pSingleRenderTarget, NULL,
           &pSingleRenderTargetView);
```

```
...

pDeviceContext->OMSetRenderTargets(1, &pSingleRenderTargetView, ..);

...

ZeroMemory(&Viewports, sizeof(D3D11_VIEWPORT)*_countof(Viewports));
Viewports[0].Width     = ...; // total screen width / 2
Viewports[0].Height    = ...; // total screen height
Viewports[0].MinDepth  = 0.0f;
Viewports[0].MaxDepth  = 1.0f;
Viewports[0].TopLeftX  = 0;
Viewports[0].TopLeftY  = 0;

Viewports[1].Width     = ...; // total screen width / 2
Viewports[1].Height    = ...; // total screen height
Viewports[1].MinDepth  = 0.0f;
Viewports[1].MaxDepth  = 1.0f;
Viewports[1].TopLeftX  = ...; // total screen width / 2
Viewports[1].TopLeftY  = 0;

pDeviceContext->RSSetViewports(2, Viewports);

...
...

NvAPI_D3D_SetSinglePassStereoMode (pDeviceOrContext, 2, 0, true);
```

If you generate a single target array, you must specify the viewport mask in the HLSL shader as 0x00020001 in a custom semantic variable, for example, `NV_VIEWPORT_MASK`.

This value indicates an independent viewport mask for the left eye and the right eye. The upper 16 bits are for right eye viewport. The second bit is set to indicate that the right eye view uses the second viewport.

The following example sets the viewport mask to 0x00020001 for a Geometry shader.

```
struct GSOutput
{
     float4 Pos : SV_POSITION;
     float4 Color : COLOR;
     float2 Tex : TEXCOORD0;

     float4 X_Right : NV_X_RIGHT; // Single Pass Stereo
     uint4 ViewportMask : NV_VIEWPORT_MASK;
}

[maxvertexcount(3)]
void GSMain(triangle VSOutput Input[3],
```

```
            inout TriangleStream<GSOutput> TriStream)
{
     GSOutput OutVtx;

     for (int v = 0; v < 3; ++v)
     {
         OutVtx.Pos    = Input[v].Pos;
         OutVtx.Color  = Input[v].Color;
         OutVtx.Tex    = Input[v].Tex;
         OutVtx.X_Right = Input[v].X_Right;
         // independent viewport mask for left eye and right eye
         // upper 16 bits for right eye viewport using Single Pass
Stereo
         // 2nd bit set indicates right eye view will use 2nd viewport
         OutVtx.ViewportMask = 0x00020001;
         ...
         ...

         TriStream.Append(OutVtx);
     }
}
```

2.3.3 Detecting the Eye for a View

When you generate the left-eye view and right-eye view, the application must be able to detect which view is being rendered.

To detect the eye for which a view is being rendered, use a system semantic in the Pixel shader. The semantic to use depends on how the view is generated.

Note: Using a system semantic in the Pixel shader to detect the eye for a view is **not** supported for use with the debug D3D runtime.

2.3.3.1 Detecting the Eye for a View Generated on a Render Target Array

If both views are generated on a render target array, use the system semantic `SV_RenderTargetArrayIndex` in the Pixel shader to detect which view is being rendered.

- For first view, the value is 0.
- For the second view, the value is 1.

```
PSMain (..., uint RTIndex : SV_RenderTargetArrayIndex)
{
..
}
```

2.3.3.2 Detecting the Eye for a View Generated on a Single Render Target

If side-by-side views are generated on a single render target, use the system semantic `SV_ViewportArrayIndex` in the Pixel shader to detect which view is being rendered.

- For first view, the value is 0.
- For the second view, the value is 1.

```
PSMain (..., uint ViewportIndex : SV_ViewportArrayIndex)
{
..
}
```

Note: The debug D3D runtime reports a linkage warning or error when the Pixel shader reads `SV_ViewportArrayIndex`. As a workaround **for debugging purposes only**, it may be output from the NVAPI-created Geometry shader with a dummy value, for example, 0. But it will be ignored by the NVIDIA display driver.

2.4 ENABLING AND DISABLING THE SINGLE PASS STEREO MODE OF THE GPU

2.4.1 Enabling the Single Pass Stereo Mode of the GPU

If your application uses the Single Pass Stereo feature, you must enable the Single Pass Stereo mode of the GPU.

To enable the Single Pass Stereo mode of the GPU, use the `NvAPI_D3D_SetSinglePassStereoMode` NVAPI as in the following example.

```
NvAPI_D3D_SetSinglePassStereoMode (pDeviceOrContext,
     2, // numViews (for two eyes)
     1, // Offset between render targets of the different views
     false); // Is the independent viewport mask enabled
```

The values of the parameters to pass to this NVAPI depend on how the left-eye view and right-eye view are generated:

- For two views generated on a render target array, set the offset to 1.
- For side-by-side views generated on a single render target, set the offset to 0 and the independent viewport mask to `true`.

For information about how to generate the left-eye view and right-eye view, see "Generating the Left-Eye and Right-Eye Views" on page 11.

2.4.2 Disabling the Single Pass Stereo Mode of the GPU

To disable the Single Pass Stereo mode of the GPU, use the `NvAPI_D3D_SetSinglePassStereoMode` NVAPI as in the following example.

```
NvAPI_D3D_SetSinglePassStereoMode (pDeviceOrContext, 1, 1, false);
```

3 DIRECTX 11 RESTRICTIONS

3.1 VIEWPORT ARRAY INDEX ALTERNATIVE

When custom semantics are used, do **not** use the viewport array index standard semantic `SV_ViewportArrayIndex` for the world space shaders. Instead, use the custom semantic for viewport mask in these shaders.

The world space shaders are as follows:

- Vertex shader
- Hull shader
- Domain shader
- Geometry shader

When you use the custom semantic for viewport mask, you must set the appropriate bits in the mask that correspond to the viewports that you want to make active. For example, to make the viewport at index *N* active, write the following code:

```
ViewportMask = (0x1 << N)
```

Do **not** write `ViewportIndex = N`.

For an exception to this requirement, see “Detecting the Eye for a View” on page 15.

3.2 MAXIMUM NUMBER OF VIEWPORTS

The maximum number of viewports supported is 16.

The number of viewports that a view uses and, therefore, the number of viewports supported per eye, depend on how the left-eye view and right-eye view are generated:

- If both views are generated on a render target array, the same viewport is used for both eyes. Therefore, the maximum number of viewports supported per eye is 16.
- If side-by-side views are generated on a single render target, independent viewports are used for each eye. Therefore, the maximum number of viewports supported per eye is 8.

If you need more than 8 viewports per eye (for example, 3×3=9 viewports) you must use a render target array for generating the left-eye view and right-eye view.

4 DIRECTX 11 API REFERENCE

4.1 STRUCTURES

4.1.1 NV_QUERY_SINGLE_PASS_STEREO_SUPPORT_PARAMS

```
typedef struct _NV_QUERY_SINGLE_PASS_STEREO_SUPPORT_PARAMS
{
    NvU32 version;
    NvU32 bSinglePassStereoSupported;
} NV_QUERY_SINGLE_PASS_STEREO_SUPPORT_PARAMS;
```

4.1.1.1 Members

`version`

Type: `NvU32`

The version of the NV_QUERY_SINGLE_PASS_STEREO_SUPPORT_PARAMS structure

`bSinglePassStereoSupported`

Type: `NvU32`

Indicates whether Single Pass Stereo is supported on the current setup

4.1.1.2 Remarks

The `NV_QUERY_SINGLE_PASS_STEREO_SUPPORT_PARAMS` structure provides information about the Single Pass Stereo capability on the current setup. This structure is used in the `NvAPI_D3D_QuerySinglePassStereoSupport()` function.

4.1.2 NV_CUSTOM_SEMANTIC

```
typedef struct _NV_CUSTOM_SEMANTIC
{
    UINT                        version;
    NV_CUSTOM_SEMANTIC_TYPE     NVCustomSemanticType;
    NvAPI_LongString            NVCustomSemanticNameString;
    BOOL                        RegisterSpecified;
    NvU32                       RegisterNum;
    NvU32                       RegisterMask;
    NvU32                       Reserved;
} NV_CUSTOM_SEMANTIC;
```

4.1.2.1 Members

`version`

Type: `UINT`

The version of the `NV_CUSTOM_SEMANTIC` structure

`NVCustomSemanticType`

Type: `NV_CUSTOM_SEMANTIC_TYPE`

`NV_X_RIGHT_SEMANTIC` for specifying the X position value for the right eye

`NV_VIEWPORT_MASK_SEMANTIC` for specifying the viewport mask

`NVCustomSemanticNameString`

Type: `NvAPI_LongString`

The name of the custom semantic

`RegisterSpecified`

Type: `NvU32`

Reserved

`RegisterNum`

Type: `NvU32`

Reserved

`RegisterMask`

Type: `NvU32`

Reserved

`Reserved`

Type: `NvU32`

Reserved

4.1.2.2 Remarks

The `NV_CUSTOM_SEMANTIC` structure is used to specify custom semantics. It is used in the following functions:

- `NvAPI_D3D11_CreateVertexShaderEx()`
- `NvAPI_D3D11_CreateHullShaderEx()`
- `NvAPI_D3D11_CreateDomainShaderEx()`
- `NvAPI_D3D11_CreateGeometryShaderEx_2()`

4.1.3 NvAPI_D3D11_CREATE_VERTEX_SHADER_EX

```
typedef struct NvAPI_D3D11_CREATE_VERTEX_SHADER_EX
{
    UINT                version;
    NvU32               NumCustomSemantics;
    NV_CUSTOM_SEMANTIC *pCustomSemantics;
    BOOL                UseWithFastGS;
    BOOL                UseSpecificShaderExt;
} NvAPI_D3D11_CREATE_VERTEX_SHADER_EX;
```

4.1.3.1 Members

`version`

Type: `UINT`

The version of the `NvAPI_D3D11_CREATE_VERTEX_SHADER_EX` structure

`NumCustomSemantics`

Type: `NvU32`

The number of custom semantic elements, up to `NV_CUSTOM_SEMANTIC_MAX`, provided in the array pointer `pCustomSemantics`

`pCustomSemantics`

Type: `NV_CUSTOM_SEMANTIC`

Used to specify custom semantics for this shader

`UseWithFastGS`

Type: `BOOL`

Reserved

`UseSpecificShaderExt`

Type: `BOOL`

`TRUE` if minimal specific shaders are being created with NVAPI shader extensions as explained in "Creating Only the Last World Space Shader in the Pipeline" on page 7.

4.1.3.2 Remarks

This structure is used to specify parameters for a custom Vertex shader. This structure is used in the `NvAPI_D3D11_CreateVertexShaderEx()` function.

4.1.4 NvAPI_D3D11_CREATE_HULL_SHADER_EX

