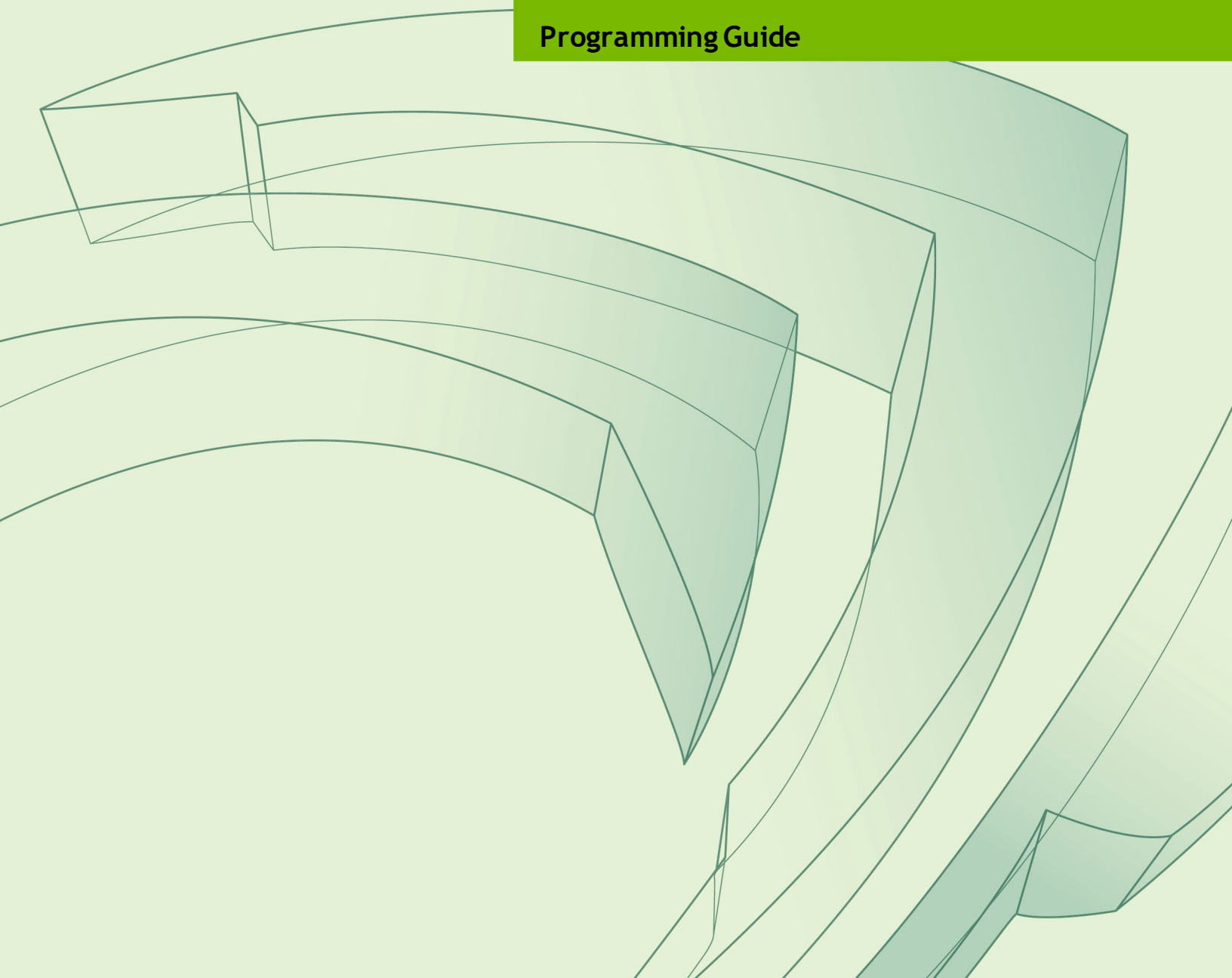
NVIDIA WARP360

PG-09216-001_v2.1 | April 2019

Programming Guide



DOCUMENT CHANGE HISTORY

PG-09216-001_v2.1

Version	Date	Description of Change
1.0.3	04/04/2018	SDK release 1.0.3
1.1	07/19/2018	SDK release 1.1
2.0	09/10/2018	SDK Release 2.0
2. 1	04/17/2019	SDK Release 2. 1

TABLE OF CONTENTS

Chapter 1. INTRODUCTION TO THE NVWARP360 SDK

The NVWarp360 SDK uses CUDA to provide warping functions for images and to provide coordinate transformations.

The images can be thought of as embedded in 3D space. Although the images are 2D, when embedded in 3D space they are generally not flat. For example, an equirectangular image is a projection of the world onto a sphere, and a 180° fisheye lens is a hemispherical image. The mathematical term for these generalized 2D imaging surfaces is an image *manifold*.

1.1 WARP PARAMETERS

The parameters of the warp are common to both the function for warping an image and the function warping coordinates.

- **The function for warping an image** takes one image and a set of parameters describing the warp as input and produces another image as output.
- **The function for warping coordinates** takes one set of coordinates and the same parameters that are used for image warping as input and produces another set of coordinates as output. The output coordinates can be the same as the input coordinates. Coordinate warping can occur either in the forward or backward direction.

Some helper functions are provided for estimating parameters.

1.2 APPLICATIONS

The NVWarp360 SDK is suitable for several different types of applications.

1.2.1 Distortion Removal

NVWarp360 can be used to remove distortion from acquired imagery. Both the fisheye and perspective image *sources* accommodate barrel distortion correction. Conversion from fisheye to perspective is another example of distortion removal.

Some security cameras are ceiling mounted and have a 360°×180° downward view that is difficult to comprehend by human viewers and machine-learning software. NVWarp360 can convert these views into sideways views without distortion for use by human viewers or machine-learning software.

1.2.2 Virtual Reality and Augmented Reality

The embedding of 2D image manifolds in 3D makes such warps useful for virtual reality (VR) or augmented reality (AR) applications.

Rendered images can be warped to match the optical properties of a head-mounted display (HMD).

Acquired imagery can be warped into a form more suitable for texture-mapping into a VR or AR environment.

1.2.3 Enhanced Perception

Warps can be used to enhance the perception by human viewers of the world around them by expanding the capabilities of the human visual system. Some insects have multiple eyes that enable them to perceive a much wider field of view than humans can. The NVWarp360 SDK can simulate this effect for humans through wider field of view projections such as Panini, equirectangular, or fisheye.

1.2.4 Rendering Environments (Panoramas)

The NVWarp360 SDK can be used to render an environment map or “panorama” for interactive viewing. The equirectangular spherical format created by the VRWorks 360 Video SDK can be reprojected by NVWarp360 to a planar or other surface with viewing parameters of yaw, pitch, roll, and zoom.

1.3 ACCURACY

Coordinate warp error is less than 1 millipixel (0.001 pixels) on a 4K image.

The accuracy of image warps is limited by the precision of bilinear interpolation, which is of the order of centipixels (0.01 pixels).

Chapter 2. WARPING IMAGES AND COORDINATES WITH NVWARP360

Use NVWarp360 to implement image and coordinate warps in your applications. The NVWarp360 API is object oriented but is accessible to C in addition to C++.

2.1 GETTING A WARP OBJECT HANDLE

To be able to use NVWarp360, an application must create a warp object and get a handle for this object. This handle is required as the first parameter in subsequent NVWarp360 function calls. A warp object is an opaque object of type `nvWarpObject` that contains parameter values and other information required for warping images and coordinates.

To get a warp object handle, call the `nvwarpCreateInstance()` function. This function allocates an opaque `nvWarpObject` object and returns a handle for this object in the out parameter `han`.

```
...
// Allocate a warp object
nvwarpHandle han;
nvwarpResult err = nvwarpCreateInstance(&han);
if (err)
   return err;
...
```

2.2 GETTING THE NVWARP360 API VERSION

To get the version of the NVWarp360 API, call the `nvwarpVersion()` function. You can call this function without creating a warp object.

This function returns an unsigned 32-bit (4-byte) integer in which the three most significant bytes represent the version number of the NVWarp360 API in the form *major.minor.revison*.

The following example shows how to check compatibility by getting the NVWarp360 API version.

```
if (NVWARP_VERSION_MAJOR (nvwarpVersion()) < kExpectedMajorVersion)
    return VERSION_TOO_EARLY;
```

2.3 SETTING AND GETTING WARP PARAMETERS

The NVWarp360 SDK provides the following types of interfaces for setting and getting parameters:

- A parameter block interface that uses a structure to set multiple parameters in a single function call
- Set and get accessor functions for individual parameters

  Some parameters also have functions for computing a value. These parameters have only one get accessor function that gets the parameter regardless of how it was set.

All these interfaces interoperate. Therefore, you can use them in any order in an application.

2.3.1 Setting Multiple Parameters in a Single Function Call

The NVWarp360 API defines the `nvwarpParams_t` structure and some helper functions to enable you to set multiple warp parameters in a single function call.

1. Declare an `nvwarpParams_t` structure.
2. If you are writing your application in C, call the `nvwarpInitParams()` function to initialize the structure before setting any of its members.

   In C++, the constructor for `nvwarpParams_t` automatically calls this function.

   The `nvwarpInitParams()` function sets all members of the structure to a default value or to Not a Number (NaN).

3. Set the members of the `nvwarpParams_t` structure to the values that you require.
4. Call one of the following functions, passing the `nvwarpParams_t` structure as a parameter to the function:
   - `nvwarpSetParams()`
   - `nvwarpComputeParamsSrcFocalLength()`
   - `nvwarpComputeParamsOutputResolution()`
   - `nvwarpComputeParamsAxialAngleRange()`
   - `nvwarpComputeBoxAxialAngleRange()`

```
// Set the warp parameters
nvwarpParams_t params;
nvwarpInitParams(&params);
params.type           = NVWARP_EQUIRECT_PERSPECTIVE;
params.srcWidth       = 4096;
params.srcHeight      = 2048;
params.srcX0          = params.srcWidth  * .5f;
params.srcY0          = params.srcHeight * .5f;
params.srcFocalLength = params.srcHeight / M_PI;
params.dstWidth       = 1024;
params.dstHeight      = 768;
params.topAngle       = +M_PI / 6.f;
params.bottomAngle    = -M_PI / 6.f;
nvwarpSetParams(han, &params);
```

2.3.2 Setting and Getting Individual Parameters

Instead of setting multiple parameters in a block with a single function call, you can access individual parameters through their own set and get accessor functions.

The form of the return value of a get accessor function depends on the format of the quantity returned:

- For a scalar quantity, the return value is a scalar.
- For a compound quantity, the return value is a pointer to the location where the result is stored.

Parameter	Set and Get Accessor Functions
Warp type	`nvwarpSetWarpType()`
	`nvwarpGetWarpType()`
CUDA block	`nvwarpSetBlock()`
	`nvwarpGetBlock()`
Pixel phase	`nvwarpSetPixelPhase()`
	`nvwarpGetPixelPhase()`
Source image width and height	`nvwarpSetSrcWidthHeight()`
	`nvwarpGetSrcWidthHeight()`

Parameter	Set and Get Accessor Functions
Destination image width and height	`nvwarpSetDstWidthHeight()`
	`nvwarpGetDstWidthHeight()`
Source image principal point	`nvwarpSetSrcPrincipalPoint()`
	`nvwarpGetSrcPrincipalPoint()`
Destination image principal point	`nvwarpSetDstPrincipalPoint()`
	`nvwarpGetDstPrincipalPoint()`
Source image circular clipping radius	`nvwarpSetSrcRadius()`
	`nvwarpGetSrcRadius()`
Distortion	`nvwarpSetDistortion()`
	`nvwarpGetDistortion()`
Controls	`nvwarpSetControl()`
	`nvwarpGetControl()`
User data	`nvwarpSetUserData()`
	`nvwarpGetUserData()`

2.3.3 Setting and Getting Parameters that Have Functions for Computing a Value

In addition to a set accessor function, some parameters also have functions for computing a value. These parameters have only one get accessor function that gets the parameter regardless of how it was set.

