NVIDIA AR SDK

Programming Guide

PG-09605-001_v0.5 | April 2020

Document History

PG-09605-001_v0.5

Version	Date	Description of Change
0.5	April 2, 2020	Beta Release

Table of Contents

List of Tables

Chapter 1. Introduction to NVIDIA AR SDK

NVIDIA® AR SDK enables real-time modeling and tracking of human faces from video. The SDK is powered by NVIDIA graphics processing units (GPUs) with Tensor Cores, and as a result, the algorithm throughput is greatly accelerated, and latency is reduced.

NVIDIA AR SDK has the following features:

- **Face detection and tracking** detects, localizes, and tracks human faces in images or videos by using bounding boxes.
- **Facial landmark detection and tracking** predicts and tracks the pixel locations of human facial landmark points and head poses in images or videos.
  The detected facial landmarks follow the *Multi-PIE 68 point mark-ups* information in Facial point annotations.
- **Face 3D mesh and tracking** reconstructs and tracks a 3D human face and its head pose from the provided facial landmarks. NVIDIA AR SDK provides a sample application that demonstrates the features listed above in real time by using a webcam or offline videos.

Chapter 2. Getting Started with NVIDIA AR SDK

2.1 Hardware and Software Requirements

NVIDIA AR SDK requires specific NVIDIA GPUs, a specific version of the Windows OS, and other associated software on which the SDK depends.

2.1.1 Hardware Requirements

NVIDIA AR SDK is compatible with GPUs that are based on the NVIDIA® Turing™ architecture. Although the SDK can run on Turing™ GPUs without Tensor Cores, it is optimized for much higher performance on GPUs with Tensor Cores.

2.1.2 Software Requirements

NVIDIA AR SDK requires a specific version of the Windows OS and other associated software on which the SDK depends. The NVIDIA CUDA®, cuDNN and TensorRT™ dependencies are bundled with the SDK Installer. See "Installing NVIDIA AR SDK and Associated Software" on page 3.

Software	**Required Version**
Windows OS	64-bit Windows 10
Microsoft Visual Studio	2015 (MSVC14.0) or later
CMake	3.12 or later
NVIDIA Graphics Driver for Windows	410.48 or later
NVIDIA CUDA Toolkit	10.1 or later
NVIDIA CUDA Deep Neural Network (cuDNN) Library	7.5.1 or later, based on the cuDNN Support Matrix
NVIDIA TensorRT	6.0.1 or later

2.2 Installing NVIDIA AR SDK and Associated Software

To develop applications with the NVIDIA AR SDK, you must install the associated software on which the SDK depends and provide the path to this software during compilation and linking.

The SDK is distributed in the following parts:

- An open source repository that includes the SDK API headers, the sample applications and their dependency libraries, and a proxy file to enable compilation without the SDK DLLs.
- An Installer that installs the SDK DLLs, the models, and the SDK dependency libraries.

> **NOTE**: The source code and sample application are hosted on GitHub at https://github.com/nvidia/BROADCAST-AR-SDK.

To use the SDK, download the source code from GitHub and install the SDK binaries. Your application needs to integrate the API headers and call the SDK APIs, and the sample app source code demonstrates how to complete these tasks.

The sample app also includes a file named `nvARProxy.cpp` that is used for linking against the SDK DLL without the need for an import library (.lib) file, allowing compilation of the open source code independently of the SDK Installer. However, the SDK Installer is still required in order to load the runtime dependencies, DLLs and models.

The SDK binaries are installed in the `C:\Program Files\NVIDIA Corporation\NVIDIA AR SDK\` directory.

> **IMPORTANT**: The installer **does not** include the face model that is required for the Face 3D Reconstruction feature. The face model must be generated and copied to the `%Program Files%\NVIDIA Corporation\NVIDIA AR SDK\models` directory, so that the feature can work with the SDK installation. See Configuring NVIDIA AR SDK with a 3D Morphable Face Model on page 4 for more information.

If your application uses the Face 3D Mesh Tracking feature, you must build and include the 3DMM model with your application package, instead of relying on the SDK installer, as it does not include this model. You may also choose to either copy the 3DMM model to the installation directory or bundle all binaries and models, including the 3DMM model, with your application package. See "Environment Variables" on page 7 for more information.

2.3 Configuring NVIDIA AR SDK with a 3D Morphable Face Model

The Face 3D Mesh Tracking feature requires a 3D Morphable Face Model (3DMM). NVIDIA AR SDK does not include a 3DMM. Therefore, if you are using the Face 3D Mesh Tracking feature, you must configure NVIDIA AR SDK with 3DMM.

> **Note**: To configure NVIDIA AR SDK with 3DMM, you can use the Surrey Face Model or your own model.

NVIDIA AR SDK provides the `ConvertSurreyFaceModel.exe` utility to convert 3DMM files to the NVIDIA `.nvf` format that is required by the SDK.

2.3.1 Using the Surrey Face Model

1. Download the following Surrey Face Model files from the eos project page on GitHub:
   - `sfm_shape_3448.bin`
   - `expression_blendshapes_3448.bin`
   - `sfm_3448_edge_topology.json`
   - `sfm_model_contours.json`
   - ibug_to_sfm.txt
2. Convert the downloaded files to the NVIDIA `.nvf` format.

```
tools/ConvertSurreyFaceModel.exe
--shape=path/sfm_shape_3448.bin
--blend_shape=path/expression_blendshapes_3448.bin
--topology=path/sfm_3448_edge_topology.json
--contours=path/sfm_model_contours.json
--ibug=path/ibug_to_sfm.txt
--out=output-path/face_model0.nvf
```

> **Note**: The `ConvertSurreyFaceModel.exe` file is distributed in the : https://github.com/nvidia/BROADCAST-AR-SDK GitHub repo.

path

The full or relative path to the folder that contains the Surrey Face Model files that you downloaded.

output-path

The full or relative path to the folder where the output `.nvf` format file should be written.

The sample application provided with NVIDIA AR SDK requires that the model file be named `face_model0.nvf`.

3. Place the `face_model0.nvf` file in the `%Program Files%\NVIDIA Corporation\NVIDIA AR SDK\models` folder, created by the SDK Installer.

2.3.2 Using your own 3DMM

1. Write a model natively in the format that is defined in "NVIDIA 3DMM File Format" on page 93.

   The sample application that is provided with the NVIDIA AR SDK requires that the model file be named `face_model0.nvf`.
2. Place the `face_model0.nvf` file in the `%Program Files%\NVIDIA Corporation\NVIDIA AR SDK\models` folder, created by the SDK Installer .

2.4 NVIDIA AR SDK Sample Application

FaceTrack is a sample Windows application that demonstrates the face tracking, landmark tracking, and 3D mesh tracking features of the NVIDIA AR SDK. The application requires a video feed from the camera that is connected to the computer on which the application is running.

2.4.1 Building the Sample Application

The open source repository includes the source code to build the sample application, and a proxy file `nvARProxy.cpp` to enable compilation without explicitly linking against the SDK DLL.

> **Note**: To download the models and runtime dependencies required by the features, you need to run the SDK Installer.

1. In the root folder of the downloaded source code, start the CMake GUI and specify the source folder and a build folder for the binary files.
   a. For the source folder, ensure that the path ends in `OSS`.
   b. For the build folder, ensure that the path ends in `OSS/build`.
2. Use CMake to configure and generate the Visual Studio solution file.
   a. Click **Configure**.
   b. When prompted to confirm that CMake can create the build folder, click **OK**.
   c. Select **Visual Studio** for the generator and **x64** for the platform.
   d. To complete configuring the Visual Studio solution file, click **Finish**.
   e. To generate the Visual Studio Solution file, click **Generate**.

f. Verify that the build folder contains the `NvAR_SDK.sln` file.

3. Use Visual Studio to generate the `FaceTrack.exe` file from the `NvAR_SDK.sln` file.
   a. In CMake, to open Visual Studio, click **Open Project**.
   b. In Visual Studio, select **Build** > **Build Solution**.

2.4.2 Running the Sample Application

A prebuilt `FaceTrack.exe` file is supplied with the NVIDIA AR SDK.

Before running the application, connect a camera to the computer on which you plan to run the sample application.

> **Note**: The application uses the video feed from this camera.

1. Open a Command Prompt window.
2. From the `samples\FaceTrack` folder, under the root folder of the NVIDIA AR SDK, execute the `run.bat` file.

   This command launches an OpenCV window with the camera feed and draws a 3D face mesh over the largest detected face.

2.4.3 Command-Line Arguments

`--model_path=`*path*

Specifies the path to the models.

`--temporal[=(true|false)]`

Optimizes the results for temporal input frames. If the input is a video, set this value to `true`.

`--offline_mode[=(true|false)]`

Specifies whether to use offline video or an online camera video as the input.

- `true`: Use offline video as the input.
- `false`: Use an online camera as the input.

`--capture_outputs[=(true|false)]`

If `--offline_mode=false`, specifies whether to enable the following features:

- Toggling video capture on and off by pressing the **C** key.
- Saving an image frame by pressing the **S** key.

Additionally, a result file that contains the detected landmarks and /or face boxes is written at the time of capture.
If `--offline_mode=true`, this argument is ignored.

`--cam_res=[`*width*`x]` *height*

If `--offline_mode=false`, specifies the camera resolution. *width* is optional. If omitted, *width* is computed from *height* to give an aspect ratio of 4:3. For example:

`--cam_res=640x480` or `--cam_res=480`.

If `--offline_mode=true`, this argument is ignored.

`--in_file=`*file*
`--in=`*file*

If `--offline_mode=true`, specifies the input video file.

If `--offline_mode=false`, this argument is ignored.

`--out_file=`*file*
`--out=`*file*

If `--offline_mode=true`, specifies the output video file.

If `--offline_mode=false`, this argument is ignored.

2.4.4 Environment Variables

Here are the environmental variables:

- **`NVAR_MODEL_DIR`**

  If the application has not provided the path to the models directory by setting the `NvAR_Parameter_Config_ModelDir` string, the SDK tries to load the models from the path in the `NVAR_MODEL_DIR` environment variable. The SDK installer sets `NVAR_MODEL_DIR` to `%ProgramFiles%\NVIDIA Corporation\NVIDIA AR SDK\models`.

- **`NV_AR_SDK_PATH`**

  By default, applications that use the SDK will try to load the SDK DLL and its dependencies from the SDK install directory, for example, `%ProgramFiles%\NVIDIA Corporation\NVIDIA AR SDK\models`. Applications might also want to include and load the SDK DLL and its dependencies directly from the application folder.

  To prevent the files from being loaded from the install directory, the application can set this environment variable to `USE_APP_PATH`. If `NV_AR_SDK_PATH` is set to `USE_APP_PATH`, instead of loading the binaries from the Program Files install directory, the SDK follows the standard OS search order to load the binaries, for example, the app folder followed by the PATH environment variable.

IMPORTANT: Set this variable **only** for the application process. Setting this variable as a user or a system variable affects other applications that use the SDK.

2.4.5 Keyboard Controls

The sample application provides the following keyboard controls to change the runtime behavior of the application:

- 1 selects the face-tracking-only mode and shows only the bounding boxes.
- 2 selects the face and landmark tracking mode and shows only landmarks.
- 3 selects face, landmark, and 3D mesh tracking mode and shows only 3D face meshes.
- W toggles the selected visualization mode on and off.
- F toggles the frame rate display.
- C toggles video saving on and off.
  - When video saving is toggled off, a file is saved with the captured video with a result file that contains the detected face box and/or landmarks.
  - This control is enabled only if `--offline_mode=false` and `--capture_outputs=true`.
- S saves an image and a result file.