```
typedef struct NvAPI_D3D11_CREATE_HULL_SHADER_EX
{
    UINT               version;
    NvU32              NumCustomSemantics;
    NV_CUSTOM_SEMANTIC *pCustomSemantics;
    BOOL               UseWithFastGS;
    BOOL               UseSpecificShaderExt;
} NvAPI_D3D11_CREATE_HULL_SHADER_EX;
```

4.1.4.1 Members

`version`

Type: `UINT`

The version of the `NvAPI_D3D11_CREATE_HULL_SHADER_EX` structure

`NumCustomSemantics`

Type: `NvU32`

The number of custom semantic elements, up to `NV_CUSTOM_SEMANTIC_MAX`, provided in the array pointer `pCustomSemantics`

`pCustomSemantics`

Type: `NV_CUSTOM_SEMANTIC`

Used to specify custom semantics for this shader

`UseWithFastGS`

Type: `BOOL`

Reserved

`UseSpecificShaderExt`

Type: `BOOL`

`TRUE` if minimal specific shaders are being created with NVAPI shader extensions as explained in "Creating Only the Last World Space Shader in the Pipeline" on page 7.

4.1.4.2 Remarks

This structure is used to specify parameters for a custom Hull shader. This structure is used in the `NvAPI_D3D11_CreateHullShaderEx()` function.

4.1.5 NvAPI_D3D11_CREATE_DOMAIN_SHADER_EX

```
typedef struct NvAPI_D3D11_CREATE_DOMAIN_SHADER_EX
{
    UINT               version;
    NvU32              NumCustomSemantics;
    NV_CUSTOM_SEMANTIC *pCustomSemantics;
    BOOL               UseWithFastGS;
    BOOL               UseSpecificShaderExt;
} NvAPI_D3D11_CREATE_DOMAIN_SHADER_EX;
```

4.1.5.1 Members

`version`

Type: `UINT`

The version of the `NvAPI_D3D11_CREATE_DOMAIN_SHADER_EX` structure

`NumCustomSemantics`

Type: `NvU32`

The number of custom semantic elements, up to `NV_CUSTOM_SEMANTIC_MAX`, provided in the array pointer `pCustomSemantics`

`pCustomSemantics`

Type: `NV_CUSTOM_SEMANTIC`

Used to specify custom semantics for this shader

`UseWithFastGS`

Type: `BOOL`

Reserved

`UseSpecificShaderExt`

Type: `BOOL`

`TRUE` if minimal specific shaders are being created with NVAPI shader extensions as explained in "Creating Only the Last World Space Shader in the Pipeline" on page 7.

4.1.5.2 Remarks

This structure is used to specify parameters for a custom Domain shader. This structure is used in the `NvAPI_D3D11_CreateDomainShaderEx()` function.

4.1.6 NvAPI_D3D11_CREATE_GEOMETRY_SHADER_EX

```
typedef struct NvAPI_D3D11_CREATE_GEOMETRY_SHADER_EX
{
    UINT                      version;

    BOOL                      UseViewportMask;
    BOOL                      OffsetRtIndexByVpIndex;
    BOOL                      ForceFastGS;
    BOOL                      DontUseViewportOrder;
    BOOL                      UseAttributeSkipMask;
    BOOL                      UseCoordinateSwizzle;
    NvAPI_D3D11_SWIZZLE_MODE *pCoordinateSwizzling;

    NvU32                     NumCustomSemantics;
    NV_CUSTOM_SEMANTIC       *pCustomSemantics;
    BOOL                      ConvertToFastGS;
    BOOL                      UseSpecificShaderExt;
} NvAPI_D3D11_CREATE_GEOMETRY_SHADER_EX;
```

4.1.6.1 Members

`version`

Type: `UINT`

The version of the `NvAPI_D3D11_CREATE_GEOMETRY_SHADER_EX` structure

`UseViewportMask`

Type: `BOOL`

FALSE for custom semantics shaders

`OffsetRtIndexByVpIndex`

Type: BOOL

FALSE for custom semantics shaders

`ForceFastGS`

Type: BOOL

TRUE if Fast GS is to be used

If TRUE, the Geometry shader must be written with `maxvertexcount(1)` and must pass through input vertex 0 to the output without modification.

`DontUseViewportOrder`

Type: BOOL

FALSE by default

Use TRUE only for API ordering, which slows performance.

`UseAttributeSkipMask`

Type: BOOL

Reserved

`UseCoordinateSwizzle`

Type: BOOL

Reserved

`pCoordinateSwizzling`

Type: NvAPI_D3D11_SWIZZLE_MODE

Reserved

`NumCustomSemantics`

Type: NvU32

The number of custom semantic elements, up to NV_CUSTOM_SEMANTIC_MAX, provided in the array pointer pCustomSemantics

`pCustomSemantics`

Type: NV_CUSTOM_SEMANTIC

Used to specify custom semantics for this shader

`ConvertToFastGS`

Type: `BOOL`

Reserved

`UseSpecificShaderExt`

Type: `BOOL`

`TRUE` if minimal specific shaders are being created with NVAPI shader extensions as explained in "Creating Only the Last World Space Shader in the Pipeline" on page 7.

4.1.6.2 Remarks

This structure is used to specify parameters for a custom Geometry shader. This structure is used in the `NvAPI_D3D11_CreateGeometryShaderEx_2()` function.

4.2 FUNCTIONS

4.2.1 NvAPI_D3D_QuerySinglePassStereoSupport

```
NVAPI_INTERFACE NvAPI_D3D_QuerySinglePassStereoSupport(
  [in]     IUnknown *pDevice,
  [inout]  NV_QUERY_SINGLE_PASS_STEREO_SUPPORT_PARAMS
           *pQuerySinglePassStereoSupportedParams
);
```

4.2.1.1 Parameters

`pDevice [in]`

Type: `IUnknown*`

Pointer to the D3D11 device `ID3D11Device*`, which inherits `IUnknown*`

`pQuerySinglePassStereoSupportedParams [inout]`

Type: `NV_QUERY_SINGLE_PASS_STEREO_SUPPORT_PARAMS*`

Pointer to the `NV_QUERY_SINGLE_PASS_STEREO_SUPPORT_PARAMS` structure

4.2.1.2 Return Value

Returns `NVAPI_OK` on success.

`bSinglePassStereoSupported` becomes `TRUE` if Single Pass Stereo is supported.

4.2.1.3 Remarks

This function determines whether the hardware supports the Single Pass Stereo feature.

4.2.2 NvAPI_D3D_SetSinglePassStereoMode

```
NVAPI_INTERFACE NvAPI_D3D_SetSinglePassStereoMode(
     [in]  IUnknown *pDevOrContext,
     [in]  NvU32 numViews,
     [in]  NvU32 renderTargetIndexOffset,
     [in]  NvU8 independentViewportMaskEnable
);
```

4.2.2.1 Parameters

`pDevOrContext [in]`

Type: `IUnknown *`

Pointer to the D3D11 device `ID3D11Device*` or `ID3D11DeviceContext*`, which inherits `IUnknown*`

`numViews [in]`

Type: `NvU32`

The number of views to render (2 for two eyes)

`renderTargetIndexOffset [in]`

Type: `NvU32`

The offset between the render targets of the different views:

- 1 if both views are generated on a render target array
- 0 if side-by-side views are generated on single render target

`independentViewportMaskEnable [in]`

Type: `NvU8`

Indicates whether the independent viewport mask is enabled:

- `false` if both views are generated on a render target array
- `true` if side-by-side views are generated on single render target

4.2.2.2 Return Value

Returns `NVAPI_OK` if all arguments are valid and initiates setting the Single Pass Stereo Mode.

4.2.2.3 Remarks

This function sets the mode of the Single Pass Stereo feature.

Note that this is an asynchronous function and returns `NVAPI_OK` if all arguments are valid. Returned value `NVAPI_OK` does not reflect that Single Pass Stereo is supported or is set in hardware. One must call `NvAPI_D3D_QuerySinglePassStereoSupport()` to confirm that the current setup supports Single Pass Stereo before calling this set-function.

4.2.3 NvAPI_D3D11_CreateVertexShaderEx

```
NVAPI_INTERFACE NvAPI_D3D11_CreateVertexShaderEx(
     [in]              ID3D11Device *pDevice,
     [in]              const void *pShaderBytecode,
     [in]              SIZE_T BytecodeLength,
     [in, optional]    ID3D11ClassLinkage *pClassLinkage,
     [in]              const NvAPI_D3D11_CREATE_VERTEX_SHADER_EX
                       *pCreateVertexShaderExArgs,
     [out]             ID3D11VertexShader **ppVertexShader
);
```

4.2.3.1 Parameters

`pDevice [in]`

Type: `ID3D11Device *`

Pointer to the D3D11 device `ID3D11Device*`

`pShaderBytecode [in]`

Type: `const void *`

Pointer to the compiled shader

`pClassLinkage [in, optional]`

Type: `ID3D11ClassLinkage*`

Pointer, which can be `NULL`, to a class linkage interface

`pCreateVertexShaderExArgs [in]`

Type: `const NvAPI_D3D11_CREATE_VERTEX_SHADER_EX*`

Pointer to the `NvAPI_D3D11_CREATE_VERTEX_SHADER_EX` structure

`ppVertexShader [out]`

Type: `ID3D11VertexShader **`

The address of a pointer to an `ID3D11VertexShader` interface

4.2.3.2 Return Value

Returns `NVAPI_OK` on success.

4.2.3.3 Remarks

This function creates a Vertex shader with custom semantics.

4.2.4 NvAPI_D3D11_CreateHullShaderEx

```
NVAPI_INTERFACE NvAPI_D3D11_CreateHullShaderEx(
     [in]              ID3D11Device *pDevice,
     [in]              const void *pShaderBytecode,
     [in]              SIZE_T BytecodeLength,
     [in, optional]    ID3D11ClassLinkage *pClassLinkage,
     [in]              const NvAPI_D3D11_CREATE_HULL_SHADER_EX
                       *pCreateHullShaderExArgs,
     [out]             ID3D11HullShader **ppHullShader
);
```

4.2.4.1 Parameters

`pDevice [in]`

Type: `ID3D11Device *`

Pointer to the D3D11 device `ID3D11Device*`

`pShaderBytecode [in]`

Type: `const void *`

Pointer to the compiled shader

`pClassLinkage [in, optional]`

Type: `ID3D11ClassLinkage*`

Pointer, which can be `NULL`, to a class linkage interface

`pCreateHullShaderExArgs [in]`

Type: `const NvAPI_D3D11_CREATE_HULL_SHADER_EX*`

Pointer to the `NvAPI_D3D11_CREATE_HULL_SHADER_EX` structure

`ppHullShader [out]`

Type: `ID3D11HullShader **`

The address of a pointer to an `ID3D11HullShader` interface

4.2.4.2 Return Value

Returns `NVAPI_OK` on success.

4.2.4.3 Remarks

This function creates a Hull shader with custom semantics.

4.2.5 NvAPI_D3D11_CreateDomainShaderEx