Parameter	Accessor Functions	Comments
Source image focal length	`nvwarpSetSrcFocalLengths()`	Explicitly sets separate focal lengths for the X direction and the Y direction to support anisotropic sampling without using view angles
	`nvwarpComputeSrcFocalLength()`	Calculates the focal length from an angular measurement and corresponding radius.
	`nvwarpGetSrcFocalLength()`	Gets the focal length of the source image.
Destination image focal length	`nvwarpSetDstFocalLengths()`	Explicitly sets separate focal lengths for the X direction and the Y direction to support anisotropic sampling without using view angles.
	`nvwarpComputeDstFocalLength()`	Calculates a single focal length from the top and bottom view angles and the destination image dimensions.
	`nvwarpComputeDstFocalLengths()`	Calculates two focal lengths from the top and bottom view angles, the left and right view angles, and the destination image dimensions.
	`nvwarpGetDstFocalLength()`	Gets the focal length of the destination image.

Parameter	Accessor Functions	Comments
Rotation	`nvwarpSetRotation()`	Sets the rotation as rotation matrix.
	`nvwarpSetEulerRotation()`	Sets the rotation as a sequence of Euler angles.
	`nvwarpGetRotation()`	Gets the rotation.

2.4 PERFORMING A WARP

All image and coordinate warps have been tested and are accurate, typically to within millipixels. However, the equirectangular projection has errors approaching 1 pixel at the singular north and south poles. Significant distortion as specified in the distortion coefficients for perspective and fisheye sources yields lower coordinate warp precision towards the periphery.

Image warps are executed on the GPU. All other warps are executed on the CPU. All image warps are asynchronous.

2.4.1 Performing an Image Warp

All images that you want to warp must be red green blue alpha images with four components. Because NVWarp360 treats all components identically the images can be RGBA images or images with other component orderings such as BGRA.

The source of all image warps is a texture. The storage for backing up the texture can be either a surface or a pitched buffer. The destination can also be either a surface or a pitched buffer but need not be the same as the type of storage for backing up the source texture.

1. Allocate a CUDA stream in one of the following ways:
   - Create a `cudaStream_t` opaque object.
   - Use the default stream 0.
2. Allocate CUDA storage, in the form of a pitched buffer or surface, for the source.

   You can allocate CUDA storage for multiple sources.
3. Make the storage for the source into a texture, typically with border 0 or border clamp.
4. Allocate CUDA storage, in the form of a pitched buffer or surface, for the destination.

   You can allocate CUDA storage for multiple destinations.
5. Set the parameters for the warp as explained in “Setting and Getting Warp Parameters,” on page 5.

6. Load the source with the image that you want to warp.
7. Call the function to perform the warp into the type of destination that you have:
   - To warp the texture into a surface, call `nvwarpWarpSurface()`.
   - To warp the texture into a pitched buffer, call `nvwarpWarpBuffer()`.

   You can warp multiple images with the same parameters in subsequent warp function calls.

Because the functions for warping images are implemented in CUDA, they are asynchronous. To perform an image warp synchronously, call `cudaDeviceSynchronize()` after the warp call.

2.4.2 Performing a Coordinate Warp

Coordinate warps are useful for rendering graphic overlays on warped images or for object selection. You can warp coordinates from the source image into the destination image or in the opposite direction, namely, from the destination image into the source image.

1. Set the parameters for the warp as explained in "Setting and Getting Warp Parameters," on page 5.
2. Call the function that warps the coordinates in the direction that you require.

From	To	Function
Source image	Destination image	`nvwarpWarpCoordinates()`
Destination image	Source image	`nvwarpInverseWarpCoordinates()`

   To warp the coordinates in place, set the address of the output coordinates equal to the address of the input coordinates.

   The result code of the function call indicates if any coordinates are out of domain.

3. If any coordinates are out of domain, check the specific coordinates for NaN values to determine which coordinates are out of domain.

2.4.3 Converting Between Pixel Coordinates and Rays

For source images and destination images, you can convert 2D pixel coordinates into 3D rays and convert 3D rays into 2D pixel coordinates.

1. Call the function that performs the conversion that you require.

Image	From	To	Function
Source image	Pixel coordinates	Rays	`nvwarpSrcToRay()`
	Rays	Pixel coordinates	`nvwarpSrcFromRay()`

Image	From	To	Function
Destination image	Pixel coordinates	Rays	`nvwarpDstToRay()`
	Rays	Pixel coordinates	`nvwarpDstFromRay()`

The input rays do not need to be normalized.

The output rays are guaranteed to be normalized, unless they are out-of-domain. The result code of the function call indicates if any conversions are out of domain.

2. If any conversions are out of domain, check the specific conversions for NaN values to determine which conversions are out of domain.

2.4.4 Converting Between Pixel Coordinates and Angular Coordinates

Converting between pixel coordinates and angular coordinates is useful for determining a view. For source images and destination images, you can convert pixel coordinates into angular coordinates and convert angular coordinates into pixel coordinates.

Call the function that performs the conversion that you require.

Image	From	To	Function
Source image	Pixel coordinates	Angular coordinates	`nvwarpComputeSrcAngularFromPixelCoordinates()`
	Angular coordinates	Pixel coordinates	`nvwarpComputeSrcPixelFromAngularCoordinates()`
Destination image	Pixel coordinates	Angular coordinates	`nvwarpComputeDstAngularFromPixelCoordinates()`
	Angular coordinates	Pixel coordinates	`nvwarpComputeDstPixelFromAngularCoordinates()`

2.4.5 Changing the Direction of the Coordinate System

In the coordinate system used in NVWarp360, the X axis points rightwards, the Y axis points downwards, and the Z axis points outwards. However, in some modeling systems, the Y axis points upwards and the Z axis points inwards instead.

To change the direction of the coordinate system in a rotation matrix specification, call the `nvwarpConvertTransformBetweenYUpandYDown()` function. This function converts the matrix in-place. Use the same function to convert the coordinate system in either direction.

2.5 CONVERTING A COLOR SPACE

To simplify the processing of videos, NVWarp360 provides a CUDA node to convert from YUV 4:2:0 in NV12 form to RGBA or BGRA. Only RGBA and BGRA formats are supported.

You specify the parameters for this conversion in a structure.

1. Allocate a CUDA stream in one of the following ways:
   - Create a `cudaStream_t` opaque object.
   - Use the default stream 0.
2. Allocate CUDA pitched buffers for the input YUV image and the output RGBA image.
3. Load the YUV image into the buffer that you allocated for it in the previous step.
4. Declare an `nvwarpYUVRGBParams_t` structure.
5. Set the members of the `nvwarpYUVRGBParams_t` structure to the values that you require.

   You can use the convenience function `nvwarpComputeYCbCr2RgbMatrix()` to compute the values in this structure **except** `width`, `height`, and `cLocation`.
6. Call the `nvwarpConvertYUVNV12ToRGBA()` function to perform the conversion.

2.6 PERFORMING MULTIPLE WARPS ON AN IMAGE

To enable an application to perform several warps on the same source image, NVWarp360 provides a convenience function to perform multiple warps in as single function call. This function is provided in the API and as source code.

1. Allocate a CUDA stream in one of the following ways:
   - Create a `cudaStream_t` opaque object.
   - Use the default stream 0.
2. Allocate CUDA pitched buffers for the input YUV image and the intermediate RGBA source.
3. Load the YUV image into the buffer that you allocated for it in the previous step.

   Note that only YUV 4:2:0 in NV12 interleaved chrominance format is supported.
4. Create a `cudaTextureObject_t` object.
5. Allocate an array of destination CUDA pitched buffers for all the destination images that you want to produce.
6. Allocate an array of `size_t` integers to store the `rowBytes` parameter for each destination buffer.

   The `rowBytes` parameter specifies the byte stride, or pitch, between pixels vertically in each pitched buffer.
7. Declare an `nvwarpYUVRGBParams_t` structure.

8. Set the members of the `nvwarpYUVRGBParams_t` structure to the values that you require.

   You can use the convenience function `nvwarpComputeYCbCr2RgbMatrix()` to compute the values in this structure **except** `width`, `height`, and `cLocation`.

9. Declare an array of `nvwarpParams_t` structures that contains one structure for each destination buffer.

10. If you are writing your application in C, call the `nvwarpInitParams()` function to initialize each structure before setting any of its members.

    In C++, the constructor for `nvwarpParams_t` automatically calls this function.

    The `nvwarpInitParams()` function sets all members of the structure to a default value or to Not a Number (NaN).

11. Set the members of each `nvwarpParams_t` structure in the array to the values that you require for each warp.

12. Call the `nvwarpMultiWarp360()` function to perform the warps, passing the arrays of parameters, buffers, and `rowBytes`.

2.7 DESTROYING A WARP OBJECT

After completing the warps, release the resources allocated for the warps by destroying the warp object. To destroy a warp object and deallocate the resources allocated to it, call the `nvwarpDestroyInstance()` function.

```
...
//Clean up
nvwarpDestroyInstance(han);
...
```

Chapter 3. EXAMPLES

The supported warp destination surfaces include perspective, fisheye, equirectangular, cylindrical, rotated cylindrical, Panini, stereographic, and simulated pushbroom. The supported source surfaces include perspective, fisheye, and equirectangular.

3.1 DRAWING A PERSPECTIVE VIEW OF AN EQUIRECTANGULAR SPHERICAL PANORAMA

This example draws a perspective view of an equirectangular spherical panorama. The example takes advantage of the fact that the default rotation is the identity.

```
// Allocate a warp object
nvwarpHandle han;
nvwarpResult err = nvwarpCreateInstance(&han);
if (err)
   return err;

// Set the warp parameters
nvwarpParams_t params;
nvwarpInitParams(&params);
params.type           = NVWARP_EQUIRECT_PERSPECTIVE;
params.srcWidth       = 4096;
params.srcHeight      = 2048;
params.srcX0          = params.srcWidth  * .5f;
params.srcY0          = params.srcHeight * .5f;
params.srcFocalLength = params.srcHeight / M_PI;
params.dstWidth       = 1024;
params.dstHeight      = 768;
```

```
params.topAngle       = +M_PI / 6.f;
params.bottomAngle    = -M_PI / 6.f;
nvWarpSetParams(han, &params);

// Perform the warp
err = nvwarpWarpBuffer(han, 0, srcTex, dstAddr, dstRowBytes);

//Clean up
nvwarpDestroyInstance(han);
```

3.2 PANNING AROUND A VIEW

This example shows how to interactively pan around a view with a click and drag of the mouse. This example uses the view that was created in the previous example.

In this example, only the pan and tilt angles are required. These angles can be converted from mouse movements from the current destination focal length No roll angles are required. The functions whose names are prefixed with `My` are user-supplied functions.

```
void MyOnClick(const Point2d *startPoint) {
  while (MyMouseDown() {
   MyPoint2D deltaPt = (GetMyMouseLocation() - *startPoint);
   float fl = nvwarpDstFocalLength(han);
   float ang[3] = { deltaPt.x / fl, deltaPt.y / fl, 0 };
   char axes[] = "yx";
   nvwarpSetEulerRotation(han, ang, axes);
   err = nvwarpWarpBuffer(han, 0, srcTex, dstAddr, dstRowBytes);
   ShowMyBuffer(dstAddr, dstRowBytes);
  }
}
```

3.3 ROTATING A FISHEYE IMAGE AND RENDERING A SIDEWAYS CYLINDRICAL VIEW

This example rotates an image from a ceiling mounted fisheye les through 180° and renders a sideways cylindrical view of the image.

This example has only one rotation angle. The example shows how to use `nvwarpComputeParamsOutputResolution()` to determine the output resolution that maintains 1:1 pixel magnification.

```
// Set the warp parameters
nvwarpParams_t params;
nvwarpInitParams(&params);
params.type           = NVWARP_FISHEYE_CYLINDER;
params.srcWidth       = 4096;
params.srcHeight      = 4096;
params.srcX0          = (params.srcWidth  - 1) * .5;
params.srcY0          = (params.srcHeight - 1) * .5;
nvwarpComputeParamsSrcFocalLength(&params, M_PI * .5, (params.srcHeight
- 1) * .5);
params.topAngle       = 0;
params.bottomAngle    = -M_PI * .25;
nvwarpComputeParamsOutputResolution(&params, 4); // 1:1 mag
AllocateMyCUDAPitchedBuffer(params.dstWidth, params.dstHeight,
&dstAddr, &dstRowBytes);
params.rotAxes[0] = 'X';
params.rotAxes[1] = 0;
params.rotAngles[0] = M_PI * 0.5;
nvWarpSetParams(han, &params);

// Perform the warp
err = nvwarpWarpBuffer(han, 0, srcTex, dstAddr, dstRowBytes);
```

Chapter 4. NVWARP360 API REFERENCE

4.1 STRUCTURES

4.1.1 nvwarpFaceDescription_t

```
typedef struct nvwarpFaceDescription_t {
    unsigned        x;
    unsigned        y;
    nvwarpFaceID_t  to;
    nvwarpFaceID_t  up;
} nvwarpFaceDescription_t;
```

4.1.1.1 Members

`x`

Type: `unsigned`

The location of the left edge of the face in the cubemap relative to the upper-left pixel in the destination buffer.