  This control is enabled only if `--offline_mode=false` and `--capture_outputs=true`.

Chapter 3. Using NVIDIA AR SDK in Applications

Use the NVIDIA AR SDK to enable an application to use the face tracking, facial landmark tracking, and 3D face mesh tracking features of the SDK.

3.1 Creating an Instance of a Feature Type

Creating an instance of a feature type provides access to configuration parameters that are used when loading an instance of the feature type and the input and output parameters that are provided at runtime when instances of the feature type are run.

1. Allocate memory for an `NvAR_FeatureHandle` structure.

   ```
   NvAR_FeatureHandle faceDetectHandle{};
   ```

2. Call the `NvAR_Create()` function.

   In the call to the function, pass the following information:

   - A value of the `NvAR_FeatureID` enumeration to identify the feature type.
   - A pointer to the variable that you declared to allocate memory for an `NvAR_FeatureHandle` structure.

   This example creates an instance of the face detection feature type:

   ```
   NvAR_Create(NvAR_Feature_FaceDetection, &faceDetectHandle)
   ```

   This function creates a handle to the feature instance, which is required in function calls to get and set the properties of the instance and to load, run, or destroy the instance.

3.2 Getting and Setting Properties for a Feature Type

To prepare to load and run an instance of a feature type, set the properties that the instance requires, such as:

- The configuration properties that are required to load the feature type.
- Input and output properties to be provided at runtime when instances of the feature type are run.

See "Key Values in the Properties of a Feature Type" on page 12 for a complete list of properties.

To set properties, NVIDIA AR SDK provides type-safe set accessor functions. If you need the value of a property that has been set by a set accessor function, use the corresponding get accessor function. See "Summary of NVIDIA AR SDK Accessor Functions" on page 11 for a complete list of get and set functions.

3.2.1 Setting Up the CUDA Stream

Some SDK features require a CUDA stream in which to run. See the [NVIDIA CUDA Toolkit Documentation]() for more information.

1. Initialize a CUDA stream by calling one of the following functions:
   - `NvAR_CudaStreamCreate()`
   - The CUDA Runtime API function `cudaStreamCreate()`

     You can use this function to avoid linking with the NVIDIA CUDA Toolkit libraries.
2. Call the `NvAR_SetCudaStream()` function and provide the following information as parameters:
   - The created filter handle..
     See "Creating an Instance of a Feature Type," on page 9 for more information.
   - The key value `NVAR_Parameter_Config(CUDAStream)`
     See "Key Values in the Properties of a Feature Type" on page 12 for more information.
   - The CUDA stream that you created in the previous step

This example sets up a CUDA stream that was created by calling the `NvAR_CudaStreamCreate()` function:

```
CUstream stream;
nvErr = NvAR_CudaStreamCreate (&stream);
nvErr = NvAR_SetCudaStream(featureHandle, NVAR_Parameter_Config(CUDAStream),
stream);
```

3.2.2 Setting the Input and Output Image Buffers

Some NVIDIA AR SDK features take a GPU `NvCVImage` structure as input. Currently, these features require input to be provided in a GPU buffer in a BGR interleaved format, where each pixel is a 24-bit unsigned `char` value. If the original buffer is on the CPU or is in planar format, it must be converted. See "Transferring Images Between CPU and GPU Buffers." on page 21 for more information.

For each image buffer, call the `NvAR_SetObject()` function, and specify the following information as parameters:

- The created filter handle.
  See "Creating an Instance of a Feature Type" on page 9 for more information.
- The key value `NVAR_Parameter_Input(Image)`.
  See "Key Values in the Properties of a Feature Type" on page 12 for more information.
- The address of the `NvCVImage` object that was created for the input image.

This example creates an input image buffer:

```
NvCVImage inputImageBuffer;
...
nvErr = NvCVImage_Alloc(&inputImageBuffer, 640, 480, NVCV_BGR, NVCV_U8,
NVCV_CHUNKY, NVCV_GPU, 1)
...
NvAR_SetObject(featureHandle, NvAR_Parameter_Input(Image),
&inputImageBuffer, sizeof(NvCVImage));
```

See "Working with Image Frames" on page 17 for more information about using the `NvCVImage` object. See "Image Functions for C and C++" on page 70 and "Image Functions for C++ Only" on page 86 for a complete list of functions that are associated with the `NvCVImage` object.

3.2.3 Summary of NVIDIA AR SDK Accessor Functions

Table 3-1: Summary of NVIDIA AR SDK Accessor Functions

Property Type	Data Type	Set and Get Accessor Function
32-bit unsigned integer	`unsigned int`	`NvAR_SetU32()`
		`NvAR_GetU32()`
32-bit signed integer	`int`	`NvAR_SetS32()`
		`NvAR_GetS32()`
Single-precision (32-bit) floating-point number	`float`	`NvAR_SetF32()`
		`NvAR_GetF32()`

Double-precision (64-bit) floating point number	`double`	`NvAR_SetF64()`
		`NvAR_GetF64()`
64-bit unsigned integer	`unsigned long long`	`NvAR_SetU64()`
		`NvAR_GetU64()`
Floating-point array	`float*`	`NvAR_SetFloatArray()`
		`NvAR_GetFloatArray()`
Object	`void*`	`NvAR_SetObject()`
		`NvAR_GetObject()`
Character string	`const char*`	`NvAR_SetString()`
		`NvAR_GetString()`
CUDA stream	`CUstream`	`NvAR_SetCudaStream()`
		`NvAR_GetCudaStream()`

3.2.4 Key Values in the Properties of a Feature Type

The key values in the properties of a feature type identify the properties that can be used with each feature type. Each key has a string equivalent and is defined by a macro that indicates the category of the property and takes a name as an input to the macro.

Here are the macros that indicate the category of a property:

- `NvAR_Parameter_Config` indicates a configuration property.
  See "Configuration Properties" on page 13 for more information.
- `NvAR_Parameter_Input` indicates an input property.
  See "Input Properties" on page 14 for more information.
- `NvAR_Parameter_Output` indicates an output property.
  See "Output Properties" on page 14 for more information.

The names are fixed keywords and are listed in `nvAR_defs.h`. The keywords might be reused with different macros, depending on whether a property is an input, an output, or a configuration property.

The property type denotes the accessor functions to set and get the property as listed in "Summary of NVIDIA AR SDK Accessor Functions," on page 10.

3.2.4.1 Configuration Properties

`NvAR_Parameter_Config(FeatureDescription)`

A description of the feature type.

String equivalent: `NvAR_Parameter_Config_FeatureDescription`

Property type: character string (`const char*`)

`NvAR_Parameter_Config(CUDAStream)`

The CUDA stream in which to run the feature.

String equivalent: `NvAR_Parameter_Config_CUDAStream`

Property type: CUDA stream (`CUstream`)

`NvAR_Parameter_Config(ModelDir)`

The path to the directory that contains the TensorRT model files that will be used to run inference for face detection or landmark detection, and the .nvf file that contains the 3D Face model, **excluding** the model file name. For details about the format of the .nvf file, see "NVIDIA 3DMM File Format," on page 93.

String equivalent: `NvAR_Parameter_Config_ModelDir`

Property type: character string (`const char*`)

`NvAR_Parameter_Config(BatchSize)`

The number of inferences to be run at one time on the GPU.

String equivalent: `NvAR_Parameter_Config_BatchSize`

Property type: unsigned integer

`NvAR_Parameter_Config(Landmarks_Size)`

The length of the output buffer that contains the X and Y coordinates in pixels of the detected landmarks. This property applies only to the landmark detection feature.

String equivalent: `NvAR_Parameter_Config_Landmarks_Size`

Property type: unsigned integer

`NvAR_Parameter_Config(LandmarksConfidence_Size)`

The length of the output buffer that contains the confidence values of the detected landmarks. This property applies only to the landmark detection feature.

String equivalent: `NvAR_Parameter_Config_LandmarksConfidence_Size`

Property type: unsigned integer

`NvAR_Parameter_Config(Temporal)`

Flag to enable optimization for temporal input frames. Enable the flag when the input is a video.

String equivalent: `NvAR_Parameter_Config_Temporal`

Property type: unsigned integer

3.2.4.2 Input Properties

`NvAR_Parameter_Input(Image)`

GPU input image buffer of type NvCVImage

String equivalent: `NvAR_Parameter_Input_Image`

Property type: object (`void*`)

NvAR_Parameter_Input(Width)

The width of the input image buffer in pixels.

String equivalent: `NvAR_Parameter_Input_Width`

Property type: integer

`NvAR_Parameter_Input(Height)`

The height of the input image buffer in pixels.

String equivalent: `NvAR_Parameter_Input_Height`

Property type: integer

`NvAR_Parameter_Input(Landmarks)`

CPU input array of type `NvAR_Point2f` that contains the facial landmark points.

String equivalent: `NvAR_Parameter_Input_Landmarks`

Property type: object (`void*`)

`NvAR_Parameter_Input(BoundingBoxes)`

Bounding boxes that determine the region of interest (ROI) of an input image that contains a face, of type `NvAR_BBoxes`.

String equivalent: `NvAR_Parameter_InputBoundingBoxes`

Property type: object (`void*`)

3.2.4.3 Output Properties

`NvAR_Parameter_Output(BoundingBoxes)`

CPU output bounding boxes of type `NvAR_BBoxes`.

String equivalent: `NvAR_Parameter_Output_BoundingBoxes`

Property type: object (`void*`)

`NvAR_Parameter_Output(BoundingBoxesConfidence)`

Float array of confidence values for each returned bounding box.

String equivalent: `NvAR_Parameter_Output_BoundingBoxesConfidence`

Property type: floating point array

`NvAR_Parameter_Output(Landmarks)`

CPU output buffer of type `NvAR_Point2f` to hold the output detected landmark key points. Refer to Facial point annotations for more information. The order of the points in the CPU buffer follows the order in MultiPIE 68-point markups.

String equivalent: `NvAR_Parameter_Output_Landmarks`

Property type: object (`void*`)

`NvAR_Parameter_Output(LandmarksConfidence)`

Float array of confidence values for each detected landmark point.

String equivalent: `NvAR_Parameter_Output_LandmarksConfidence`

Property type: floating point array

`NvAR_Parameter_Output(Pose)`

CPU array of type `NvAR_Quaternion` to hold the output-detected pose as an XYZW quaternion.

String equivalent: `NvAR_Parameter_Output_Pose`

Property type: object (`void*`)

`NvAR_Parameter_Output(FaceMesh)`

CPU 3D face Mesh of type `NvAR_FaceMesh`.

String equivalent: `NvAR_Parameter_Output_FaceMesh`

Property type: object (`void*`)

`NvAR_Parameter_Output(RenderingParams)`

CPU output structure of type `NvAR_RenderingParams` that contains the rendering parameters that might be used to render the 3D face mesh.

String equivalent: `NvAR_Parameter_Output_RenderingParams`

Property type: object (`void*`)

3.2.5 Getting the Value of a Property of a Feature Type

To get the value of a property of a feature, call the get accessor function that is appropriate for the data type of the property. In the call to the function, pass the following information:

- The feature handle to the feature instance.
- The key value that identifies the property that you are getting.
- The location in memory where you want the value of the property to be written.