```
NVAPI_INTERFACE NvAPI_D3D11_CreateDomainShaderEx(
     [in]              ID3D11Device *pDevice,
     [in]              const void *pShaderBytecode,
     [in]              SIZE_T BytecodeLength,
     [in, optional]    ID3D11ClassLinkage *pClassLinkage,
     [in]              const NvAPI_D3D11_CREATE_DOMAIN_SHADER_EX
                       *pCreateDomainShaderExArgs,
     [out]             ID3D11DomainShader **ppDomainShader
);
```

4.2.5.1 Parameters

`pDevice [in]`

Type: `ID3D11Device *`

Pointer to the D3D11 device `ID3D11Device*`

`pShaderBytecode [in]`

Type: `const void *`

Pointer to the compiled shader

`pClassLinkage [in, optional]`

Type: `ID3D11ClassLinkage*`

Pointer, which can be `NULL`, to a class linkage interface

`pCreateDomainShaderExArgs [in]`

Type: `const NvAPI_D3D11_CREATE_DOMAIN_SHADER_EX*`

Pointer to the `NvAPI_D3D11_CREATE_DOMAIN_SHADER_EX` structure

`ppDomainShader [out]`

Type: `ID3D11DomainShader **`

The address of a pointer to an `ID3D11DomainShader` interface

4.2.5.2 Return Value

Returns `NVAPI_OK` on success.

4.2.5.3 Remarks

This function creates a Domain shader with custom semantics.

4.2.6 NvAPI_D3D11_CreateGeometryShaderEx_2

```
NVAPI_INTERFACE NvAPI_D3D11_CreateGeometryShaderEx_2(
     [in]              ID3D11Device *pDevice,
     [in]              const void *pShaderBytecode,
     [in]              SIZE_T BytecodeLength,
     [in, optional]    ID3D11ClassLinkage *pClassLinkage,
     [in]              const NvAPI_D3D11_CREATE_GEOMETRY_SHADER_EX
                       *pCreateGeometryShaderExArgs,
     [out]             ID3D11GoemetryShader **ppGeometryShader
);
```

4.2.6.1 Parameters

`pDevice [in]`

Type: `ID3D11Device *`

Pointer to the D3D11 device `ID3D11Device*`

`pShaderBytecode [in]`

Type: `const void *`

Pointer to the compiled shader

`pClassLinkage [in, optional]`

Type: `ID3D11ClassLinkage*`

Pointer, which can be `NULL`, to a class linkage interface

`pCreateGeometryShaderExArgs [in]`

Type: `const NvAPI_D3D11_CREATE_GEOMETRY_SHADER_EX*`

Pointer to the `NvAPI_D3D11_CREATE_GEOMETRY_SHADER_EX` structure

`ppGeometryShader [out]`

Type: `ID3D11GoemetryShader **`

The address of a pointer to an `ID3D11GeometryShader` interface

4.2.6.2 Return Value

Returns `NVAPI_OK` on success.

4.2.6.3 Remarks

This function creates a Geometry shader with custom semantics.

5 DIRECTX 12 RENDERING WITH SINGLE PASS STEREO

Use the NVAPI programming interfaces for the Single Pass Stereo feature to enable rendering of both the left and right eye views in a single draw call as explained in the sections that follow.

5.1 QUERYING THE FEATURE CAPABILITY

To query the Single Pass Stereo feature capability, call the `NvAPI_D3D12_QuerySinglePassStereoSupport` NVAPI.

```
ComPtr<ID3D12Device> pDevice;

NV_QUERY_SINGLE_PASS_STEREO_SUPPORT_PARAMS Stereo = {0};

Stereo.version = NV_QUERY_SINGLE_PASS_STEREO_SUPPORT_PARAMS_VER;

NvAPI_D3D12_QuerySinglePassStereoSupport (pDevice.Get(), &Stereo);

// Stereo.bSinglePassStereoSupported will be TRUE if supported
```

5.2 CREATING AN HLSL SHADER PIPELINE BY USING NVAPI

Creating an HLSL pipeline involves using an NVAPI to create one or more graphics Pipeline State Objects (PSOs). A graphics PSO encapsulates the individual shader stages and other graphics states for the pipeline.

When you create a PSO, you determine which shader stages to encapsulate in the PSO in one of the following ways:

- Creating the PSO with all world space shader PSO extensions. For details, see "Creating the NVAPI PSO Extensions for All World Space Shaders" on page 37.
- Creating the PSO with **only** the last world space shader PSO extension. If you create the PSO in this way, you must set a flag when calling NVAPI functions as explained in "Creating the NVAPI PSO Extension for Only the Last World Space Shader " on page 43.

5.2.1 Defining the X Component for an HLSL Shader

An HLSL shader must compute the x component of the vertex position for both the eye views in a single pass. Therefore, you must define the x component of the view position of each eye as follows:

- For the left eye, define the x component of the view position in the regular .x component of the `SV_POSITION` variable.
- For the right eye, define the x component of the view position in a custom semantic variable, for example, `NV_X_RIGHT`.

The string name of the custom semantic variable is programmable. To associate the Single Pass Stereo custom semantic with the x position of the right eye view, you must use the NVAPI function to create the PSO.

5.2.2 Creating a Graphics PSO by Using NVAPI

An HLSL shader requires a graphics PSO.

1. Create and fill the following structures:
   - A regular PSO descriptor (`D3D12_GRAPHICS_PIPELINE_STATE_DESC *pPSODesc`). Fill this structure in the same way as for a regular PSO that would be created without NVAPI.
   - Additional information (`NVAPI_D3D12_PSO_EXTENSION_DESC** ppExtensions`). Fill this structure with information about the PSO extensions that are associated with individual shader stages.

2. Call the `NvAPI_D3D12_CreateGraphicsPipelineState` NVAPI function, passing the structures that you created in the previous step.

```
ComPtr<ID3D12PipelineState> pPSO;
ComPtr<ID3D12Device> pDevice;
ComPtr<ID3D12RootSignature> pRootSignature;

...

// Describe the graphics pipeline state object (PSO) using
// regular PSO descriptor
D3D12_GRAPHICS_PIPELINE_STATE_DESC psoDesc = {};
psoDesc.InputLayout = { inputElementDescs, _countof(inputElementDescs)
};
psoDesc.pRootSignature = pRootSignature.Get();
psoDesc.VS = {
    reinterpret_cast<UINT8*>(pVertexShader->GetBufferPointer()),
    pVertexShader->GetBufferSize() };
psoDesc.HS = {
    reinterpret_cast<UINT8*>(pHullShader->GetBufferPointer()),
    pHullShader->GetBufferSize() };
psoDesc.DS = {
    reinterpret_cast<UINT8*>(pDomainShader->GetBufferPointer()),
    pDomainShader->GetBufferSize() };
psoDesc.GS = {
    reinterpret_cast<UINT8*>(pGeometryShader->GetBufferPointer()),
    pGeometryShader->GetBufferSize() };
psoDesc.PS = {
    reinterpret_cast<UINT8*>(pPixelShader->GetBufferPointer()),
    pPixelShader->GetBufferSize() };
psoDesc.RasterizerState = CD3DX12_RASTERIZER_DESC(D3D12_DEFAULT);
psoDesc.BlendState = CD3DX12_BLEND_DESC(D3D12_DEFAULT);
psoDesc.DepthStencilState.DepthEnable = FALSE;
psoDesc.DepthStencilState.StencilEnable = FALSE;
psoDesc.SampleMask = UINT_MAX;
psoDesc.PrimitiveTopologyType = D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE;
psoDesc.NumRenderTargets = 1;
psoDesc.RTVFormats[0] = DXGI_FORMAT_R8G8B8A8_UNORM;
psoDesc.SampleDesc.Count = 1;

// Create and describe extensions as per the requirement
// e.g. Creating ALL world-space shaders using NVAPI would need four
// extension descriptors as follows

UINT numExtensions;

const NVAPI_D3D12_PSO_EXTENSION_DESC* ppPSOExtensionsDesc[4];
NVAPI_D3D12_PSO_VERTEX_SHADER_DESC vsExDesc;   // For Vertex Shader
NVAPI_D3D12_PSO_HULL_SHADER_DESC hsExDesc;     // For Hull Shader
NVAPI_D3D12_PSO_DOMAIN_SHADER_DESC dsExDesc;   // For Domain Shader
NVAPI_D3D12_PSO_GEOMETRY_SHADER_DESC gsExDesc; // For Geometry Shader
```

```
// Fill up the descriptors
// numExtensions should denote the number of extensions i.e. the number
// of shader stage created using NVAPI for this PSO. This will be 4 in
// case shaders of all four world-space stages are created using NVAPI
// for current PSO.

...

// Initialize NvAPI
NvAPI_Initialize();

...

NvAPI_D3D12_CreateGraphicsPipelineState(pDevice.Get(), &psoDesc,
numExtensions, ppPSOExtensionsDesc, &pPSO);
```

5.2.3 Creating the NVAPI PSO Extensions for All World Space Shaders

Any shaders that are being used in a PSO for a Single Pass Stereo rendering must be provided with the corresponding PSO extension information. They can be all or any combination of the following shaders:

- Vertex shader
- Hull shader
- Domain shader
- Geometry shader

5.2.3.1 Creating the NVAPI PSO Extension for the Vertex World Space Shader

The following example creates a Vertex shader to compute the vertex positions for both the left eye view and the right eye view.

```
cbuffer ConstantBuffer : register( b0 )
{
    matrix World;
    matrix View;
    matrix ViewRight;
    matrix Projection;
}

struct VSOutput
{
     float4 Pos : SV_POSITION;
     float4 Color : COLOR;
     float2 Tex : TEXCOORD0;
```

```
     float4 X_Right : NV_X_RIGHT; // Custom Semantic for Single Pass
Stereo
     uint4 ViewportMask : NV_VIEWPORT_MASK;
}

VSOutput VSMain (float4 Pos : POSITION,
                 float4 Color : COLOR,
                 float2 Tex : TEXCOORD0)
{
    VSOutput output = (VSOutput)0;
    float PosRight  = 0.0f;

    // Compute the vertex positions
    output.Pos = mul(mul(mul(Pos, World), View), Projection);
    PosRight   = mul(mul(mul(Pos, World), ViewRight), Projection);
    ...
    X_Right = PosRight.x; // Store in Custom Semantic for
                          // Single Pass Stereo
    ...
    output.ViewportMask = 0x00000001;
}
```

Note: You must declare the custom semantic for Single Pass Stereo as `float4`.

Note: You must declare the custom semantic for Viewport Mask as `uint4`.

After the shader is compiled to Direct3D shader bytecode, fill the PSO extension structure `NVAPI_D3D12_PSO_VERTEX_SHADER_DESC` and use the `NvAPI_D3D12_CreateGraphicsPipelineState` NVAPI to create the PSO.

```
#define MAX_EXT_COUNT 4 // e.g. Creating all world-space shaders using
NVAPI
ComPtr<ID3D12PipelineState> pPSO;
ComPtr<ID3D12Device> pDevice;
UINT numExtensions;
const NVAPI_D3D12_PSO_EXTENSION_DESC*
ppPSOExtensionsDesc[MAX_EXT_COUNT];
NVAPI_D3D12_PSO_VERTEX_SHADER_DESC vsExDesc;

...

