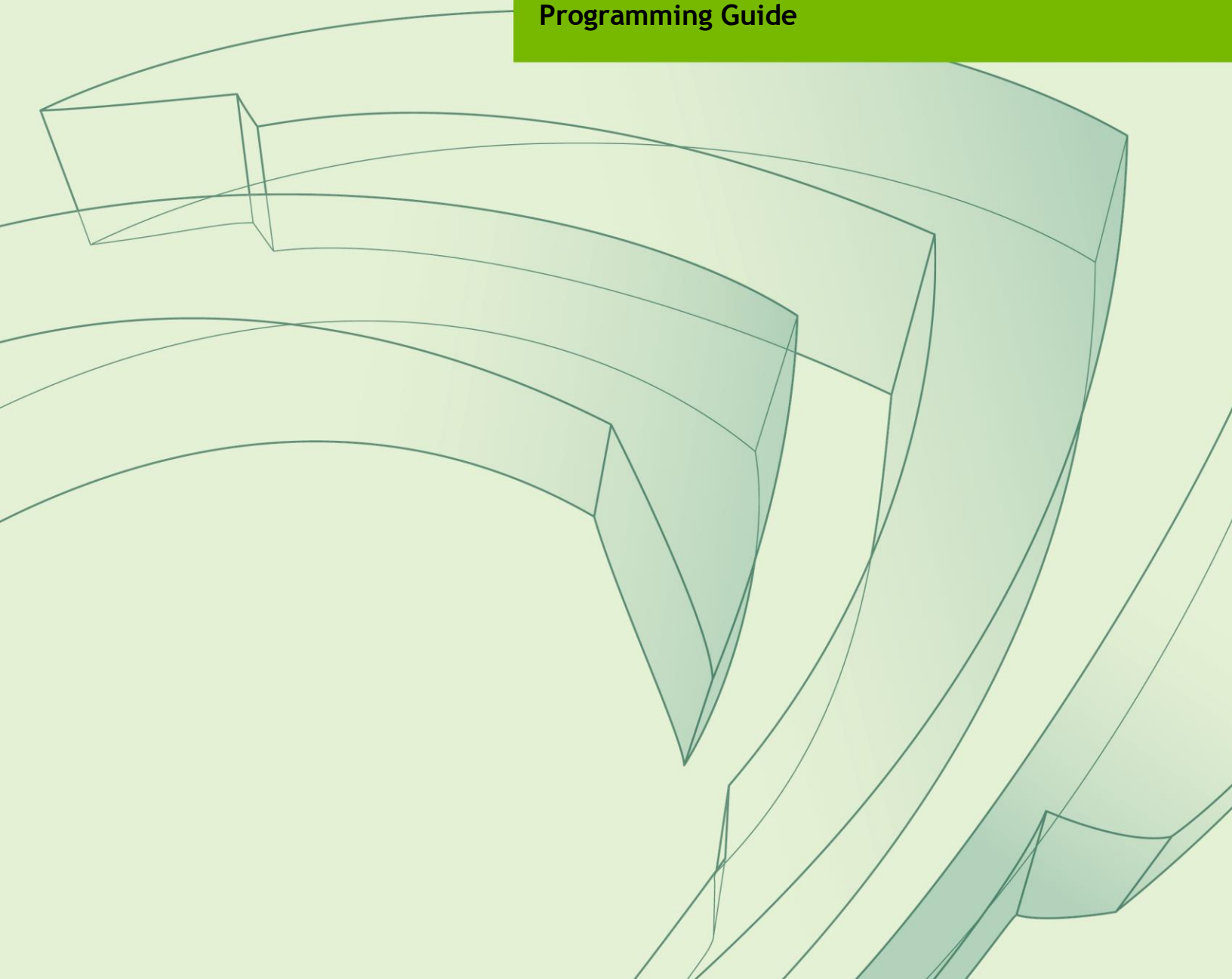
NVIDIA MULTI-PROJECTION SDK

PG-07866-001_v05 | March 2017

Programming Guide



DOCUMENT CHANGE HISTORY

PG-07866-001_v05

Version	Date	Authors	Description of Change
01	2015-11-19	NR	Initial release
02	2016-05-11	OA, AP	Updated for Pascal GPU features
03	2016-06-28	AP	Updated to reflect the changes in projection configuration functions
04	2016-10-28	AP	Updated for Direct X 12 support
05	2017-03-06	SB	Added a disclaimer about lack of support on Quadro GP100 cards

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1. INTRODUCTION TO THE NVIDIA MULTI-PROJECTION SDK

The NVIDIA Multi-Projection SDK enables applications to support hardware features of Maxwell and Pascal GPU families that improve rendering performance for virtual reality (VR) headsets.

The primary aim of the SDK is to allow applications to render fewer pixels while keeping image quality good enough for VR. To meet this aim, the SDK provides support for these advanced projection types:

- Multi-Resolution Shading (MRS)
- Lens Matched Shading (LMS)

Additionally, the SDK makes it easier to implement stereo rendering through the Single Pass Stereo hardware feature, by using either a regular planar projection or a lens-matched projection.

Note: Lens Matched Shading and Single Pass Stereo are not supported on Quadro GP100 cards.

1.1 VR HEADSET OPTICS

When you wear a VR headset, you view the display through an optical system of one or more lenses per eye. The lenses are essential for enabling you to comfortably focus on the screen and for expanding the apparent field of view. However, in most cases, the lenses also introduce pincushion distortion, causing straight lines in the image to appear curved inward. An example of this distortion in an Oculus DK2 lens is shown in Figure 1.

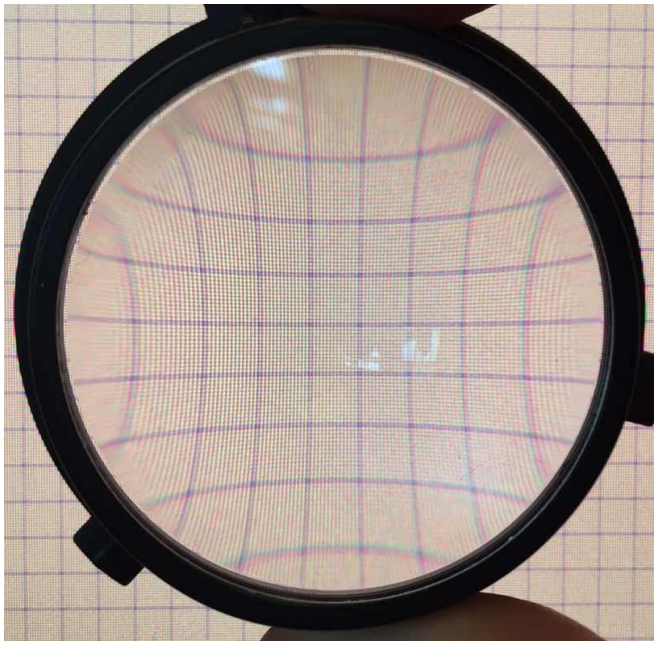


Figure 1. Pincushion Distortion and Chromatic Aberration in an Oculus DK2 Lens

To compensate for this behavior, VR applications use the GPU to render a barrel-distorted image, as shown in Figure 2. The barrel distortion in the image cancels out the pincushion distortion of the lens, so that when this image is presented on the headset display, the user perceives a geometrically correct 3D scene.

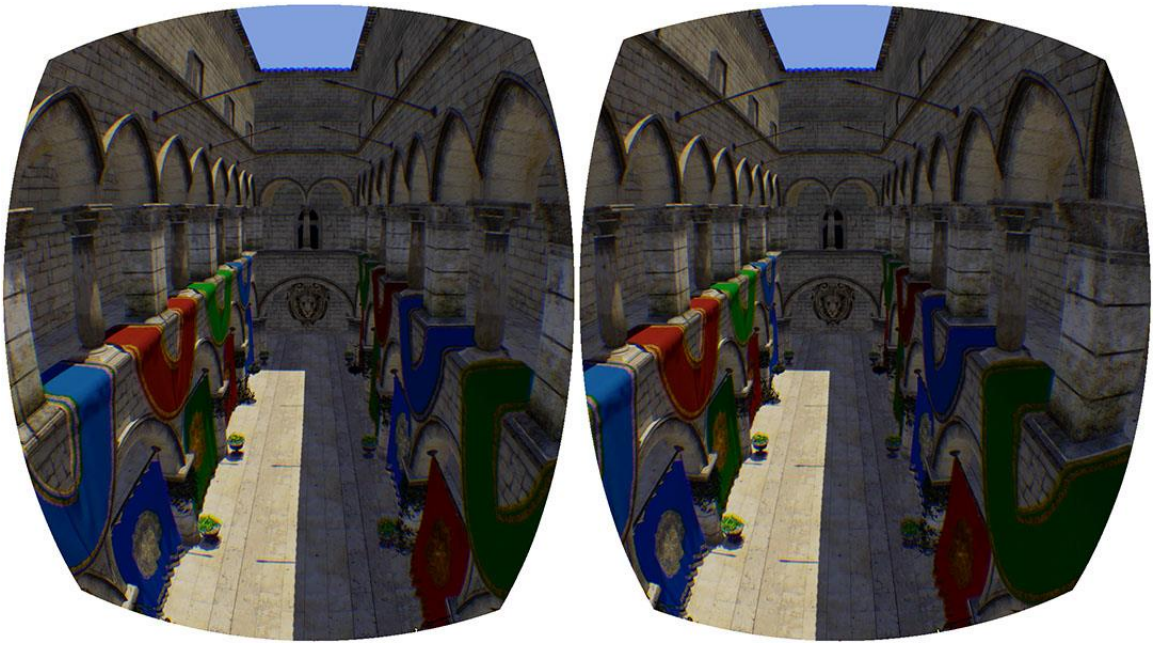


Figure 2. Barrel-Distorted Image as Presented in a Headset Display

However, GPUs cannot *natively* render to the nonlinear, barrel-distorted image space. Their rasterization hardware is designed to ensure that straight lines in 3D remain straight when projected on the screen. However, with barrel distortion, straight lines in 3D become curved on screen. VR applications overcome this problem by first rendering an ordinary perspective projection, then resampling to the distorted space as a post process as shown in Figure 3.

Figure 3. VR Rendering and Display Process

1.2 DISTORTION AND OVER-RENDERING

The barrel-distorted image that is ultimately presented in the headset has fewer pixels than the ordinary perspective-projection image that a VR application initially renders. In other words, when rendering the initial undistorted frame, we're rasterizing and shading many more pixels than can actually be displayed.

This excessive rendering isn't evenly distributed over the image. Figure 4 shows the shrinkage of edge regions by comparing the initial rendered image on the left with the final barrel-distorted image on the right:

- In the center of the frame, marked by green circles, the initial render matches almost 1:1 with the final distorted output.
- At the periphery of the frame, marked by red boxes, the barrel distortion shrinks the image significantly. In this area, the resolution of the distorted output is much lower than the initial render.

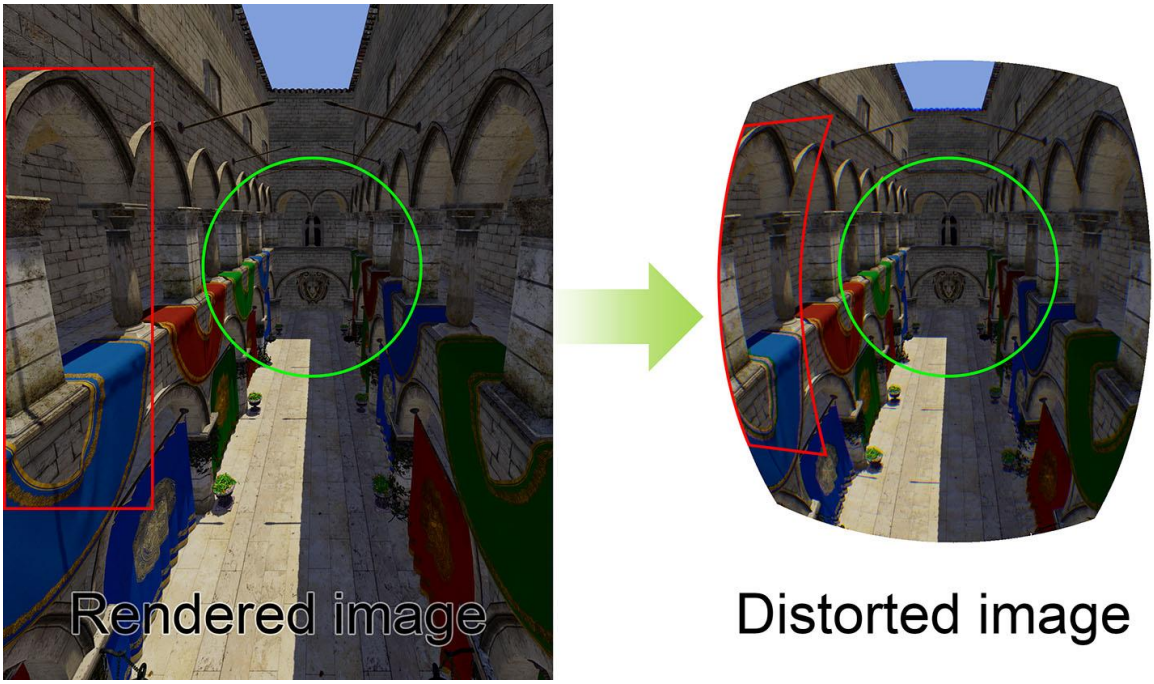


Figure 4. Comparison of Initial Rendered and Barrel-Distorted Image

The conclusion is that we're rendering an excessive number of pixels, that is oversampling, in the outer regions of the image where users don't spend a lot of time looking anyway.