This example determines the length of the `NvAR_Point2f` output buffer that was returned by the landmark detection feature:

```
unsigned int OUTPUT_SIZE_KPTS;
NvAR_GetU32(landmarkDetectHandle, NvAR_Parameter_Config(Landmarks_Size),
&OUTPUT_SIZE_KPTS);
```

3.2.6 Setting a Property for a Feature Type

To set a property for a Feature Type:

1. Allocate memory for all inputs and outputs that are required by the feature type and any other properties that might be required.
2. Call the set accessor function that is appropriate for the data type of the property.

   In the call to the function, pass the following information:

   - The feature handle to the feature instance.
   - The key value that identifies the property that you are setting.
   - A pointer to the value to which you want to set the property.

This example sets the file path to the file that contains the output 3D face model:

```
const char *modelPath = "file/path/to/model";
NvAR_SetString(landmarkDetectHandle, NvAR_Parameter_Config(ModelDir),
modelPath);
```

This example sets up the input image buffer in GPU memory, which is required by the face detection feature type:

> **Note**: It sets up an 8-bit chunky/interleaved BGR array.

```
NvCVImage InputImageBuffer;
NvCVImage_Alloc(&inputImageBuffer, input_image_width, input_image_height,
NVCV_BGR, NVCV_U8, NVCV_CHUNKY, NVCV_GPU, 1) ;
NvAR_SetObject(landmarkDetectHandle, NvAR_Parameter_Input(Image),
&InputImageBuffer, sizeof(NvCVImage));
```

See "List of Properties for AR Features" on page 17 for more information about the properties and input and output requirements for each feature.

> **Note**: The listed property name is the input to the macro that defines the key value for the property.

3.3 Loading a Feature Instance

You can load the feature after setting the configuration properties that are required to load an instance of a feature type.

To load a feature instance, call the `NvAR_Load()` function and specify the handle that was created for the feature instance when the instance was created. See "Creating an Instance of a Feature Type," on page 9 for more information

This example loads an instance of the face detection feature type:

```
NvAR_Load(faceDetectHandle);
```

3.4 Running a Feature Instance

Before you can run the feature instance, load an instance of a feature type and set the user-allocated input and output memory buffers that are required when the feature instance is run.

To run a feature instance, call the `NvAR_Run()` function and specify the handle that was created for the feature instance when the instance was created. See "Creating an Instance of a Feature Type" on page 9 for more information.

This example shows how to run a face detection feature instance:

```
NvAR_Run(faceDetectHandle);
```

3.5 Destroying a Feature Instance

When a feature instance is no longer required, you need to destroy it to free the resources and memory that the feature instance allocated internally. Memory buffers are provided as input and to hold the output of a feature and must be separately deallocated.

To destroy a feature instance, call the `NvAR_Destroy()` function and specify the handle that was created for the feature instance when the instance was created. See "Creating an Instance of a Feature Type" on page 9 for more information.

```
NvAR_Destroy(faceDetectHandle);
```

3.6 Working with Image Frames

NVIDIA AR SDK features accept image buffers as `NvCVImage` objects. The image buffers can be CPU or GPU buffers, but for performance reasons, the effect filters require GPU buffers . The SDK provides functions to convert an image representation to `NvCVImage` and the ability to transfer the images between the CPU and GPU buffers.

3.6.1 Converting Image Representations to NvCVImage Objects

NVIDIA AR SDK provides functions to convert OpenCV images and other image representations to `NvCVImage` objects. Each function places a wrapper around an existing buffer, and the wrapper prevents the buffer from being freed when the destructor of the wrapper is called.

3.6.1.1 Converting OpenCV Images

Use the wrapper functions that the SDK provides specifically for RGB OpenCV images.

> **Note**: The SDK provides wrapper functions only for RGB images. No wrapper functions are provided for YUV images.

- To create an `NvCVImage` object wrapper for an OpenCV image, use the `NVWrapperForCVMat()` function.

```
//Allocate source and destination OpenCV images
cv::Mat srcCVImg(   );
cv::Mat dstCVImg(...);

// Declare source and destination NvCVImage objects
NvCVImage srcCPUImg;
NvCVImage dstCPUImg;

NVWrapperForCVMat(&srcCVImg, &srcCPUImg);
NVWrapperForCVMat(&dstCVImg, &dstCPUImg);
```

- To create an OpenCV image wrapper for an `NvCVImage` object, use the `CVWrapperForNvCVImage()` function.

```
// Allocate source and destination NvCVImage objects
NvCVImage srcCPUImg(...);
NvCVImage dstCPUImg(...);

//Declare source and destination OpenCV images
cv::Mat srcCVImg;
cv::Mat dstCVImg;

CVWrapperForNvCVImage (&srcCPUImg, &srcCVImg);
CVWrapperForNvCVImage (&dstCPUImg, &dstCVImg);
```

3.6.1.2 Converting Other Image Representations

Call the `NvCVImage_Init()` function to place a wrapper around an existing buffer (`srcPixelBuffer`).

```
NvCVImage src;
nvErr = NvCVImage_Init(&src, 640, 480, 1920, srcPixelBuffer, NVCV_BGR,
NVCV_U8, NVCV_INTERLEAVED, NVCV_CPU);
```

3.6.2 Allocating an NvCVImage Object Buffer

You can allocate the buffer for an `NvCVImage` object by using the `NvCVImage` allocation constructor or image functions. For each of these options, the buffer is automatically freed by the destructor when the images go out of scope.

3.6.2.1 Using the NvCVImage Allocation Constructor to Allocate a Buffer

The `NvCVImage` allocation constructor creates an object to which memory has been allocated and that has been initialized. See "Allocation Constructor" on page 86.

The following optional allocation constructor parameters determine the properties of the resulting `NvCVImage` object:

- The pixel organization determines whether blue, green, and red are in separate planes or interleaved.
- The memory type determines whether the buffer resides on the GPU or the CPU.
- The byte alignment determines the gap between consecutive scanlines.

The following examples show how to use the these optional parameters and determine the properties of the `NvCVImage` object.

- This example creates an object without setting the optional parameters . In this object, the blue, green, and red components are interleaved in each pixel, the buffer resides on the CPU, and the byte alignment is the default alignment:

  ```
  NvCVImage cpuSrc(
    srcWidth,
    srcHeight,
    NVCV_BGR,
    NVCV_U8
  );
  ```

- This example creates an object with identical pixel organization, memory type, and byte alignment to the previous example by explicitly setting the optional parameters:

> **Note**: As in the previous example, the blue, green, and red components are interleaved in each pixel, the buffer resides on the CPU, and the byte alignment is by default, optimized for maximum performance.

.

```
NvCVImage src(
  srcWidth,
  srcHeight,
  NVCV_BGR,
  NVCV_U8,
  NVCV_INTERLEAVED,
  NVCV_CPU,
  0
);
```

- This example creates an object in which the blue, green, and red components are in separate planes, the buffer resides on the GPU, and the byte alignment ensures that no gap exists between one scanline and the next scanline:

```
NvCVImage gpuSrc(
  srcWidth,
  srcHeight,
  NVCV_BGR,
  NVCV_U8,
  NVCV_PLANAR,
  NVCV_GPU,
  1
);
```

3.6.2.2 Using Image Functions to Allocate a Buffer

By declaring an empty image, you can defer buffer allocation.

1. Declare an empty `NvCVImage` object.

```
NvCVImage xfr;
```

2. Allocate or reallocate the buffer for the image.
   - To allocate the buffer, call the `NvCVImage_Alloc()` function.

     You can allocate a buffer with this function when the image is part of a state structure and where you won't know the size the image until later.
   - To reallocate a buffer, call `NvCVImage_Realloc()`.

     This function checks for an allocated buffer and, if it is big enough, reshapes the buffer, , before freeing the buffer and calling `NvCVImage_Alloc()`.

3.6.3 Transferring Images Between CPU and GPU Buffers

If the memory types of the input and output image buffers are different, an application can transfer images between CPU and GPU buffers.

3.6.3.1 Transferring Input Images from a CPU Buffer to a GPU Buffer

1. Create an `NvCVImage` object to use as a staging GPU buffer that has the same dimensions and format as the source CPU buffer.

   ```
   NvCVImage srcGpuPlanar(inWidth, inHeight, NVCV_BGR, NVCV_F32,
   NVCV_PLANAR, NVCV_GPU,1)
   ```

2. Create a staging buffer in one of the following ways:
   - To avoid allocating memory in a video pipeline, create a GPU buffer that has the same dimensions and format as required for the input to the video effect filter.

     ```
     NvCVImage srcGpuStaging(inWidth, inHeight, srcCPUImg.pixelFormat,
     srcCPUImg.componentType, srcCPUImg.planar, NVCV_GPU)
     ```

   - To simplify your application program code, declare an empty staging buffer.

     ```
     NvCVImage srcGpuStaging;
     ```

     An appropriate buffer will be allocated or reallocated as needed.

3. Call the `NvCVImage_Transfer()` function to copy the source CPU buffer contents into the final GPU buffer via the staging GPU buffer.

   ```
   //Read the image into srcCPUImg
   NvCVImage_Transfer(&srcCPUImg, &srcGPUPlanar, 1.0f, stream,
   &srcGPUStaging)
   ```

3.6.3.2 Transferring Output Images from a GPU Buffer to a CPU Buffer

1. Create an `NvCVImage` object that can be used as a staging GPU buffer and that has the same dimensions and format as the destination CPU buffer.

   ```
   NvCVImage dstGpuPlanar(outWidth, outHeight, NVCV_BGR, NVCV_F32,
   NVCV_PLANAR, NVCV_GPU, 1)
   ```

2. Create a staging buffer in one of the following ways:
   - To avoid allocating memory in a video pipeline, create a GPU buffer that has the same dimensions and format as the output of the video effect filter.

     ```
     NvCVImage dstGpuStaging(outWidth, outHeight, dstCPUImg.pixelFormat,
     dstCPUImg.componentType, dstCPUImg.planar, NVCV_GPU)
     ```

   - To simplify your application program code, declare an empty staging buffer.

     ```
     NvCVImage dstGpuStaging;
     ```

     An appropriately sized buffer will be allocated as needed.

3. Call the `NvCVImage_Transfer()` function to copy the GPU buffer contents into the destination CPU buffer via the staging GPU buffer.

```
//Retrieve the image from the GPU to CPU, perhaps with conversion.
NvCVImage_Transfer(&dstGpuPlanar, &dstCPUImg, 1.0f, stream,
&dstGpuStaging);
```

3.7 List of Properties for AR Features

3.7.1 Face Tracking Property Values

Table 3-2: Configuration Properties for Face Tracking

Property Name	Value
`FeatureDescription`	String is free-form text that describes the feature. The string is set by the SDK and cannot be modified by the user.
`CUDAStream`	The CUDA stream. Set by the user.
`ModelDir`	String that contains the path to the folder that contains the TensorRT package files. Set by the user.
`Temporal`	Unsigned integer, 1/0 to enable/disable the temporal optimization of face detection. If enabled, only one face is returned. See "Face Detection on page 28" for more information. Set by the user.