// Describe the graphics pipeline state object (PSO) using regular PSO
descriptor
D3D12_GRAPHICS_PIPELINE_STATE_DESC pPSODesc = ...;

...
```

```
// Custom Semantics for Vertex Shader
memset(&vsExDesc, 0, sizeof(vsExDesc));
vsExDesc.psoExtension = NV_PSO_VERTEX_SHADER_EXTENSION;
vsExDesc.version = NV_VERTEX_SHADER_PSO_EXTENSION_DESC_VER;
vsExDesc.baseVersion = NV_PSO_EXTENSION_DESC_VER;
vsExDesc.NumCustomSemantics = 2;
pCustomSemantics = (NV_CUSTOM_SEMANTIC *)malloc(2 *
sizeof(NV_CUSTOM_SEMANTIC));
memset(vsExDesc.pCustomSemantics, 0, (2 * sizeof(NV_CUSTOM_SEMANTIC)));

vsExDesc.pCustomSemantics[0].version = NV_CUSTOM_SEMANTIC_VERSION;
vsExDesc.pCustomSemantics[0].NVCustomSemanticType =
NV_X_RIGHT_SEMANTIC;
strcpy_s(&(vsExDesc.pCustomSemantics[0].NVCustomSemanticNameString[0]),
NVAPI_LONG_STRING_MAX, "NV_X_RIGHT");

vsExDesc.pCustomSemantics[1].version = NV_CUSTOM_SEMANTIC_VERSION;
vsExDesc.pCustomSemantics[1].NVCustomSemanticType =
NV_VIEWPORT_MASK_SEMANTIC;
strcpy_s(&(vsExDesc.pCustomSemantics[1].NVCustomSemanticNameString[0]),
NVAPI_LONG_STRING_MAX, "NV_VIEWPORT_MASK");
ppPSOExtensionsDesc[numExtensions++] = &vsExDesc;

...

// Describe other PSO extensions, if required
// Create PSO using NVAPI and free up memory of pCustomSemantics
```

5.2.3.2 Creating the NVAPI PSO Extension for the Hull World Space Shader

If the application uses a Hull shader during Single Pass Stereo rendering, create the PSO by using the NVAPI with the PSO extension for the Hull shader. For the Hull shader, use the same custom semantics as for Single Pass Stereo.

After the Hull shader is compiled to Direct3D shader bytecode, fill the PSO extension structure `NVAPI_D3D12_PSO_HULL_SHADER_DESC` and use the `NvAPI_D3D12_CreateGraphicsPipelineState` NVAPI to create the PSO.

```
#define MAX_EXT_COUNT 4 // e.g. Creating all world-space shaders using
NVAPI
ComPtr<ID3D12PipelineState> pPSO;
ComPtr<ID3D12Device> pDevice;
UINT numExtensions;
const NVAPI_D3D12_PSO_EXTENSION_DESC*
ppPSOExtensionsDesc[MAX_EXT_COUNT];
NVAPI_D3D12_PSO_HULL_SHADER_DESC hsExDesc;

...
```

```
// Describe the graphics pipeline state object (PSO) using regular PSO
descriptor
D3D12_GRAPHICS_PIPELINE_STATE_DESC pPSODesc = ...;

...

// Custom Semantics for Hull Shader
memset(&hsExDesc, 0, sizeof(hsExDesc));
hsExDesc.psoExtension = NV_PSO_HULL_SHADER_EXTENSION;
hsExDesc.version = NV_HULL_SHADER_PSO_EXTENSION_DESC_VER;
hsExDesc.baseVersion = NV_PSO_EXTENSION_DESC_VER;
hsExDesc.NumCustomSemantics = 2;
hsExDesc.pCustomSemantics = (NV_CUSTOM_SEMANTIC *)malloc(2 *
sizeof(NV_CUSTOM_SEMANTIC));
memset(hsExDesc.pCustomSemantics, 0, (2 * sizeof(NV_CUSTOM_SEMANTIC)));

hsExDesc.pCustomSemantics[0].version = NV_CUSTOM_SEMANTIC_VERSION;
hsExDesc.pCustomSemantics[0].NVCustomSemanticType =
NV_X_RIGHT_SEMANTIC;
strcpy_s(&(hsExDesc.pCustomSemantics[0].NVCustomSemanticNameString[0]),
NVAPI_LONG_STRING_MAX, "NV_X_RIGHT");

hsExDesc.pCustomSemantics[1].version = NV_CUSTOM_SEMANTIC_VERSION;
hsExDesc.pCustomSemantics[1].NVCustomSemanticType =
NV_VIEWPORT_MASK_SEMANTIC;
strcpy_s(&(hsExDesc.pCustomSemantics[1].NVCustomSemanticNameString[0]),
NVAPI_LONG_STRING_MAX, "NV_VIEWPORT_MASK");

ppPSOExtensionsDesc[numExtensions++] = &hsExDesc;

...

// Describe other PSO extensions, if required
// Create PSO using NVAPI and free up memory of pCustomSemantics
```

5.2.3.3 Creating the NVAPI PSO Extension for the Domain World Space Shader

If the application uses a Domain shader during Single Pass Stereo rendering, create the PSO by using the NVAPI with the Domain shader PSO extension. For the Domain Shader, use the same custom semantics as for Single Pass Stereo.

After the Domain shader is compiled to Direct3D shader bytecode, fill the PSO extension structure `NVAPI_D3D12_PSO_DOMAIN_SHADER_DESC` and use the `NvAPI_D3D12_CreateGraphicsPipelineState` NVAPI to create the PSO.

```
#define MAX_EXT_COUNT 4 // e.g. Creating all world-space shaders using
NVAPI
ComPtr<ID3D12PipelineState> pPSO;
```

```
ComPtr<ID3D12Device> pDevice;
UINT numExtensions;
const NVAPI_D3D12_PSO_EXTENSION_DESC*
ppPSOExtensionsDesc[MAX_EXT_COUNT];
NVAPI_D3D12_PSO_DOMAIN_SHADER_DESC dsExDesc;

...

// Describe the graphics pipeline state object (PSO) using regular PSO
descriptor
D3D12_GRAPHICS_PIPELINE_STATE_DESC pPSODesc = ...;

...

// Custom Semantics for Domain Shader
memset(&dsExDesc, 0, sizeof(dsExDesc));
dsExDesc.psoExtension = NV_PSO_DOMAIN_SHADER_EXTENSION;
dsExDesc.version = NV_DOMAIN_SHADER_PSO_EXTENSION_DESC_VER;
dsExDesc.baseVersion = NV_PSO_EXTENSION_DESC_VER;
dsExDesc.NumCustomSemantics = 2;
dsExDesc.pCustomSemantics = (NV_CUSTOM_SEMANTIC *)malloc(2 *
sizeof(NV_CUSTOM_SEMANTIC));
memset(dsExDesc.pCustomSemantics, 0, (2 * sizeof(NV_CUSTOM_SEMANTIC)));

dsExDesc.pCustomSemantics[0].version = NV_CUSTOM_SEMANTIC_VERSION;
dsExDesc.pCustomSemantics[0].NVCustomSemanticType =
NV_X_RIGHT_SEMANTIC;
strcpy_s(&(dsExDesc.pCustomSemantics[0].NVCustomSemanticNameString[0]),
NVAPI_LONG_STRING_MAX, "NV_X_RIGHT");

dsExDesc.pCustomSemantics[1].version = NV_CUSTOM_SEMANTIC_VERSION;
dsExDesc.pCustomSemantics[1].NVCustomSemanticType =
NV_VIEWPORT_MASK_SEMANTIC;
strcpy_s(&(dsExDesc.pCustomSemantics[1].NVCustomSemanticNameString[0]),
NVAPI_LONG_STRING_MAX, "NV_VIEWPORT_MASK");

ppPSOExtensionsDesc[numExtensions++] = &dsExDesc;

...

// Describe other PSO extensions, if required
// Create PSO using NVAPI and free up memory of pCustomSemantics
```

5.2.3.4 Creating the NVAPI PSO Extension for the Geometry World Space Shader

If the application uses a Geometry shader during Single Pass Stereo rendering, create the PSO by using the NVAPI with the Geometry shader PSO extension. For the Geometry shader, use the same custom semantics as for Single Pass Stereo.

After the shader is compiled to Direct3D shader bytecode, fill the PSO extension structure `NVAPI_D3D12_PSO_GEOMETRY_SHADER_DESC` and use the `NvAPI_D3D12_CreateGraphicsPipelineState` NVAPI to create the PSO.

```
#define MAX_EXT_COUNT 4 // e.g. Creating all world-space shaders using
NVAPI
ComPtr<ID3D12PipelineState> pPSO;
ComPtr<ID3D12Device> pDevice;
UINT numExtensions;
const NVAPI_D3D12_PSO_EXTENSION_DESC*
ppPSOExtensionsDesc[MAX_EXT_COUNT];
NVAPI_D3D12_PSO_GEOMETRY_SHADER_DESC gsExDesc;

...

// Describe the graphics pipeline state object (PSO) using regular PSO
descriptor
D3D12_GRAPHICS_PIPELINE_STATE_DESC pPSODesc = ...;

...

// Custom Semantics for Geometry Shader
memset(&gsExDesc, 0, sizeof(gsExDesc));
gsExDesc.ForceFastGS = false;   // True, if using FastGS
gsExDesc.psoExtension = NV_PSO_GEOMETRY_SHADER_EXTENSION;
gsExDesc.version = NV_GEOMETRY_SHADER_PSO_EXTENSION_DESC_VER;
gsExDesc.baseVersion = NV_PSO_EXTENSION_DESC_VER;
gsExDesc.NumCustomSemantics = 2;
gsExDesc.pCustomSemantics = (NV_CUSTOM_SEMANTIC *)malloc(2 *
sizeof(NV_CUSTOM_SEMANTIC));
memset(gsExDesc.pCustomSemantics, 0, (2 * sizeof(NV_CUSTOM_SEMANTIC)));

gsExDesc.pCustomSemantics[0].version = NV_CUSTOM_SEMANTIC_VERSION;
gsExDesc.pCustomSemantics[0].NVCustomSemanticType =
NV_X_RIGHT_SEMANTIC;
strcpy_s(&(gsExDesc.pCustomSemantics[0].NVCustomSemanticNameString[0]),
NVAPI_LONG_STRING_MAX, "NV_X_RIGHT");

gsExDesc.pCustomSemantics[1].version = NV_CUSTOM_SEMANTIC_VERSION;
gsExDesc.pCustomSemantics[1].NVCustomSemanticType =
NV_VIEWPORT_MASK_SEMANTIC;
strcpy_s(&(gsExDesc.pCustomSemantics[1].NVCustomSemanticNameString[0]),
NVAPI_LONG_STRING_MAX, "NV_VIEWPORT_MASK");

ppPSOExtensionsDesc[numExtensions++] = &gsExDesc;

...