In VR, to see the periphery of the display, you have to point your eyes without turning your head. Pointing your eyes without turning your head is uncomfortable to do for long periods of time. People instinctively turn their heads toward what they want to look at, bringing it to the middle of the image.

We could try to solve this efficiency issue by reducing the overall rendering resolution to do less oversampling in the outer regions, but then we would undersample in the central region, which is very noticeable.

1.3 CONTROLLING RENDERING RESOLUTION

Ideally, we would like to render every part of the image at just a high enough resolution to provide enough detail in the final distorted output, and no more. To achieve this aim, the initial undistorted render should have spatially varying resolution. The ideal resolution would be highest in the center and would then fall off smoothly toward the edges of the image.

Figure 5 shows a grid that represents the ideal rendering resolution in undistorted space. In this figure, the grid spacing is constant in post-distortion pixels.

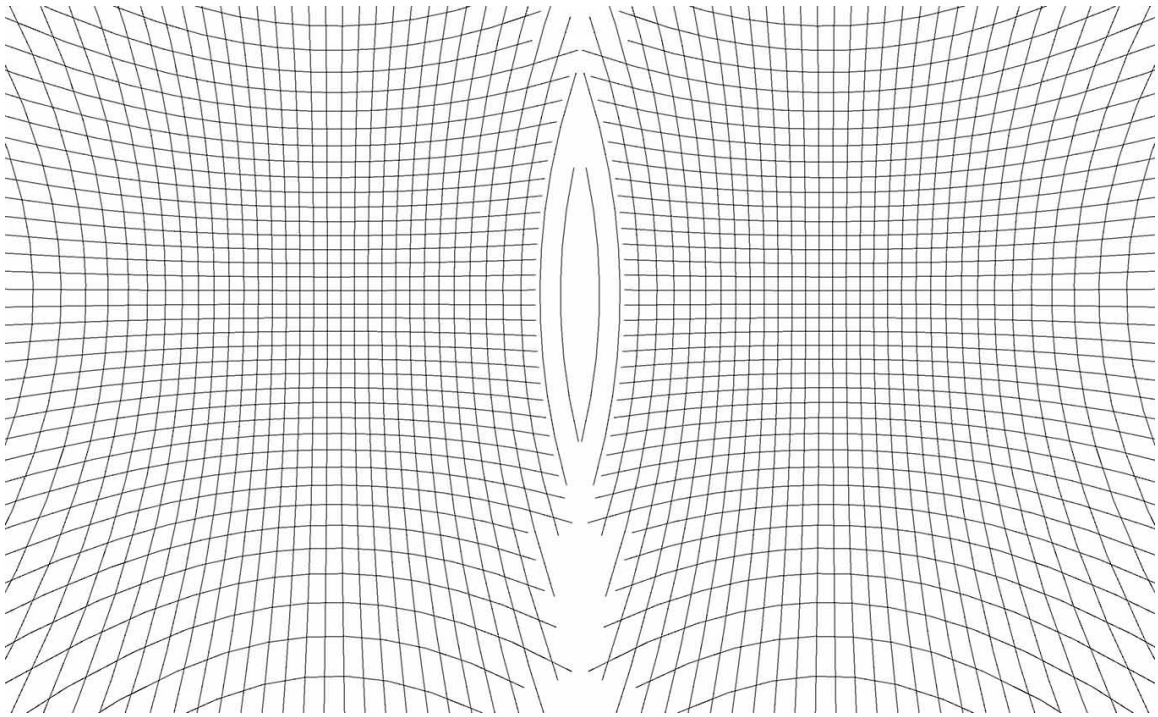


Figure 5. Ideal Rendering Resolution in Undistorted Space

Unfortunately, we can't actually render to an image with such a continuously varying resolution. The GPU rasterizer is built to work only with linear transforms, which means that straight lines in 3D space always remain straight in a 2D projection of that space. But we *can* use multiple projections with different properties to approximate this varying resolution. The SDK provides two ways to approximate varying resolution:

- Multi-Resolution Shading (MRS)
- Lens Matched Shading (LMS)

1.3.1 Multi-Resolution Shading (MRS)

Multi-Resolution Shading (MRS) uses multiple viewports combined with scissor rectangles to create a multi-resolution projection. The resolution inside each viewport is constant, but the resolutions of different viewports are different. We can arrange them in such a way that the viewports will represent one complicated projection of a scene without seams. The central area will have higher resolution than the peripheral areas.

Figure 6 shows a grid that represents multi-resolution rendering. In this figure, grid spacing is constant in shaded pixels. The purple lines represent the viewport boundaries.

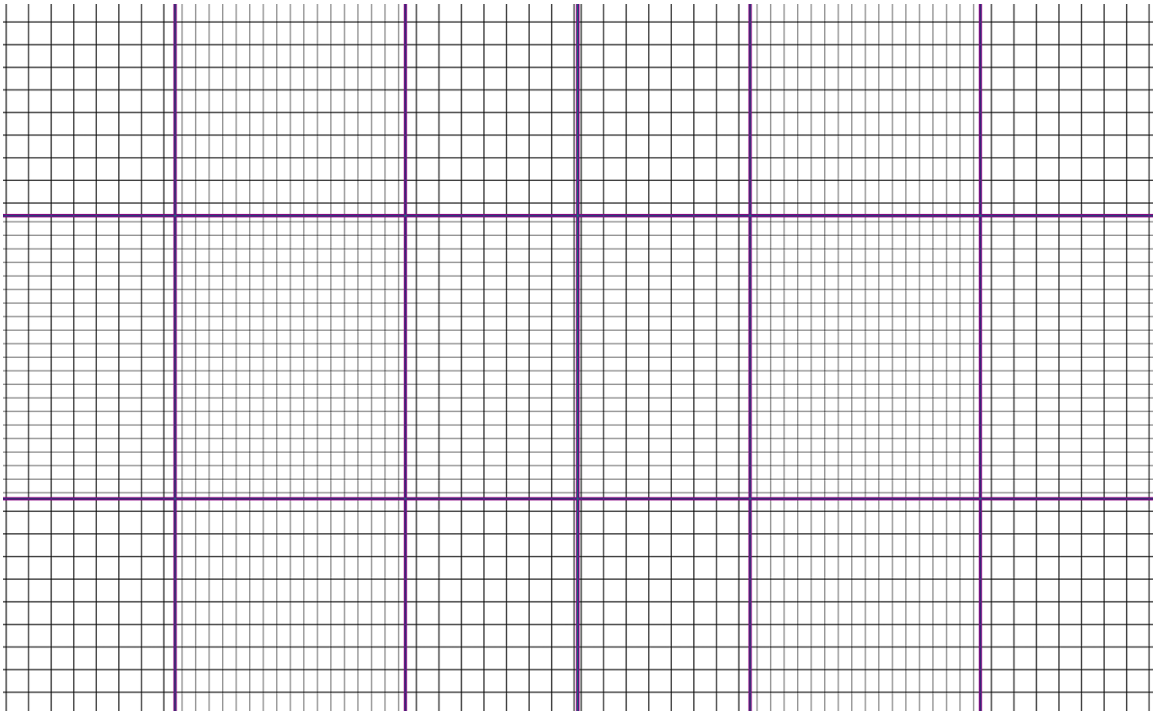


Figure 6.Multi-Resolution Rendering

The most important property of this projection is that all the viewports share the same view and projection transform matrices. The only different parts are the window transforms, namely, viewport scale and bias. As a result, a clip-space triangle can be sent into any of the viewports without modification, and doing so will produce a correct image of the triangle.

The Multi-Projection Acceleration feature enables NVIDIA Maxwell GPUs to send one triangle into multiple viewports. Using this feature, applications can render into a multi-resolution projection in a single rendering pass by computing the viewport mask in the geometry shader. A multi-resolution image has fewer pixels than an ordinary, single-resolution image would have. This lower number of pixels considerably reduces the pixel shading workload, and can significantly speed up rendering overall.

Figure 7 shows how viewports are laid out in a normal single-resolution image and a multi-resolution image.

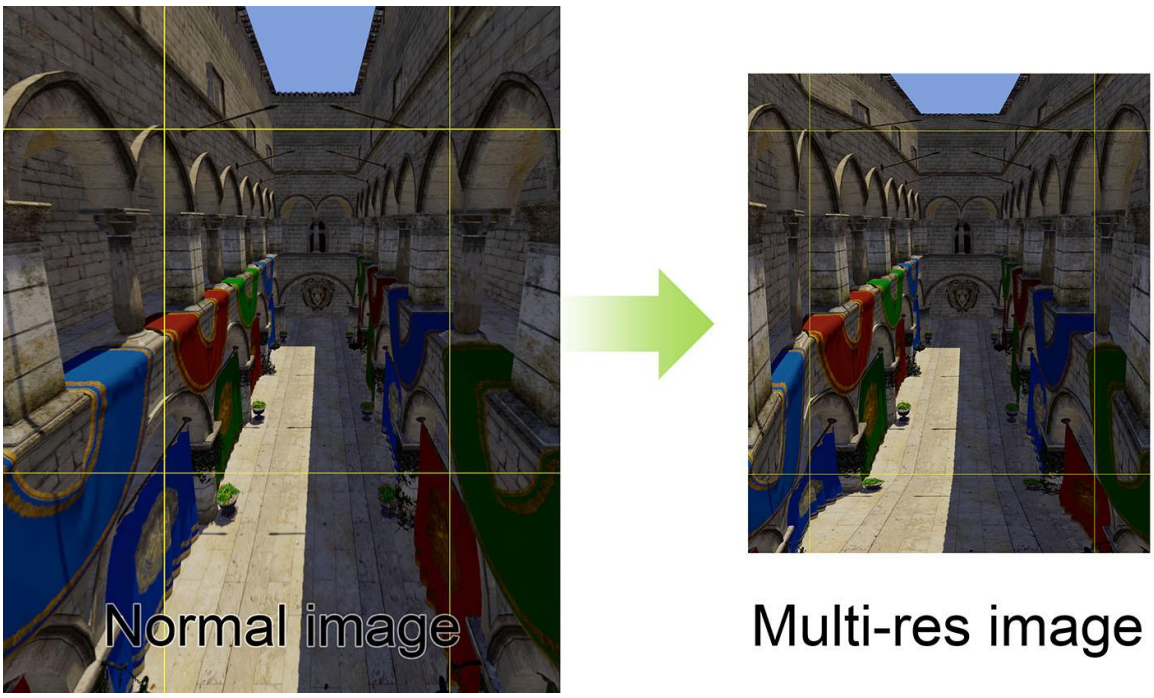


Figure 7. Viewports in Single-Resolution and Multi-Resolution Images

1.3.2 Lens-Matched Shading (LMS)

Lens-Matched Shading (LMS) uses multiple viewports combined with scissor rectangles and linear transformations to create a lens-matched projection. In a lens-matched projection, the triangle vertices are transformed before they are rasterized. The transformation applied is very simple:

$$w' = w + Ax + By$$

When such transformation with nonzero A and B factors is applied uniformly over the entire viewport, the image rotates in 3D space around an axis parallel to the screen, as shown in Figure 8.

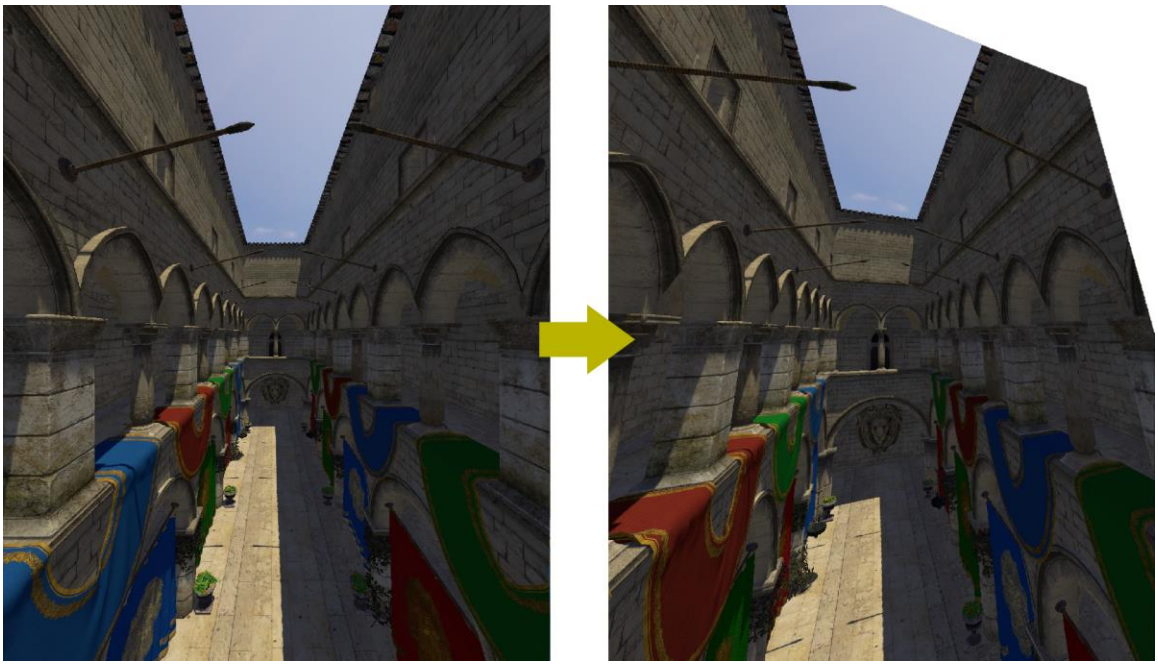


Figure 8. Applying a Uniform W Transformation to an Image

We can subdivide the screen into four viewports (top-left, top-right, bottom-left and bottom-right) and apply similar transformations to all viewports, the only difference being the signs of A and B factors. Let's choose the signs so that each viewport is rotated away from the viewer, making the image in the corners smaller, as shown in Figure 9.