Table 3-3: Input Properties for Face Tracking

Property Name	Value
`Image`	Interleaved (or chunky) 8-bit BGR input image in a CUDA buffer of type `NvCVImage`. To be allocated and set by the user.

Table 3-4: Output Properties for Face Tracking

Property Name	Value
`BoundingBoxes`	`NvAR_BBoxes` structure that holds the detected face boxes. To be allocated by the user.
`BoundingBoxesConfidence`	**Optional:** An array of single-precision (32-bit) floating-point numbers that contains the confidence values for each detected face box. To be allocated by the user.

3.7.2 Landmark Tracking Property Values

Table 3-5: Configuration Properties for Landmark Tracking

Property Name	Value
`FeatureDescription`	String that describes the feature.
`CUDAStream`	The CUDA stream. Set by the user.
`ModelDir`	String that contains the path to the folder that contains the TensorRT package files. Set by the user.
`BatchSize`	• The default value is 1. • The maximum value is 8.
`Landmarks_Size`	Unsigned integer, 68. The X and Y values for 68 detected key points are returned.
`LandmarksConfidence_Size`	Unsigned integer, 68. The confidence values for 68 detected key points are returned.
`Temporal`	Unsigned integer, 1/0 to enable/disable the temporal optimization of landmark detection. If enabled, only one input bounding box is supported as the input. See "Landmark Detection" on page 29 for more information. Set by the user.

Table 3-6: Input Properties for Landmark Tracking

Property Name	Value
`Image`	Interleaved (or chunky) 8-bit BGR input image in a CUDA buffer of type `NvCVImage`. To be allocated and set by the user.
`BoundingBoxes`	**Optional:** `NvAR_BBoxes` structure that contains the number of bounding boxes that are equal to `BatchSize` on which to run landmark detection. If not specified as an input property, face detection is automatically run on the input image. See "Landmark Detection" on page 29 for more information. To be allocated by the user.

Table 3-7: Output Properties for Landmark Tracking

Property Name	Value
`Landmarks`	`NvAR_Point2f` array, which must be large enough to hold the number of points given by the product of `NvAR_Parameter_Config(BatchSize)` and `NvAR_Parameter_Config(Landmarks_Size)`. To be allocated by the user.
`Pose`	**Optional:** `NvAR_Quaternion` array, which must be large enough to hold the number of quaternions equal to `NvAR_Parameter_Config(BatchSize)`. To be allocated by the user.
`LandmarksConfidence`	**Optional:** An array of single-precision (32-bit) floating-point numbers, which must be large enough to hold the number of confidence values given by the product of `NvAR_Parameter_Config(BatchSize)` and `NvAR_Parameter_Config(LandmarksConfidence_Size)`. To be allocated by the user.

`BoundingBoxes`	**Optional:** `NvAR_BBoxes` structure that contains the detected face through face detection performed by the landmark detection feature. See "Landmark Detection" on page 29 for more information. To be allocated by the user.

3.7.3 Face 3D Mesh tracking Property Values

Table 3-8: Configuration Properties for Face 3D Mesh Tracking

Property Name	Value
`FeatureDescription`	String that describes the feature.
`ModelDir`	String that contains the path to the face model, and the TensorRT package files. See "Alternative usage of Face 3D Mesh Feature" on page 31 for more information. Set by the user.
`CUDAStream`	**Optional:** The CUDA stream. See "Alternative usage of Face 3D Mesh Feature on page 31" for more information. Set by the user.
`Temporal`	**Optional:** Unsigned integer, 1/0 to enable/disable the temporal optimization of face and landmark detection. See "Alternative usage of Face 3D Mesh Feature" on page 31 for more information. Set by the user.
`LandmarksConfidence_Size`	Unsigned integer, 68. The confidence values for 68 detected key points are returned if landmark detection is run internally. See "Alternative usage of Face 3D Mesh Feature" on page 31 for more information.

Table 3-9: Input Properties for Face 3D Mesh Tracking

Property Name	Value
`Width`	The width of the input image buffer that contains the face to which the face model is to be fitted. Set by the user.
`Height`	The height of the input image buffer that contains the face to which the face model is to be fitted. Set by the user.
`Landmarks`	**Optional:** An `NvAR_Point2f` array that contains the landmark points of size `NvAR_Parameter_Config(Landmarks_Size)` that is returned by the landmark detection feature. If landmarks are not provided to this feature, an input image must be provided. See "Alternative usage of Face 3D Mesh Feature" on page 31 for more information. To be allocated by user.
`Image`	**Optional:** An interleaved (or chunky) 8-bit BGR input image in a CUDA buffer of type `NvCVImage`. If an input image is not provided as input, the landmark points must be provided to this feature as input. See "Alternative usage of Face 3D Mesh Feature" on page 31 for more information. To be allocated by the user.

Table 3-10: Output Properties for Face 3D Mesh Tracking

Property Name	Value
FaceMesh	NvAR_FaceMesh structure that contains the output face mesh. To be allocated by the user.
RenderingParams	NvAR_RenderingParams structure that contains the rendering parameters for drawing the face mesh that is returned by this feature. To be allocated by the user.
Landmarks	**Optional:** An NvAR_Point2f array, which must be large enough to hold the number of points of size NvAR_Parameter_Config(Landmarks_Size). See "Alternative usage of Face 3D Mesh Feature" on page 31 for more information. To be allocated by the user.
Pose	**Optional:** NvAR_Quaternion array pointer, to hold one quaternion. See "Alternative usage of Face 3D Mesh Feature" on page 31 for more information. To be allocated by the user.
LandmarksConfidence	**Optional:** An array of single-precision (32-bit) floating-point numbers, which must be large enough to hold the number of confidence values of size Parameter_Config(LandmarksConfidence_Size). See "Alternative usage of Face 3D Mesh Feature" on page 31 for more information. To be allocated by the user.
BoundingBoxes	**Optional:** NvAR_BBoxes structure that contains the detected face that is determined internally. See "Alternative usage of Face 3D Mesh Feature" on page 31 for more information. To be allocated by the user.
BoundingBoxesConfidence	**Optional:** An array of single-precision (32-bit) floating-point numbers that contain the confidence values for each detected face box. See "Alternative usage of Face 3D Mesh Feature" on page 31 for more information. To be allocated by the user.

3.8 Using the AR Features

3.8.1 Face Detection and Tracking

3.8.1.1 Face Detection for Static Frames (Images)

To obtain detected bounding boxes, you can explicitly instantiate and run the face detection feature as below, with the feature taking an image buffer as input.

This example runs the Face Detection AR feature with an input image buffer and output memory to hold bounding boxes:

```
//Set input image buffer
NvAR_SetObject(faceDetectHandle, NvAR_Parameter_Input(Image),
&inputImageBuffer, sizeof(NvCVImage));

//Set output memory for bounding boxes
NvAR_BBoxes = output_boxes{};
output_bboxes.boxes = new NvAR_Rect[25];
output_bboxes.max_boxes = 25;
NvAR_SetObject(faceDetectHandle, NvAR_Parameter_Output(BoundingBoxes),
&output_bboxes, sizeof(NvAR_BBoxes));

//OPTIONAL – Set memory for bounding box confidence values if desired

NvAR_Run(faceDetectHandle);
```

3.8.1.2 Face Tracking for Temporal Frames (Videos)

If `Temporal` is enabled, for example when you process a video frame instead of an image, only one face is returned. The largest face appears for the first frame, and this face is subsequently tracked over following frames.

However, explicitly calling the face detection feature is not the only way to obtain a bounding box that denotes detected faces. See "Landmark Detection and Tracking" on page 29 and "Face 3D Mesh and Tracking" on page 31 for more information about how to use the Landmark Detection or Face3D Reconstruction AR features and return a face bounding box.

3.8.2 Landmark Detection and Tracking

3.8.2.1 Landmark Detection for Static Frames (Images)

Typically, the input to the landmark detection feature is an input image and a batch (up to 8) of bounding boxes. These boxes denote the regions of the image that contain the faces on which you want to run landmark detection.

This example runs the Landmark Detection AR feature after obtaining bounding boxes from Face Detection:

```
//Set input image buffer
NvAR_SetObject(landmarkDetectHandle, NvAR_Parameter_Input(Image),
&inputImageBuffer, sizeof(NvCVImage));

//Pass output bounding boxes from face detection as an input on which
//landmark detection is to be run
NvAR_SetObject(landmarkDetectHandle, NvAR_Parameter_Input(BoundingBoxes),
&output_bboxes, sizeof(NvAR_BBoxes));

//Set output buffer to hold detected facial keypoints
std::vector<NvAR_Point2f> facial_landmarks;
facial_landmarks.assign(OUTPUT_SIZE_KPTS, {0.f, 0.f});
NvAR_SetObject(landmarkDetectHandle, NvAR_Parameter_Output(Landmarks),
facial_landmarks.data(),sizeof(NvAR_Point2f));

NvAR_Run(landmarkDetectHandle);
```

3.8.2.2 Alternative usage of Landmark Detection

However, as described in "Configuration Properties for Landmark Detection on page 23", the Landmark Detection AR feature supports some optional parameters that determine how the feature can be run. If bounding boxes are not provided to the Landmark Detection AR feature as inputs, face detection is automatically run on the input image, and the largest face bounding box is selected on which to run landmark detection. If `BoundingBoxes` is set as an output property, the property is populated with the selected bounding box that contains the face on which the landmark detection was run.

`Landmarks` is not an optional property and, to explicitly run this feature, this property must be set with a provided output buffer.

3.8.2.3 Landmark Tracking for Temporal Frames (Videos)

Additionally, if `Temporal` is enabled, for example when you process a video stream and face detection is run explicitly, only one bounding box is supported as an input for landmark detection. When face detection is not explicitly run, by providing an input image instead of a bounding box, the largest detected face is automatically selected. The detected face and landmarks are then tracked as an optimization across temporally related frames.

> **Note**: The internally determined bounding box can be queried from this feature but is not required for the feature to run.

This example uses the Landmark Detection AR feature to obtain landmarks directly from the image, without first explicitly running Face Detection:

```
//Set input image buffer
NvAR_SetObject(landmarkDetectHandle, NvAR_Parameter_Input(Image),
&inputImageBuffer, sizeof(NvCVImage));

//Set output memory for landmarks
std::vector<NvAR_Point2f> facial_landmarks;
facial_landmarks.assign(batchSize * OUTPUT_SIZE_KPTS, {0.f, 0.f});
NvAR_SetObject(landmarkDetectHandle, NvAR_Parameter_Output(Landmarks),
facial_landmarks.data(),sizeof(NvAR_Point2f));

//OPTIONAL – Set output memory for bounding box if desired
NvAr_BBoxes = output_boxes{};
output_bboxes.boxes = new NvAR_Rect[25];
output_bboxes.max_boxes = 25;
NvAR_SetObject(landmarkDetectHandle, NvAR_Parameter_Output(BoundingBoxes),
&output_bboxes, sizeof(NvAr_BBoxes));

//OPTIONAL – Set output memory for pose, landmark confidence, or even
bounding box confidence if desired

NvAR_Run(landmarkDetectHandle);
```

3.8.3 Face 3D Mesh and Tracking

3.8.3.1 Face 3D Mesh for static frames (Images)

Typically, the input to Face 3D Mesh feature is an input image and a set of detected landmark points corresponding to the face on which we want to run 3D reconstruction.