// Describe other PSO extensions, if required
// Create PSO using NVAPI and free up memory of pCustomSemantics
```

5.2.4 Creating the NVAPI PSO Extension for Only the Last World Space Shader

You can use NVAPI to create the PSO by providing PSO extension information **only** for the last world space shader stage in the Single Pass Stereo rendering pipeline.

Fill the appropriate NVAPI PSO extension while creating the PSO according to the type of world space shader as shown in the following table.

Last World Space Shader	NVAPI
Vertex	`NVAPI_D3D12_PSO_VERTEX_SHADER_DESC`
Domain	`NVAPI_D3D12_PSO_DOMAIN_SHADER_DESC`
Geometry	`NVAPI_D3D12_PSO_GEOMETRY_SHADER_DESC`

For example, the Single Pass Stereo rendering pipeline might contain a Vertex shader, Geometry shader, and Pixel shader in that order. In this situation, create a PSO with the PSO extension only for the Geometry shader.

When you use the NVAPI to create the PSO with only the last world space Shader PSO extension, set `UseSpecificShaderExt` to `TRUE`.

The following example uses the NVAPI to create a Geometry last world shader.

```
ComPtr<ID3D12PipelineState> pPSO;
ComPtr<ID3D12Device> pDevice;
UINT numExtensions;

const NVAPI_D3D12_PSO_EXTENSION_DESC* ppPSOExtensionsDesc[1];
NVAPI_D3D12_PSO_GEOMETRY_SHADER_DESC gsExDesc;

...

// Describe the graphics pipeline state object (PSO) using regular PSO
descriptor
D3D12_GRAPHICS_PIPELINE_STATE_DESC pPSODesc = ...;

// Custom Semantics for Geometry Shader
memset(&gsExDesc, 0, sizeof(gsExDesc));
gsExDesc.ForceFastGS = false;   // True, if using FastGS
gsExDesc.psoExtension = NV_PSO_GEOMETRY_SHADER_EXTENSION;
gsExDesc.version = NV_GEOMETRY_SHADER_PSO_EXTENSION_DESC_VER;
gsExDesc.baseVersion = NV_PSO_EXTENSION_DESC_VER;
gsExDesc.NumCustomSemantics = 2;
gsExDesc.pCustomSemantics = (NV_CUSTOM_SEMANTIC *)malloc(2 *
sizeof(NV_CUSTOM_SEMANTIC));
memset(gsExDesc.pCustomSemantics, 0, (2 * sizeof(NV_CUSTOM_SEMANTIC)));

gsExDesc.pCustomSemantics[0].version = NV_CUSTOM_SEMANTIC_VERSION;
```

```
gsExDesc.pCustomSemantics[0].NVCustomSemanticType =
NV_X_RIGHT_SEMANTIC;
strcpy_s(&(gsExDesc.pCustomSemantics[0].NVCustomSemanticNameString[0]),
NVAPI_LONG_STRING_MAX, "NV_X_RIGHT");

gsExDesc.pCustomSemantics[1].version = NV_CUSTOM_SEMANTIC_VERSION;
gsExDesc.pCustomSemantics[1].NVCustomSemanticType =
NV_VIEWPORT_MASK_SEMANTIC;
strcpy_s(&(gsExDesc.pCustomSemantics[1].NVCustomSemanticNameString[0]),
NVAPI_LONG_STRING_MAX, "NV_VIEWPORT_MASK");

gsExDesc.UseSpecificShaderExt = true;

ppPSOExtensionsDesc[0] = &gsExDesc;
numExtensions = 1;

NvAPI_D3D12_CreateGraphicsPipelineState(pDevice.Get(), &psoDesc,
numExtensions, ppPSOExtensionsDesc, &pPSO);

free(gsExDesc.pCustomSemantics);
```

5.2.5 Using Single Pass Stereo with FastGS

Fast Geometry (FastGS) shader enables you to write Geometry shaders with certain restrictions to make them run more efficiently on the GPU.

Note: FastGS is not supported with the optimization disabled (/Od) HLSL compilation option.

You can use FastGS with the Single Pass Stereo feature when creating a PSO using NVAPI with PSO extensions for all the world space shaders or with PSO extensions for only some of the world space shaders.

5.2.5.1 Using FastGS when Creating NVAPI PSO Extensions for All the World Space Shaders

The following example HLSL code shows how to use Single Pass Stereo with FastGS.

```
struct VSOutput
{
     float4 Pos : SV_POSITION;
     float4 Color : COLOR;
     float2 Tex : TEXCOORD0;

     float4 X_Right : NV_X_RIGHT; // Single Pass Stereo
}

struct GSOutput
```

```
{
    VSOutput Passthrough;

    uint4 ViewportMask : NV_VIEWPORT_MASK;
};

VSOutput VSMain (float4 position : POSITION,
                 float4 color : COLOR,
                 float2 Tex : TEXCOORD0)
{
     VSOutput OutVtx;

     OutVtx.Pos = position;
     ...
     ...
     OutVtx.X_Right = (...); // right eye X value computation
     ...
     ...
     return OutVtx;
}

[maxvertexcount(1)]
void FastGSMain(triangle VSOutput In[3],
               inout TriangleStream<GSOutput>
                TriStream)
{
    GSOutput output;

    output.Passthrough = In[0];
    output.ViewportMask= 0x00000001;

    TriStream.Append(output);
}
```

To create the PSO using NVAPI with PSO extensions for the Fast Geometry Shader, fill the `NVAPI_D3D12_PSO_GEOMETRY_SHADER_DESC` with `ForceFastGS = true` as shown in the following example.

```
NVAPI_D3D12_PSO_GEOMETRY_SHADER_DESC gsExDesc;

// Describe the graphics pipeline state object (PSO) using regular PSO
descriptor
...
// Specify custom semantics for Geometry Shaders in gsExDesc
...
gsExDesc.ForceFastGS = true;
...
gsExDesc.UseViewportMask = false;

 NvAPI_D3D12_CreateGraphicsPipelineState(pDevice.Get(), &psoDesc,
numExtensions, ppPSOExtensionsDesc, &pPSO);
```

5.2.5.2 Using FastGS when Creating NVAPI PSO Extensions for Only Some of the World Space Shaders

You might want to use FastGS when creating a PSO using NVAPI with PSO extensions for only some of the world space shaders. In this situation, you cannot do so only for the last world space shader in the pipeline. Instead, you must be create the PSO with extensions for both of the following world space shaders:

- The FastGS Geometry shader
- The world space shader immediately before the FastGS Geometry shader in the Single Pass Stereo rendering pipeline

All remaining shaders can be created without PSO extensions.

For example:

- The Single Pass Stereo rendering pipeline might contain the Vertex shader and the FastGS Geometry shader in that order.

  In this situation, create the PSO using NVAPI with PSO extensions for the Vertex shader **and** the FastGS Geometry shader by using the NVAPI for creating the last world space shader.

- The Single Pass Stereo rendering pipeline might contain the Vertex shader, Hull shader, Domain shader, and the FastGS Geometry shader in that order.

  In this situation, create PSO using NVAPI with PSO extensions **only** for Domain shader and the FastGS Geometry shader by using the NVAPI for creating the last world space shader. You can omit PSO extensions for theVertex Shader and the Hull Shader.

For information about how to create a PSO using NVAPI with PSO extensions for the last world space shader, see “Creating the NVAPI PSO Extension for Only the Last World Space Shader” on page 43.

5.3 GENERATING THE LEFT-EYE AND RIGHT-EYE VIEWS

Generate the left-eye view and right view with Single Pass Stereo by using one of the following methods:

- **Generating both views on a render target array.** In this method, the view for each eye occupies one element in the render target array. The left-eye view occupies the

first element in the array and the right-eye view occupies the second element in the array.

- **Generating side-by-side views on a single render target.** In this method, the two views are generated side-by-side directly on a single render target.

5.3.1 Generating Both Views on a Render Target Array

The following example creates a render target array in which the right-eye view is generated on the second element of the array. Note that each element of this array has a width equal to half the width of a regular screen or `SwapChain` width. The width of each element is this size because each RTA element holds only a single eye view which is copied onto a `SwapChain` containing side-by-side views before the view is presented.

```
ComPtr<ID3D12GraphicsCommandList> pCommandList;
ComPtr<ID3D12Resource> pRenderTargetForRTA; // Render Targets for RTA
method
ComPtr<IDXGISwapChain3> pSwapChain;
ComPtr<ID3D12DescriptorHeap> pRtvHeap;
ComPtr<ID3D12DescriptorHeap> pSrvHeap;
UINT rtvDescriptorSize;

...

// Create SwapChain with twice the width of each view (side-by-side)

...

// Create descriptor heaps.

// Describe and create a render target view (RTV) descriptor heap.
D3D12_DESCRIPTOR_HEAP_DESC rtvHeapDesc = {};
rtvHeapDesc.NumDescriptors = FrameCount;
rtvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_RTV;
rtvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_NONE;
pDevice->CreateDescriptorHeap(&rtvHeapDesc, IID_PPV_ARGS(&pRtvHeap));

// Describe and create a shader resource view (SRV) heap for the
texture.
D3D12_DESCRIPTOR_HEAP_DESC srvHeapDesc = {};
srvHeapDesc.NumDescriptors = 1;
srvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;
srvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;
 pDevice->CreateDescriptorHeap(&srvHeapDesc, IID_PPV_ARGS(&pSrvHeap));

rtvDescriptorSize = pDevice-
>GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_RTV);

...
```

```
// Descriptor for the RTA Render Targets we will create for this method
CD3DX12_RESOURCE_DESC renderTargetDescForRTA;

ZeroMemory(&renderTargetDescForRTA, sizeof (CD3DX12_RESOURCE_DESC));

renderTargetDescForRTA.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;
renderTargetDescForRTA.Alignment = 0;
renderTargetDescForRTA.Width = ...;     // Total screen width / 2 (i.e.
width per view)
renderTargetDescForRTA.Height = ...;    // Total screen height
renderTargetDescForRTA.DepthOrArraySize = 2; // For RTA
renderTargetDescForRTA.MipLevels = 1;
renderTargetDescForRTA.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
renderTargetDescForRTA.SampleDesc.Count = 1;
renderTargetDescForRTA.SampleDesc.Quality = 0;
renderTargetDescForRTA.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;
renderTargetDescForRTA.Flags = D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET;

...

D3D12_CLEAR_VALUE clearValue = { DXGI_FORMAT_R8G8B8A8_UNORM, { 0.0f,
0.2f, 0.4f, 1.0f } };

pDevice->CreateCommittedResource(
    &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
    D3D12_HEAP_FLAG_NONE,
    &renderTargetDescForRTA,
    D3D12_RESOURCE_STATE_GENERIC_READ,
    &clearValue,
    IID_PPV_ARGS(&pRenderTargetForRTA));

// Create frame resources.
CD3DX12_CPU_DESCRIPTOR_HANDLE rtvHandle(pRtvHeap-
>GetCPUDescriptorHandleForHeapStart());

pDevice->CreateRenderTargetView(pRenderTargetForRTA.Get(), nullptr,
rtvHandle);
rtvHandle.Offset(1, rtvDescriptorSize);

...