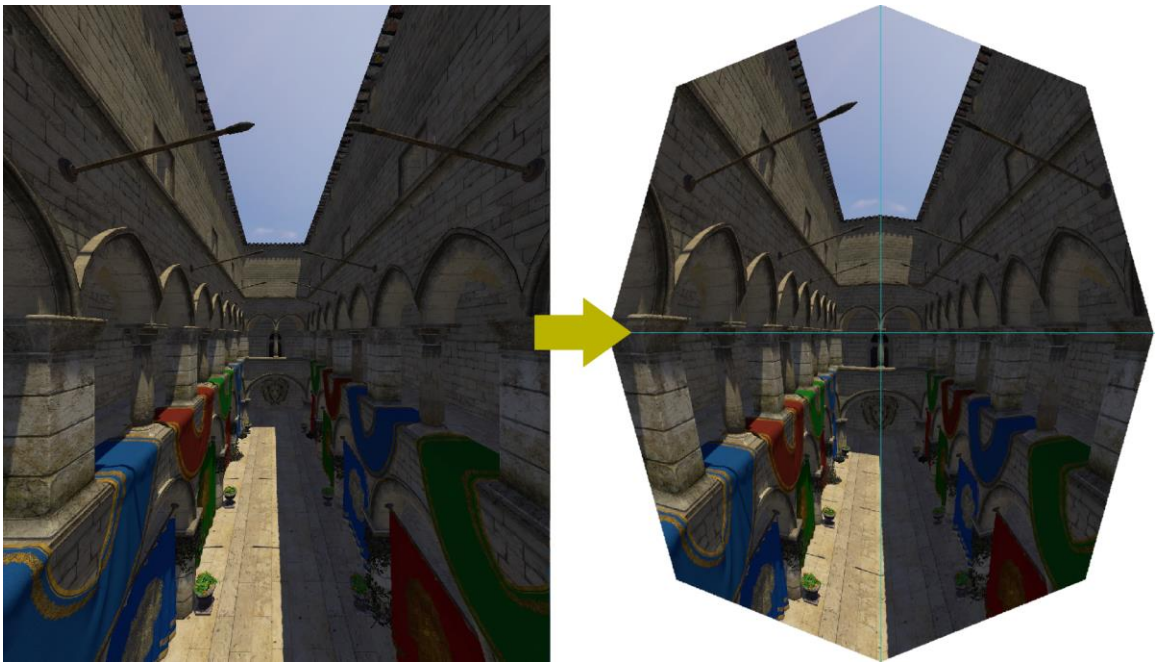


Figure 9. Applying W Transformations with Different Signs for A and B Factors to Different Viewports

This transformation cannot be produced by viewport multicast alone. Therefore, NVIDIA Pascal GPUs introduce a new hardware feature called Modified W that applies the W transformation before projecting triangles to window space. The transformation is based on A and B factors set before rendering. The factors can be different for each viewport, which allows us to warp the image in a way that resembles barrel distortion, as shown in Figure 9.

1.3.3 Postprocessing Multi-Projection Images

Multi-Resolution and Lens-Matched projections share one important property: they are continuous. Adjacent pixels belonging to different viewports will correspond to similar world-space projection rays and are likely to hit the same world-space objects. This feature allows hardware bilinear filtering to work normally across viewport boundaries. Many types of image-space processing that rely on neighboring pixels can also work without modification on a multi-resolution or lens-matched image.

However, in general, most image-space post-processing passes that an application performs will need to be modified to recognize the shape of the modified projection image. For example, blurring and filtering passes will need to adjust the size and shape of the kernel to compensate for the different resolutions in different parts of a multi-resolution image, or compute offsets in clip space instead of window space to work correctly with a lens-matched image. Passes that reconstruct a 3D position from the depth buffer, such as deferred shading or volumetric effects, will also need to be modified. This is discussed further in this document.

Finally, at some point near the end of the frame, the modified projection image must be resampled back to a planar image, or flattened, so that it can be submitted to the VR headset API (which doesn't recognize modified projections). This resampling pass can often be incorporated into an existing final post-processing pass. But even if it can't, the cost of this additional pass is minimal compared to the time saved by using MRS or LMS throughout the rest of the frame. UI elements such as HUDs can also be rendered at full-resolution at this stage, because they are unlikely to look good if they are rendered in a modified projection.

1.3.4 Integrating MRS and LMS Support in Applications

The SDK provides a unified interface for integrating Multi-Resolution and Lens-Matched projection support into applications. For convenience, regular planar projection is also defined with a compatible interface. As a result, both shader-side code and CPU-side code working with different projections can be similar or, in some cases, identical. The SDK includes functions to calculate viewports and scissor areas that must be set for a specific projection configuration, coordinate transforms that match the modified projection for shaders and for the CPU, and more.

Chapter 2. INTEGRATING MRS OR LMS INTO AN EXISTING 3D RENDERER

By adding support for multi-resolution and lens-matched shading to an existing 3D renderer, you can enhance the performance of the renderer. You get the greatest performance benefit when all rendering passes, from the initial depth pre-pass to the final stages of post-processing, can work with images in modified projection format.

To enable all rendering passes to work with images in modified projection format requires a significant amount of effort, especially if your renderer has many different passes.

Integrating MRS or LMS in an existing 3D renderer involves these tasks:

1. Setting up the projection
2. Performing the geometry rendering passes
3. Performing the image space passes
4. Performing the flattening pass

2.1 SETTING UP THE PROJECTION

When rendering a scene normally, the GPU performs these tasks:

1. Transforming the geometry in vertex or tessellation shaders using view and projection matrices
2. Culling and clipping the geometry
3. Normalizing the vertex positions by dividing their X, Y and Z components by W
4. Applying the window transform according to the viewport parameters

When applying the window transform, the GPU scales and offsets the positions so that X and Y are expressed in pixels relative to the render target origin and Z is fitted into the viewport depth range.

When you render a scene with the multi-projection SDK, you use view and projection matrices to transform the geometry in vertex or tessellation shaders as normal. However, the GPU processes the geometry and vertex positions differently. The fast geometry shader, viewport multicast hardware and modified W hardware together implement a more advanced version of the window transform. Furthermore, that transform has many more parameters than the viewport location and size.

The SDK defines configuration structures that describe the projection parameters for each supported projection type:

```
template<Projection Type> struct Configuration;
template<> struct Configuration<Projection::MULTI_RES> {...}
template<> struct Configuration<Projection::LENS_MATCHED> {...}
```

2.1.1 Multi-Resolution Projection Configuration

Multi-resolution projection is a version of planar projection that has more detail in some region in the middle and less detail in other areas. Therefore, to configure the multi-resolution projection, you must specify the following parameters:

- Location and size of the high-detail region, defined as fractions of the final render target into which the rendered scene will be flattened.
- Pixel densities in all regions relative to the final flattened image, defined separately in X and Y directions.
- Location of the MRS projection area on a render target, or its bounding box.

These parameters are defined in the following structure:

```
template<>
struct Configuration<Projection::MULTI_RES>
{
      enum { Width = 3, Height = 3, Count = Width * Height };
      float CenterWidth;
      float CenterHeight;
      float CenterX;
      float CenterY;
      float DensityScaleX[Width];
      float DensityScaleY[Height];
};
```

To illustrate the meanings of these parameters, consider a multi-resolution projection after it has been flattened into a regular projection because it is defined in terms of the final render target size. In this case, parameters optimized for an Oculus Rift headset lens distortion were used:

```
CenterWidth = 0.61
CenterHeight = 0.49
CenterX = 0.57
CenterY = 0.41
DensityScaleX = { 0.7, 1.0, 0.7 }
DensityScaleY = { 0.7, 1.0, 0.7 }
```

flattenedSize

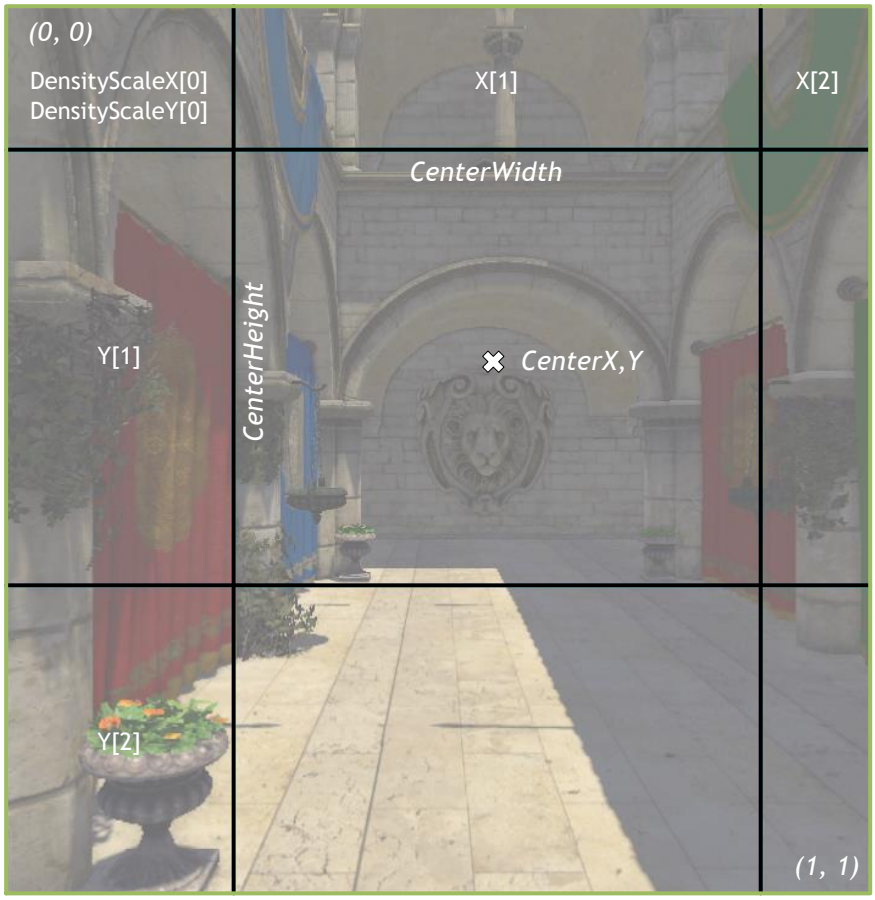


Figure 10. Multi-Resolution Projection after Flattening

The projection before flattening is very different, as shown in Figure 11. The rendered image is smaller than the final flattened image. Its position and size on the render target are stored in the `BoundingRect` field of the `ProjectionViewports` structure.

If you look closely, you can notice that the central part has the same size before and after upscaling, but the other parts have smaller size. These sizes are controlled by density scale parameters as follows:

- The central part has the same size because the `DensityScaleX[1]` and `DensityScaleY[1]` parameters that control the center scaling are set to 1.0.
- Other parts have a smaller size because the other density scales are less than 1.0, shrinking those parts of the image.

The scales don't necessarily have to be less than 1.0. They can be greater than 1.0, effectively applying supersampling to some parts of the image.

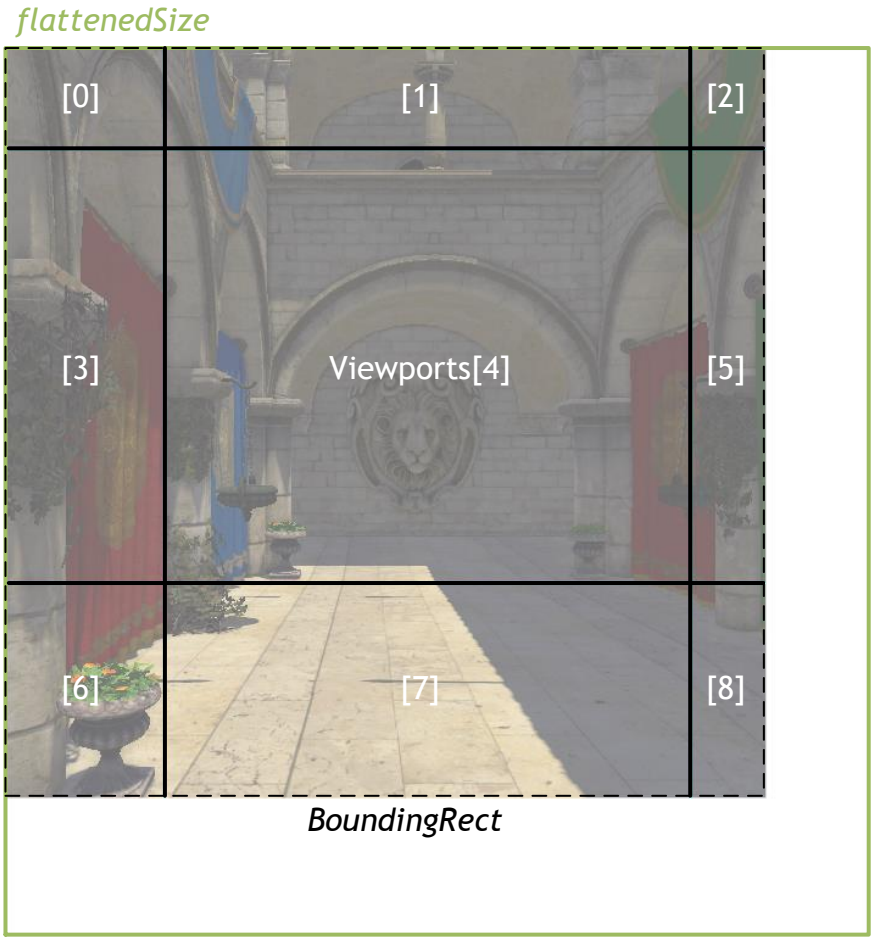


Figure 11. Multi-Resolution Projection Before Flattening

2.1.2 Lens Matched Projection Configuration

As described in Chapter 1, lens matched projection is produced by subdividing the image into 4 quadrants and applying different A and B coefficients to the W equation in each quadrant. We could use completely unrelated coefficients in each quadrant, but because we want the projection to be continuous and filterable, the coefficients have to satisfy two conditions:

- A(up) = A(down)
- B(left) = B(right)

This way, we have only 4 independent coefficients: A(left), A(right), B(up), B(down).