`y`

Type: `unsigned`

The location of the top edge of the face in the cubemap relative to the upper-left pixel in the destination buffer.

`to`

Type: `nvwarpFaceID_t`

The ID of the look-at direction for the face. The look-at direction specifies which face is being chosen, for example, front or top. For possible values, see "nvwarpFaceID_t," on page 24.

`up`

Type: `nvwarpFaceID_t`

The ID of the up direction for the face. The up direction specifies the orientation of the face. For possible values, see "nvwarpFaceID_t," on page 24.

4.1.1.2 Remarks

This structure specifies the format of one of the six faces in a cubemap. It is used by the `nvwarpIntoCubemap()` function.

4.1.2 nvwarpHandle

```
typedef struct nvwarpObject *nvwarpHandle;
```

This structure encapsulates the state of the warp object. It is a pointer to an opaque object of type `nvwarpObject`. Most function calls include this handle as the first parameter.

4.1.3 nvwarpObject

```
struct nvwarpObject;
```

This structure represents an opaque warp object that is allocated by the `nvwarpCreateInstance()` function and deallocated by the `nvwarpDestroyInstance()` function.

4.1.4 nvwarpParams_t

```
typedef struct nvwarpParams_t
{
    nvwarpType_t    type;
    uint32_t        srcWidth,
                    srcHeight;
    float           srcX0,
```

```
                    srcY0;
    float           srcFocalLen;
    float           srcRadius;
    float           dist[5];
    uint32_t        dstWidth,
                    dstHeight;
    float           rotAngles[3];
    char            rotAxes[4];
    float           topAngle,
                    bottomAngle;
    void*           userData;
    float           control[4];
} nvwarpParams_t;
```

4.1.4.1 Members

`type`

Type: `nvwarpType_t`

The type of the warp. For details, see "nvwarpType_t," on page 25.

`srcWidth`

Type: `uint32_t`

The width of the source image in pixels.

`srcHeight`

Type: `uint32_t`

The height of the source image in pixels.

`srcX0`

Type: `float`

The X coordinate of the center of projection of the source image, in pixel coordinates. Typically, it is (`srcWidth`-1)/2, but may be different if the image is calibrated or wraps around.

`srcY0`

Type: `float`

The Y coordinate of the center of projection of the source image, in pixel coordinates. Typically, it is (`srcHeight`-1)/2, but may be different if the image is calibrated or wraps around.

`srcFocalLen`

Type: `float`

The focal length of the source, which specifies the angular pixel density of the image. The focal length (in pixels or pixels/radian) can be acquired from the image EXIF data, by multiplying the focal length in millimeters by the pixel density in pixels per millimeter. The relevant EXIF tags are as follows:

- `FocalLength` (37386)
- `FocalPlaneXResolution` (41486)
- `FocalPlaneYResolution` (41487)
- `FocalPlaneResolutionUnit` (41488)

`srcRadius`

Type: `float`

Circular clipping radius for fisheye sources. 0 implies no clipping (unimplemented).

`dist`

Type: `float`

A five-element array containing the distortion coefficients in the distortion polynomial for a fisheye source or a perspective source.

- For fisheye distortion, the distortion polynomial from which the radial distortion is calculated uses four radial coefficients:

$$r' = r(1 + k_0 r^2 + k_1 r^4 + k_2 r^6 + k_3 r^8)$$

- For perspective distortion, the distortion polynomial from which the radial distortion is calculated uses a total of five coefficients:
  - Three radial coefficients:

$$r' = r(1 + k_0 r^2 + k_1 r^4 + k_2 r^6)$$

  - Two tangential coefficients:

$$x' = x + [2k_3 xy + k_4(r^2 + 2x^2)]$$

$$y' = y + [k_3(r^2 + 2y^2) + 2k_4 xy]$$

The remaining coefficient (k_4 for a fisheye lens) is unused and should be set to zero. Tangential distortion is typically found only in compound lenses such as zoom lenses and other SLR lenses; typical small embedded cameras have simple lenses, with both k_3 and k_4 set to 0. The distortion coefficients are unused for equirectangular sources.

In these calculations, the radius has been normalized by the focal length. The higher-order coefficients take effect closer to the periphery. These coefficients are

compatible with OpenCV. For more information, see Camera calibration with OpenCV.

If the `dist` member is `NULL`, all distortion coefficients are set to zero.

`dstWidth`

Type: `uint32_t`

The width of the destination image in pixels.

`dstHeight`

Type: `uint32_t`

The height of the destination image in pixels.

`rotAngles`

Type: `float`

A three-element array containing the view rotation angles (Euler angles) in radians.

`rotAxes`

Type: `char`

A four-element array of three-character C-strings terminated by `\0` that represent the axes about which the view rotation angles `rotAngles` are specified.

The case of each character specifies whether rotation is about the positive or negative axes:

- Upper case `X`, `Y`, and `Z` specify rotation about the positive X, Y and Z axes.
- Lower case `x`, `y`, and `z` specify rotation about the negative X, Y and Z axes.

The direction of positive rotation for each axis is as follows:

- **X axis:** upwards about the rightward-pointing X axis
- **Y axis:** rightwards about the downward-pointing Y axis
- **Z axis:** clockwise about the outward-pointing Z axis

The default is `YXZ`, which corresponds to yaw, pitch, and roll, as set by `nvwarpInitParams()`.

These rotations and axes specify the rotation of a *camera* to view an image that is considered to be straight ahead, for example, to pan around a panorama. To rotate the *image* and keep the camera upright, to embed the image in a 3D image, specify the angles and axes in reverse order.

`topAngle`

Type: `float`

The top field of view angle in radians.

`bottomAngle`

Type: `float`

The bottom field of view angle in radians.

`userData`

Type: `void*`

A pointer to user data. This data is useful for callbacks and resynchronizing with asynchronous processes.

`control`

Type: `float`

A four-element array of projection specific controls, with values nominally between 0 and 1, although some controls allow larger or smaller values.

The Panini, Stereographic and Pushbroom projections use `control[0]`. For these projections, `control[0]=0` makes them identical to the perspective output.

The canonical stereographic has `control[0]=1`.

4.1.4.2 Remarks

This structure contains warp parameters. In C++, the constructor for `nvwarpParams_t` automatically calls `nvwarpInitParams()` to set the members of the structure to a default value or to NaN. In C, call this function to initialize the structure before setting any its members.

This structure is used in the following functions:

- `nvwarpComputeBoxAxialAngleRange()`
- `nvwarpComputeParamsAxialAngleRange()`
- `nvwarpComputeParamsOutputResolution()`
- `nvwarpComputeParamsSrcFocalLength()`
- `nvwarpGetParams()`
- `nvwarpInitParams()`
- `nvwarpSetParams()`

4.1.5 nvwarpYUVRGBParams_t

```
typedef struct nvwarpYUVRGBParams_t
{
    uint32_t width;
    uint32_t height;
```

```
    uint32_t cLocation;
    float    ry,
             rcb,
             rcr,
             gy,
             gcb,
             gcr,
             by,
             bcb,
             bcr;
    float    yOffset,
             cOffset;
} nvwarpYUVRGBParams_t;
```

4.1.5.1 Members

`width`

Type: `uint32_t`

The width of the Y and RGB channels. Chroma has half the width but, because it is interleaved in NV12, it is necessarily even.

`height`

Type: `uint32_t`

The height of the Y and RGB channels. Because chroma has half the height, this height must be even.

`cLocation`

Type: `uint32_t`

Specifies where chroma is sampled in relation to luma as follows:

- 0 specifies chroma sampled cosited horizontally with luma.
- 1 specifies chroma sampled halfway between luma samples horizontally.

To determine which value to use, get the video header.

`ry`

Type: `float`

Coefficients for R with respect to Y, including normalization scaling.

`rcb`

Type: `float`

Coefficients for R with respect to Cb, including normalization scaling.

`rcr`

Type: `float`

Coefficients for R with respect to Cr, including normalization scaling.

`gy`

Type: `float`

Coefficients for G with respect to Y, including normalization scaling.

`gcb`

Type: `float`

Coefficients for G with respect to Cb, including normalization scaling.

`gcr`

Type: `float`

Coefficients for G with respect to Cr, including normalization scaling.

`by`

Type: `float`

Coefficients for B with respect to Y, including normalization scaling.

`bcb`

Type: `float`

Coefficients for B with respect to Cb, including normalization scaling.

`bcr`

Type: `float`

Coefficients for B with respect to Cr, including normalization scaling.

`yOffset`

Type: `float`

Offset of luma, typically 16.

`cOffset`

Type: `float`

Offset of chroma, typically 128.

4.1.5.2 Remarks

This structure specifies the parameters for conversion by a CUDA node from YUV 4:2:0 in NV12 form to RGBA. It is used by the following functions:

- `nvwarpComputeYCbCr2RgbMatrix()`
- `nvwarpConvertYUVNV12ToRGBA()`
- `nvwarpMultiWarp360()`

4.2 ENUMERATIONS

4.2.1 nvwarpFaceID_t

```
typedef enum nvwarpFaceID_t {
    NVWARP_FRONT    = 0
    NVWARP_RIGHT    = 1
    NVWARP_BACK     = 2,
    NVWARP_LEFT     = 3,
    NVWARP_TOP      = 4,
    NVWARP_BOTTOM   = 5
} nvwarpFaceID_t;
```

4.2.1.1 Values

`NVWARP_FRONT = 0`

Toward the front face.

`NVWARP_RIGHT = 1`

Toward the right face.

`NVWARP_BACK = 2`

Toward the back face.

`NVWARP_LEFT = 3`

Toward the left face.

`NVWARP_TOP = 4`

Toward the top face.

`NVWARP_BOTTOM = 5`

Toward the bottom face.

4.2.1.2 Remarks

The `nvwarpFaceID_t` enumeration is used in `nvwarpFaceDescription_t` to define both the look-at direction and the up direction for each face in a cubemap.

4.2.2 nvwarpType_t

```
typedef enum nvwarpType_t
{
    NVWARP_EQUIRECT_CYLINDER,
    NVWARP_EQUIRECT_EQUIRECT,
    NVWARP_EQUIRECT_FISHEYE,
    NVWARP_EQUIRECT_PANINI,
    NVWARP_EQUIRECT_PERSPECTIVE,
    NVWARP_EQUIRECT_PUSHBROOM,
    NVWARP_EQUIRECT_STEREOGRAPHIC,
    NVWARP_EQUIRECT_VERTCYLINDER,
    NVWARP_FISHEYE_CYLINDER,
    NVWARP_FISHEYE_EQUIRECT,
    NVWARP_FISHEYE_FISHEYE,
    NVWARP_FISHEYE_PANINI,
    NVWARP_FISHEYE_PERSPECTIVE,
    NVWARP_FISHEYE_PUSHBROOM,
    NVWARP_FISHEYE_VERTCYLINDER,
    NVWARP_PERSPECTIVE_EQUIRECT,
    NVWARP_PERSPECTIVE_PANINI,
    NVWARP_PERSPECTIVE_PERSPECTIVE,
} nvwarpType_t;
```

4.2.2.1 Values

Each enumerator represents a combination of source warp type and destination warp type, encoded as a concatenation of the types as follows:

`NVWARP_`*source-warp-type*`_`*destination-warp-type*

For example, `NVWARP_EQUIRECT_CYLINDER` represents a combination of warp types that warps an `EQUIRECT` source into a `CYLINDER` destination.

Only useful combinations of source warp type and destination warp type are implemented. Combinations that are not useful are not implemented.

The possible source warp types are as follows:

- `EQUIRECT`
- `FISHEYE`
- `PERSPECTIVE`

All warp types are possible destination warp types.