// Enabling Single Pass Stereo using RTA method
NvAPI_D3D12_SetSinglePassStereoMode (pCommandList,
    2, // numViews (for two eyes)
    1, // Offset between render targets of the different views
    false); // Since there’s a single viewport, we don’t need
            // independent viewport mask enabled
...

pCommandList->OMSetRenderTargets(1, &rtvHandle, FALSE, nullptr);
```

```
...

// Render using single draw call

...

// Indicate that the render target will be used as a copy source
pCommandList->ResourceBarrier(1,
&CD3DX12_RESOURCE_BARRIER::Transition(pRenderTargetForRTA.Get(),
D3D12_RESOURCE_STATE_RENDER_TARGET, D3D12_RESOURCE_STATE_COPY_SOURCE));

// Copy each view from temporary RTA Render Targets onto the
// double-width SwapChain

ComPtr<ID3D12Resource> swapChainBuffer;

pSwapChain->GetBuffer(0, IID_PPV_ARGS(&swapChainBuffer));

D3D12_TEXTURE_COPY_LOCATION dst = {swapChainBuffer.Get(),
D3D12_TEXTURE_COPY_TYPE_SUBRESOURCE_INDEX, 0};
D3D12_TEXTURE_COPY_LOCATION src = {pRenderTargetForRTA.Get(),
D3D12_TEXTURE_COPY_TYPE_SUBRESOURCE_INDEX, 0};
// Source box with dimensions per view
D3D12_BOX srcBox = {
    0,
    0,
    0,
    ...,    // Total screen width / 2 (i.e. width per view)
    ...,    // Total screen height
    1};

// Copy View 0
src.SubresourceIndex = 0;
pCommandList->CopyTextureRegion(
    &dst,
    0,
    0,
    0,
    &src,
    &srcBox);

// Copy View 1
src.SubresourceIndex = 1;
pCommandList->CopyTextureRegion(
    &dst,
    ...,      // DstX: Offset of total screen width / 2
    0,
    0,
    &src,
    &srcBox); // Note the DstX has an offset of one eye view width

...
```

```
// Present
```

If you generate a render target array, you must specify the viewport mask in the HLSL shader as 0x00000001 in a custom semantic variable, for example, NV_VIEWPORT_MASK.

The following example sets the viewport mask to 0x00000001 for a Geometry shader.

```
struct GSOutput
{
     float4 Pos : SV_POSITION;
     float4 Color : COLOR;
     float2 Tex : TEXCOORD0;

     float4 X_Right : NV_X_RIGHT; // Single Pass Stereo
     uint4 ViewportMask : NV_VIEWPORT_MASK;
}

[maxvertexcount(3)]
void GSMain(triangle VSOutput Input[3],
            inout TriangleStream<GSOutput> TriStream)
{
     GSOutput OutVtx;

     for (int v = 0; v < 3; ++v)
     {
         OutVtx.Pos    = Input[v].Pos;
         OutVtx.Color  = Input[v].Color;
         OutVtx.Tex    = Input[v].Tex;
         OutVtx.X_Right = Input[v].X_Right;

         OutVtx.ViewportMask = 0x00000001;
         ...
         TriStream.Append(OutVtx);
     }
}
```

5.3.2 Generating Side-by-Side Views on a Single Render Target

If you generate side-by-side views on a single render target, you must create one viewport for each eye view.

The following example creates a single render target with two viewports.

```
ComPtr<ID3D12GraphicsCommandList> pCommandList

// Create SwapChain with twice the width of each view (side-by-side)

...
```

```
// Setup two side-by-side viewports, one for each view
D3D12_VIEWPORT viewports[2];
memset(viewports, 0, sizeof(viewports));

viewports[0].TopLeftX = 0;
viewports[0].TopLeftY = 0;
viewports[0].Width = ...;    // Total screen width / 2 (i.e. width per
eye view)
viewports[0].Height = ...;   // Total screen height
viewports[0].MinDepth = 0.0f; // Total screen height
viewports[0].MaxDepth = 1.0f;

viewports[1].TopLeftX = ...; // Total screen width / 2 (Offset for the
second view)
viewports[1].TopLeftY = 0;
viewports[1].Width = ...;    // Total screen width / 2 (i.e. width per
view)
viewports[1].Height = ...;   // Total screen height
viewports[1].MinDepth = 0.0f;
viewports[1].MaxDepth = 1.0f;

pCommandList->RSSetViewports(2, &viewports[0]);

// Setup Scissor Rects (one for for each view)

scissorRect[0].left = 0;
scissorRect[0].top = 0;
scissorRect[0].right = ...;  // Total screen width / 2 (i.e. width per
eye view)
scissorRect[0].bottom = ...; // Total screen height

scissorRect[1].left = ...;   // Total screen width / 2 (Offset for the
second view)
scissorRect[1].top = 0;
scissorRect[1].right = ...;  // Upto total screen width (making the
width of the Scissor Rect as width / 2)
scissorRect[1].bottom = ...; // Total screen height

...

// Enabling Single Pass Stereo using Side-by-side method
NvAPI_D3D12_SetSinglePassStereoMode (pCommandList.Get(),
    2, // numViews (for two eyes)
    0, // Offset between render targets of the different views.
       // Since there is a single Render Target, we set this as 0.
    true); // We need to provide viewport mask corresponding to each
           // viewport, so we set this as true.

...

// Render using single draw call
```

```
...

// Present
```

If you generate a single target array, you must specify the viewport mask in the HLSL shader as 0x00020001 in a custom semantic variable, for example, NV_VIEWPORT_MASK.

This value indicates an independent viewport mask for the left eye and the right eye. The upper 16 bits are for right eye viewport. The second bit is set to indicate that the right eye view uses the second viewport.

The following example sets the viewport mask to 0x00020001 for a Geometry shader.

```
struct GSOutput
{
     float4 Pos : SV_POSITION;
     float4 Color : COLOR;
     float2 Tex : TEXCOORD0;

     float4 X_Right : NV_X_RIGHT; // Single Pass Stereo
     uint4 ViewportMask : NV_VIEWPORT_MASK;
}

[maxvertexcount(3)]
void GSMain(triangle VSOutput Input[3],
            inout TriangleStream<GSOutput> TriStream)
{
     GSOutput OutVtx;

     for (int v = 0; v < 3; ++v)
     {
         OutVtx.Pos    = Input[v].Pos;
         OutVtx.Color  = Input[v].Color;
         OutVtx.Tex    = Input[v].Tex;
         OutVtx.X_Right = Input[v].X_Right;
         // independent viewport mask for left eye and right eye
         // upper 16 bits for right eye viewport using Single Pass
Stereo
         // 2nd bit set indicates right eye view will use 2nd viewport
         OutVtx.ViewportMask = 0x00020001;
         ...
         ...

         TriStream.Append(OutVtx);
     }
}
```

5.3.3 Detecting the Eye for a View

When you generate the left-eye view and right-eye view, the application must be able to detect which view is being rendered.

To detect the eye for which a view is being rendered, use the system semantic `SV_RenderTargetArrayIndex` or `SV_ViewportArrayIndex` in the Pixel shader based on the Single Pass Stereo method used. The semantic to use depends on how the view is generated.

Note: The D3D runtime reports a linkage error while creating a PSO when only the Pixel shader reads `SV_RenderTargetArrayIndex` or `SV_ViewportArrayIndex` but the previous shader does not output the same value. To avoid this error, the value must be output from a Geometry shader with a dummy value, for example, 0. This dummy assignment is ignored by the NVIDIA display driver. This Geometry shader must be in the PSO created using NVAPI with the Geometry shader PSO extension. Only in that case, the dummy assignment will be ignored. Also, make sure that the input signature of the Pixel Shader matches the output signature of the Geometry shader when you follow this procedure.

5.3.3.1 Detecting the Eye for a View Generated on a Render Target Array

If both views are generated on a render target array, use the system semantic `SV_RenderTargetArrayIndex` in the Pixel shader to detect which view is being rendered.

- For first view, the value is 0.
- For the second view, the value is 1.

```
struct GSOutput
{
    ...
    uint renderTargetIndex : SV_RenderTargetArrayIndex;
};

[maxvertexcount(3)]
void GSMain (triangle VSOutput input[3], inout TriangleStream<GSOutput>
OutStream)
{
    GSOutput output = (GSOutput)0;

    for (int v = 0; v < 3; v++)
    {
        ...
        output.renderTargetIndex = 0; // Dummy assignment. This will be
ignored by driver
```

```
        OutStream.Append(output);
    }
}

float4 PSMain(GSOutput input) : SV_TARGET
{
    ...
    if (input.renderTargetIndex == 0)
        // View 0 (Left eye)
    else if (input.renderTargetIndex == 1)
        // View 1 (Right eye)
}
```

5.3.3.2 Detecting the Eye for a View Generated on a Single Render Target

If side-by-side views are generated on a single render target, use the system semantic `SV_ViewportArrayIndex` in the Pixel shader to detect which view is being rendered.

- For first view, the value is 0.
- For the second view, the value is 1.

```
struct GSOutput
{
    ...
    uint viewportIndex : SV_ViewportArrayIndex;
};

[maxvertexcount(3)]
void GSMain (triangle VSOutput input[3], inout TriangleStream<GSOutput>
OutStream)
{
    GSOutput output = (GSOutput)0;

    for (int v = 0; v < 3; v++)
    {
        ...
        output.viewportIndex = 0; // Dummy assignment. This will be
ignored by driver

        OutStream.Append(output);
    }
}

float4 PSMain(GSOutput input) : SV_TARGET
{
    ...
    if (input.viewportIndex == 0)
        // View 0 (Left eye)
    else if (input.viewportIndex == 1)
        // View 1 (Right eye)
}
```

5.4 ENABLING AND DISABLING THE SINGLE PASS STEREO MODE OF THE GPU

5.4.1 Enabling the Single Pass Stereo Mode of the GPU

If your application uses the Single Pass Stereo feature, you must enable the Single Pass Stereo mode of the GPU.

To enable the Single Pass Stereo mode of the GPU, use the `NvAPI_D3D12_SetSinglePassStereoMode` NVAPI as in the following example.

You must enable Single Pass Stereo on a `D3D12_COMMAND_LIST_TYPE_DIRECT` Graphics Command List. The state of Single Pass Stereo persists until the Command List is closed. As soon as `pCommandList->Close()` is called, the state of Single Pass Stereo is reset back to disabled.

The Single Pass Stereo state is restricted only to the Command List on which `NvAPI_D3D12_SetSinglePassStereoMode` is called. Therefore, all render calls made only to this specific Command List are affected. Any other Command List for which Single Pass Stereo has not been enabled will function normally.

The values of the parameters to pass to this NVAPI depend on how the left-eye view and right-eye view are generated:

- For two views generated on a render target array, set the offset to 1.