We can also control the scale of the lens matched projection. Similar to the equation coefficients, we could use completely separate X and Y scaling factors for all quadrants,

but the continuity constraint leaves only 4 independent factors: X(left), X(right), Y(up), Y(down).

The configuration is defined by the following structure:

```
template<>
struct Configuration<Projection::LENS_MATCHED>
{
      float WarpLeft;
      float WarpRight;
      float WarpUp;
      float WarpDown;

      float RelativeSizeLeft;
      float RelativeSizeRight;
      float RelativeSizeUp;
      float RelativeSizeDown;
};
```

The meaning of these parameters is illustrated in Figure 12. Note that the octagon doesn't fill the entire bounding box. Instead, it is just centered inside that box or clipped if it doesn't fit.

The four `RelativeSize`*XXX* parameters are defined as fractions of the flattened image size, just like the center size or pixel densities in the multi-resolution projection configuration. Note that the presets supplied with the SDK are calculated assuming that the size of the flattened image is the recommended size of the render target provided by VR HMD runtime. If any other flattened size is used, the image will be oversampled or undersampled.

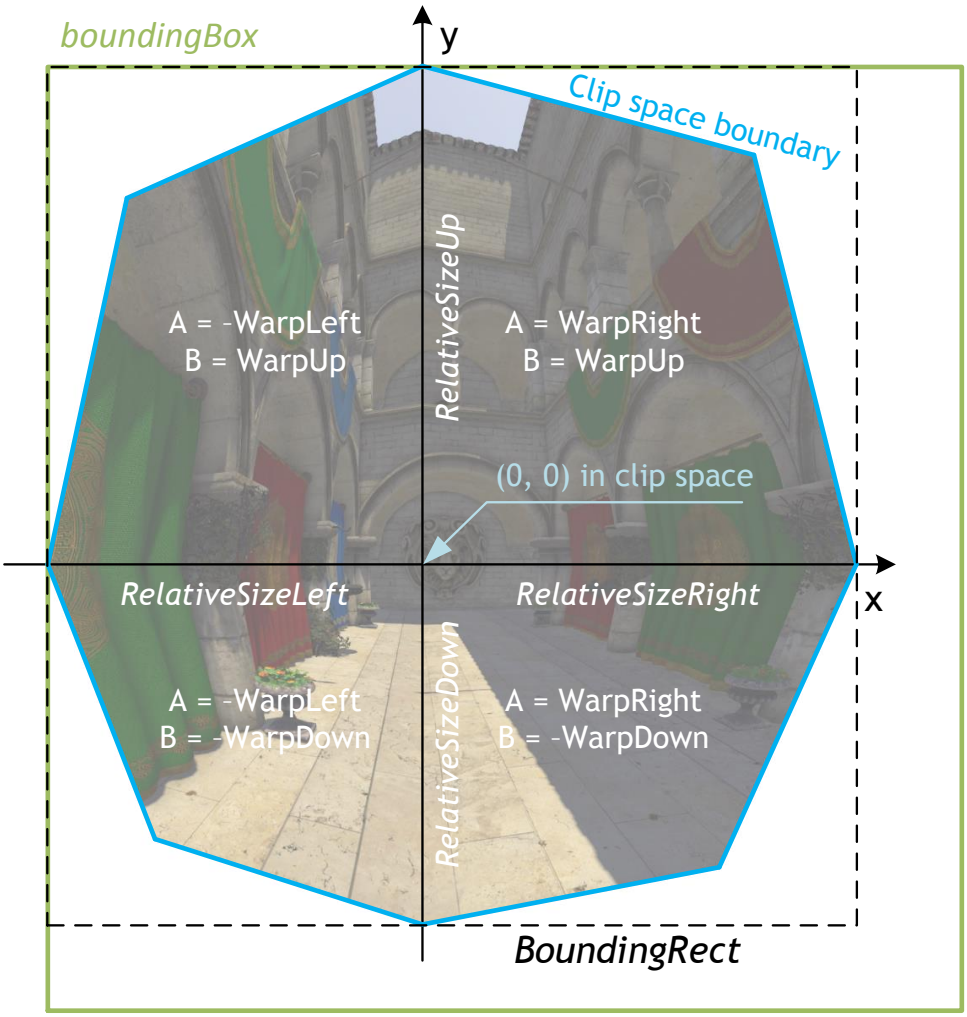


Figure 12. Lens Matched Projection Before Flattening

Because the Lens-Matched projection modifies W values, it modifies depth. This fact has significant implications on many rendering passes, as discussed in the next section. For reasons not fully understood, it also results in catastrophic loss of depth accuracy when a regular perspective projection matrix is used. This loss presents itself as Z-fighting at moderate distances, self-shadowing and noise artifacts in SSAO, and misplaced shadows when deferred shading is used. The solution to that is to use reverse infinite projection, where depth is computed as 1/W instead of traditional Z/W.

2.1.3 Computing Viewports and Constant Buffer Data

When you have a projection configuration, you must compute the rendering state for configuring the GPU, and data for constant buffers that are required for correct operation of the projection shaders.

All this information is normally stored in a `Data` structure:

```
template<Projection Type>
struct Data
{
      ProjectionViewports           Viewports;
      FastGSCBData                  FastGsCbData;
      RemapCBData                   RemapCbData;
};
```

The contents of `FastGSCBData` and `RemapCBData` structures are not important to applications. It is sufficient to pass these structures to the shaders through constant buffers.

Viewports are important to applications. Viewports are defined in a structure as follows:

```
struct ProjectionViewports
{
      enum { MaxCount = 16 };

      Viewport          Viewports[MaxCount];
      ScissorRect       Scissors[MaxCount];
      int               NumViewports;

      ScissorRect       BoundingRect;
      Float2            FlattenedSize;
}
```

To compute the `Data` structure from a projection configuration, call the `CalculateViewportsAndBufferData` function, passing it the configuration, the flattened image size, and the bounding box to place the projection:

```
template<Projection Type>
void CalculateViewportsAndBufferData(
      const Float2& flattenedSize,
      const Viewport& boundingBox,
      const Configuration<Type>& configuration,
      Data<Type>& ref_data);
```

2.2 GEOMETRY PASSES

Geometry passes are passes that render geometry into the main eye views, for example:

- Depth pre-pass
- G-buffer fill
- Forward rendering of transparencies and particles

Performing geometry passes involves these tasks:

1. Creating appropriate geometry shaders and enabling them during rendering
2. Configuring the graphics pipeline with the viewports, scissor rectangles and maybe Modified W factors that correspond to the specific projection
3. Setting up the depth or stencil buffer to cull pixels outside of the octagon so that you get maximum efficiency from the LMS projection

2.2.1 Creating Geometry Shaders

The SDK provides implementations of generic geometry shaders for all regular rendering cases, including:

- Multi-resolution projections
- Lens matched projections
- Lens matched projections in combination with Single Pass Stereo

To generate a geometry shader for your needs, include these files in a blank HLSL file:

- `nv_vr.hlsli`
- `nv_vr_geometry_shader.hlsli`

The specific projection to use and some other options are controlled by setting preprocessor symbols before including these files.

`NV_VR_PROJECTION`

Selects the projection type. The following values are accepted:

- `NV_VR_PROJECTION_PLANAR`
- `NV_VR_PROJECTION_MULTI_RES`
- `NV_VR_PROJECTION_LENS_MATCHED`

`NV_VR_FASTGS_FUNCTION_NAME`

Controls the function name for the generated geometry shader. The default value is `main`.

`NV_VR_FASTGS_PASSTHROUGH_STRUCT`

Sets the name of the structure with attributes that must be passed through to the pixel shader. This structure must include a system-interpreted position attribute.

`NV_VR_FASTGS_POSITION_ATTRIBUTE`

Sets the name of the position attribute in the pass-through structure.

`NV_VR_FASTGS_CONSTANT_DATA`

Sets the expression that will be used in the geometry shader to access the `FastGSCBData` structure. Normally, it is the name of a global constant.

`NV_VR_FASTGS_EMULATION`

A Boolean that enables generation of a regular geometry shader instead of a fast geometry shader.

Set it to 1 to experiment with extended projections on a machine without a compatible GPU. The rendered image will be equivalent, but rendering will be significantly slower.

This mode is also useful to ensure compatibility with graphics debuggers that do not properly support NVIDIA D3D extensions.

`NV_VR_FASTGS_SINGLE_PASS_STEREO`

A Boolean that defines whether the geometry shader is intended for use with single-pass stereo. It is compatible only with Planar and Lens Matched projections.

`NV_VR_FASTGS_X_RIGHT_ATTRIBUTE`

Sets the name of the attribute in the pass-through structure that holds the X coordinate for the right eye. Required if `NV_VR_FASTGS_SINGLE_PASS_STEREO` is set to 1.

`NV_VR_FASTGS_VIEWPORT_MASK_COMPATIBILITY`

A Boolean that makes the FastGS shader write the viewport mask to `SV_ViewportArrayIndex` attribute instead of an `NV_VIEWPORT_MASK` attribute. Write the viewport mask to `SV_ViewportArrayIndex` attribute to achieve compatibility with older driver versions that do not recognize custom semantics.

`NV_VR_FASTGS_OUTPUT_VIEWPORT_INDEX`

A Boolean that makes the FastGS shader output a dummy viewport index to the `SV_ViewportArrayIndex` attribute in addition to outputting the viewport mask to the `NV_VIEWPORT_MASK` attribute. This mode is useful in Single Pass Stereo rendering mode when the pixel shader needs to read `SV_ViewportArrayIndex` to find the eye index. If the pixel shader reads this attribute but the geometry shader does not write it, the D3D debug runtime issues a shader linkage error, but the pixel shader still works correctly.

The following example shows a typical geometry shader file for lens matched projection:

```
#define NV_VR_PROJECTION                    NV_VR_PROJECTION_LENS_MATCHED
#define NV_VR_FASTGS_PASSTHROUGH_STRUCT     VertexShaderOutput
#define NV_VR_FASTGS_POSITION_ATTRIBUTE     posClip
#define NV_VR_FASTGS_CONSTANT_DATA          g_vrFastGSCBData
```

```
#include "nv_vr.hlsli"

struct VertexShaderOutput
{
     Vertex vtx;
     float4 posClip : SV_Position;
};

cbuffer MyConstantBuffer : register(b0)
{
     NV_VR_FastGSCBData g_vrFastGSCBData;
};

#include "nv_vr_geometry_shader.hlsli"
```

Use the `gs_5_0` shader profile to compile these shaders in the same ways any other geometry shader.

However, how to create a D3D shader object depends on the specific configuration used by the geometry shader and on the D3D version being used:

- In emulation mode, which is specified by setting `NV_VR_FASTGS_EMULATION` to 1, use the regular method for the D3D version being used:
  - For a D3D11 device, use the regular `CreateGeometryShader` method.
  - For a D3D12 device, use the regular `CreateGraphicsPipelineState` method.