Warp Type	Name	Description
EQUIRECT	Equirectangular Spherical	Spherical image indexed by longitude and latitude. Commonly used to represent 360°×180° panoramas, though neither the domain nor the shape is restricted.
FISHEYE	Equidistant Fisheye, Equidistant Azimuthal	Spherical image, with a round or rectangular boundary. Captured by wide angle lenses. Nominally 180° across the diameter but may vary from 120° to 360°. The fisheye source can have a distortion curve to model non-ideal lenses.
PERSPECTIVE	Perspective, Projective, Rectilinear	The most common type of projection captured by cameras, where straight lines appear straight. The perspective source can have a distortion curve to model non-ideal lenses.
CYLINDER	Horizontally-panned cylinder	Typical cylindrical projection, with a vertical axis, panned horizontally. Useful for sideward views.
VERTCYLINDER	Vertically-panned cylinder	Rotated cylinder, so that the axis is horizontal and can be panned vertically. Useful for downward views.
PANINI	Panini (sometimes spelled “Pannini”)	A projection similar to the perspective projection because it tries to keep straight lines straight but can have a wide horizontal field of view of up to 240° or above. The Panini projection can be considered a cylindrical version of the spherical stereographic projection. The Panini destination has a control parameter to vary the amount of warp: • 0 is identical to the perspective destination. • Increasing the control value pulls in more content from the sides. • Negative values slightly greater than -1 are allowed but are not useful.
STEREOGRAPHIC	Stereographic, Little Planet	A spherical azimuthal image, similar to the equidistant fisheye but more compressed toward the edges. Panoramic photographers use a downward stereographic projection similar to a little planet, achieved with control=1. The stereographic destination has a control parameter to vary the amount of warp: • 0 is identical to the perspective destination. • 1 is the canonical little planet view. • Increasing the control value pulls in more content from all around. • Negative values slightly greater than -1 are allowed but are not useful.

Warp Type	Name	Description
PUSHBROOM	Simulated pushbroom	A simulation of a pushbroom camera, although pushbroom characteristics are achieved only at a particular depth. The pushbroom destination has a control parameter to vary the amount of warp: • 0 yields a perspective projection. • A positive value selects the depth at which the pushbroom effect is achieved. Values up to 1 and greater are useful.

4.2.2.2 Remarks

The `nvwarpType_t` enumeration is used to specify combinations of source warp type and destination warp type. It is used by the following functions:

- `nvwarpComputeSrcFocalLength()`
- `nvwarpGetWarpType()`
- `nvwarpSetWarpType()`

4.3 FUNCTIONS

4.3.1 nvwarpComputeBoxAxialAngleRange

```
nvwarpResult nvwarpComputeBoxAxialAngleRange(
    const nvwarpParams_t *params,
    const float srcBoundingBox[4],
    float minMaxXY[4]);
```

4.3.1.1 Parameters

`params`

Type: `const nvwarpParams_t *`

Pointer to the `nvwarpParams_t` structure that contains values to be used in the calculation. For details of this structure, see "nvwarpParams_t," on page 17.

Ensure that the following members of this structure are set before this function is called:

- `srcWidth`
- `srcHeight`
- `srcFocalLength`
- `srcX0`
- `srcY0`
- `dist`

`srcBoundingBox`

Type: `const float`

A four-element array containing the position in pixels of each edge of the bounding box to be used in the calculation in the following order:

1. Left
2. Top
3. Right
4. Bottom

`minMaxXY`

Type: `float`

A four-element array to which the axial angle range angles are written is this order:

1. Minimum longitude
2. Maximum longitude
3. Minimum latitude
4. Maximum latitude

4.3.1.2 Return Value

Returns `NVWARP_SUCCESS` on success.

4.3.1.3 Remarks

This function optimizes the projection rectangle by calculating the axial angle range from the specified bounding box. The axial angle range is defined by the maximum and minimum longitude and latitude angles These angles correspond to the angles on the coordinate axes.

To perform more general conversions from pixel to angular coordinates, use these functions:

- `nvwarpComputeSrcAngularFromPixelCoordinates()`
- `nvwarpComputeDstAngularFromPixelCoordinates()`

4.3.2 nvwarpComputeDstAngularFromPixelCoordinates

```
nvwarpResult nvwarpComputeDstAngularFromPixelCoordinates(
    const nvwarpHandle han,
    uint32_t numPts,
    const float *pts2D,
    float *ang2D
);
```

4.3.2.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`numPts`

Type: `uint32_t`

The number of coordinate pairs to be converted.

`pts2D`

Type: `const float *`

A pointer to an array of input pixel coordinates to be converted.

`ang2D`

Type: `float *`

A pointer to an array into which the converted output angular coordinates are to be placed.

4.3.2.2 Return Value

Returns one of the following result codes:

- `NVWARP_SUCCESS` on success
- `NVWARP_ERR_DOMAIN` if any coordinate is out of domain
- `NVWARP_ERR_UNIMPLEMENTED` if the specified warp type is unsupported

4.3.2.3 Remarks

This function converts the pixel coordinates of a destination image to angular coordinates. To enable the coordinates to be converted in place, the array used for the input coordinates can also be used for the output coordinates.

4.3.3 nvwarpComputeDstFocalLength

```
void nvwarpComputeDstFocalLength(
    nvwarpHandle han,
    float topAngle,
    float bottomAngle,
    uint32_t dstWidth,
    uint32_t dstHeight
);
```

4.3.3.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`topAngle`

Type: `float`

The top view angle in radians.

`bottomAngle`

Type: `float`

The bottom view angle in radians.

`dstWidth`

Type: `uint32_t`

The width of the destination image in pixels.

`dstHeight`

Type: `uint32_t`

The height of the destination image in pixels.

4.3.3.2 Return Value

None.

4.3.3.3 Remarks

This function calculates a single destination focal length from the top and bottom view angles and the destination image dimensions. The destination focal lengths, principal points and image dimensions are updated after this function is called.

4.3.4 nvwarpComputeDstFocalLengths

```
nvwarpResult nvwarpComputeDstFocalLengths(
    nvwarpHandle han,
    float topAxialAngle,
    float botAxialAngle,
    float leftAxialAngle,
    float rightAxialAngle,
```

```
    uint32_t dstWidth,
    uint32_t dstHeight
);
```

4.3.4.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`topAxialAngle`

Type: `float`

The top view angle in radians.

`botAxialAngle`

Type: `float`

The bottom view angle in radians.

`leftAxialAngle`

Type: `float`

The left view angle in radians.

`rightAxialAngle`

Type: `float`

The right view angle in radians.

`dstWidth`

Type: `uint32_t`

The width of the destination image in pixels.

`dstHeight`

Type: `uint32_t`

The height of the destination image in pixels.

4.3.4.2 Return Value

Returns `NVWARP_SUCCESS` on success.

4.3.4.3 Remarks

This function calculates two destination focal lengths from the top and bottom view angles, the left and right view angles, and the destination image dimensions. The destination focal lengths, principal points and image dimensions are updated after this function is called.

4.3.5 nvwarpComputeDstPixelFromAngularCoordinates

```
nvwarpResult nvwarpComputeDstPixelFromAngularCoordinates(
    const nvwarpHandle han,
    uint32_t numPts,
    const float *ang2D,
    float *pts2D
);
```

4.3.5.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`numPts`

Type: `uint32_t`

The number of coordinate pairs to be converted.

`ang2D`

Type: `const float *`

A pointer to an array of input angular coordinates to be converted, specified as pairs of `{ longitude, latitude }` angles in radians.

`pts2D`

Type: `float *`

A pointer to an array in which to place the converted output pixel coordinates.

4.3.5.2 Return Value

Returns one of the following result codes:

- `NVWARP_SUCCESS` on success

- NVWARP_ERR_DOMAIN if any coordinate is out of domain
- NVWARP_ERR_UNIMPLEMENTED if the specified warp type is unsupported

4.3.5.3 Remarks

This function converts the angular coordinates of a destination image to pixel coordinates.

4.3.6 nvwarpComputeParamsAxialAngleRange

```
nvwarpResult nvwarpComputeParamsAxialAngleRange(
    const nvwarpParams_t *params,
    float minMaxXY[4]);
```

4.3.6.1 Parameters

`params`

Type: `const nvwarpParams_t *`

Pointer to the `nvwarpParams_t` structure that contains values to be used in the calculation. For details of this structure, see "nvwarpParams_t," on page 17.

Ensure that the following members of this structure are set before this function is called:

- `srcWidth`
- `srcHeight`
- `srcFocalLength`
- `srcX0`
- `srcY0`
- `dist`

`minMaxXY`

Type: `float`

A four-element array to which the axial angle range angles are written is this order:

1. Minimum longitude
2. Maximum longitude
3. Minimum latitude
4. Maximum latitude

4.3.6.2 Return Value

Returns `NVWARP_SUCCESS` on success.

4.3.6.3 Remarks

This function optimizes the projection rectangle by calculating the axial angle range from the dimensions of the source image set in the specified the `nvwarpParams_t` structure. The axial angle range is defined by the maximum and minimum longitude and latitude angles These angles correspond to the angles on the coordinate axes.

To perform more general conversions from pixel to angular coordinates, use these functions:

- `nvwarpComputeSrcAngularFromPixelCoordinates()`
- `nvwarpComputeDstAngularFromPixelCoordinates()`

4.3.7 nvwarpComputeParamsOutputResolution

```
nvwarpResult nvwarpComputeParamsOutputResolution(
     nvwarpParams_t *params,
     float aspectRatio
);
```

4.3.7.1 Parameters

`params`

Type: `nvwarpParams_t *`

Pointer to the `nvwarpParams_t` structure that contains values to be used in the calculation. For details of this structure, see “nvwarpParams_t,” on page 17.

Ensure that the following members of this structure are set before this function is called:

- `topAngle`
- `bottomAngle`
- `srcFocalLen`
- `dist`

This function adjusts the destination focal length to ensure that the `topAngle` member corresponds to the top edge of the destination image and the `bottomAngle` member corresponds to the bottom edge of the destination image.

`aspectRatio`

Type: `float`

The ratio of the width of the destination image to the height of the destination image.

4.3.7.2 Return Value

Returns `NVWARP_SUCCESS` on success.

4.3.7.3 Remarks

This function calculates the height of the destination image from the specified aspect ratio to create a destination image in which the focal length is equal to the focal length of the source image, rounded up or rounded down to the closest even integer.

For the spatial images that the NVWarp360 SDK supports, you can use this function to set the resolution of the destination image equal to the resolution of the source image. For these images, the most meaningful measure of resolution is *angular pixel density*, that is, the number of pixels per degree or per radian. Because the angular pixel density at the center of projection is equal to the focal length, the resolution of the source image and destination image can be matched by matching their focal lengths.

The destination width is the height multiplied by the aspect ratio. This function updates the `dstWidth` and `dstHeight` members of the specified `nvwarpParams_t` structure.

To meet specific requirements for the dimensions of the destination image, or to lower the resolution, you can change the `dstWidth` and `dstHeight` parameters after calling this function. Note that if `dstHeight` set by this function is subsequently reduced to one third or less, aliasing may start to be visible in the destination image.

4.3.8 nvwarpComputeParamsSrcFocalLength

```
nvwarpResult nvwarpComputeParamsSrcFocalLength(
    nvwarpParams_t *params,
    float angle,
    float radius
);
```

4.3.8.1 Parameters

`params`

Type: `nvwarpParams_t *`

Pointer to the `nvwarpParams_t` structure that contains values to be used in the calculation. For details of this structure, see “nvwarpParams_t,” on page 17.

Ensure that the following members of this structure are set before this function is called:

- `type`
- `srcX0`

- srcY0
- dist

`angle`

Type: `float`

The angle in radians from the optical axis, which corresponds to the specified radius from the source's principal point.

`radius`

Type: `float`

The radius in pixels from the source's principal point, which corresponds to the specified angle from the optical axis.

4.3.8.2 Return Value

Returns `NVWARP_SUCCESS` on success.

4.3.8.3 Remarks

This function calculates the focal length of the source image from measurements of the radius and its corresponding angle. Typically, one of the following pairs of values is used for the radius and angle:

- For measurements in the horizontal direction:
  - **Radius:** half the source width
  - **Angle:** half the horizontal field of view
- For measurements in the vertical direction:
  - **Radius:** half of the source height
  - **Angle:** half the vertical field of view

However, any pair of measurements of the radius and its corresponding angle can be used. Note that because camera-acquired images do not have a perfectly symmetric field of view (FOV), the pairs of values typically used are not exact but yield a good approximation.