```
// Enabling Single Pass Stereo using RTA method
NvAPI_D3D12_SetSinglePassStereoMode (pCommandList.Get(),
     2, // numViews (for two eyes)
     1, // Offset between render targets of the different views
     false); // Since there's a single viewport, we don't need
             // independent viewport mask enabled
```

- For side-by-side views generated on a single render target, set the offset to 0 and the independent viewport mask to `true`.

```
// Enabling Single Pass Stereo using Side-by-side method
NvAPI_D3D12_SetSinglePassStereoMode (pCommandList.Get(),
     2, // numViews (for two eyes)
     0, // Offset between render targets of the different views.
        // Since there is a single Render Target, we set this as 0.
     true); // We need to provide viewport mask corresponding to each
            // viewport, so we set this as true.
```

For more information about how to generate the left-eye view and right-eye view, see “Generating the Left-Eye and Right-Eye Views” on page 46.

5.4.2 Disabling the Single Pass Stereo Mode of the GPU

To disable the Single Pass Stereo mode of the GPU, use the `NvAPI_D3D12_SetSinglePassStereoMode` NVAPI as in the following example.

```
NvAPI_D3D12_SetSinglePassStereoMode (pCommandList.Get(), 1, 1, false);
```

6 DIRECTX 12 RESTRICTIONS

6.1 VIEWPORT ARRAY INDEX ALTERNATIVE

When custom semantics are used, do **not** use the viewport array index standard semantic `SV_ViewportArrayIndex` for the world space shaders. Instead, use the custom semantic for viewport mask in these shaders.

The world space shaders are as follows:

- Vertex shader
- Hull shader
- Domain shader
- Geometry shader

When you use the custom semantic for viewport mask, you must set the appropriate bits in the mask that correspond to the viewports that you want to make active. For example, to make the viewport at index *N* active, write the following code:

```
ViewportMask = (0x1 << N)
```

Do **not** write `ViewportIndex = N`. Hardware will ignore any writes performed to `SV_ViewportArrayIndex` variable inside any custom semantic shader.

For an exception to this requirement, see "Detecting the Eye for a View" on page 53.

6.2 MAXIMUM NUMBER OF VIEWPORTS

The maximum number of viewports supported is 16.

The number of viewports that a view uses and, therefore, the number of viewports supported per eye, depend on how the left-eye view and right-eye view are generated:

- If both views are generated on a render target array, the same viewport is used for both eyes. Therefore, the maximum number of viewports supported per eye is 16.
- If side-by-side views are generated on a single render target, independent viewports are used for each eye. Therefore, the maximum number of viewports supported per eye is 8.

If you need more than 8 viewports per eye (for example, 3 × 3 = 9 viewports) you must use a render target array for generating the left-eye view and right-eye view.

6.3 CACHED PSO

A Graphics Pipeline State Object (PSO) created using NVAPI `NvAPI_D3D12_CreateGraphicsPipelineState` does not support the Cached PSO feature of DirectX 12. Do not call `GetCachedBlob()` with such a PSO.

7 DIRECTX 12 API REFERENCE

7.1 STRUCTURES

7.1.1 NV_QUERY_SINGLE_PASS_STEREO_SUPPORT_PARAMS

```
typedef struct _NV_QUERY_SINGLE_PASS_STEREO_SUPPORT_PARAMS
{
    NvU32 version;
    NvU32 bSinglePassStereoSupported;
} NV_QUERY_SINGLE_PASS_STEREO_SUPPORT_PARAMS;
```

7.1.1.1 Members

`version`

Type: `NvU32`

The version of the `NV_QUERY_SINGLE_PASS_STEREO_SUPPORT_PARAMS` structure

`bSinglePassStereoSupported`

Type: `NvU32`

Indicates whether Single Pass Stereo is supported on the current setup

7.1.1.2 Remarks

The `NV_QUERY_SINGLE_PASS_STEREO_SUPPORT_PARAMS` structure provides information about the Single Pass Stereo capability on the current setup. This structure is used in the `NvAPI_D3D12_QuerySinglePassStereoSupport()` function.

7.1.2 NV_CUSTOM_SEMANTIC

```
typedef struct _NV_CUSTOM_SEMANTIC
{
    UINT                        version;
    NV_CUSTOM_SEMANTIC_TYPE     NVCustomSemanticType;
    NvAPI_LongString            NVCustomSemanticNameString;
    BOOL                        RegisterSpecified;
    NvU32                       RegisterNum;
    NvU32                       RegisterMask;
    NvU32                       Reserved;
} NV_CUSTOM_SEMANTIC;
```

7.1.2.1 Members

`version`

Type: `UINT`

The version of the `NV_CUSTOM_SEMANTIC` structure

`NVCustomSemanticType`

Type: `NV_CUSTOM_SEMANTIC_TYPE`

It can have either of the following types:

`NV_X_RIGHT_SEMANTIC` for specifying the X position value for the right eye

`NV_VIEWPORT_MASK_SEMANTIC` for specifying the viewport mask

`NVCustomSemanticNameString`

Type: `NvAPI_LongString`

The name of the custom semantic

`RegisterSpecified`

Type: `NvU32`

Reserved

`RegisterNum`

Type: `NvU32`

Reserved

`RegisterMask`

Type: `NvU32`

Reserved

`Reserved`

Type: `NvU32`

Reserved

7.1.2.2 Remarks

The `NV_CUSTOM_SEMANTIC` structure is used to specify custom semantics. It is a part of following structures which are used in creating corresponding extended shaders:

- `NVAPI_D3D12_PSO_VERTEX_SHADER_DESC`
- `NVAPI_D3D12_PSO_HULL_SHADER_DESC`
- `NVAPI_D3D12_PSO_DOMAIN_SHADER_DESC`
- `NVAPI_D3D12_PSO_GEOMETRY_SHADER_DESC`

7.1.3 NVAPI_D3D12_PSO_EXTENSION_DESC

```
struct NVAPI_D3D12_PSO_EXTENSION_DESC
{
    NvU32            baseVersion;
    NV_PSO_EXTENSION psoExtension;
};
```

7.1.3.1 Members

`baseVersion`

Type: `NvU32`

The version of the `NVAPI_D3D12_PSO_EXTENSION_DESC` structure. Always use `NV_PSO_EXTENSION_DESC_VER` for this.

`psoExtension`

Type: `NV_PSO_EXTENSION`

The type of the PSO extension. This can have either of the following values which are of importance for Single Pass Stereo:

- `NV_PSO_VERTEX_SHADER_EXTENSION`
- `NV_PSO_HULL_SHADER_EXTENSION`
- `NV_PSO_DOMAIN_SHADER_EXTENSION`
- `NV_PSO_GEOMETRY_SHADER_EXTENSION`

7.1.3.2 Remarks

This structure is inherited in PSO extension structures for the corresponding shader stages.

7.1.4 NVAPI_D3D12_PSO_VERTEX_SHADER_DESC

```
struct NVAPI_D3D12_PSO_VERTEX_SHADER_DESC
{
    NvU32 version;
    NV_PSO_EXTENSION psoExtension;
    NvU32 NumCustomSemantics;
    NV_CUSTOM_SEMANTIC *pCustomSemantics;
    BOOL UseWithFastGS;
    BOOL UseSpecificShaderExt;
};
```

7.1.4.1 Members

`version`

Type: `NvU32`

The version of the `NVAPI_D3D12_PSO_VERTEX_SHADER_DESC` structure. You should assign `NV_VERTEX_SHADER_PSO_EXTENSION_DESC_VER` to this.

`psoExtension`

Type: `NV_PSO_EXTENSION`

The type of the PSO extension. You should assign `NV_PSO_VERTEX_SHADER_EXTENSION` to this.

`NumCustomSemantics`

Type: `NvU32`

The number of custom semantic elements, up to `NV_CUSTOM_SEMANTIC_MAX`, provided in the array pointer `pCustomSemantics`

`pCustomSemantics`

Type: `NV_CUSTOM_SEMANTIC`

Used to specify custom semantics for this shader

`UseWithFastGS`

Type: `BOOL`

Reserved

`UseSpecificShaderExt`

Type: `BOOL`

`TRUE` if minimal specific shaders are being created with NVAPI shader extensions as explained in “Creating the NVAPI PSO Extension for Only the Last World Space Shader” on page 43.

7.1.4.2 Remarks

This structure is used to specify parameters while creating a custom Vertex shader in a PSO using NVAPI. This structure is used in the `NvAPI_D3D12_CreateGraphicsPipelineState()` function.

7.1.5 NVAPI_D3D12_PSO_HULL_SHADER_DESC

```
struct NVAPI_D3D12_PSO_HULL_SHADER_DESC
{
    NvU32 version;
    NV_PSO_EXTENSION psoExtension;
    NvU32 NumCustomSemantics;
    NV_CUSTOM_SEMANTIC *pCustomSemantics;
    BOOL UseWithFastGS;
    BOOL UseSpecificShaderExt;
};
```

7.1.5.1 Members

`version`

Type: `NvU32`

The version of the `NVAPI_D3D12_PSO_HULL_SHADER_DESC` structure. You should assign `NV_HULL_SHADER_PSO_EXTENSION_DESC_VER` to this.

`psoExtension`

Type: `NV_PSO_EXTENSION`

The type of the PSO extension. You should assign `NV_PSO_HULL_SHADER_EXTENSION` to this.

`NumCustomSemantics`

Type: `NvU32`

The number of custom semantic elements, up to `NV_CUSTOM_SEMANTIC_MAX`, provided in the array pointer `pCustomSemantics`

`pCustomSemantics`

Type: `NV_CUSTOM_SEMANTIC`

Used to specify custom semantics for this shader

`UseWithFastGS`

Type: `BOOL`

Reserved

`UseSpecificShaderExt`

Type: `BOOL`

`TRUE` if minimal specific shaders are being created with NVAPI shader extensions as explained in "Creating the NVAPI PSO Extension for Only the Last World Space Shader" on page 43.

7.1.5.2 Remarks

This structure is used to specify parameters while creating a custom Hull shader in a PSO using NVAPI. This structure is used in the `NvAPI_D3D12_CreateGraphicsPipelineState()` function.

7.1.5.3 NVAPI_D3D12_PSO_DOMAIN_SHADER_DESC