Here is the typical usage of this feature, where the detected facial keypoints from the Landmark Detection feature are passed as input to this feature:

```
//Set facial keypoints from Landmark Detection as an input
NvAR_SetObject(faceFitHandle, NvAR_Parameter_Input(Landmarks),
facial_landmarks.data(),sizeof(NvAR_Point2f));

//Set output memory for face mesh
NvAR_FaceMesh face_mesh = new NvAR_FaceMesh();
face_mesh->vertices = new NvAR_Vector3f[FACE_MODEL_NUM_VERTICES];
face_mesh->tvi = new NvAR_Vector3u16[FACE_MODEL_NUM_INDICES];
NvAR_SetObject(faceFitHandle, NvAR_Parameter_Output(FaceMesh), face_mesh,
sizeof(NvAR_FaceMesh));

//Set output memory for rendering parameters
NvAR_RenderingParams rendering_params = new NvAR_RenderingParams();
NvAR_SetObject(faceFitHandle, NvAR_Parameter_Output(RenderingParams),
rendering_params, sizeof(NvAR_RenderingParams));

NvAR_Run(faceFitHandle);
```

3.8.3.2 Alternative usage of Face 3D Mesh Feature

Similar to the alternative usage of the Landmark detection feature, the Face 3D Mesh AR feature can be used to determine the detected face bounding box, the facial keypoints, and a 3D face mesh and its rendering parameters.

Instead of the facial keypoints of a face, if an input image is provided, the face and the facial keypoints are automatically detected and used to run the face mesh fitting. When run this way, if `BoundingBoxes` and/or `Landmarks` are set as optional output properties for this feature, these properties will be populated with the bounding box that contains the face and the detected facial keypoints, respectively.

`FaceMesh` and `RenderingParams` are not optional properties for this feature, and to run the feature, these properties must be set with user-provided output buffers.

Additionally, if this feature is run without providing facial keypoints as an input, the path pointed to by the `ModelDir` config parameter must also contain the face and landmark detection TRT package files. Optionally, the `CUDAStream` and the `Temporal` flag can be set for those features.

3.8.3.3 Face 3D Mesh Tracking for Temporal Frames (Videos)

If the `Temporal` flag is set and face and landmark detection are run internally, we will optimize those features for temporally related frames. This means that face and facial keypoints will be tracked across frames, and only one bounding box will be returned, if requested, as an output. The `Temporal` flag is not supported by the Face 3D Mesh feature if Landmark Detection and/or Face Detection features are called explicitly. In that case, you will have to provide the flag directly to those features.

> **Note**: The facial keypoints and/or the face bounding box that were determined internally can be queried from this feature but are not required for the feature to run.

This example uses the Mesh Tracking AR feature is used to obtain the face mesh directly from the image, without explicitly running Landmark Detection or Face Detection:

```
//Set input image buffer instead of providing facial keypoints
NvAR_SetObject(landmarkDetectHandle, NvAR_Parameter_Input(Image),
&inputImageBuffer, sizeof(NvCVImage));

//Set output memory for face mesh
NvAR_FaceMesh face_mesh = new NvAR_FaceMesh();
face_mesh->vertices = new NvAR_Vector3f[FACE_MODEL_NUM_VERTICES];
face_mesh->tvi = new NvAR_Vector3u16[FACE_MODEL_NUM_INDICES];
NvAR_SetObject(faceFitHandle, NvAR_Parameter_Output(FaceMesh), face_mesh,
sizeof(NvAR_FaceMesh));

//Set output memory for rendering parameters
NvAR_RenderingParams rendering_params = new NvAR_RenderingParams();
NvAR_SetObject(faceFitHandle, NvAR_Parameter_Output(RenderingParams),
rendering_params, sizeof(NvAR_RenderingParams));

//OPTIONAL - Set facial keypoints as an output
NvAR_SetObject(faceFitHandle, NvAR_Parameter_Output(Landmarks),
facial_landmarks.data(),sizeof(NvAR_Point2f));

//OPTIONAL – Set output memory for bounding boxes, or other parameters, such
as pose, bounding box/landmarks confidence, etc.

NvAR_Run(faceFitHandle);
```

Chapter 4. NVIDIA AR SDK API Reference

4.1 Structures

The structures in the NVIDIA AR SDK are defined in the following header files:

- `nvAR.h`
- `nvAR_defs.h`

The structures defined in the `nvAR_defs.h` header file are mostly data types.

4.1.1 NvAR_BBoxes

```
struct NvAR_BBoxes {
  NvAR_Rect *boxes;
  uint8_t num_boxes;
  uint8_t max_boxes;
};
```

4.1.1.1 Members

`boxes`

Type: `NvAR_Rect *`

Pointer to an array of bounding boxes that are allocated by the user.

`num_boxes`

Type: `uint8_t`

The number of bounding boxes in the array.

max_boxes

Type: uint8_t

The maximum number of bounding boxes that can be stored in the array as defined by the user.

4.1.1.2 Remarks

This structure is returned as the output of the face detection feature.

Defined in: nvAR_defs.h.

4.1.2 NvAR_FaceMesh

```
struct NvAR_FaceMesh {
  NvAR_Vec3<float> *vertices;
  size_t num_vertices;
  NvAR_Vec3<unsigned short> *tvi;
  size_t num_tri_idx;
};
```

4.1.2.1 Members

vertices

Type: NvAR_Vec3<float>*

Pointer to an array of vectors that represent the mesh 3D vertex positions.

num_vertices

Type: size_t

The number of vertices in the array pointed to by the vertices parameter.

tvi

Type: NvAR_Vec3<unsigned short> *

Pointer to an array of vectors that represent the mesh triangle's vertex indices.

num_tri_idx

Type: size_t

The number of mesh triangle vertex indices.

4.1.2.2 Remarks

This structure is returned as an output of the Mesh Tracking feature.

Defined in: `nvAR_defs.h`.

4.1.3 NvAR_Frustum

```
struct NvAR_Frustum {
  float left = -1.0f;
  float right = 1.0f;
  float bottom = -1.0f;
  float top = 1.0f;
};
```

4.1.3.1 Members

`left`

Type: `float`

The X coordinate of the top-left corner of the viewing frustum.

`right`

Type: `float`

The X coordinate of the bottom-right corner of the viewing frustum.

`bottom`

Type: `float`

The Y coordinate of the bottom-right corner of the viewing frustum.

`top`

Type: `float`

The Y coordinate of the top-left corner of the viewing frustum.

4.1.3.2 Remarks

This structure represents a camera viewing frustum for an orthographic camera. As a result, it contains only the left, the right, the top, and the bottom coordinates in pixels It does **not** contain a near or a far clipping plane.

Defined in: `nvAR_defs.h`.

4.1.4 NvAR_FeatureHandle

```
typedef struct nvAR_Feature *NvAR_FeatureHandle;
```

4.1.4.1 Remarks

This type defines the handle of a feature that is defined by the SDK. It is used to reference the feature at runtime, when the feature is executed, and must be destroyed when it is no longer required.

Defined in: `nvAR_defs.h`.

4.1.5 NvAR_Point2f

```
typedef struct NvAR_Point2f {
  float x, y;
} NvAR_Point2f;
```

4.1.5.1 Members

`x`

  Type: `float`

  The X coordinate of the point in pixels.

`y`

  Type: `float`

  The Y coordinate of the point in pixels.

4.1.5.2 Remarks

This structure represents the X and Y coordinates of one point in 2D space.

Defined in: `nvAR_defs.h`.

4.1.6 NvAR_Quaternion

```
struct NvAR_Quaternion {
  float x, y, z, w;
};
```

4.1.6.1 Members

`x`

Type: `float`

The first coefficient of the complex part of the quaternion.

`y`

Type: `float`

The second coefficient of the complex part of the quaternion.

`z`

Type: `float`

The third coefficient of the complex part of the quaternion.

`w`

Type: `float`

The scalar coefficient of the quaternion.

4.1.6.2 Remarks

This structure represents the coefficients in the quaternion that are expressed in the following equation:

$$q = xi + yj + zk + w$$

Defined in: `nvAR_defs.h`.

4.1.7 NvAR_Rect

```
typedef struct NvAR_Rect {
  float x, y, width, height;
} NvAR_Rect;
```

4.1.7.1 Members

`x`

Type: `float`

The X coordinate of the top left corner of the bounding box in pixels.

`y`

Type: `float`

The Y coordinate of the top left corner of the bounding box in pixels.

`width`

Type: `float`

The width of the bounding box in pixels.

`height`

Type: `float`

The height of the bounding box in pixels.

4.1.7.2 Remarks

This structure represents the position and size of a rectangular 2D bounding box.

Defined in: `nvAR_defs.h`.

4.1.8 NvAR_RenderingParams

```
struct NvAR_RenderingParams {
  NvAR_Frustum frustum;
  NvAR_Quaternion rotation;
  NvAR_Vec3<float> translation;
};
```

4.1.8.1 Members

`frustum`

Type: `NvAR_Frustum`

The camera viewing frustum for an orthographic camera.

`rotation`

Type: `NvAR_Quaternion`

The rotation of the camera relative to the mesh.

`translation`

Type: `NvAR_Vec3<float>`

The translation of the camera relative to the mesh.

4.1.8.2 Remarks

This structure defines the parameters that are used to draw a 3D face mesh in a window on the computer screen, so that the face mesh is aligned with the corresponding video frame. The projection matrix is constructed from the `frustum` parameter, and the model view matrix is constructed from the `rotation` and `translation` parameters.

Defined in: `nvAR_defs.h`.

4.1.9 NvAR_Vec3

```
template <typename T>
struct NvAR_Vec3 {
  T vec[3];
  NvAR_Vec3() { vec[0] = vec[1] = vec[2] = 0; }
  NvAR_Vec3(T t_x, T t_y, T t_z) {
    vec[0] = t_x;
    vec[1] = t_y;
    vec[2] = t_z;
  }
  NvAR_Vec3(const NvAR_Vec3 &t) {
    vec[0] = t.vec[0];
    vec[1] = t.vec[1];
    vec[2] = t.vec[2];
  }
};
```

4.1.9.1 Members

`vec`

Type: templated array of size 3, usually `int` or `float`

A vector of size 3.

4.1.9.2 Remarks

This structure represents a 3D vector.

Defined in: `nvAR_defs.h`.

4.1.10 NvAR_Vector2f

```
typedef struct NvAR_Vector2f {
  float x, y;
} NvAR_Vector2f;
```

4.1.10.1 Members

x

Type: float

The X component of the 2D vector.

y

Type: float

The Y component of the 2D vector.

4.1.10.2 Remarks

This structure represents a 2D vector.

Defined in: nvAR_defs.h.

4.1.11 NvCVImage

```
typedef struct NvCVImage {
  unsigned int              width;
  unsigned int              height;
  unsigned int              pitch;
  NvCVImage_PixelFormat   pixelFormat;
  NvCVImage_ComponentType  componentType;
  unsigned char             pixelBytes;
  unsigned char             componentBytes;
  unsigned char             numComponents;
  unsigned char             planar;
  unsigned char             gpuMem;
  unsigned char             colorspace;
  unsigned char             batch;
  void                      *pixels;
  void                      *deletePtr;
  void                      (*deleteProc)(void *p);
  unsigned long long        bufferBytes;

} NvCVImage;
```

4.1.11.1 Members

`width`

Type: `unsigned int`

The width of the image in pixels.