4.3.9 nvwarpComputeSrcAngularFromPixelCoordinates

```
nvwarpResult nvwarpComputeSrcAngularFromPixelCoordinates(
     const nvwarpHandle han,
     uint32_t numPts,
     const float *pts2D,
     float *ang2D
);
```

4.3.9.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`numPts`

Type: `uint32_t`

The number of coordinate pairs to be converted.

`pts2D`

Type: `const float *`

A pointer to an array of the input pixel coordinates to be converted.

`ang2D`

Type: `float *`

A pointer to an array of the output angular coordinates converted from the input pixel coordinates. To enable the coordinates to be converted in place, the array used for the input coordinates can also be used for the output coordinates.

4.3.9.2 Return Value

Returns one of the following result codes:

- `NVWARP_SUCCESS` on success
- `NVWARP_ERR_DOMAIN` if any coordinate is out of domain
- `NVWARP_ERR_UNIMPLEMENTED` if the specified warp type is unsupported

4.3.9.3 Remarks

This function converts the pixel coordinates of a source image to angular coordinates.

4.3.10 nvwarpComputeSrcFocalLength

```
nvwarpResult nvwarpComputeSrcFocalLength(
    nvwarpHandle han,
    nvwarpType_t warpType,
    float radius,
    float angle,
    float dist[5]
);
```

4.3.10.1 Parameters

`han`

Type: `nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`warpType`

Type: `nvwarpType_t`

The type of the warp. For details, see "nvwarpType_t," on page 25.

`radius`

Type: `float`

The radius in pixels from the source's principal point, which corresponds to the specified angle of inclination from the optical axis.

`angle`

Type: `float`

The angle, in radians of inclination from the optical axis, which corresponds to the specified radius from the source's principal point.

`dist`

Type: `float`

A five-element array containing the distortion coefficients in the distortion polynomial for a fisheye source or a perspective source.

- For fisheye distortion, the distortion polynomial from which the radial distortion is calculated uses four radial coefficients:

$$r' = r(1 + k_0 r^2 + k_1 r^4 + k_2 r^6 + k_3 r^8)$$

- For perspective distortion, the distortion polynomial from which the radial distortion is calculated uses three radial coefficients:

$$r' = r(1 + k_0 r^2 + k_1 r^4 + k_2 r^6)$$

The remaining coefficients are reserved for tangential distortion correction and should be set to zero.

In these calculations, the radius has been normalized by the focal length. The higher-order coefficients take effect closer to the periphery.

If the source image contains no distortion or if the warp type does not support distortion, set `dist` to `NULL`.

4.3.10.2 Return Value

Returns `NVWARP_SUCCESS` on success.

4.3.10.3 Remarks

This function calculates the focal length of the source image from an angular measurement and corresponding radius, for example, half horizontal field of view (HFOV) and half width.

4.3.11 nvwarpComputeSrcPixelFromAngularCoordinates

```
nvwarpResult nvwarpComputeSrcPixelFromAngularCoordinates(
    const nvwarpHandle han,
    uint32_t numPts,
    const float *ang2D,
    float *pts2D
);
```

4.3.11.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`numPts`

Type: `uint32_t`

The number of coordinate pairs to be converted.

`ang2D`

Type: `const float *`

A pointer to the array of input angular coordinates to be converted, specified as pairs of `{ longitude, latitude }` angles in radians.

`pts2D`

Type: `float *`

A pointer to an array where the converted output pixel coordinates are to be stored.

4.3.11.2 Return Value

Returns one of the following result codes:

- `NVWARP_SUCCESS` on success
- `NVWARP_ERR_DOMAIN` if any coordinate is out of domain
- `NVWARP_ERR_UNIMPLEMENTED` if the specified warp type is unsupported

4.3.11.3 Remarks

This function converts the angular coordinates of a source image to pixel coordinates.

4.3.12 nvwarpComputeYCbCr2RgbMatrix

```
void nvwarpComputeYCbCr2RgbMatrix(
    nvwarpYUVRGBParams_t *p,
    uint32_t matrix_coefficients,
    uint32_t video_full_range_flag,
    uint32_t bit_depth,
    uint32_t normalized_input,
    uint32_t normalized_output
);
```

4.3.12.1 Parameters

`p`

Type: `nvwarpYUVRGBParams_t *`

Pointer to the `nvwarpYUVRGBParams_t` structure the values in which are to be computed. For details of this structure, see "nvwarpYUVRGBParams_t," on page 21.

`matrix_coefficients`

Type: `uint32_t`

The RGB coefficients to be used in the computation. The matrix coefficients are symbolic for various kinds of chromaticity standards.

Coefficient	Standard
709, 1 or 2	BT.709
601, 5, or 6	BT.601
2020, 9, or 10	BR.2020
4 or 0xFCC	FCC
240 or 7	SMPTE 240M

The matrix coefficients produced by this function are for RGB. If you require BGR, swap the order of the B and R coefficients. Regardless of whether this matrix has been initialized for RGB or BGR, the last component is always A (set to 255). Therefore, only RGBA and BGRA formats are supported.

`video_full_range_flag`

Type: `uint32_t`

An integer that indicates whether the video is standard or full range:

- 0 for standard video range (16-240)
- 1 for full range (0-255)

Typical video is standard video range. Full range may be used in production.

`bit_depth`

Type: `uint32_t`

Set to 8, which is the only depth that `nvwarpConvertYUVNV12ToRGBA()` supports.

`normalized_input`

Type: `uint32_t`

An integer that indicates whether the input is normalized:

- 0 if the input is not normalized
- 1 if the input is normalized

Set to 0 for `nvwarpConvertYUVNV12ToRGBA()`.

`normalized_output`

Type: `uint32_t`

An integer that indicates whether the output is normalized:

- 0 if the output is not normalized
- 1 if the output is normalized.

Set to 1 for `nvwarpConvertYUVNV12ToRGBA()`.

4.3.12.2 Return Value

None.

4.3.12.3 Remarks

This function computes the values in the `nvwarpYUVRGBParams_t` structure to be used in a color space conversion **except** `width`, `height`, and `cLocation`.

4.3.13 nvwarpConvertTransformBetweenYUpandYDown

```
void nvwarpConvertTransformBetweenYUpandYDown(
    const float fr[9],
    float to[9]
);
```

4.3.13.1 Parameters

`fr`

Type: `const float`

A nine-element array containing a 3×3 matrix for the current coordinate system.

`to`

Type: `float`

A nine-element array containing a 3×3 matrix for the target coordinate system. This parameter may be the same as `fr` for an in-place conversion

4.3.13.2 Return Value

None.

4.3.13.3 Remarks

This utility function changes a rotation matrix specified in one right-handed coordinate system to another coordinate system, where both coordinate systems have either Y-up or Y-down, and both have X to the right. The Y-down convention is used in Warp360.

4.3.14 nvwarpConvertYUVNV12ToRGBA

```
nvwarpResult nvwarpConvertYUVNV12ToRGBA(
    cudaStream_t stream,
    const nvwarpYUVRGBParams_t *params,
    const void *yuv,
    size_t yuvRowBytes,
    void *dst,
    size_t dstRowBytes
);
```

4.3.14.1 Parameters

`stream`

Type: `cudaStream_t`

The CUDA stream on which the conversion is to be executed. This stream can be allocated with `cudaStreamCreate()` or the default stream 0 can be used.

`params`

Type: `const nvwarpYUVRGBParams_t *`

Pointer to the `nvwarpYUVRGBParams_t` structure that contains values to be used in the conversion. For details of this structure, see "nvwarpYUVRGBParams_t," on page 21.

`yuv`

Type: `const void *`

Pointer to a pitched buffer containing the input YUV 4:2:0 image in NV12 format. This function converts the image into RGBA or BGRA.

`yuvRowBytes`

Type: `size_t`

The byte stride between pixels vertically in the pitched buffer specified in `yuv`.

`dst`

Type: `void *`

Pointer to a pitched buffer that is to contain the output converted RGBA or BGRA color space.

`dstRowBytes`

Type: `size_t`

The byte stride between pixels vertically in the pitched buffer specified in `dst`.

4.3.14.2 Return Value

Returns `NVWARP_SUCCESS` on success.

4.3.14.3 Remarks

This function performs a color space conversion from YUV 4:2:0 in interleaved chroma NV12 format to RGBA or BGRA.

4.3.15 nvwarpCreateInstance

```
nvwarpResult nvwarpCreateInstance(
    nvwarpHandle *han
);
```

4.3.15.1 Parameters

`han`

Type: `nvwarpHandle *`

A pointer to an opaque `nvWarp` object that is allocated by this function.

4.3.15.2 Return Value

Returns `NVWARP_SUCCESS` on success.

4.3.15.3 Remarks

This function allocates an opaque `nvWarp` object and returns a handle for this object in the out parameter `han`.

4.3.16 nvwarpDestroyInstance

```
void nvwarpDestroyInstance(
    nvwarpHandle han
);
```

4.3.16.1 Parameters

`han`

Type: `nvwarpHandle`

The handle for the opaque `nvWarp` object to be destroyed.

4.3.16.2 Return Value

None.

4.3.16.3 Remarks

This function destroys an opaque `nvWarp` object and deallocates the resources allocated to it.

4.3.17 nvwarpDstFromRay

```
nvwarpResult nvwarpDstFromRay(
    const nvwarpHandle han,
    uint32_t numRays,
    const float *rays3D,
    float *pts2D
);
```

4.3.17.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`numRays`

Type: `uint32_t`

The number of rays to be converted.

`rays3D`

Type: `const float *`

A pointer to the array of input 3D rays to be converted. These input rays do not need to be normalized.

`pts2D`

Type: `float *`

A pointer to a place to store the converted output 2D pixel coordinates.

4.3.17.2 Return Value

Returns one of the following result codes:

- `NVWARP_SUCCESS` on success
- `NVWARP_ERR_DOMAIN` if any conversions are out of domain
- `NVWARP_ERR_UNIMPLEMENTED` if the specified warp type is unsupported

4.3.17.3 Remarks

This function converts 3D rays from a destination image to 2D pixel coordinates.

4.3.18 nvwarpDstToRay

```
nvwarpResult nvwarpDstToRay(
    const nvwarpHandle han,
    uint32_t numPts,
    const float *pts2D,
    float *rays3D
);
```

4.3.18.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`numPts`

Type: `uint32_t`

The number of coordinate pairs to be converted.

`pts2D`

Type: `const float *`

A pointer to an array of the input 2D pixel coordinates to be converted.

`rays3D`

Type: `float *`

A pointer to an array into which the converted output 3D rays will be placed.

The output rays are guaranteed to be normalized, unless they are out-of-domain, in which case they are NaN. Note that this parameter must be different than the array of pixel coordinates to be converted because rays are 3D and points are 2D.

4.3.18.2 Return Value

Returns one of the following result codes:

- `NVWARP_SUCCESS` on success
- `NVWARP_ERR_DOMAIN` if any conversions are out of domain
- `NVWARP_ERR_UNIMPLEMENTED` if the specified warp type is unsupported

4.3.18.3 Remarks

This function converts 2D pixel coordinates of a destination image to 3D rays.

4.3.19 nvwarpErrorStringFromCode

```
const char* nvwarpErrorStringFromCode(
    nvwarpResult err
);
```

4.3.19.1 Parameters

`err`

Type: `nvwarpResult`

The `nvwarpResult` result code to be converted.

4.3.19.2 Return Value

A pointer to a human-readable string representation of the `nvwarpResult` result code.

4.3.19.3 Remarks

This function converts an `nvwarpResult` result code into a human-readable string.

4.3.20 nvwarpGetBlock

```
void nvwarpGetBlock(
    const nvwarpHandle han,
    dim3 *dim_block
);
```

4.3.20.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`dim_block`

Type: `dim3 *`

Pointer to a CUDA `dim3` structure.

4.3.20.2 Return Value

None.

4.3.20.3 Remarks

This function writes the CUDA block to the address specified by the parameter `dim_block`. The CUDA block determines the distribution of work.

4.3.21 nvwarpGetControl

```
float nvwarpGetControl(
    const nvwarpHandle han,
    uint32_t index
);
```

4.3.21.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`index`

Type: `uint32_t`

The index of the control whose value is to be retrieved. Only one control (`index=0`) is used.

4.3.21.2 Return Value

Returns the value of the control with the specified index, or NaN if there is no control with the specified index. Only one control (`index=0`) is used.