```
struct NVAPI_D3D12_PSO_DOMAIN_SHADER_DESC
{
    NvU32 version;
    NV_PSO_EXTENSION psoExtension;
    NvU32 NumCustomSemantics;
    NV_CUSTOM_SEMANTIC *pCustomSemantics;
    BOOL UseWithFastGS;
    BOOL UseSpecificShaderExt;
};
```

7.1.5.4 Members

`version`

Type: `NvU32`

The version of the `NVAPI_D3D12_PSO_DOMAIN_SHADER_DESC` structure. You should assign `NV_DOMAIN_SHADER_PSO_EXTENSION_DESC_VER` to this.

`psoExtension`

Type: `NV_PSO_EXTENSION`

The type of the PSO extension. You should assign `NV_PSO_DOMAIN_SHADER_EXTENSION` to this.

`NumCustomSemantics`

Type: `NvU32`

The number of custom semantic elements, up to `NV_CUSTOM_SEMANTIC_MAX`, provided in the array pointer `pCustomSemantics`

`pCustomSemantics`

Type: `NV_CUSTOM_SEMANTIC`

Used to specify custom semantics for this shader

`UseWithFastGS`

Type: `BOOL`

Reserved

`UseSpecificShaderExt`

Type: `BOOL`

`TRUE` if minimal specific shaders are being created with NVAPI shader extensions as explained in "Creating the NVAPI PSO Extension for Only the Last World Space Shader" on page 43.

7.1.5.5 Remarks

This structure is used to specify parameters while creating a custom Domain shader in a PSO using NVAPI. This structure is used in the `NvAPI_D3D12_CreateGraphicsPipelineState()` function.

7.1.6 NVAPI_D3D12_PSO_GEOMETRY_SHADER_DESC

```
struct NVAPI_D3D12_PSO_GEOMETRY_SHADER_DESC
{
    NvU32                      version;
    NV_PSO_EXTENSION           psoExtension;
    BOOL                       UseViewportMask;
    BOOL                       OffsetRtIndexByVpIndex;
    BOOL                       ForceFastGS;
    BOOL                       DontUseViewportOrder;
    BOOL                       UseAttributeSkipMask;
    BOOL                       UseCoordinateSwizzle;
    NvAPI_D3D11_SWIZZLE_MODE *pCoordinateSwizzling;
    NvU32                      NumCustomSemantics;
    NV_CUSTOM_SEMANTIC         *pCustomSemantics;
    BOOL                       ConvertToFastGS;
    BOOL                       UseSpecificShaderExt;
};
```

7.1.6.1 Members

`version`

Type: `NvU32`

The version of the `NVAPI_D3D12_PSO_GEOMETRY_SHADER_DESC` structure. You should assign `NV_GEOMETRY_SHADER_PSO_EXTENSION_DESC_VER` to this.

`psoExtension`

Type: `NV_PSO_EXTENSION`

The type of the PSO extension. You should assign `NV_PSO_GEOMETRY_SHADER_EXTENSION` to this.

`UseViewportMask`

Type: `BOOL`

`FALSE` for custom semantics shaders

`OffsetRtIndexByVpIndex`

Type: `BOOL`

`FALSE` for custom semantics shaders

`ForceFastGS`

Type: `BOOL`

`TRUE` if Fast GS is to be used

If `TRUE`, the Geometry shader must be written with `maxvertexcount(1)` and must pass through input vertex 0 to the output without modification.

`DontUseViewportOrder`

Type: `BOOL`

`FALSE` by default

Use `TRUE` only for API ordering, which slows performance.

`UseAttributeSkipMask`

Type: `BOOL`

Reserved

`UseCoordinateSwizzle`

Type: `BOOL`

Reserved

`pCoordinateSwizzling`

Type: `NvAPI_D3D11_SWIZZLE_MODE`

Reserved

`NumCustomSemantics`

Type: NvU32

The number of custom semantic elements, up to NV_CUSTOM_SEMANTIC_MAX, provided in the array pointer pCustomSemantics

pCustomSemantics

Type: NV_CUSTOM_SEMANTIC

Used to specify custom semantics for this shader

ConvertToFastGS

Type: BOOL

Reserved

UseSpecificShaderExt

Type: BOOL

TRUE if minimal specific shaders are being created with NVAPI shader extensions as explained in "Creating the NVAPI PSO Extension for Only the Last World Space Shader" on page 43.

7.1.6.2 Remarks

This structure is used to specify parameters while creating a custom Geometry shader in a PSO using NVAPI. This structure is used in the NvAPI_D3D12_CreateGraphicsPipelineState() function.

7.2 FUNCTIONS

7.2.1 NvAPI_D3D12_QuerySinglePassStereoSupport

```
NVAPI_INTERFACE NvAPI_D3D12_QuerySinglePassStereoSupport(
  [in]     ID3D12Device *pDevice,
  [inout]  NV_QUERY_SINGLE_PASS_STEREO_SUPPORT_PARAMS
           *pQuerySinglePassStereoSupportedParams
);
```

7.2.1.1 Parameters

pDevice [in]

Type: ID3D12Device *

Pointer to the D3D12 device `ID3D12Device*`

`pQuerySinglePassStereoSupportedParams [inout]`

Type: `NV_QUERY_SINGLE_PASS_STEREO_SUPPORT_PARAMS*`

Pointer to the `NV_QUERY_SINGLE_PASS_STEREO_SUPPORT_PARAMS` structure

7.2.1.2 Return Value

Returns `NVAPI_OK` on success.

`bSinglePassStereoSupported` becomes `TRUE` if Single Pass Stereo is supported.

7.2.1.3 Remarks

This function determines whether the hardware supports the Single Pass Stereo feature.

7.2.2 NvAPI_D3D12_SetSinglePassStereoMode

```
NVAPI_INTERFACE NvAPI_D3D12_SetSinglePassStereoMode(
  [in]     ID3D12GraphicsCommandList* pCommandList,
  [in]     NvU32 numViews,
  [in]     NvU32 renderTargetIndexOffset,
  [in]     NvU8 independentViewportMaskEnable
);
```

7.2.2.1 Parameters

`pCommandList [in]`

Type: `ID3D12GraphicsCommandList *`

Pointer to the `D3D12_COMMAND_LIST_TYPE_DIRECT` Graphics Command List that will be used for setting Single Pass Stereo mode and further rendering.

`numViews [in]`

Type: `NvU32`

The number of views to render (2 for two eyes)

`renderTargetIndexOffset [in]`

Type: `NvU32`

The offset between the render targets of the different views:

- 1 if both views are generated on a render target array
- 0 if side-by-side views are generated on single render target

`independentViewportMaskEnable [in]`

Type: `NvU8`

Indicates whether the independent viewport mask is enabled:

- `false` if both views are generated on a render target array
- `true` if side-by-side views are generated on single render target

7.2.2.2 Return Value

Returns `NVAPI_OK` on success.

7.2.2.3 Remarks

This function sets the mode of the Single Pass Stereo feature.

Note that Single Pass Stereo state persists on a particular Command List till it is closed. The state is reset to default (disabled) for every newly created Command List. See "Enabling and Disabling the Single Pass Stereo Mode of the GPU" on page 55.

You must call `NvAPI_D3D12_QuerySinglePassStereoSupport()` to confirm that the current setup supports Single Pass Stereo before calling this set-function.

7.2.3 NvAPI_D3D12_CreateGraphicsPipelineState

```
NVAPI_INTERFACE NvAPI_D3D12_CreateGraphicsPipelineState(
  [in]     ID3D12Device *pDevice,
  [in]     const D3D12_GRAPHICS_PIPELINE_STATE_DESC *pPSODesc,
  [in]     NvU32 numExtensions,
  [in]     const NVAPI_D3D12_PSO_EXTENSION_DESC** ppExtensions,
  [out]    ID3D12PipelineState **ppPSO
);
```

7.2.3.1 Parameters

`pDevice [in]`

Type: `ID3D12Device *`

Pointer to the D3D12 device `ID3D12Device*`

`pPSODesc [in]`

Type: `const D3D12_GRAPHICS_PIPELINE_STATE_DESC *`

Pointer to regular the PSO descriptor. This should be filled exactly similar to creating a normal PSO without using NVAPI.

`numExtensions`

Type: `NvU32 [in]`

Number of PSO extensions being used for the current PSO being created.

`ppExtensions [in]`

Type: `const NVAPI_D3D12_PSO_EXTENSION_DESC**`

Pointer to the array of `NVAPI_D3D12_PSO_EXTENSION_DESC`. Each element points to one of the following structure variables (maximum of one of each type):

- `NVAPI_D3D12_PSO_VERTEX_SHADER_DESC`
- `NVAPI_D3D12_PSO_HULL_SHADER_DESC`
- `NVAPI_D3D12_PSO_DOMAIN_SHADER_DESC`
- `NVAPI_D3D12_PSO_GEOMETRY_SHADER_DESC`

Note that only PSO extensions for Single Pass Stereo are listed here. The full list of PSO extensions may contain more entries.

`ppPSO [out]`

Type: `ID3D12PipelineState **`

The address of a pointer to an output `ID3D12PipelineState` interface that is created by the function.

7.2.3.2 Return Value

Returns `NVAPI_OK` on success.

7.2.3.3 Remarks

This function creates a Graphics Pipeline State Object with custom semantics associated to the shader stages for which the additional information is provided via `NVAPI_D3D12_PSO_*_SHADER_DESC` structures.

Note that "Cached PSO" functionality is not supported with the Graphics Pipeline State Object created using this NvAPI. `GetCachedBlob()` should not be called with such a PSO.

Notice

The information provided in this specification is believed to be accurate and reliable as of the date provided. However, NVIDIA Corporation ("NVIDIA") does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This publication supersedes and replaces all other specifications for the product that may have been previously supplied.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and other changes to this specification, at any time and/or to discontinue any product or service without notice. Customer should obtain the latest relevant specification before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer. NVIDIA hereby expressly objects to applying any customer general terms and conditions with regard to the purchase of the NVIDIA product referenced in this specification.

NVIDIA products are not designed, authorized or warranted to be suitable for use in medical, military, aircraft, space or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on these specifications will be suitable for any specified use without further testing or modification. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to ensure the product is suitable and fit for the application planned by customer and to do the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this specification. NVIDIA does not accept any liability related to any default, damage, costs or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this specification, or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this specification. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA. Reproduction of information in this specification is permissible only if reproduction is approved by NVIDIA in writing, is reproduced without alteration, and is accompanied by all associated conditions, limitations, and notices.

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. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the NVIDIA terms and conditions of sale for the product.

VESA DisplayPort

DisplayPort and DisplayPort Compliance Logo, DisplayPort Compliance Logo for Dual-mode Sources, and DisplayPort Compliance Logo for Active Cables are trademarks owned by the Video Electronics Standards Association in the United States and other countries.

HDMI

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

ROVI Compliance Statement

NVIDIA Products that support Rovi Corporation's Revision 7.1.L1 Anti-Copy Process (ACP) encoding technology can only be sold or distributed to buyers with a valid and existing authorization from ROVI to purchase and incorporate the device into buyer's products.

This device is protected by U.S. patent numbers 6,516,132; 5,583,936; 6,836,549; 7,050,698; and 7,492,896 and other intellectual property rights. The use of ROVI Corporation's copy protection technology in the device must be authorized by ROVI Corporation and is intended for home and other limited pay-per-view uses only, unless otherwise authorized in writing by ROVI Corporation. Reverse engineering or disassembly is prohibited.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

Trademarks

NVIDIA and the NVIDIA logo 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