`height`

Type: `unsigned int`

The height of the image in pixels.

`pitch`

Type: `unsigned int`

The vertical byte stride between pixels.

`pixelFormat`

Type: `NvCVImage_PixelFormat`

The format of the pixels in the image.

`componentType`

Type: `NvCVImage_ComponentType`

The data type used to represent each component of the image.

`pixelBytes`

Type: `unsigned char`

The number of bytes in a chunky pixel.

`componentBytes`

Type: `unsigned char`

The number of bytes in each pixel component.

`numComponents`

Type: `unsigned char`

The number of components in each pixel.

`planar`

Type: `unsigned char`

Specifies the organization of the pixels in the image.

- 0: Chunky
- 1: Planar

`gpuMem`

Type: `unsigned char`

Specifies the type of memory in which the image data buffer is stored. The different types of memory have different address spaces.

- 0: CPU memory
- 1: CUDA memory

`colorspace`

Type: `unsigned char`

Specifies a logical OR group of YUV color space types, for example:

```
my422.colorspace = NVCV_709 | NVCV_VIDEO_RANGE | NVCV_CHROMA_COSITED;
```

YUV Color Spaces on page 46 .

Always set the colorspace for 420 or 422 YUV images. The default colorspace is `NVCV_601 | NVCV_VIDEO_RANGE | NVCV_CHROMA_COSITED`.

`batch`

Type: `unsigned char`

Set this parameter to 1.

`pixels`

Type: `void`

Pointer to pixel (0,0) in the image.

`deletePtr`

Type: `void`

Buffer memory to be deleted (can be `NULL`).

`deleteProc`

Type: `void`

The function to call instead of `free()` to delete the pixel buffer. To call `free()`, set this parameter to `NULL`. The image allocators use `free()` for CPU buffers and `cudaFree()` for GPU buffers.

`bufferBytes`

Type: `unsigned long long`

The maximum amount of memory in bytes available through pixels.

4.1.11.2 Remarks

This structure defines the properties of an image in an image buffer that is provided as input to an effect filter.

The members can be set by using the setter functions in the NVIDIA AR SDK API.

Defined in: `nvCVImage.h`.

4.2 Enumerations

The enumerations in the NVIDIA AR SDK are defined in the header file `nvCVImage.h`.

4.2.1 NvCVImage_ComponentType

This enumeration defines the data type used to represent one component of a pixel.

`NVCV_TYPE_UNKNOWN  = 0`

Unknown component data type.

`NVCV_U8 = 1`

Unsigned 8-bit integer.

`NVCV_U16 = 2`

Unsigned 16-bit integer.

`NVCV_S16 = 3`

Signed 16-bit integer.

`NVCV_F16 = 4`

16-bit floating-point number.

`NVCV_U32`

Unsigned 32-bit integer.

`NVCV_S32 = 6`

Signed 32-bit integer.

`NVCV_F32 = 7`

32-bit floating-point number (`float`).

`NVCV_U64 =8`

Unsigned 64-bit integer.

`NVCV_S64 = 9`

Signed 64-bit integer.

`NVCV_F64 = 10`

64-bit floating-point (`double`).

4.2.2 NvCVImage_PixelFormat

This enumeration defines the order of the components in a pixel.

`NVCV_FORMAT_UNKNOWN`

Unknown pixel format.

`NVCV_Y`

Luminance (gray).

`NVCV_A`

Alpha (opaque).

`NVCV_YA`

Luminance, alpha.

`NVCV_RGB`

Red, green, blue.

`NVCV_BGR`

Blue, green, red.

`NVCV_RGBA`

Red, green, blue, alpha.

`NVCV_BGRA`

Blue, green, red, alpha.

`NVCV_YUV420`

Luminance and subsampled Chrominance (Y, Cb, Cr).

`NVCV_YUV422`

Luminance and subsampled Chrominance (Y, Cb, Cr).

4.3 Type Definitions

4.3.1 Pixel Organizations

The components of the pixels in an image can be organized in the following ways.

- **Interleaved** pixels (also known as **chunky** pixels) are compact and are arranged so that the components of each pixel in the image are contiguous.

  Planar pixels are arranged so that the individual components, for example, the red components, of all pixels in the image are grouped.

> **Note**: Typically, pixels are interleaved. However, many neural networks perform better with planar pixels.

In the pixel organization descriptions, square brackets ([]) are used to indicate how groups of pixel components are arranged, for example:

- [VYUY] indicates that groups of V, Y, U and Y components are interleaved.
- [Y][U][V] indicates that the individual Y, U, and V components of all pixels are grouped together.
- [Y][UV] indicates that groups of Y components and groups of U and V components are interleaved.

See YUV pixel formats for more information.

The NVIDIA AR SDK API defines the following types to specify the pixel organization:

- `NVCV_INTERLEAVED 0`
- `NVCV_CHUNKY 0`

  Each of these types specifies interleaved, or chunky, pixels in which the components of each pixel in the image are adjacent.

`NVCV_PLANAR 1`

  This type specifies planar pixels, in which the individual components of all pixels in the image are grouped together.

`NVCV_UYVY 2`

  This type specifies UYVY pixels, which are in interleaved YUV 4:2:2 format (default for 4:2:2 and default for non-YUV).

  Pixels are arranged in [UYVY] groups.

`NVCV_VYUY 4`

  This type specifies VYUY pixels, which are in interleaved YUV 4:2:2 format.

  Pixels are arranged in [VYUY] groups.

`NVCV_YUYV 6`

`NVCV_YUY2 NVCV_YUYV`

  Each of these types specifies YUYV pixels, which are in interleaved YUV 4:2:2 format.

  Pixels are arranged in [YUYV] groups.

`NVCV_YVYU 8`

  This type specifies YVYU pixels, which are in interleaved YUV 4:2:2 format.

  Pixels are arranged in [YVYU] groups.

`NVCV_YUV 3`

`NVCV_I420 NVCV_YUV`

`NVCV_IYUV NVCV_YUV`

  Each of these types specifies a planar YUV arrangement: planar YUV 4:2:2 format or planar YUV 4:2:0 format.

  Pixels are arranged in [Y], [U], [V] groups.

`NVCV_YVU` 5

`NVCV_YV12` `NVCV_YVU`

Each of these types specifies YV12 pixels, which are in planar YUV 4:2:2 format or planar YUV 4:2:0 format.

Pixels are arranged in [Y], [V], and [U] groups.

`NVCV_YCUV` 7

`NVCV_NV12` `NVCV_YCUV`

Each of these types specifies NV12 pixels, which are in the semiplanar YUV 4:2:2 format or the semiplanar YUV 4:2:0 format (default for 4:2:0).

Pixels are arranged in [Y] and [UV] groups.

`NVCV_YCVU` 9

`NVCV_NV21` `NVCV_YCVU`

Each of these types specifies NV21 pixels, which are in the semiplanar YUV 4:2:2 format or the semiplanar YUV 4:2:0 format.

Pixels are arranged in [Y] and [VU] groups.

> **Tip**: FlipY is supported only with the planar 4:2:2 formats UYVY, VYUY, YUYV, YVYU and not with other planar or semiplanar formats.

4.3.2 YUV Color Spaces

The NVIDIA AR SDK API defines these types to specify the YUV color spaces:

`NVCV_601` 0

This type specifies the Rec.601 YUV color space, which is typically used for standard definition (SD) video.

`NVCV_709` 1

This type specifies the Rec.709 YUV colorspace, which is typically used for high definition (HD) video.

`NVCV_VIDEO_RANGE` 0

This type specifies the video range [16, 235].

`NVCV_FULL_RANGE` 4

This type specifies the video range [0, 255].

`NVCV_CHROMA_COSITED` 0
`NVCV_CHROMA_MPEG2` `NVCV_CHROMA_COSITED`

Each of these types specifies a color space in which the chroma is sampled at the same location as the luma samples horizontally.

NVCV_CHROMA_INTSTITIAL 8
NVCV_CHROMA_MPEG1 NVCV_CHROMA_INTSTITIAL

Each of these types specifies a color space in which the chroma is sampled between luma samples horizontally.

4.3.3 Memory Types

Image data buffers can be stored in different types of memory. The different types of memory have different address spaces.

NVCV_CPU 0

This buffer is stored in CPU memory.

NVCV_GPU 1

This buffer is stored in CUDA memory.

NVCV_CUDA 1

This buffer is stored in CUDA memory.

4.4 Functions

4.4.1 NvAR_Create

```
NvAR_Result NvAR_Create(
  NvAR_FeatureID featureID,
  NvAR_FeatureHandle *handle
);
```

4.4.1.1 Parameters

featureID [in]

Type: NvAR_FeatureID

The type of feature to be created.

handle[out]

Type: NvAR_FeatureHandle *

A handle to the newly created feature instance.

4.4.1.2 Return Value

Returns one of the following values:

- NVCV_SUCCESS on success
- NVCV_ERR_FEATURENOTFOUND
- NVCV_ERR_INITIALIZATION

4.4.1.3 Remarks

This function creates an instance of the specified feature type and writes a handle to the feature instance to the handle out parameter.

4.4.2 NvAR_Destroy

```
NvAR_Result NvAR_Destroy(
  NvAR_FeatureHandle handle
);
```

4.4.2.1 Parameters

handle [in]

Type: NvAR_FeatureHandle

The handle to the feature instance to be released.

4.4.2.2 Return Value

Returns one of the following values:

- NVCV_SUCCESS on success
- NVCV_ERR_FEATURENOTFOUND

4.4.2.3 Remarks

This function releases the feature instance with the specified handle. Because handles are not reference counted, the handle is invalid after this function is called.

4.4.3 NvAR_Load

```
NvAR_Result NvAR_Load(
  NvAR_FeatureHandle handle,
);
```

4.4.3.1 Parameters

`handle [in]`

Type: `NvAR_FeatureHandle`

The handle to the feature instance to load.

4.4.3.2 Return Value

Returns one of the following values:

- `NVCV_SUCCESS` on success
- `NVCV_ERR_MISSINGINPUT`
- `NVCV_ERR_FEATURENOTFOUND`
- `NVCV_ERR_INITIALIZATION`
- `NVCV_ERR_UNIMPLEMENTED`

4.4.3.3 Remarks

This function loads the specified feature instance and validates any configuration properties that were set for the feature instance.

4.4.4 NvAR_Run

```
NvAR_Result NvAR_Run(
  NvAR_FeatureHandle handle,
);
```

4.4.4.1 Parameters

`handle[in]`

Type: const `NvAR_FeatureHandle`

The handle to the feature instance to be run.

4.4.4.2 Return Value

Returns one of the following values:

- `NVCV_SUCCESS` on success
- `NVCV_ERR_GENERAL`
- `NVCV_ERR_FEATURENOTFOUND`
- `NVCV_ERR_MEMORY`
- `NVCV_ERR_MISSINGINPUT`
- `NVCV_ERR_PARAMETER`

4.4.4.3 Remarks

This function validates the input/output properties that are set by the user, runs the specified feature instance with the input properties that were set for the instance, and writes the results to the output properties set for the instance. The input and output properties are set by the accessor functions. See "Summary of NVIDIA AR SDK Accessor Functions on page 10" for more information.