4.3.21.3 Remarks

This function retrieves the value of the control with the specified index.

4.3.22 nvwarpGetDistortion

```
void nvwarpGetDistortion(
    const nvwarpHandle han,
    float d[5]
);
```

4.3.22.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`d`

Type: `float`

A five-element array to which the coefficients in the distortion polynomial are to be written.

4.3.22.2 Return Value

None.

4.3.22.3 Remarks

This function gets the coefficients in the distortion polynomial for a fisheye source or a perspective source.

4.3.23 nvwarpGetDstFocalLength

```
float nvwarpGetDstFocalLength(
    const nvwarpHandle han,
    float *fy
);
```

4.3.23.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`fy`

Type: `float *`

The address to which the focal length of the destination image in the Y direction is to be written. If only the focal length is the X direction is needed, set `fy` to `NULL`. Because most applications use the same focal length for X and Y, `NULL` is commonly passed for `fy`.

4.3.23.2 Return Value

Returns the focal length of the destination image in the X direction.

4.3.23.3 Remarks

This function returns the focal length of the destination image in the X direction and writes the focal length in the Y direction to the address specified by the parameter `fy`. It can get focal lengths that have been set by any of the following functions:

- `nvwarpSetDstFocalLengths()`
- `nvwarpComputeDstFocalLength()`
- `nvwarpComputeDstFocalLengths()`

4.3.24 nvwarpGetDstPrincipalPoint

```
void nvwarpGetDstPrincipalPoint(
    const nvwarpHandle han,
    float xy[2],
    uint32_t relToCenter
    );
```

4.3.24.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`xy`

Type: `float`

A two-element array to which the X coordinate and Y coordinate of the principal point of the destination image are to be written.

`relToCenter`

Type: `uint32_t`

A `uint32_t` variable that specifies the coordinate system of the principal point:

`0` Relative to pixel (0,0)

`1` Relative to the center of the image

4.3.24.2 Return Value

None.

4.3.24.3 Remarks

This function gets the principal point of the destination image.

4.3.25 nvwarpGetDstWidthHeight

```
void nvwarpGetDstWidthHeight(
    const nvwarpHandle han,
    uint32_t wh[2]
);
```

4.3.25.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`wh`

Type: `uint32_t`

A two-element array to which the width and height of the destination image are to be written.

4.3.25.2 Return Value

None.

4.3.25.3 Remarks

This function writes the width and height of the destination image to the array specified by the parameter `wh`.

4.3.26 nvwarpGetParams

```
void nvwarpGetParams(
    const nvwarpHandle han,
    nvwarpParams_t *params
);
```

4.3.26.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`params`

Type: `const nvwarpParams_t *`

Pointer to an `nvwarpParams_t` structure into which the warp parameters are to be written. For details of this structure, see "nvwarpParams_t," on page 17.

4.3.26.2 Return Value

None.

4.3.26.3 Remarks

This function gets the values of the warp parameters that the application has set, regardless of which functions have been used to set them. However, restrictions apply if parameters have been set by certain functions:

- The `nvwarpParams_t` structure has only a single source focal length. Therefore, if `nvwarpSetSrcFocalLengths()` was used to set a different focal length for X and Y, only the X focal length is returned in this structure.
- The `nvwarpParams_t` structure does not have a field to return the destination principal point if it is changed with `nvwarpGetDstPrincipalPoint()`.
- The top and bottom angles reflect the values last set by `nvwarpSetParams()` or `nvwarpComputeDstFocalLength()`. They are not recomputed if any of the following functions is called:
  - `nvwarpSetDstFocalLengths()`
  - `nvwarpComputeDstFocalLengths()`
  - `nvwarpSetDstPrincipalPoint()`
- Only the rotation angles for the rotation axis sequences `YXZ`, `ZXY` and `ZXZ` are returned. If a different sequence is used to specify the rotation, it is returned as `YXZ`. Note, however, that the associated rotation matrix is the same.

4.3.27 nvwarpGetPixelPhase

```
uint32_t nvwarpGetPixelPhase(
    const nvwarpHandle han
);
```

4.3.27.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

4.3.27.2 Return Value

Returns the pixel phase.

4.3.27.3 Remarks

This function gets the pixel phase, which determines whether pixels are considered to be sampled on the integers or the integers plus-one-half.

4.3.28 nvwarpGetRotation

```
void nvwarpGetRotation(
    const nvwarpHandle han,
    float R[9]
);
```

4.3.28.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`R`

Type: `float`

A nine-element array in which the rotation matrix is to be written.

4.3.28.2 Return Value

None.

4.3.28.3 Remarks

This function writes the rotation matrix to the address specified by the parameter `R`. It can get a rotation matrix that has been set by any of the following functions:

- `nvwarpSetRotation()`
- `nvwarpSetEulerRotation()`

4.3.29 nvwarpGetSrcFocalLength

```
float nvwarpGetSrcFocalLength(
    const nvwarpHandle han,
    float *fy
);
```

4.3.29.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`fy`

Type: `float *`

The address to which the focal length of the source image in the Y direction is to be written. If only the focal length is the X direction is needed, set `fy` to `NULL`. Because most applications use the same focal length for X and Y, `NULL` is commonly passed for `fy`.

4.3.29.2 Return Value

Returns the focal length of the source image in the X direction.

4.3.29.3 Remarks

This function returns the focal length of the source image in the X direction and writes the focal length in the Y direction to the address specified by the parameter `fy`. It can get focal lengths that have been set by any of the following functions:

- `nvwarpSetSrcFocalLengths()`
- `nvwarpComputeSrcFocalLength()`

4.3.30 nvwarpGetSrcPrincipalPoint

```
void nvwarpGetSrcPrincipalPoint(
    const nvwarpHandle han,
    float xy[2],
    uint32_t relToCenter
);
```

4.3.30.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`xy`

Type: `float`

A two-element array to which the X coordinate and Y coordinate of the principal point of the source image are to be written.

`relToCenter`

Type: `uint32_t`

A `uint32_t` variable that is to contain the position relative to which the principal point is specified:

`0` Relative to pixel (0,0)

`1` Relative to the center of the image

4.3.30.2 Return Value

None.

4.3.30.3 Remarks

This function gets the principal point of the source image.

4.3.31 nvwarpGetSrcRadius

```
float nvwarpGetSrcRadius(
    const nvwarpHandle han
);
```

4.3.31.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

4.3.31.2 Return Value

Returns the circular clipping radius of the source image.

4.3.31.3 Remarks

This function gets the radius to be used for circular clipping of round fisheye source images.

4.3.32 nvwarpGetUserData

```
void* nvwarpGetUserData(
    nvwarpHandle han
);
```

4.3.32.1 Parameters

`han`

Type: `nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

4.3.32.2 Return Value

None.

4.3.32.3 Remarks

This function gets the value of the pointer set by a previous call to `nvwarpSetUserData()` or `nvwarpSetParams()`. The user data pointer is useful in callbacks from asynchronous image warps.

4.3.33 nvwarpGetSrcWidthHeight

```
void nvwarpGetSrcWidthHeight(
    const nvwarpHandle han,
    uint32_t wh[2]
);
```

4.3.33.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`wh`

Type: `uint32_t`

A two-element array to which the width and height of the source image are to be written.

4.3.33.2 Return Value

None.

4.3.33.3 Remarks

This function writes the width and height of the source image to the array specified by the parameter `wh`.

4.3.34 nvwarpGetWarpType

```
nvwarpType_t nvwarpGetWarpType(
    const nvwarpHandle han
);
```

4.3.34.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

4.3.34.2 Return Value

Returns the warp type. For details, see “nvwarpType_t,” on page 25.

4.3.34.3 Remarks

This function gets the warp type.

4.3.35 nvwarpInitParams

```
void nvwarpInitParams(
    nvwarpParams_t *params
);
```

4.3.35.1 Parameters

`params`

Type: `nvwarpParams_t *`

Pointer to the `nvwarpParams_t` structure that is to be initialized. For details of this structure, see "nvwarpParams_t," on page 17.

4.3.35.2 Return Value

None.

4.3.35.3 Remarks

This function initializes the specified `nvwarpParams_t` structure by setting all its members to a default value or to NaN.

In C++, the constructor for the `nvwarpParams_t` structure automatically calls this function. In C, call this function to initialize the structure before setting any of its members.

4.3.36 nvwarpIntoCubemap

```
nvwarpResult nvwarpIntoCubemap(
    nvwarpHandle han,
    cudaStream_t stream,
    cudaTextureObject_t srcTex,
    const nvwarpFaceDescription_t faceDesc[6],
    unsigned faceDim,
    int interpolateEdges,
    void *dstAddr,
    size_t dstRowBytes
);
```

4.3.36.1 Parameters

`han`

Type: `nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

This object should be initialized properly for the source and warp type, typically by using `NVWARP_EQUIRECT_PERSPECTIVE` but `NVWARP_FISHEYE_PERSPECTIVE` can be used for a 360 fisheye source.

`stream`

Type: `cudaStream_t`

The CUDA stream in which the conversion and warping are to be executed. This stream can be allocated with `cudaStreamCreate()` or the default stream 0 can be used.

`srcTex`

Type: `cudaTextureObject_t`

The texture that is used as the source of the image warp, appropriately initialized and buffered. If the source wraps around, clamp the edges.

`faceDesc`

Type: array of `nvwarpFaceDescription_t` structures

A six-element array of `nvwarpFaceDescription_t` structures in which each element specifies the format of one of the six faces in the cubemap. For details of this structure, see "nvwarpFaceDescription_t," on page 16.

The order of the elements in the array is irrelevant. Any permutation of elements yields the same result.

`faceDim`

Type: `unsigned int`

The length, in linear pixels, of one side of each face in the cubemap. The dimensions of each face are calculated from this parameter as (`faceDim` × `faceDim`).

`interpolateEdges`

Type: `int`

Specifies how the edges of the cubemap will be handled:

0 The pixels will be sampled a half pixel away from the edges, a more uniform distribution

1 The cubemap will interpolate the edges. This setting necessarily duplicates the pixels on adjacent faces.

>1 The cubemap will extrapolate beyond the edges by half pixels. This setting can be useful for motion estimation.

`dstAddr`

Type: `void*`

A pointer to pixel (0,0) in the pitched buffer into which the warped destination image is to be placed.

`dstRowBytes`

Type: `size_t`

The byte stride, or pitch, between pixels vertically in the pitched buffer specified in `dstAddr`.

4.3.36.2 Return Value

Returns `NVWARP_SUCCESS` on success.

4.3.36.3 Remarks

This function warps the source image into a cubemap.

The following figures show the results of the warp for different types of source images.

Figure 1 Equirectangular Source Image

Figure 2 H-Cross Format

Figure 3 FBTGLR Format

Figure 4 Pano2VR Format

Figure 5 Continuous Format

4.3.37 nvwarpInverseWarpCoordinates

```
nvwarpResult nvwarpInverseWarpCoordinates(
    const nvwarpHandle han,
    uint32_t numPts,
    const float *inPtsXY,
    float *outPtsXY
);
```

4.3.37.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`numPts`

Type: `uint32_t`

The number of coordinate pairs to be warped.

`inPtsXY`

Type: `const float *`

A pointer to an array of the destination coordinates to be warped.

`outPtsXY`

Type: `float *`

A pointer to an array into which the transformed source coordinates are to be placed. The coordinate warping occurs in-place if `outPtsXY` is equal to `inPtsXY`.

4.3.37.2 Return Value

Returns one of the following result codes:

- `NVWARP_SUCCESS` on success
- `NVWARP_ERR_DOMAIN` if any coordinates are out of domain
- `NVWARP_ERR_UNIMPLEMENTED` if the specified warp type is unsupported

4.3.37.3 Remarks

This function warps coordinates from the destination image into the source image.

To warp coordinates in the opposite direction, namely, from the source image into the destination image, call the `nvwarpWarpCoordinates()` function.

4.3.38 nvwarpMultiWarp360

```
nvwarpResult nvwarpMultiWarp360(
    cudaStream_t stream,
    const nvwarpYUVRGBParams_t *yuvParams,
    const void *yuvBuffer,
    size_t yuvRowBytes,
    void *rgbBuffer,
    size_t rgbRowBytes,
    cudaTextureObject_t rgbTex,
    uint32_t numWarps,
    const nvwarpParams_t *paramArray,
    void **dstBuffers,
    const size_t *dstRowBytes
);
```

4.3.38.1 Parameters

`stream`

Type: `cudaStream_t`

The CUDA stream in which the conversion and warping are to be executed. This stream can be allocated with `cudaStreamCreate()` or the default stream 0 can be used.