  When using the shaders in emulation mode, omit any NVAPI calls that configure the VR-related hardware features such as Modified W or Single Pass Stereo.
- In Maxwell compatibility mode on D3D11, which is specified by using multi-resolution projection with `NV_VR_FASTGS_VIEWPORT_MASK_COMPATIBILITY` set to 1, use the following NVAPI call:

```
NvAPI_D3D11_CREATE_FASTGS_EXPLICIT_DESC FastGSArgs =
{
         NVAPI_D3D11_CREATEFASTGSEXPLICIT_VER,
         NV_FASTGS_USE_VIEWPORT_MASK
};
ID3D11GeometryShader* pGeometryShader = nullptr;
NvAPI_D3D11_CreateFastGeometryShaderExplicit(pDevice, code,
codeSize, nullptr, &FastGSArgs, &pGeometryShader);
```

- In Maxwell compatibility mode on D3D12, use the following NVAPI call:

```
NVAPI_D3D12_PSO_CREATE_FASTGS_EXPLICIT_DESC FastGSExtension = {};
FastGSExtension.baseVersion = NV_PSO_EXTENSION_DESC_VER;
FastGSExtension.psoExtension = NV_PSO_EXPLICIT_FASTGS_EXTENSION;
FastGSExtension.version = NV_FASTGS_EXPLICIT_PSO_EXTENSION_VER;
FastGSExtension.flags = NV_FASTGS_USE_VIEWPORT_MASK;

NVAPI_D3D12_PSO_EXTENSION_DESC extensions[] = {
         &FastGSExtension
```

```
};

ID3D12PipelineState* pPSO = nullptr;
NvAPI_D3D12_CreateGraphicsPipelineState(pDevice, &desc,
ARRAYSIZE(extensions), extensions, &pPSO);
```

- In all other cases when running on D3D11, use the following NVAPI call:

```
NV_CUSTOM_SEMANTIC semantics[] = {
{ NV_CUSTOM_SEMANTIC_VERSION, NV_X_RIGHT_SEMANTIC,
"NV_X_RIGHT", false, 0, 0, 0 },
{ NV_CUSTOM_SEMANTIC_VERSION, NV_VIEWPORT_MASK_SEMANTIC,
NV_VIEWPORT_MASK", false, 0, 0, 0 }
};

NvAPI_D3D11_CREATE_GEOMETRY_SHADER_EX ExDesc = {};
ExDesc.version = NVAPI_D3D11_CREATEGEOMETRYSHADEREX_2_VERSION;
ExDesc.ForceFastGS = true;
ExDesc.NumCustomSemantics = ARRAYSIZE(semantics);
ExDesc.pCustomSemantics = semantics;

ID3D11GeometryShader* pGeometryShader = nullptr;
NvAPI_D3D11_CreateGeometryShaderEx_2(pDevice, code, codeSize,
nullptr, &ExDesc, &pGeometryShader);
```

- In all other cases when running on D3D12, use the following NVAPI call:

```
NV_CUSTOM_SEMANTIC semantics[] = {
{ NV_CUSTOM_SEMANTIC_VERSION, NV_X_RIGHT_SEMANTIC,
"NV_X_RIGHT", false, 0, 0, 0 },
{ NV_CUSTOM_SEMANTIC_VERSION, NV_VIEWPORT_MASK_SEMANTIC,
NV_VIEWPORT_MASK", false, 0, 0, 0 }
};

NVAPI_D3D12_PSO_GEOMETRY_SHADER_DESC FastGSExtension = {};
FastGSExtension.baseVersion = NV_PSO_EXTENSION_DESC_VER;
FastGSExtension.psoExtension = NV_PSO_GEOMETRY_SHADER_EXTENSION;
FastGSExtension.version = NV_GEOMETRY_SHADER_PSO_EXTENSION_DESC_VER;
FastGSExtension.ForceFastGS = true;
FastGSExtension.NumCustomSemantics = ARRAYSIZE(semantics);
FastGSExtension.pCustomSemantics = semantics;

NVAPI_D3D12_PSO_EXTENSION_DESC extensions[] = {
         &FastGSExtension
};

ID3D12PipelineState* pPSO = nullptr;
NvAPI_D3D12_CreateGraphicsPipelineState(pDevice, &desc,
ARRAYSIZE(extensions), extensions, &pPSO);
```

Notes:

- The semantic names used in the `semantics` array must match the actual attribute semantics produced by the geometry shader.
- The `NV_X_RIGHT` semantic can be omitted if the application is not using Single Pass Stereo.

2.2.2 Setting Other Rendering State

Besides geometry shaders, you must set other rendering states:

- Viewports
- Scissor rectangles
- Modified W factors for a lens matched projection

2.2.2.1 Setting Viewports and Scissor Rectangles

The `ProjectionViewports` structure has all the necessary information to set the viewports and scissor rectangles. Structures defined in `nv_vr.h` map directly to D3D structures as follows:

- The `Viewport` structure maps directly to the `D3D11_VIEWPORT` structure.
- The `ScissorRect` structure maps directly to the `D3D11_RECT` structure.

To set the viewports and scissor rectangles:

1. Copy the `Viewports` and `Scissors` arrays into proper D3D structures.
2. Pass the D3D structure into which you copied the `Viewports` array to `RSSetViewports`.
3. Pass the D3D structure into which you copied the `Scissors` array to `RSSetScissorRects`.

CAUTION: Do not use a type cast to pass the `Viewports` array to `RSSetViewports` and the `Scissors` array to `RSSetScissorRects`. Using a type cast for this purpose is unsafe and is incompatible with techniques such as temporal antialiasing that requires viewports to be jittered.

2.2.2.2 Setting Modified W Factors

To set Modified W factors, call the `NvAPI_D3D(12)_SetModifiedWMode` NVAPI function, as shown in this example:

```
NV_MODIFIED_W_PARAMS wParams = { 0 };
wParams.version = NV_MODIFIED_W_PARAMS_VER;
wParams.numEntries = 4;
wParams.modifiedWCoefficients[0].fA = -WarpLeft;
wParams.modifiedWCoefficients[0].fB = WarpUp;
wParams.modifiedWCoefficients[1].fA = WarpRight;
```

```
wParams.modifiedWCoefficients[1].fB = WarpUp;
wParams.modifiedWCoefficients[2].fA = -WarpLeft;
wParams.modifiedWCoefficients[2].fB = -WarpDown;
wParams.modifiedWCoefficients[3].fA = WarpRight;
wParams.modifiedWCoefficients[3].fB = -WarpDown;
NvAPI_D3D_SetModifiedWMode(pDevice, &wParams); // D3D11
NvAPI_D3D12_SetModifiedWMode(pCommandList, &wParams); // D3D12
```

To disable the Modified W mode, call the same function but set the `numEntries` field to 0 and do not fill the `modifiedWCoefficients` array.

2.2.3 Culling Pixels for Lens Matched Shading

Modifying the W component of the vertex position, which is necessary to implement LMS, affects the geometry of the effective clip space in a subtle way. Triangles that were outside the clip space in the X or Y direction are now visible in the corners of the projection area just outside of the octagon.

The FastGS shader can conservatively remove triangles that were outside the clip space. Specifically, it removes triangles that should be completely invisible, but does not remove triangles that cross the octagon boundary. The results of such triangle-level culling are shown in Figure 13, which shows a lens matched projection before flattening. The figure shows how the detailed meshes such as curtains disappear outside the octagon, but the larger meshes such as the floor are still present.

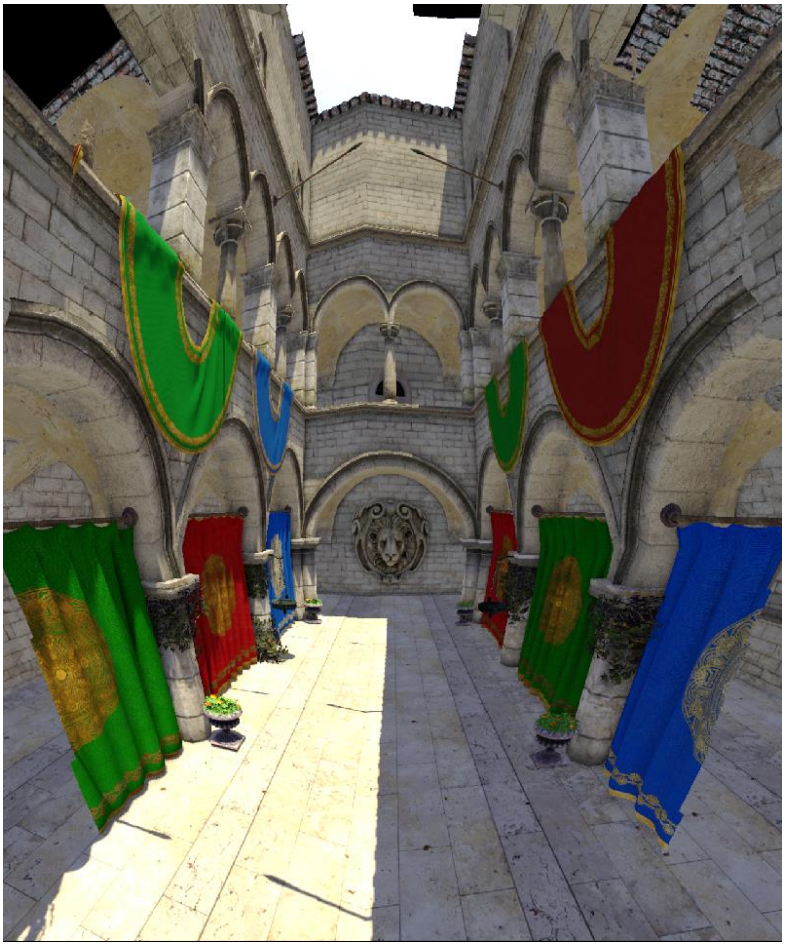


Figure 13. Lens-Matched Projection with Triangle Culling Only

You could remove extra pixels outside of the octagon by computing the clip space position in the pixel shader and discarding a pixel if it is out of bounds. But culling pixels that way is not likely to impove performance much because the culled pixels are still rasterized and pixel shader threads are still launched.

A more efficient solution culls pixels earlier in the graphics pipeline by using a stencil test or a depth test.

2.2.3.1 Culling Pixels with a Stencil Test

To cull pixels with a stencil test:

1. Clear the stencil buffer.
2. Draw a full-screen quad by using the lens-matched projection. This produces the octagon on screen and writes a constant into the stencil buffer for the octagon.
3. Use stencil test to draw the geometry only inside the octagon.

2.2.3.2 Culling Pixels with a Depth Test

Culling pixels with a depth test is more efficient and universal than culling pixels with a stencil test. The application is not restricted to using depth buffer formats that include stencil.

To cull pixels with a depth test:

1. Clear the depth buffer by using the near plane depth instead of far plane depth.

   The depth test now fails for all geometry.

2. Draw a full screen quad at the far plane using the lens matched projection in overwrite-depth mode.

   This will fill the octagon with far plane depth, allowing other geometry to be visible inside it.

   When a 1/W projection is used, the far plane is at infinity, but that is not a problem because the vertex shader can just write `0` into the `SV_Position.z` component to specify the far plane.

   To improve depth accuracy, set the minimum and maximum viewport depth to a single value that is equal to the far plane depth. Setting them to the far plane depth makes the octagon have the same depth in all its pixels.

3. Draw the scene using a regular depth test.

2.3 IMAGE-SPACE PASSES

Individualized treatment is required for passes that operate in image space, for example:

- SSAO
- Tile-based deferred lighting
- Screen-space reflections
- Depth-based atmospheric effects
- Depth of field
- Motion blur
- Temporal antialiasing
- Bloom

Such individualized treatment is required to make image-space passes aware of the non-uniform layout of the modified projection image that they're operating on.

To enable image-space passes to work with modified projection images, the multi-projection SDK defines HLSL functions that implement the clip-to-window and window-to-clip transforms for all supported projection types. The parameters for the

transform must be supplied by a constant buffer. The SDK defines the structure of these parameters and provides routines to calculate them. You can incorporate the data into an existing constant buffer if convenient, or create a new buffer.

2.3.1 Position Mapping in a Multi-Resolution Projection

For multi-resolution shading, a mapping that translates X and Y (or U and V) from one space to another is sufficient. Such a mapping is sufficient because the MRS projection only affects the X and Y components, leaving the Z and W components unmodified.

To translate 2D coordinates from one space to another, use the functions that operate on UV only:

```
float2 NV_VR_MapUV_LinearToVR(NV_VR_RemapCBData cbData, float2
linearUV);

float2 NV_VR_MapUV_VRToLinear(NV_VR_RemapCBData cbData, float2
vrUV);
```

The behavior of these functions is identical to the behavior of the same functions in the older NVIDIA Multi-Resolution SDK.

You can also use these functions for other projection types, and they are sufficient for simple 2D passes such as flattening or blurs.

2.3.2 Position Mapping in a Lens-Matched Projection

For lens-matched shading, a mapping that translates X and Y (or U and V) from one space to another is insufficient. To work with 3D positions in a lens matched projection, you must perform the full modified projection transform or its inverse by using these functions:

```
float4 NV_VR_MapWindowToClip(
     NV_VR_RemapCBData cbData,
     float3            windowPos,
     bool              normalize = true);

float3 NV_VR_MapClipToWindow(
     NV_VR_RemapCBData cbData,
     float4            clipPos,
     bool              normalize = true);
```

These functions match the transform that is done by the GPU, including the viewport position and size, up to math precision limits.

These functions are defined for all supported projection kinds. Shader code that uses these functions is likely to work without modifications with Planar, Multi-Resolution and Lens-Matched projections.

2.3.3 Making Image-Space Passes Compatible with Modified Projections

Image space passes often operate on the world space positions or window space positions of surfaces represented by individual pixels. The only way of obtaining this position information that works across all types of modified projections is to use the functions that the SDK provides for this purpose:

- The only way to obtain a world space position for a pixel is to use the `NV_VR_MapWindowToClip` function.
- Similarly, the only way to obtain a window space position is to use the `NV_VR_MapClipToWindow` function.

An image space pass can be perfumed with or with a conversion of full 3-D positions.

2.3.3.1 Image Space Pass with Conversion of Full 3D Positions

An image space pass that includes a conversion of full 3D positions follows these steps:

1. Draw a full screen quad or do a full screen compute dispatch.
2. Obtain the pixel coordinates and depth information, which collectively define a window space position:
   a) Read or compute the pixel coordinates:
      – For a full screen quad, read them from `SV_Position.xy`.
      – For a compute dispatch, compute them as `(SV_DispatchThreadID.xy + 0.5)`.
   b) If the pass uses depth information, read the depth buffer by using the pixel coordinates.
3. Convert the window space position obtained in the previous step to a clip space position by using the `NV_VR_MapWindowToClip` function.
4. If the pass uses world space positions or view space positions, compute the relevant position:
   - To compute the world space position, multiply the clip space position by the inverse of the view-projection matrix.
   - To compute the view space position, multiply the clip space position by the inverse of the projection matrix.
5. Normalize the final position by dividing its X, Y, and Z components by the W component.