4.4.5 NvAR_GetCudaStream

```
NvAR_GetCudaStream(
   NvAR_FeatureHandle handle,
   const char *name,
   const CUStream *stream
);
```

4.4.5.1 Parameters

`handle`

Type: `NvAR_FeatureHandle`

The handle to the feature instance from which you want to get the CUDA stream.

`name`

Type: `const char *`

The `NvAR_Parameter_Config(CUDAStream)` key value. Any other key value returns an error.

`stream`

Type: `const CUStream *`

Pointer to the CUDA stream where the CUDA stream retrieved is to be written.

4.4.5.2 Return Value

Returns one of the following values:

- `NVCV_SUCCESS` on success
- `NVCV_ERR_PARAMETER`
- `NVCV_ERR_SELECTOR`
- `NVCV_ERR_MISSINGINPUT`
- `NVCV_ERR_GENERAL`
- `NVCV_ERR_MISMATCH`

4.4.5.3 Remarks

This function gets the CUDA stream in which the specified feature instance will run and writes the CUDA stream to be retrieved to the location that is specified by the parameter `stream`.

4.4.6 NvAR_CudaStreamCreate

```
NvCV_Status NvAR_CudaStreamCreate(
  CUstream *stream
);
```

4.4.6.1 Parameters

`stream [out]`

Type: `CUstream *`

The location in which to store the newly allocated CUDA stream.

4.4.6.2 Return Value

- `NVFVX_SUCCESS` on success
- `NVCV_ERR_CUDA_VALUE` if a CUDA parameter is not within its acceptable range

4.4.6.3 Remarks

This function creates a CUDA stream. It is a wrapper for the CUDA Runtime API function `cudaStreamCreate()` that you can use to avoid linking with the NVIDIA CUDA Toolkit libraries. This function and `cudaStreamCreate()` are equivalent and interchangeable.

4.4.7 NvAR_CudaStreamDestroy

```
void NvAR_CudaStreamDestroy(
  CUstream stream
);
```

4.4.7.1 Parameters

`stream [in]`

Type: `CUstream`

The CUDA stream to destroy.

4.4.7.2 Return Value

Does not return a value.

4.4.7.3 Remarks

This function destroys a CUDA stream. It is a wrapper for the CUDA Runtime API function `cudaStreamDestroy()` that you can use to avoid linking with the NVIDIA CUDA Toolkit libraries. This function and `cudaStreamDestroy()` are equivalent and interchangeable.

4.4.8 NvAR_GetF32

```
NvAR_GetF32(
   NvAR_FeatureHandle handle,
   const char *name,
   float *val
);
```

4.4.8.1 Parameters

`handle`

Type: `NvAR_FeatureHandle`

The handle to the feature instance from which you want to get the specified 32-bit floating-point parameter.

`name`

Type: `const char *`

The key value that is used to access the 32-bit float parameters as defined in `nvAR_defs.h` and in “Key Values in the Properties of a Feature Type” on page 12.

`val`

Type: `float*`

Pointer to the 32-bit floating-point number where the value retrieved is to be written.

4.4.8.2 Return Value

Returns one of the following values:

- `NVCV_SUCCESS` on success
- `NVCV_ERR_PARAMETER`
- `NVCV_ERR_SELECTOR`
- `NVCV_ERR_GENERAL`
- `NVCV_ERR_MISMATCH`

4.4.8.3 Remarks

This function gets the value of the specified single-precision (32-bit) floating-point parameter for the specified feature instance and writes the value to be retrieved to the location that is specified by the `val` parameter.

4.4.9 NvAR_GetF64

```
NvAR_GetF64(
   NvAR_FeatureHandle handle,
   const char *name,
   double *val
);
```

4.4.9.1 Parameters

`handle`

Type: `NvAR_FeatureHandle`

The handle to the feature instance from which you want to get the specified 64-bit floating-point parameter.

`name`

Type: `const char *`

The key value used to access the 64-bit double parameters as defined in `nvAR_defs.h` and in "Key Values in the Properties of a Feature Type" on page 12.

`val`

Type: `double*`

Pointer to the 64-bit double-precision floating-point number where the retrieved value will be written.

4.4.9.2 Return Value

Returns one of the following values:

- `NVCV_SUCCESS` on success
- `NVCV_ERR_PARAMETER`
- `NVCV_ERR_SELECTOR`
- `NVCV_ERR_GENERAL`
- `NVCV_ERR_MISMATCH`

4.4.9.3 Remarks

This function gets the value of the specified double-precision (64-bit) floating-point parameter for the specified feature instance and writes the retrieved value to the location that is specified by the `val` parameter.

4.4.10 NvAR_GetF32Array

```
NvAR_GetFloatArray (
   NvAR_FeatureHandle handle,
   const char *name,
   const float** vals,
   int *count
);
```

4.4.10.1 Parameters

`handle`

Type: `NvAR_FeatureHandle`

The handle to the feature instance from which you want to get the specified float array.

`name`

Type: `const char *`

See in “Key Values in the Properties of a Feature Type on page 12 for a complete list of key values.

`vals`

Type: `const float**`

Pointer to an array of floating-point numbers where the retrieved values will be written.

`count`

Type: `int *`

Currently unused. The number of elements in the array that is specified by the `vals` parameter.

4.4.10.2 Return Value

Returns one of the following values:

- NVCV_SUCCESS on success
- NVCV_ERR_PARAMETER
- NVCV_ERR_SELECTOR
- NVCV_ERR_MISSINGINPUT
- NVCV_ERR_GENERAL
- NVCV_ERR_MISMATCH

4.4.10.3 Remarks

This function gets the values in the specified floating-point-array for the specified feature instance and writes the retrieved values to an array at the location that is specified by the vals parameter.

4.4.11 NvAR_GetObject

```
NvAR_GetObject(
   NvAR_FeatureHandle handle,
   const char *name,
   const void **ptr,
   unsigned long typeSize
);
```

4.4.11.1 Parameters

handle

Type: NvAR_FeatureHandle

The handle to the feature instance from which you can get the specified object.

name

Type: const char *

See in "Key Values in the Properties of a Feature Type on page 12 for a complete list of key values.

ptr

Type: const void**

A pointer to the memory that is allocated for the objects defined in "Structures on page 33".

`typeSize`

Type: `unsigned long`

- The size of the item to which the pointer points. If the size does not match, an `NVCV_ERR_MISMATCH` is returned.

4.4.11.2 Return Value

Returns one of the following values:

- `NVCV_SUCCESS` on success
- `NVCV_ERR_PARAMETER`
- `NVCV_ERR_SELECTOR`
- `NVCV_ERR_MISSINGINPUT`
- `NVCV_ERR_GENERAL`
- `NVCV_ERR_MISMATCH`

4.4.11.3 Remarks

This function gets the specified object for the specified feature instance and stores the object in the memory location that is specified by the `ptr` parameter.

4.4.12 NvAR_GetS32

```
NvAR_GetS32(
   NvAR_FeatureHandle handle,
   const char *name,
   int *val
);
```

4.4.12.1 Parameters

`handle`

Type: `NvAR_FeatureHandle`

The handle to the feature instance from which you get the specified 32-bit signed integer parameter.

`name`

Type: `const char *`

The key value that is used to access the signed integer parameters as defined in `nvAR_defs.h` and in "Key Values in the Properties of a Feature Type on page 12.

val

Type: int*

Pointer to the 32-bit signed integer where the retrieved value will be written.

4.4.12.2 Return Value

Returns one of the following values:

- NVCV_SUCCESS on success
- NVCV_ERR_PARAMETER
- NVCV_ERR_SELECTOR
- NVCV_ERR_GENERAL
- NVCV_ERR_MISMATCH

4.4.12.3 Remarks

This function gets the value of the specified 32-bit signed integer parameter for the specified feature instance and writes the retrieved value to the location that is specified by the val parameter.

4.4.13 NvAR_GetString

```
NvAR_GetString(
   NvAR_FeatureHandle handle,
   const char *name,
   const char** str
);
```

4.4.13.1 Parameters

handle

Type: NvAR_FeatureHandle

The handle to the feature instance from which you get the specified character string parameter.

name

Type: const char *

See in “Key Values in the Properties of a Feature Type on page 12 for a complete list of key values.

str

Type: const char**

The address where the requested character string pointer is stored.

4.4.13.2 Return Value

Returns one of the following values:

- NVCV_SUCCESS on success
- NVCV_ERR_PARAMETER
- NVCV_ERR_SELECTOR
- NVCV_ERR_MISSINGINPUT
- NVCV_ERR_GENERAL
- NVCV_ERR_MISMATCH

4.4.13.3 Remarks

This function gets the value of the specified character string parameter for the specified feature instance and writes the retrieved string to the location that is specified by the str parameter.

4.4.14 NvAR_GetU32

```
NvAR_GetU32(
   NvAR_FeatureHandle handle,
   const char *name,
   unsigned int* val
);
```

4.4.14.1 Parameters

handle

Type: NvAR_FeatureHandle

The handle to the feature instance from which you want to get the specified 32-bit unsigned integer parameter.

name

Type: const char *

The key value that is used to access the unsigned integer parameters as defined in nvAR_defs.h and in "Key Values in the Properties of a Feature Type on page 12.

val

Type: unsigned int*

Pointer to the 32-bit unsigned integer where the retrieved value will be written.

4.4.14.2 Return Value

Returns one of the following values:

- NVCV_SUCCESS on success
- NVCV_ERR_PARAMETER
- NVCV_ERR_SELECTOR
- NVCV_ERR_GENERAL
- NVCV_ERR_MISMATCH

4.4.14.3 Remarks

This function gets the value of the specified 32-bit unsigned integer parameter for the specified feature instance and writes the retrieved value to the location that is specified by the `val` parameter.

4.4.15 NvAR_GetU64

```
NvAR_GetU64(
   NvAR_FeatureHandle handle,
   const char *name,
   unsigned long long *val
);
```

4.4.15.1 Parameters

`handle`

Type: `NvAR_FeatureHandle`

The handle to the returned feature instance from which you get the specified 64-bit unsigned integer parameter.

`name`

Type: `const char *`

The key value used to access the unsigned 64-bit integer parameters as defined in `nvAR_defs.h` and in “Key Values in the Properties of a Feature Type on page 12.

`val`

Type: `unsigned long long*`

Pointer to the 64-bit unsigned integer where the retrieved value will be written.

4.4.15.2 Function Return Values

Returns one of the following values:

- NVCV_SUCCESS on success
- NVCV_ERR_PARAMETER
- NVCV_ERR_SELECTOR
- NVCV_ERR_GENERAL
- NVCV_ERR_MISMATCH

4.4.15.3 Remarks

This function gets the value of the specified 64-bit unsigned integer parameter for the specified feature instance and writes the retrieved value to the location specified by the val parameter.

4.4.16 NvAR_SetCudaStream

```
NvAR_SetCudaStream(
   NvAR_FeatureHandle handle,
   const char *name,
   CUStream stream
);
```

4.4.16.1 Parameters

handle

Type: NvAR_FeatureHandle

The handle to the feature instance that is returned for which you want to set the CUDA stream.

name

Type: const char *

The NvAR_Parameter_Config(CUDAStream) key value. Any other key value returns an error.

stream

Type: CUStream

The CUDA stream in which to run the feature instance on the GPU.