`yuvParams`

Type: `const nvwarpYUVRGBParams_t *`

If the color space of the source image is YUV, set this parameter to point to the `nvwarpYUVRGBParams_t` structure that contains values to be used for converting the YUV color space specified in the parameter `yuvBuffer`.

If the color space of the source image is RGBA or BGRA, set `yuvParams` to `NULL`.

For details of the `nvwarpYUVRGBParams_t` structure, see "nvwarpYUVRGBParams_t," on page 21.

`yuvBuffer`

Type: `const void *`

Pointer to a pitched CUDA buffer containing an input YUV color space that is to be converted.

If the color space of the source image is RGBA or BGRA, set `yuvBuffer` to `NULL`.

`yuvRowBytes`

Type: `size_t`

The byte stride, or pitch, between pixels vertically in the pitched buffer specified in `yuvBuffer`.

`rgbBuffer`

Type: `void *`

Pointer to a pitched CUDA buffer that is to contain an RGBA or BGRA image. This buffer is fed into each warp.

If the color space of the source image is YUV, this buffer is an intermediate buffer that is to receive an RGBA image that has been converted from YUV. The parameter `yuvBuffer` must point to an input YUV color space and the parameter `yuvParams` must point to values to be used for converting the YUV color space.

If the color space of the source image is RGBA or BGRA, preload this buffer with a source RGB image.

`rgbRowBytes`

Type: `size_t`

The byte stride, or pitch, between pixels vertically in the pitched buffer specified in `rgbBuffer`.

`rgbTex`

Type: `cudaTextureObject_t`

A texture that has been allocated and initialized appropriately for the pitched CUDA buffer specified by `rgbBuffer`.

`numWarps`

Type: `uint32_t`

The number of warps to be performed on the image.

`paramArray`

Type: `const nvwarpParams_t *`

Pointer to an array of `nvwarpParams_t` structures that contain the parameters to be used for each warp. For details of this structure, see "nvwarpParams_t," on page 17.

`dstBuffers`

Type: `void **`

Pointer to an array of CUDA pitched buffers to contain the results of each warp.

`dstRowBytes`

Type: `const size_t *`

Pointer to the `rowBytes` of each buffer specified in `dstBuffers`. The `rowBytes` specifies byte stride, or pitch, between pixels vertically in the pitched buffer. The parameters, buffers and `rowBytes` arrays are in correspondence.

4.3.38.2 Return Value

Returns `NVWARP_SUCCESS` on success.

4.3.38.3 Remarks

This function performs several warps on the same source image. The warps are represented by the following arrays:

- An array of warp parameters specified by the parameter `paramArray`
- An array of destination buffers specified by the parameter `dstBuffers`
- An array of `rowBytes` (Y-strides, pitch) for each buffer specified by the parameter `dstRowBytes`

The source code for this function is distributed in the SDK.

4.3.39 nvwarpSetBlock

```
void nvwarpSetBlock(
    nvwarpHandle han,
    dim3 dim_block
);
```

4.3.39.1 Parameters

`han`

Type: `nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`dim_block`

Type: `dim3`

The CUDA block.

4.3.39.2 Return Value

None.

4.3.39.3 Remarks

This function sets the CUDA block, which determines the distribution of work.

4.3.40 nvwarpSetControl

```
void nvwarpSetControl(
    nvwarpHandle han,
    uint32_t index,
    float control
);
```

4.3.40.1 Parameters

`han`

Type: `nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`index`

Type: `uint32_t`

The index of the control to be set. Only one control (`with index=0`) is used, but not all warps use this control.

`control`

Type: `float`

The value to be set in the control with the specified index.

4.3.40.2 Return Value

None.

4.3.40.3 Remarks

This function sets a control value for a warp.

4.3.41 nvwarpSetDistortion

```
void nvwarpSetDistortion(
    nvwarpHandle han,
    const float d[5]
);
```

4.3.41.1 Parameters

`han`

Type: `nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`d`

Type: `const float`

A five-element array containing the distortion coefficients in the distortion polynomial for a fisheye source or a perspective source.

- For fisheye distortion, the distortion polynomial from which the radial distortion is calculated uses four radial coefficients:

$$r' = r(1 + k_0 r^2 + k_1 r^4 + k_2 r^6 + k_3 r^8)$$

- For perspective distortion, the distortion polynomial from which the radial distortion is calculated uses three radial coefficients:

$$r' = r(1 + k_0 r^2 + k_1 r^4 + k_2 r^6)$$

The remaining coefficients are reserved for tangential distortion correction and should be set to zero.

In these calculations, the radius has been normalized by the focal length. The higher-order coefficients take effect closer to the periphery.

If the `d` parameter is `NULL`, all distortion coefficients are set to zero.

4.3.41.2 Return Value

None.

4.3.41.3 Remarks

This function sets the coefficients in the distortion polynomial for a fisheye source or a perspective source.

4.3.42 nvwarpSetDstFocalLengths

```
void nvwarpSetDstFocalLengths(
    nvwarpHandle han,
    float fl,
    float fy
);
```

4.3.42.1 Parameters

`han`

Type: `nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`fl`

Type: `float`

The focal length in the X direction in pixels.

`fy`

Type: `float`

The focal length in the Y direction in pixels. If `fy` is 0, the function uses the same focal length as the focal length in the X direction.

4.3.42.2 Return Value

None.

4.3.42.3 Remarks

This function explicitly sets the focal lengths of the destination image, which specify the angular pixel density of the image. To support anamorphic cinema and other systems that use anisotropic sampling, set separate focal lengths for the X direction and the Y direction. Otherwise, set the same focal length in each direction. In anisotropic sampling, the pixels are rectangular or otherwise not square in shape.

To preserve the pixel density of the source image in the destination image, set the destination focal lengths equal to the source focal lengths. Otherwise, you can zoom in or zoom out by setting different focal lengths for the source image and the target image:

- To zoom in, set the destination focal lengths to longer than the source focal lengths.
- To zoom out, set the destination focal lengths to shorter than the source focal lengths.

However, as the ratio of the destination to source focal lengths decreases, aliasing becomes more apparent. To avoid aliasing, keep this ratio greater than 0.3. Aliasing also depends on the source detail.

4.3.43 nvwarpSetDstPrincipalPoint

```
void nvwarpSetDstPrincipalPoint(
    nvwarpHandle han,
    const float xy[2],
    uint32_t relToCenter
);
```

4.3.43.1 Parameters

`han`

Type: `nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`xy`

Type: `const float`

A two-element array that contains the X coordinate and Y coordinate in pixels of the principal point relative to the position specified by the parameter `relToCenter`.

`relToCenter`

Type: `uint32_t`

The position relative to which the principal point is specified:

`0` Relative to pixel (0,0)

`1` Relative to the center of the image

If `relToCenter` is nonzero, you must call `nvwarpSetDstWidthHeight()` to set the dimensions of the source image *before* calling this function.

If the principal point is specified relative to the center of the image, you do not need to set the pixel phase.

4.3.43.2 Return Value

None.

4.3.43.3 Remarks

This function sets the principal point, which is the center of projection, of the destination image.

This function allows you to generate a view that is horizontally asymmetrical. If you set the destination principal point through the parameter block API, you can generate only horizontally symmetrical views.

If the destination view is specified using the top and bottom angles, the destination principal point is computed and does not need to be set explicitly.

4.3.44 nvwarpSetDstWidthHeight

```
void nvwarpSetDstWidthHeight(
    nvwarpHandle han,
    uint32_t w,
    uint32_t h
);
```

4.3.44.1 Parameters

`han`

Type: `nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`w`

Type: `uint32_t`

The width of the destination image in pixels.

`h`

Type: `uint32_t`

The height of the destination image in pixels.

4.3.44.2 Return Value

None.

4.3.44.3 Remarks

This function sets the width and height of the destination image.

4.3.45 nvwarpSetEulerRotation

```
void nvwarpSetEulerRotation(
    nvwarpHandle han,
    const float *angles,
    const char *axes
);
```

4.3.45.1 Parameters

`han`

Type: `nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`angles`

Type: `const float *`

Pointer to an array containing a series of view rotation angles (Euler angles) in radians. For traditional Euler angles this array contains three elements, but it can be any size greater than 0.

`axes`

Type: `const char *`

A 0-terminated C-string that lists the axes of rotation. The length of the `axes` string determines how many rotations are performed and does not need to be three characters. However, strings longer than three characters are redundant and shorter strings suffice for many applications.

The case of each character specifies whether rotation is about the positive or negative axes:

- Upper case `X`, `Y`, and `Z` specify rotation about the positive X, Y and Z axes.
- Lower case `x`, `y`, and `z` specify rotation about the negative X, Y and Z axes.

The direction of positive rotation for each axis is as follows:

- **X axis:** upwards about the rightward-pointing X axis
- **Y axis:** rightwards about the downward-pointing Y axis
- **Z axis:** clockwise about the outward-pointing Z axis

These rotations and axes specify the rotation of a *camera* to view an image that is considered to be straight ahead, for example, to pan around a panorama. To rotate the *image* and keep the camera upright, which embeds the image in 3D, specify the angles and axes in reverse order.

The same axis may appear more than once, for example, `ZXZ`. This specification is for a coordinate system where the Y axis points downwards and the Z axis points outwards. For a coordinate system where the Y axis points upwards and Z axis points inwards, invert the case of all axis and Z axis specifications.

4.3.45.2 Return Value

None.

4.3.45.3 Remarks

This function sets the rotation as a sequence of generalized Euler angles.

4.3.46 nvwarpSetParams

```
nvwarpResult nvwarpSetParams(
    nvwarpHandle han,
    const nvwarpParams_t *params
);
```

4.3.46.1 Parameters

`han`

Type: `nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`params`

Type: `const nvwarpParams_t *`

Pointer to the `nvwarpParams_t` structure that contains values to be used in the warp. For details of this structure, see "nvwarpParams_t," on page 17.

4.3.46.2 Return Value

Returns `NVWARP_SUCCESS` on success.

4.3.46.3 Remarks

This function sets warp parameters to the values set in the specified `nvwarpParams_t` structure.

4.3.47 nvwarpSetPixelPhase

```
void nvwarpSetPixelPhase(
    nvwarpHandle han,
    uint32_t phase
);
```

4.3.47.1 Parameters

`han`

Type: `nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`phase`

Type: `uint32_t`

The pixel phase, which determines how pixels are considered to be sampled:

`0` Sampled on the integers.

The X and Y coordinates of the pixels are as follows:

X: `{ 0, 1, …, width-1 }`
Y: `{ 0, 1, …, height-1 }`

`1` Sampled on the integers-plus one-half.

The X and Y coordinates of the pixels are as follows:

X: `{ 0.5, 1.5, …, width-0.5 }`
Y: `{ 0.5, 1.5, …, height-0.5 }`

The default is `0`.

4.3.47.2 Return Value

None.

4.3.47.3 Remarks

This function sets the pixel phase, which determines whether pixels are considered to be sampled on the integers or the integers plus-one-half. The phase must be set correctly for coordinate warps to render graphical overlays that match features exactly. It is also used to interpret `srcX0` and `srcY0` when the source center of projection is set.

4.3.48 nvwarpSetRotation

```
void nvwarpSetRotation(
    nvwarpHandle han,
    const float R[9]
);
```

4.3.48.1 Parameters

`han`

Type: `nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`R`

Type: `const float *`

Pointer to a nine-element array containing the 3×3 rotation matrix. This matrix is necessarily an orthogonal matrix that can be interpreted as a list of the remappings of the X, Y and Z axes to specify a view.

4.3.48.2 Return Value

None.

4.3.48.3 Remarks

This function sets the rotation as a rotation matrix to specify the destination *view*. If you want to specify the *embedding* of an image in 3D, pass the transposed matrix as the `R` parameter to `nvwarpSetRotation()` instead. The matrix is specified in a right-handed screen space, with X to the right, Y downwards and Z outwards. To convert from a 3D modeling convention instead, with Y upward and Z inward, call `nvwarpConvertTransformBetweenYUpandYDown()` before calling this function. The source is centered in the view with the identity (or unit) matrix.