Note: Normalization is necessary only if the pass works with the individual coordinates of the homogenous position or requires a position in regular 3D space. There is no need to normalize positions before doing vector-matrix multiplications in homogenous space.

2.3.3.2 Image Space Pass without Conversion of Full 3D Positions

A lens-matched projection might not require conversion of full 3D positions between coordinate spaces. In such a projection, you can deal with coordinates in a way that makes the lens-matched projection similar to the multi-resolution projection. You need only to map X and Y coordinates between spaces. The depth remains the same.

To deal with coordinates in a lens-matched projection that doesn't require converting full 3D positions, add a separate rendering pass that performs these operations:

- Copies the depth buffer into a new render target
- Performs the `MapWindowToClip` conversion on pixel data
- Outputs the Z component of its result

The following example shows a pixel shader that performs the `MapWindowToClip` conversion.

```
float UnwarpDethPS(float4 position : SV_Position): SV_Target
{
      float warpedDepth = DepthBuffer[position.xy].x;
      float flatDepth = NV_VR_MapWindowToClip(CBData,
            float3(position.xy, warpedDepth)).z;
      return flatDepth;
}
```

2.3.4 Avoiding Common Mistakes with Modified Projections

Modified projections are not linear over the entire screen. Therefore, some well-known techniques for simplifying the coordinate transformation math in image space passes cannot be used for modified projections:

- Passing coordinates or directions from a full-screen quad vertex shader, specifically:
  - Passing UV, NDC-space (X,Y) coordinates from a full-screen quad vertex shader
  - Passing view or world space ray directions from a full-screen quad vertex shader

  UV coordinates that are passed from a full-screen quad can be used to sample G-buffer textures, but cannot be used to reconstruct positions without remapping. Furthermore, this technique can be used only when the viewport starts at 0, so it cannot be used for side-by-side stereo.

  UV or NDC coordinates or rays passed from a full-screen quad that is drawn by using the modified projection can be used to reconstruct positions. But they cannot be used in an LMS projection, which modifies depth. And they can no longer be used to sample the G-buffer in any type of modified projection.

- Linearizing depth based on zNear and zFar parameters extracted from a projection matrix

  Linearizing depth does not work directly with lens matched projection because this projection modifies depth values. Therefore, a mapping from window-space depth to NDC-space depth is necessary first. Furthermore, lens matched shading requires 1/W projection to achieve good depth accuracy. The linearization math must also handle that requirement: Extracting the near plane and far plane from a 1/W projection matrix yields zNear = ∞ and zFar = 1.0.

2.3.5 Using Threads Efficiently in a Lens Matched Projection

Drawing a full screen quad or doing a full screen dispatch over a lens matched projection image causes many threads to be discarded. The discarded threads are launched for pixels located outside the LMS octagon. This behavior can be avoided or mitigated in the following ways:

- A full screen quad can be drawn by using the modified projection. Then the quad covers only the pixels inside the octagon.
- A full screen quad can be drawn by using the depth test or stencil test, so that hardware pixel-level culling is applied as described in section 2.2.3.
- In a compute dispatch, threads can be terminated immediately after the clip space position is computed. However, if the compute shader uses shared memory and barriers, terminating only some threads in a group is not an option. In this case, entire thread groups can be terminated if all threads in them are outside the clip space.

  The temporal AA shader in the sample in `taa_cs.hlsl` contains a good example of terminating thread groups in which all threads are outside the clip space.

2.3.6 Other Considerations for Image-Space Passes

- Some passes sample only a small pixel neighborhood, for example:
  - Downsampling
  - Upsampling
  - Edge detection
  - Very slight blurring or sharpening effects

  These passes may not require any special treatment at all because pixels that are adjacent in an ordinary full-resolution image are still adjacent in the modified projection image.
- Passes with a large filter kernel, such as bloom or SSAO, must alter the size and shape of the filter kernel to compensate for the differing viewport resolutions.

For the following reasons, the position of each individual tap must be remapped for each pixel individually:

- To allow the kernel cross viewport boundaries without producing artifacts
- To use correct and consistent kernel shape in lens matched projection

For details, refer to the SSAO implementation in the sample.

- Passes that perform a histogram of the image, such as for HDR exposure adaptation, must assign weight to pixels on the basis of the number of modified projection pixels for each flattened pixel. These passes must give more weight to pixels in the outer regions.

2.4 THE FLATTENING PASS

Before an image is finally displayed or sent to the VR API, the image must be flattened. Flattening an image resamples the image from a modified MRS or LMS projection format back to a standard planar projection image.

2.4.1 Implementing the Flattening Pass

To implement the flattening pass:

1. Draw a full screen quad or triangle.
2. In the pixel shader, do the following:
   a) Compute the clip space (X,Y) or UV coordinates that the output pixel corresponds to.
   b) Use one of the following mapping functions to compute the window-space coordinates in the modified projection texture:
      - `NV_VR_MapClipToWindow`
      - `NV_VR_MapUVToWindow`
   c) Multiply those window-space coordinates by inverse input texture size
   d) Sample the input texture.

2.4.2 Remapping Parameters for Side-by-Side Stereo

In an application that renders side-by-side stereo, the remapping parameters include the location of the viewports for each eye. Therefore, you must ensure that the remapping parameters for the correct eye are used.

For example, you might erroneously use the left eye parameters for both eyes. If the projection parameters for both eyes are the same, the left eye image appears in both parts of the output texture. If the projection parameters for both eyes are different, the results may be even worse.

2.4.3 Determining when to Flatten an Image

To avoid adding an extra pass to the renderer, incorporate the flattening pass into an existing final post-processing pass, such as tone mapping or color correction.

However, if you need to integrate modified projections incrementally into a renderer, perform the flattening pass earlier in the frame. Early passes can be converted to use modified projection first. Flattening can then be applied before later passes that don't yet support modified projections are executed. As more passes are converted, the flattening pass can be moved later in the frame.

2.4.4 Rendering 2D UI Elements

For best results, use a planar projection in full resolution to render any 2D UI elements such as HUDs, menus, chat messages, and so forth. Do not use a modified projection for 2D UI elements because the modified projection is likely to be obviously apparent on text and sharp edges.

Because you should use a planar projection to render 2D UI elements, perform the flattening pass before rendering UI elements and text.

Chapter 3. RENDERING WITH SINGLE PASS STEREO

3.1 EFFICIENT STEREO RENDERING

Working with VR means rendering two slightly different scene views on each frame: one for left eye and one for right eye. There are several ways to render two views, some more efficient than others.

3.1.1 Overview of Stereo Rendering Approaches

3.1.1.1 Sequential Rendering

The simplest way is to render the views sequentially, going through all rendering passes first for one eye and then for the other eye. Until very recently, most engines were rendering the views sequentially. Sequential rendering is inefficient because the scene is the same and the views are very similar. However, it is not possible to increase efficiency merely by discarding computations that are not required.

3.1.1.2 Geometry Shader Expansion with Multiple Viewports

One obvious thing to try is to set up 2 viewports and use a geometry shader to expand each triangle into two, compute two clip space positions, and set the viewport indices. This method works and cuts CPU and tessellation workloads by half. But adding a geometry shader to a graphics pipeline that doesn't already have one usually reduces overall performance significantly, which defeats the purpose of drawing two views in one pass.

3.1.1.3 Geometry Instancing

Another approach is to use geometry instancing, drawing every mesh twice just by calling `DrawInstanced(…, 2)` instead of `Draw(…)`. However, there is an issue with DirectX 11 because it doesn't let you specify viewport index from a vertex shader. But there is also a workaround: use one viewport for both views, scale and offset the geometry according to the view being rendered, and output a user clip plane distance to create a boundary between the views. This approach is known as instanced stereo rendering. It does not have any negative performance implications unlike the previously described GS-based approach, but it doesn't have any major positive implications either: the only thing it saves is CPU time required to issue 2x the amount of draw calls. Geometry is expanded in the earliest stage of GPU pipeline, so the GPU doesn't benefit from instanced stereo at all.

3.1.1.4 Single-Pass Stereo Rendering

With Pascal GPU architecture, we introduce a new hardware feature that addresses the stereo rendering efficiency issue: Single Pass Stereo (SPS). The idea is simple: we make the last shader operating on geometry (that can be vertex, domain or geometry shader) output two positions for each vertex instead of one. Then the hardware that does viewport-related operations expands each triangle into two, using the left position for one instance and the right position for another instance. So, the GPU doesn't have to do all the geometry processing twice, and it also doesn't have to pay the cost of adding a geometry shader to a pipeline that does not otherwise use one.

3.1.1.5 Comparison of Geometry Instancing and Single-Pass Stereo Rendering

To compare instanced stereo with single pass stereo, let's take a look at the graphics pipeline diagram below. The CPU submits draw calls, then the driver translates them into GPU commands. The GPU gathers vertex attributes from memory, runs the vertex, tessellation and geometry shaders, then culls and clips triangles, maps them to window space, rasterizes them, and runs the pixel shader, which outputs the final color. Every stage of the pipeline consumes some resources and every stage can be a bottleneck. So, the later geometry is expanded, the better.

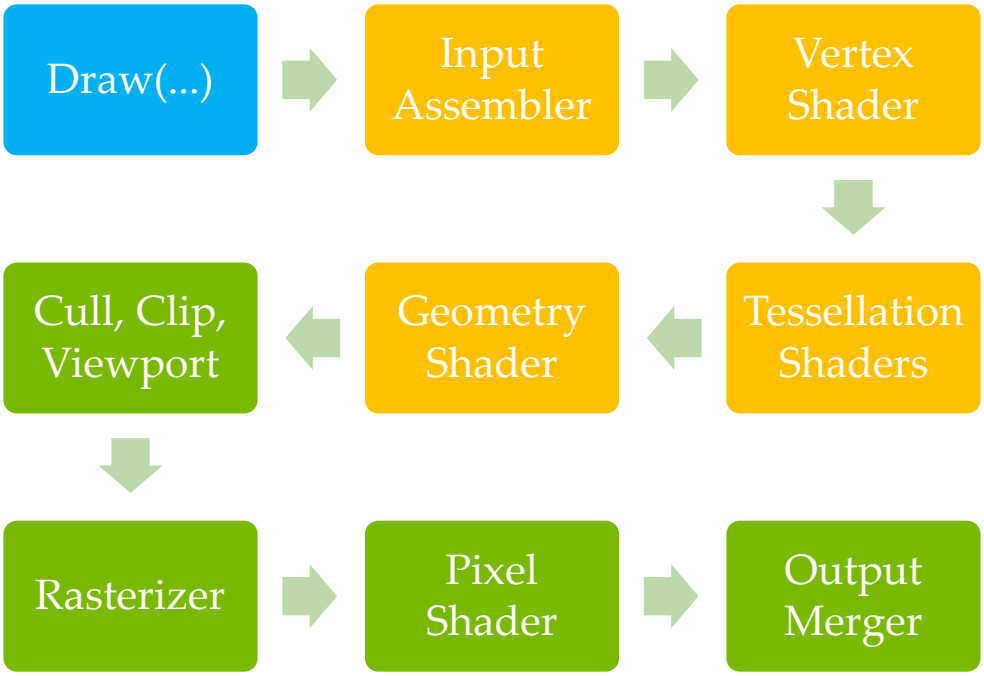


Figure 14. Effects of Instanced Stereo and Single Pass Stereo Modes on the Graphics Pipeline

Instanced stereo reduces the CPU workload, but geometry is expanded even before the input assembler; the improved part is painted blue on the diagram. Single Pass Stereo expands the geometry later, just before culling and viewport transform stage, therefore optimizing every stage up to the geometry shader; these optimized stages are painted orange.

3.1.2 Enabling Single Pass Stereo in an Application

To use single pass stereo, an application should:

- Use a vertex, tessellation or geometry (VTG) shader that computes a separate clip space X coordinate for the right eye and outputs that coordinate into an attribute with a custom semantic, for example `NV_X_RIGHT`. The shader should also output a viewport mask to identify which viewports correspond to which eye. The viewport mask should also go to an attribute with a custom semantic, such as `NV_VIEWPORT_MASK`.
- Create all enabled VTG shaders:
  - When working with D3D11, create the shaders by using these NVAPI functions:
    - `NvAPI_D3D11_CreateVertexShaderEx`
    - `NvAPI_D3D11_CreateHullShaderEx`
    - `NvAPI_D3D11_CreateDomainShaderEx`
    - `NvAPI_D3D11_CreateGeometryShaderEx_2`

- When working with D3D12, create the shaders as follows:
  1. Create the Pipeline State Object (PSO) by using the `NvAPI_D3D12_CreateGraphicsPipelineState` function.
  2. Include the custom semantics declaration into the extensions for all enabled VTG shaders by using the following extension structures:
     - `NVAPI_D3D12_PSO_VERTEX_SHADER_DESC`
     - `NVAPI_D3D12_PSO_HULL_SHADER_DESC`
     - `NVAPI_D3D12_PSO_DOMAIN_SHADER_DESC`
     - `NVAPI_D3D12_PSO_GEOMETRY_SHADER_DESC`

- Enable the SPS mode:
  - On D3D11, use the NVAPI function `NvAPI_D3D_SetSinglePassStereoMode`.
  - On D3D12, use the NVAPI function `NvAPI_D3D12_SetSinglePassStereoMode`.
- Depending on the stereo render target layout, either set at least two viewports, or bind a render target array with at least two slices.