4.3.49 nvwarpSetSrcFocalLengths

```
void nvwarpSetSrcFocalLengths(
    nvwarpHandle han,
    float fl,
    float fy
);
```

4.3.49.1 Parameters

`han`

Type: `nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`fl`

Type: `float`

The focal length in the X direction in pixels.

`fy`

Type: `float`

The focal length in the Y direction in pixels. If `fy` is 0, the function uses the same focal length as the focal length in the X direction.

4.3.49.2 Return Value

None.

4.3.49.3 Remarks

This function explicitly sets the focal lengths of the source image, which specify the angular pixel density of the image. To support anisotropic sampling, set separate focal lengths for the X direction and the Y direction. In anisotropic sampling, the pixels are rectangular or otherwise not square in shape.

The focal length (in pixels or pixels/radian) can be acquired from the image EXIF data, by multiplying the focal length in millimeters by the pixel density in pixels per millimeter. The relevant EXIF tags are as follows:

- `FocalLength` (37386)
- `FocalPlaneXResolution` (41486)
- `FocalPlaneYResolution` (41487)
- `FocalPlaneResolutionUnit` (41488)

A camera calibration package, such as the package in OpenCV, can be also used to recover the focal length in addition to the principal point and distortion coefficients.

4.3.50 nvwarpSetSrcPrincipalPoint

```
void nvwarpSetSrcPrincipalPoint(
    nvwarpHandle han,
    const float xy[2],
    uint32_t relToCenter
);
```

4.3.50.1 Parameters

`han`

Type: `nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`xy`

Type: `const float`

A two-element array that contains the X coordinate and Y coordinate of the principal point relative to the position specified by the parameter `relToCenter` in pixels.

`relToCenter`

Type: `uint32_t`

The position relative to which the principal point is specified:

`0` Relative to pixel (0,0)

`1` Relative to the center of the image

If `relToCenter` is nonzero, you must call `nvwarpSetSrcWidthHeight()` to set the dimensions of the source image before calling this function.

If the principal point is specified relative to the center of the image, you do not need to set the pixel phase.

4.3.50.2 Return Value

None.

4.3.50.3 Remarks

This function sets the principal point, which is the center of projection, of the source image.

4.3.51 nvwarpSetSrcRadius

```
void nvwarpSetSrcRadius(
    nvwarpHandle han,
    float r
);
```

4.3.51.1 Parameters

`han`

Type: `nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`r`

Type: `float`

The fisheye circular clipping radius in pixels. 0 implies no clipping.

4.3.51.2 Return Value

None.

4.3.51.3 Remarks

This function sets the radius to be used for circular clipping of round fisheye source images. In this implementation, no circular clipping is performed.

4.3.52 nvwarpSetUserData

```
void nvwarpSetUserData(
    nvwarpHandle han,
    void *userData
);
```

4.3.52.1 Parameters

`han`

Type: `nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`userData`

Type: `void *`

The pointer to be stored with the warp instance. This pointer does not need to be a valid pointer because it is returned only by `nvwarpGetUserData()` and is not otherwise used.

4.3.52.2 Return Value

None.

4.3.52.3 Remarks

This function stores a pointer in the warp instance. The user data pointer is useful in callbacks from asynchronous image warps or for any other purpose that may arise.

4.3.53 nvwarpSetSrcWidthHeight

```
void nvwarpSetSrcWidthHeight(
    nvwarpHandle han,
    uint32_t w,
    uint32_t h
);
```

4.3.53.1 Parameters

`han`

Type: `nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`w`

Type: `uint32_t`

The width of the source image in pixels.

`h`

Type: `uint32_t`

The height of the source image in pixels.

4.3.53.2 Return Value

None.

4.3.53.3 Remarks

This function sets the width and height of the source image.

4.3.54 nvwarpSetWarpType

```
nvwarpResult nvwarpSetWarpType(
    nvwarpHandle han,
    nvwarpType_t type
);
```

4.3.54.1 Parameters

`han`

Type: `nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`type`

Type: `nvwarpType_t`

The type of the warp. For details, see “nvwarpType_t,” on page 25.

4.3.54.2 Return Value

Returns `NVWARP_SUCCESS` on success.

4.3.54.3 Remarks

This function sets the warp type.

4.3.55 nvwarpSrcFromRay

```
nvwarpResult nvwarpSrcFromRay(
    const nvwarpHandle han,
    uint32_t numRays,
    const float *rays3D,
    float *pts2D
);
```

4.3.55.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`numRays`

Type: `uint32_t`

The number of rays to be converted.

`rays3D`

Type: `const float *`

A pointer to the array of input 3D rays to be converted. The input rays do not need to be normalized.

`pts2D`

Type: `float *`

A pointer to an array where the converted output 2D pixel coordinates are to be placed.

4.3.55.2 Return Value

Returns one of the following result codes:

- `NVWARP_SUCCESS` on success
- `NVWARP_ERR_DOMAIN` if any conversions are out of domain
- `NVWARP_ERR_UNIMPLEMENTED` if the specified warp type is unsupported

4.3.55.3 Remarks

This function converts 3D rays from a source image to 2D pixel coordinates.

4.3.56 nvwarpSrcToRay

```
nvwarpResult nvwarpSrcToRay(
    const nvwarpHandle han,
    uint32_t numPts,
    const float *pts2D,
    float *rays3D
);
```

4.3.56.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`numPts`

Type: `uint32_t`

The number of coordinate pairs to be converted to rays.

`pts2D`

Type: `const float *`

A pointer to the array of input 2D pixel coordinates to be converted.

`rays3D`

Type: `float *`

A pointer to an array where the converted output 3D rays are to be placed.

The output rays are guaranteed to be normalized, unless they are out-of-domain, in which case they are NaN.

4.3.56.2 Return Value

Returns one of the following result codes:

- `NVWARP_SUCCESS` on success
- `NVWARP_ERR_DOMAIN` if any conversions are out of domain
- `NVWARP_ERR_UNIMPLEMENTED` if the specified warp type is unsupported

4.3.56.3 Remarks

This function converts 2D pixel coordinates from a source image to 3D rays.

4.3.57 nvwarpVersion

```
uint32_t nvwarpVersion();
```

4.3.57.1 Parameters

None.

4.3.57.2 Return Value

Returns an unsigned 32-bit (4-byte) integer in which the three most significant bytes represent the version number of the NVWarp360 API in the form *major.minor.revison*.

Byte	Contents
Most significant	Major version number. Typically, the major version number is incremented when API changes, other major changes, or backwards incompatibilities occur.
Second most significant	Minor version number. The minor version is incremented when minor functionality changes such as new projections are added
Third most significant	Revision number. The revision is incremented when a bug fix has been released.
Least significant	Always 0. Reserved for internal use.

For example, version 2.0.0 is represented as 33554432, which is equivalent to 0x02000000.

4.3.57.3 Remarks

This function gets the version number of the NVWarp360 API.

4.3.58 nvwarpWarpBuffer

```
nvwarpResult nvwarpWarpBuffer(
    const nvwarpHandle han,
    cudaStream_t stream,
    cudaTextureObject_t srcTex,
    void *dstAddr,
    size_t dstRowBytes
);
```

4.3.58.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`stream`

Type: `cudaStream_t`

The CUDA stream that will perform the warping. The stream can be allocated with `cudaStreamCreate()` or the default stream 0 can be used.

`srcTex`

Type: `cudaTextureObject_t`

The texture that is used as the source of the image warp.

`dstAddr`

Type: `void *`

A pointer to the pitched buffer into which the warped destination image is to be placed.

`dstRowBytes`

Type: `size_t`

The byte stride, or pitch, between pixels vertically in the pitched buffer specified in `dstAddr`.

4.3.58.2 Return Value

Returns `NVWARP_SUCCESS` on success.

4.3.58.3 Remarks

This function warps the specified source image texture and places it in the specified pitched buffer.

4.3.59 nvwarpWarpCoordinates

```
nvwarpResult nvwarpWarpCoordinates(
    const nvwarpHandle han,
    uint32_t numPts,
    const float *inPtsXY,
    float *outPtsXY
);
```

4.3.59.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`numPts`

Type: `uint32_t`

The number of coordinate pairs to be warped.

`inPtsXY`

Type: `const float *`

A pointer to an array of the source coordinates to be warped.

`outPtsXY`

Type: `float *`

A pointer to an array into which the destination warped coordinates are to be placed. The coordinate warping occurs in-place if `outPtsXY` is equal to `inPtsXY`.

4.3.59.2 Return Value

Returns one of the following result codes:

- `NVWARP_SUCCESS` on success
- `NVWARP_ERR_DOMAIN` if any coordinates are out of domain
- `NVWARP_ERR_UNIMPLEMENTED` if the specified warp type is unsupported

4.3.59.3 Remarks

This function warps coordinates from the source image into the destination image.

To warp coordinates in the opposite direction, namely, from the destination image into the source image, call the `nvwarpInverseWarpCoordinates()` function.

4.3.60 nvwarpWarpSurface

```
nvwarpResult nvwarpWarpSurface(
    const nvwarpHandle han,
    cudaStream_t stream,
    cudaTextureObject_t srcTex,
    cudaSurfaceObject_t dstSurface
);
```

4.3.60.1 Parameters

`han`

Type: `const nvwarpHandle`

The handle for the opaque `nvWarp` object that was created by an earlier call to `nvwarpCreateInstance()`.

`stream`

Type: `cudaStream_t`

The CUDA stream that will perform the warping. The stream can be allocated with `cudaStreamCreate()` or the default stream 0 can be used.

`srcTex`

Type: `cudaTextureObject_t`

The source image texture, which is used as input to the warp.

`dstSurface`

Type: `cudaSurfaceObject_t`

The surface in which the warped image is to be placed.

4.3.60.2 Return Value

Returns `NVWARP_SUCCESS` on success.

4.3.60.3 Remarks

This function warps the specified image texture and places it in the specified surface.

4.4 RESULT CODES

Set and get accessor functions that merely access values in the opaque `nvwarpObject` object do not return a result code.

Functions that compute images or values return an `nvwarpResult` result code that indicates the result of the computation or validation of parameters.

`nvwarpResult` Result Code	**Meaning**
`NVWARP_SUCCESS`	The operation was successful.
`NVWARP_ERR_GENERAL`	An unspecified error has occurred.
`NVWARP_ERR_UNIMPLEMENTED`	The requested feature has not yet been implemented.
`NVWARP_ERR_DOMAIN`	The warped coordinates are outside of the domain. This error occurs when the square root of a negative number in the warp equation is taken. This error does not indicate whether the result is within the image bounds.
`NVWARP_ERR_MISSING_PARAMETERS`	Some required parameters have not been specified.
`NVWARP_ERR_PARAMETER`	A parameter has an invalid value.
`NVWARP_ERR_INITIALIZATION`	Initialization has not completed successfully.
`NVWARP_ERR_CUDA_MEMORY`	There is not enough CUDA memory for the operation specified.
`NVWARP_ERR_CUDA_LAUNCH`	CUDA could not launch the specified kernel.
`NVWARP_ERR_CUDA_NO_KERNEL`	No suitable CUDA kernel image has been found for this GPU, which means that this GPU's compute model is not supported.
`NVWARP_ERR_CUDA_DRIVER`	The CUDA driver version is too low for CUDA runtime version.
`NVWARP_ERR_CUDA`	An unspecified error has been reported by the CUDA runtime.
`NVWARP_ERR_CUDA_APPROXIMATE`	An accurate calculation is not available, so an approximation returned. This result can occur only when *coordinates* (not *images*) from *perspective* sources are being warped with nonzero tangential distortion (nonzero `dist[3]` and `dist[4]` parameters).

Common CUDA errors are translated into a specific `nvwarpResult` result code. Less common errors are translated into the `NVWARP_ERR_CUDA` result code. To retrieve the specific CUDA error, call `cudaGetLastError()`.

To convert an `nvwarpResult` result code into a human-readable string, call the `nvwarpErrorStringFromCode()` function.

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.

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.

Third-Party Notice

Portions provided under the following license terms:

PaniniGeneral.c 15Jan2010 TKS

Copyright © 2010, Thomas K Sharpless

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the <organization> nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL <COPYRIGHT HOLDER> BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This is the reference implementation of the General Pannini Projection, an elaboration of the basic Pannini projection discovered by Bruno Postle and Thomas Sharpless in December 2008 in paintings by Gian Paolo Pannini (1691-1765).

Copyright