For more information on single pass stereo related NVAPI functions, see *Single Pass Stereo Programming Guide*.

3.1.3 Putting Features Together

NVIDIA Maxwell and Pascal GPU architectures introduce several graphics pipeline extensions related to geometry processing.

Maxwell added Viewport Multicast, enabling each triangle to be sent into several viewports with the same position and attributes, and Coordinate Swizzle, changing the order of vertex position components independently for each viewport. These extensions are known as Multi-Projection Acceleration, and they are available as follows:

- On D3D11, they are available through the `NvAPI_D3D11_CreateGeometryShaderEx_2` function. Therefore, using a regular or fast geometry shader is required to take advantage of them.
- On D3D12, they are available through the `NvAPI_D3D12_CreateGraphicsPipelineState` function and the `NVAPI_D3D12_PSO_GEOMETRY_SHADER_DESC` and `NVAPI_D3D12_PSO_CREATE_FASTGS_EXPLICIT_DESC` extension structures.

Pascal adds Modified W and Single Pass Stereo, where the former changes the W component of vertex coordinates based on their X and Y components, and the latter duplicates every triangle and uses different values of X for each copy.

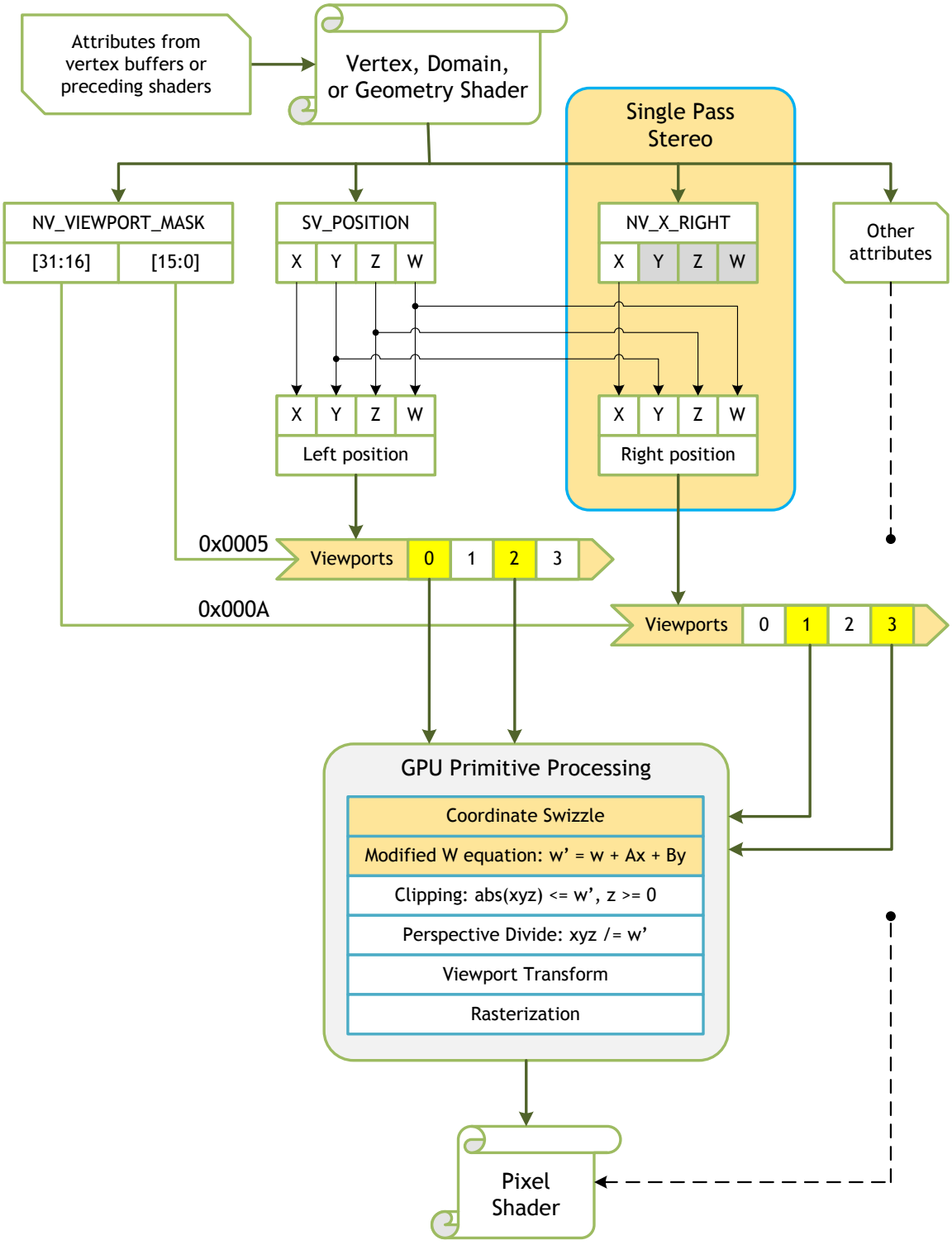


Figure 15. Single Pass Stereo, Viewport Multi-cast, Coordinate Swizzle, and Modified W in the Graphics Pipeline

The diagram above shows how all these features fit together in the fixed function hardware inside the GPU. This is the stage between the last geometry processing shader stage (vertex, domain or geometry shader) and the pixel shader. Normally this stage performs only clipping, perspective divide, viewport transform, and rasterization. Features introduced in Maxwell and Pascal are highlighted with gold fill on the diagram.

The first new feature in the pipeline is Single Pass Stereo. If enabled, it picks up the attribute with the special semantic name (normally `NV_X_RIGHT`) and combines it with `SV_Position` to produce the position for the right view. From this point on, the pipeline works with two identical triangles only differing in position and maybe viewport mask. When the `independentViewportMaskEnable` parameter of `NvAPI_D3D(12)_SetSinglePassStereoMode` function is set to true, the higher 16 bits of the viewport mask are used for the right view and the lower 16 bits are used for the left view; when that parameter is set to false, the lower 16 bits are used for both views. Also, when the `renderTargetIndexOffset` parameter of the same function is nonzero, the right view triangle is sent into a different render target array slice.

Then the triangle is replicated some more by Viewport Multi-cast. Depending on the viewport mask effective for the current view, it can be sent into 1 to 16 viewports. Combined with Single Pass Stereo, this means a triangle can become 32 triangles in 16 viewports and 2 render target array slices at this point. Each viewport can have its own size and position, scissor rectangle, coordinate swizzle pattern, and Modified W parameters.

After that, Coordinate Swizzle is applied. This feature can swap the X, Y, Z and W components of every vertex's position in arbitrary order and change their signs – depending on viewport state. Swizzle patterns are bound to the geometry shader by the `NvAPI_D3D11_CreateGeometryShaderEx_2` function, so this feature cannot be used without a geometry shader. Similarly, on D3D12, the swizzle patterns are bound through the `NVAPI_D3D12_PSO_GEOMETRY_SHADER_DESC` or `NVAPI_D3D12_PSO_CREATE_FASTGS_EXPLICIT_DESC` extensions.

After swizzling, W components of vertex positions can be changed if the Modified W feature is activated. Then the triangle is passed to the standard part of graphics pipeline, including culling, clipping, viewport transform and rasterization.

Note that Single Pass Stereo, Coordinate Swizzle and Modified W and are only applied to the system-interpreted clip space positions. If the application for example passes a world space position or a vector to camera through custom attributes, those will not be affected. Be extra careful with view dependent attributes like camera vectors because they will be the same for both views, which is generally incorrect.

3.2 STEREO LAYOUTS

Single Pass Stereo can be used to render two different kinds of stereo layouts: side-by-side or render target array (RTA). Each has its strengths and weaknesses, but the performance of these layouts is approximately the same. Choosing one or the other is mostly a matter of minimizing engine modifications, i.e. which one will be less troublesome to integrate.

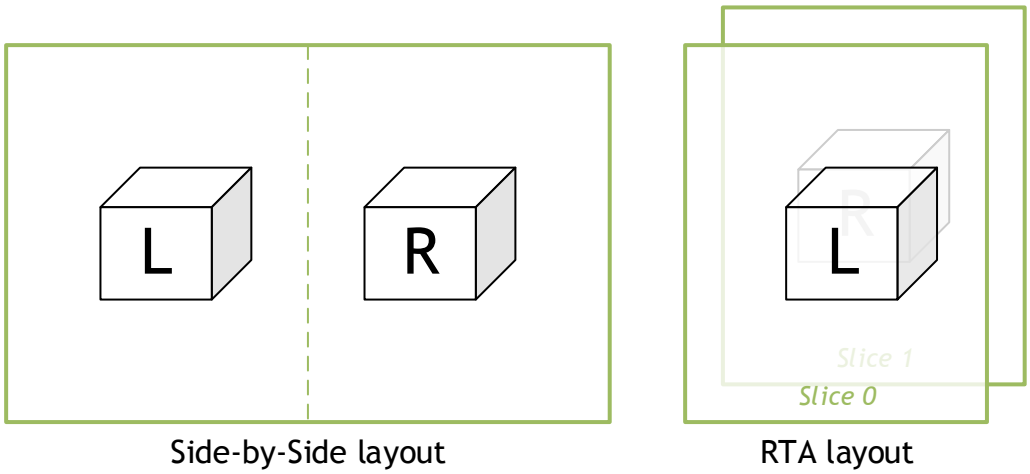


Figure 16. Stereo Layouts

3.2.1 Side-by-Side Layout

Images for both eyes are located on the same render target, one on the left half and one on the right half. For a planar projection, there should be two viewports, each covering the corresponding eye's image. For a lens-matched projection, there should be 8 viewports: 4 for the left eye and 4 for the right eye. To get the viewports and configuration for the right eye, use these SDK functions:

- `CalculateMirroredConfig`
- `CalculateViewportsAndBufferData`

By default, the GPU will send both triangles (left and right ones) into the same viewport, which is not what we need in side-by-side mode. The solution is to output separate viewport masks for two views. This mode is enabled by the last parameter of `NvAPI_D3D_SetSinglePassStereoMode` function, `independentViewportMaskEnable`. When it's active, the 32-bit viewport mask attribute will be interpreted as two masks: the lower 16 bits are for the left view, and the upper 16 bits are for the right view. Let's say we are using a planar projection and want to output the left view into viewport 0 and the right view into viewport 1. We set up the viewport mask in the shader as follows:

```
ViewportMaskLeft = 0x0001          // only viewport 0
```

```
ViewportMaskRight = 0x0002          // only viewport 1
ViewportMask = ViewportMaskLeft | (ViewportMaskRight << 16)
```

If we are using a Lens Matched projection, we need to use a geometry shader for culling and viewport mask computation, and the SDK implements this geometry shader for the single pass stereo mode as well. Refer to section 2.2.1 for more information about SDK geometry shaders.

Multi-Res projection is not compatible with SPS in side-by-side layout because one MRS view requires 9 viewports, and two views require 18 viewports. GPUs support only 16 viewports, which is insufficient. There is a way to enable MRS in instanced stereo mode by joining the right column of viewports of the left view with the left column of viewports of the right view, but it's not implemented in the SDK.

3.2.2 Render Target Array (RTA) Layout

Images for each eye are located on separate render target array slices. For a planar projection, one viewport is sufficient for both views because they overlap in XY plane. The hardware will take care of generating proper render target array indices for each triangle, you just need to activate the appropriate mode using one of these functions:

```
NvAPI_D3D_SetSinglePassStereoMode(pDevice, 2, 1, false); // D3D11
NvAPI_D3D12_SetSinglePassStereoMode(pCommandList, 2, 1, false); //D3D12
```

The third parameter is called `renderTargetIndexOffset`, and setting it to 1 means that left view triangles will go to slice 0 and right view triangles will go to slice 1.

RTA Single Pass Stereo can be compatible with MRS or LMS projections also. For LMS, same or different projection configurations can be used for the views. If the configurations are the same (which means they are symmetric), 4 viewports are sufficient, but the FastGS shader still has to compute 2 viewport masks, and the GPU has to be configured to use separate masks for different views. In the general case when the configurations are different, set 8 viewports, compute 2 viewport masks and shift the right mask to the left by 20 bits: 16 to get to the right view mask component, and 4 more because the right view uses viewports 4-8 instead of 0-3. This mode is implemented in the FastGS provided with the SDK.

Compatibility of RTA Single Pass Stereo with MRS is limited to symmetric projections only because of the same viewport count limitations: the GPU only supports 16 simultaneous viewports but using different configurations needs 18 viewports. So, you can bind 9 viewports and use them for both views and compute 2 viewport masks in the GS. But the problem with this approach is that MRS configurations optimized for real HMDs are asymmetric, so there is no obvious value in such mode. Therefore, it is not implemented in the SDK.

Chapter 4. SDK REFERENCE

The SDK is provided fully open-source so that you can easily incorporate it into your code and adapt it to your application's needs, for example, by porting it to your renderer's math library, fine-tuning coordinate systems, or adding features. The SDK also provides a sample application that demonstrates how all the pieces fit together.

The SDK contains the following files:

`nv_vr.h`

Header that should be included into the client code, defining all CPU-side SDK types and functions

`nv_multi_res.cpp`

Implementation of the MRS projection

`nv_lens_matched_shading.cpp`

Implementation of the LMS projection

`nv_planar.cpp`

Implementation of the regular planar projection

`nv_vr.hlsli`

HLSL include file that defines macros, structures and functions for all projections supported by the SDK

`nv_vr_geometry_shader.hlsli`

HLSL include file that defines geometry shader code for culling geometry and computing viewport masks for all projections and Single Pass Stereo rendering

4.1 MAIN SDK HEADER FILE (C++)

The main header file of the SDK is `nv_vr.h`. All the types and functions are contained within the `Nv::VR` namespace. Include this header in your application.

4.1.1 Enumerations

```
enum class Projection
{
      PLANAR,
      MULTI_RES,
      LENS_MATCHED
};
```

The SDK defines the following types of special rendering:

- Multi-resolution shading
- Lens matched shading

Additionally, planar projection is defined to reduce the number of special projection-dependent code paths in applications.

Note that it doesn't matter which projection type is used. The interface remains the same for all operations. Implementation differences are hidden from the client.

4.1.2 Common Types

```
struct Viewport
struct ScissorRect
struct Float2
struct Float3
struct Float4
struct ScaleBias
```

These simple structures serve as in and out parameters to the methods of the library. They are collections of integers or floating point numbers.

```
struct FastGSCBData
struct RemapCBData
```

`FastGSCBData`

Contains the data required for the FastGS to cull triangles against viewports. It matches the same-named structure in HLSL, and should be incorporated into a constant buffer.

`RemapCBData`

Contains the data required for the HLSL helper functions to map clip-space positions to window-space and back. It matches the same-named structure in HLSL, and should be incorporated into a constant buffer. Note that this structure uses MRS names for the data fields. LMS data is written into these fields, but shaders interpret the data in a different way.

`template<Projection Type> struct Configuration`

Contains the user-editable parameters that define the layout of a multi-resolution or lens-matched image. Every projection type defines its own specialization. See section 2.1 for more information about these parameters. Configuration parameters are suitable for being exposed as tunable values in a developer UI or console.

`ProjectionViewports`

Stores the precise viewport and scissor rectangles to set on the graphics pipeline for rendering into any projection. The number of viewports and scissor rectangles is specified by the `NumViewports` field and depends on the projection:

- For MRS, 9 (3×3) is used
- For LMS 4 (2×2) is used
- For planar projection, 1 is used

`Data`

The set of `FastGSCBData`, `RemapCBData` and `ProjectionViewports` structures. It is returned as the output from `Configuration` in one of the library functions that are described in section 4.1.3.

```
struct Data
{
   ProjectionViewports           Viewports;
   FastGSCBData                  FastGsCbData;
   RemapCBData                   RemapCbData;
};
```

4.1.3 Functions

```
template<Projection Type>
void CalculateViewportsAndBufferData(
  const Float2& flattenedSize,
  const Viewport& boundingBox,
  const Configuration<Type>& configuration,
  Data& ref_data);
```

This function is the core method in the library. Given a projection configuration, size of flat image, and a bounding box, calculates everything that is required to set up the GPU state and render using a modified projection and stores it in the `Data` structure.

See section 4.1.2 for information about the `Data` structure.

See section 2.1 for more information about the flattened size and how it affects various projections.

```
template<Projection Type>
Float2 CalculateProjectionSize(
  const Float2& flattenedSize,
  const Configuration<Type>& configuration);
```

Given a projection configuration and size of flat image, calculates the size of the modified projection. This function can be used to determine the render target size that is required for the modified projection.

```
template<Projection Type>
void CalculateMirroredConfig(
  const Configuration<Type>& sourceConfig,
  Configuration<Type>& ref_mirroredConfig);
```

Given a `Configuration` structure, this function calculates another configuration that is a left-to-right mirror image of it. In stereo rendering, viewport configurations for the left eye and right eye should generally be mirror images of each other.

```
template<Projection Type>
float CalculateRenderedArea(
  const Configuration<Type>& configuration,
  const ProjectionViewports& viewports);
```

Given a `Configuration` structure and `ProjectionViewports` structure, this function calculates the number of pixels that will be rendered with this configuration.

```
template<Projection Type>
Float3 MapClipToWindow(const Data<Type>& data, const Float4& clipPos);
```

This function maps homogenous clip space coordinates to window-space pixel coordinates. The input position is normalized, but not clipped against clip space bounds.

```
template<Projection Type>
Float4 MapWindowToClip(const Data<Type>& data, const Float3& windowPos,
bool normalize = true);
```

This function maps window-space pixel coordinates, including depth, to clip space coordinates. The input position is not clipped against window bounds. The output position is normalized, that is, it has W component = 1.0 when `normalize` is set to `true`.

```
template<Projection Type>
Float2 MapClipToWindow(const Data<Type>& data, const Float2& clipPos)
```

This function is a simplified version of the `MapClipToWindow` function that is described previously.

```
template<Projection Type>
Float2 MapWindowToClip(const Data<Type>& data, const Float2& windowPos)
```

This function is a simplified version of the `MapWindowToClip` function that is described previously.

4.1.4 Planar, MRS, and LMS Projection Specializations

Each projection type defines its own specializations of the configuration data structures.

```
namespace LensMatched
{
     typedef Nv::VR::Configuration<Projection::LENS_MATCHED>
     Configuration;
}
```

Moreover, multi-resolution and lens-matched projections have configuration presets to be used with certain HMDs.

4.2 PROJECTION TYPE IMPLEMENTATION FILES (C++)

The projection type implementation files are as follows:

- `nv_multi_res.cpp`
- `nv_lens_matched_shading.cpp`
- `nv_planar.cpp`

These files contain the interface implementation for each of the projection types by defining template function specializations. Configuration presets are also defined in these files.

4.3 HLSL INCLUDE FILE FOR SUPPORTED PROJECTION TYPES

The HLSL include file for supported projection types is `nv_vr.hlsli`.

Include this file in user shader code to use SDK shader functionality.

```
#define NV_VR_PROJECTION_PLANAR          0
#define NV_VR_PROJECTION_MULTI_RES       1
#define NV_VR_PROJECTION_LENS_MATCHED    2

#ifndef NV_VR_PROJECTION
#define NV_VR_PROJECTION NV_VR_PROJECTION_PLANAR
#endif
```

At the start of the file, projection type macros are defined. Define the `NV_VR_PROJECTION` macro to one of these projection types in your shaders. Otherwise planar projection is selected, which specifies regular rendering.

```
struct NV_VR_FastGSCBData
struct NV_VR_RemapCBData
```

HLSL counterparts of the structures defined in `nv_vr.h` which are used to cull geometry in FastGS and perform window-to-clip or clip-to-window coordinate mappings.

```
uint2 NV_VR_CalculateVertexSideMasks(NV_VR_FastGSCBData CBData,
float4 Position)
```

This function calculates a bitmask of the results of comparing a vertex position against the left, top, right, and bottom sides of all the viewports in the multi-resolution or lens-matched image. It is used internally by `NV_VR_CalculateViewportMask()`.

```
uint NV_VR_CalculateViewportMask(NV_VR_FastGSCBData CBData,
float4 Position0, float4 Position1, float4 Position2)
```

Given an `NV_VR_FastGSCBData` and the clip-space vertex positions of a triangle, this function culls the triangle against all the multi-resolution or lens-matched viewports and returns a bitmask of which viewports it conservatively intersects. It is used by geometry shaders.

```
float4 NV_VR_MapWindowToClip(NV_VR_RemapCBData cbData,
float3 windowPos, bool normalize = true)
```

Each of the projection types defines this function. It maps the window space pixel position to clip space coordinates. See section 2.3 for more information about how to use this function.

```
float3 NV_VR_MapClipToWindow(NV_VR_RemapCBData cbData, float4 clipPos,
bool normalize = true)
```

Each of the projection types defines this function. It maps clip space coordinates to the window space pixel position. See section 2.3 for more information about how to use this function.

```
float2 NV_VR_MapUVToWindow(NV_VR_RemapCBData cbData, float2 linearUV)
float2 NV_VR_MapUV_LinearToVR(NV_VR_RemapCBData cbData, float2
linearUV)
float2 NV_VR_MapUV_VRToLinear(NV_VR_RemapCBData cbData, float2 vrUV)
```

These functions are helper functions that operate on UV coordinates instead of clip space position or window space position. These functions are wrappers around the functions `NV_VR_MapWindowToClip()` and `NV_VR_MapClipToWindow()`.

For example, the SDK sample shader flatten pixel shader uses `NV_VR_MapUVToWindow()` to unproject a multi-resolution or lens-matched image to the regular form of image.

4.4 GEOMETRY SHADER HLSL INCLUDE FILE

The Geometry shader HLSL include file is `nv_vr_geometry_shader.hlsli`.

Include this file to create a FastGS or regular Geometry Shader that passes through a specified structure of vertex attributes and performs triangle culling against the viewports.

The number of viewports used depends on the type of shading:

- For multi-resolution shading, 9 (3×3) viewports are used.
- For lens-matched shading, 4 (2×2) viewports are used.

To properly define the culling method, define several macros before including this shader. These macros are listed and explained in section 2.2.1. For an example of how these macros are used, see the `world_gs.hlsl` or `safezone_gs.hlsl` shader in the sample application.

Chapter 5. SAMPLE APPLICATION

The SDK contains a sample application that uses multi-resolution shading or lens-matched shading to render the Crytek Sponza test scene or the San Miguel test scene to either a VR headset or a screen.

5.1 APPLICATION VERSIONS

Two versions of the application are provided to demonstrate two different ways of rendering the test scene.

5.1.1 `demo_dx11` Version

The `demo_dx11` version renders the scene through native D3D11 APIs. This version of the sample was included with the previous versions of the SDK.

The source code of this version is stored in the `demo/dx11` folder of the SDK package.

5.1.2 `demo_nvrhi` Version

The `demo_nvrhi` version uses a rendering API abstraction layer called NVRHI. This layer allows application code to be written in a platform-independent manner and to work without modification on both the D3D11 runtime and the D3D12 runtime.

To enable this version of the application to run on the D3D11 runtime and the D3D12 runtime, two rendering backends are provided:

- For D3D11: `GFSDK_NVRHI_D3D11.cpp`
- For D3D12: `GFSDK_NVRHI_D3D12.cpp`

Each rendering backend translates NVRHI commands to the native commands of the respective low-level rendering API. Because most of the sample code is platform independent, all the NVAPI functions described in this document are used in these two files, not in the sample rendering code.

The source code of this version is stored in the `demo/dx12` folder of the SDK package.

5.2 LAUNCHING THE APPLICATION

You launch the sample application in one of the following ways:

- Open the Visual Studio solution file, and compile and launch the project.
- Use the supplied binaries.

5.3 CONTROLLING THE APPLICATION

You can use the WSAD keys and the left mouse button to move the camera around the scene. The [**Q**] and [**E**] keys move the camera up and down. [**Shift**] accelerates the camera, and [**Ctrl**] slows it down.

XInput-compatible controllers are also supported. The triggers move the camera up and down, and the right bumper accelerates the camera.

5.4 APPLICATION PARAMETERS IN THE UI

Figure 17 shows the UI of the sample application.

Figure 17. Sample Application UI

The UI contains the following groups of parameters:

FPS

This group contains various performance counters.

Rendering

This group provides options for:

- Selecting the scene from the available options Sponza and San Miguel
- Toggling v-sync
- Setting MSAA modes
- Setting how many times a scene should be rendered
- Toggling SSAO and temporal anti-aliasing
- Changing sample shaders and hot reloading them

VR and Stereo

This group provides options to control how the test scene is rendered:

- If HMD is present, you can activate VR rendering. Oculus DK2/CV1 and HTC Vive devices are supported.
- If HMD is not present, you can activate emulated VR rendering by clicking **Activate Desktop Stereo**.

While stereo rendering is enabled, the following additional modes can be used:

- **Instanced Stereo**, which uses instancing to submit geometry.
- **Single Pass Stereo**, which is a special feature of the Pascal GPU to draw stereo geometry. When no Pascal GPU is detected, this feature is emulated using regular geometry shaders.

Multi-Res shading
Lens-Matched shading

These groups control whether special rendering is enabled and provide options for:

- Changing configuration parameters
- Choosing presets
- Drawing viewport splits
- Disabling flattening to see how the image appears before it is flattened

The **Lens Matched Shading** group also contains a parameter called **Resolution Scale**. This parameter is not part of the lens matched projection configuration. But it directly affects the configuration because all four octagon diagonal sizes are scaled by using this factor.

You can use the Resolution Scale parameter to easily compare quality and performance with various undersampling and oversampling settings. For example, we found that using the Oculus Rift CV1 preset with **Resolution Scale** set to 0.85 provides acceptable quality with best performance.

5.5 PERFORMANCE TESTING

For performance testing, the application uses Direct3D 11 or Direct3D12 timestamp queries to measure GPU busy time. To get the most accurate results, enable full-screen mode [**Alt+Enter**] or VR mode.

Note that the NVRHI version of the sample does not measure rendering performance by default. As a result, the GPU time counters in the UI normally display 0. To see the actual GPU time spent on scene rendering, select the **Measure GPU times** option adjacent to the counters. This option is deselected by default to maximize rendering

performance because the timestamp query results are read back to the CPU on the same frame, which affects overall FPS value.

You can also use the **Repeat Rendering** control to cause the application to repeat all scene rendering multiple times per frame. Repeating the rendering increases the GPU load, which may assist in getting more reliable numbers.

You are unlikely to see any performance benefit from enabling Instanced Stereo or Single Pass Stereo with the Sponza scene. The reason is that the scene is relatively low-poly and, therefore, its rendering performance is limited by pixel workload, not geometry. To observe Single Pass Stereo speed-ups, switch to the San Miguel scene.

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.

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.

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