4.4.16.2 Return Value

Returns one of the following values:

- NVCV_SUCCESS on success
- NVCV_ERR_PARAMETER
- NVCV_ERR_SELECTOR
- NVCV_ERR_GENERAL
- NVCV_ERR_MISMATCH

4.4.16.3 Remarks

This function sets the CUDA stream, in which the specified feature instance will run, to the parameter stream.

Defined in: nvAR.h.

4.4.17 NvAR_SetF32

```
NvAR_SetF32(
   NvAR_FeatureHandle handle,
   const char *name,
   float val
);
```

4.4.17.1 Parameters

handle

Type: NvAR_FeatureHandle

The handle to the feature instance for which you want to set the specified 32-bit floating-point parameter.

name

Type: const char *

The key value used to access the 32-bit float parameters as defined in nvAR_defs.h and in "Key Values in the Properties of a Feature Type on page 12.

val

Type: float

The 32-bit floating-point number to which the parameter is to be set.

4.4.17.2 Return Value

Returns one of the following values:

- NVCV_SUCCESS on success
- NVCV_ERR_PARAMETER
- NVCV_ERR_SELECTOR
- NVCV_ERR_GENERAL
- NVCV_ERR_MISMATCH

4.4.17.3 Remarks

This function sets the specified single-precision (32-bit) floating-point parameter for the specified feature instance to the `val` parameter.

4.4.18 NvAR_SetF64

```
NvAR_SetF64(
   NvAR_FeatureHandle handle,
   const char *name,
   double val
);
```

4.4.18.1 Parameters

`handle`

Type: `NvAR_FeatureHandle`

The handle to the feature instance for which you want to set the specified 64-bit floating-point parameter.

`name`

Type: `const char *`

The key value used to access the 64-bit float parameters as defined in `nvAR_defs.h` and in "Key Values in the Properties of a Feature Type on page 12.

`val`

Type: `double`

The 64-bit double-precision floating-point number to which the parameter will be set.

4.4.18.2 Return Value

Returns one of the following values:

- NVCV_SUCCESS on success
- NVCV_ERR_PARAMETER
- NVCV_ERR_SELECTOR
- NVCV_ERR_GENERAL
- NVCV_ERR_MISMATCH

4.4.18.3 Remarks

This function sets the specified double-precision (64-bit) floating-point parameter for the specified feature instance to the val parameter.

4.4.19 NvAR_SetF32Array

```
NvAR_SetFloatArray(
   NvAR_FeatureHandle handle,
   const char *name,
   float* vals,
   int count
);
```

4.4.19.1 Parameters

handle

Type: NvAR_FeatureHandle

The handle to the feature instance for which you want to set the specified float array.

name

Type: const char *

See in "Key Values in the Properties of a Feature Type on page 12 for a complete list of key values.

vals

Type: float*

An array of floating-point numbers to which the parameter will be set.

count

Type: int

Currently unused. The number of elements in the array that is specified by the vals parameter.

4.4.19.2 Return Value

Returns one of the following values:

- NVCV_SUCCESS on success
- NVCV_ERR_PARAMETER
- NVCV_ERR_SELECTOR
- NVCV_ERR_GENERAL
- NVCV_ERR_MISMATCH

4.4.19.3 Remarks

This function assigns the array of floating-point numbers that are defined by the vals parameter to the specified floating-point-array parameter for the specified feature instance.

4.4.20 NvAR_SetObject

```
NvAR_SetObject(
   NvAR_FeatureHandle handle,
   const char *name,
   void *ptr,
   unsigned long typeSize
);
```

4.4.20.1 Parameters

handle

Type: NvAR_FeatureHandle

The handle to the feature instance for which you want to set the specified object.

name

Type: const char *

See in "Key Values in the Properties of a Feature Type on page 12 for a complete list of key values.

ptr

Type: void*

A pointer to memory that was allocated to the objects that were defined in "Structures on page 33".

`typeSize`

Type: `unsigned long`

The size of the item to which the pointer points. If the size doesn't match, an `NVCV_ERR_MISMATCH` is returned.

4.4.20.2 Return Value

Returns one of the following values:

- `NVCV_SUCCESS` on success
- `NVCV_ERR_PARAMETER`
- `NVCV_ERR_SELECTOR`
- `NVCV_ERR_GENERAL`
- `NVCV_ERR_MISMATCH`

4.4.20.3 Remarks

This function assigns the memory of the object that was specified by the `ptr` parameter to the specified object parameter for the specified feature instance.

4.4.21 NvAR_SetS32

```
NvAR_SetS32(
   NvAR_FeatureHandle handle,
   const char *name,
   int val
);
```

4.4.21.1 Parameters

`handle`

Type: `NvAR_FeatureHandle`

The handle to the feature instance for which you want to set the specified 32-bit signed integer parameter.

`name`

Type: `const char *`

The key value used to access the signed 32-bit integer parameters as defined in `nvAR_defs.h` and in "Key Values in the Properties of a Feature Type on page 12.

`val`

Type: `int`

The 32-bit signed integer to which the parameter will be set.

4.4.21.2 Return Value

Returns one of the following values:

- `NVCV_SUCCESS` on success
- `NVCV_ERR_PARAMETER`
- `NVCV_ERR_SELECTOR`
- `NVCV_ERR_GENERAL`
- `NVCV_ERR_MISMATCH`

4.4.21.3 Remarks

This function sets the specified 32-bit signed integer parameter for the specified feature instance to the `val` parameter.

4.4.22 NvAR_SetString

```
NvAR_SetString(
   NvAR_FeatureHandle handle,
   const char *name,
   const char* str
);
```

4.4.22.1 Parameters

`handle`

Type: `NvAR_FeatureHandle`

The handle to the feature instance for which you want to set the specified character string parameter.

`name`

Type: `const char *`

See "Key Values in the Properties of a Feature Type on page 12 for a complete list of key values.

`str`

Type: `const char*`

Pointer to the character string to which you want to set the parameter.

4.4.22.2 Return Value

Returns one of the following values:

- NVCV_SUCCESS on success
- NVCV_ERR_PARAMETER
- NVCV_ERR_SELECTOR
- NVCV_ERR_GENERAL
- NVCV_ERR_MISMATCH

4.4.22.3 Remarks

This function sets the value of the specified character string parameter for the specified feature instance to the str parameter.

4.4.23 NvAR_SetU32

```
NvAR_SetU32(
   NvAR_FeatureHandle handle,
   const char *name,
   unsigned int val
);
```

4.4.23.1 Function Parameters

handle

Type: NvAR_FeatureHandle

The handle to the feature instance for which you want to set the specified 32-bit unsigned integer parameter.

name

Type: const char *

The key value used to access the unsigned 32-bit integer parameters as defined in nvAR_defs.h and in “Summary of NVIDIA AR SDK Accessor Functions on page 10.

val

Type: unsigned int

The 32-bit unsigned integer to which you want to set the parameter.

4.4.23.2 Function Return Values

Returns one of the following values:

- NVCV_SUCCESS on success
- NVCV_ERR_PARAMETER
- NVCV_ERR_SELECTOR
- NVCV_ERR_GENERAL
- NVCV_ERR_MISMATCH

4.4.23.3 Remarks

This function sets the value of the specified 32-bit unsigned integer parameter for the specified feature instance to the val parameter.

4.4.24 NvAR_SetU64

```
NvAR_SetU64(
   NvAR_FeatureHandle handle,
   const char *name,
   unsigned long long val
);
```

4.4.24.1 Parameters

handle

Type: NvAR_FeatureHandle

The handle to the feature instance for which you want to set the specified 64-bit unsigned integer parameter.

name

Type: const char *

The key value used to access the unsigned 64-bit integer parameters as defined in nvAR_defs.h andin "Key Values in the Properties of a Feature Type on page 12. .

val

Type: unsigned long long

The 64-bit unsigned integer to which you want to set the parameter.

4.4.24.2 Return Value

Returns one of the following values:

- `NVCV_SUCCESS` on success
- `NVCV_ERR_PARAMETER`
- `NVCV_ERR_SELECTOR`
- `NVCV_ERR_GENERAL`
- `NVCV_ERR_MISMATCH`

4.4.24.3 Remarks

This function sets the value of the specified 64-bit unsigned integer parameter for the specified feature instance to the `val` parameter .

`VCV_CHUNKY`

Each of these types specifies interleaved, or chunky, pixels in which the components of each pixel in the image are adjacent.

`NVCV_PLANAR 1`

This type specifies planar pixels, in which the individual components of all pixels in the image are grouped together.

`NVCV_UYVY 2`

This type specifies UYVY pixels, which are in interleaved YUV 4:2:2 format (default for 4:2:2 and default for non-YUV).

Pixels are arranged in [UYVY] groups.

`NVCV_VYUY 4`

This type specifies VYUY pixels, which are in interleaved YUV 4:2:2 format.

Pixels are arranged in [VYUY] groups.

`NVCV_YUYV 6`

`NVCV_YUY2 NVCV_YUYV`

Each of these types specifies YUYV pixels, which are in interleaved YUV 4:2:2 format.

Pixels are arranged in [YUYV] groups.

`NVCV_YVYU 8`

This type specifies YVYU pixels, which are in interleaved YUV 4:2:2 format.

Pixels are arranged in [YVYU] groups.

`NVCV_YUV 3`

`NVCV_I420 NVCV_YUV`

`NVCV_IYUV NVCV_YUV`

Each of these types specifies a planar YUV arrangement: planar YUV 4:2:2 format or planar YUV 4:2:0 format.

Pixels are arranged in [Y], [U], [V] groups.

`NVCV_YVU` 5

`NVCV_YV12 NVCV_YVU`

Each of these types specifies YV12 pixels, which are in planar YUV 4:2:2 format or planar YUV 4:2:0 format.

Pixels are arranged in [Y], [V], and [U] groups.

`NVCV_YCUV` 7

`NVCV_NV12 NVCV_YCUV`

Each of these types specifies NV12 pixels, which are in the semiplanar YUV 4:2:2 format or the semiplanar YUV 4:2:0 format (default for 4:2:0).

Pixels are arranged in [Y] and [UV] groups.

`NVCV_YCVU` 9

`NVCV_NV21 NVCV_YCVU`

Each of these types specifies NV21 pixels, which are in the semiplanar YUV 4:2:2 format or the semiplanar YUV 4:2:0 format.

Pixels are arranged in [Y] and [VU] groups.

> **Note**: FlipY is supported only with the planar 4:2:2 formats (UYVY, VYUY, YUYV, YVYU) and not with other planar or semiplanar formats.

4.5 Image Functions for C and C++

The image functions are defined in the `nvCVImage.h` header file. The image API is object oriented but is accessible in C and C++.

4.5.1 CVWrapperForNvCVImage

```
void CVWrapperForNvCVImage(
  const NvCVImage *vfxIm,
  cv::Mat *cvIm
);
```

4.5.1.1 Parameters

`vfxIm [in]`

Type: `const NvCVImage *`

Pointer to an allocated `NvCVImage` object.

`cvIm [out]`

Type: `cv::Mat *`

Pointer to an empty OpenCV image that has been appropriately initialized to access the buffer of the `NvCVImage` object. An empty OpenCV image is created by the default `cv::Mat` constructor.

4.5.1.2 Return Value

Does not return a value.

4.5.1.3 Remarks

This function creates an OpenCV image wrapper for an `NvCVImage` object.

4.5.2 NvCVImage_Alloc

```
NvCV_Status NvCVImage_Alloc(
  NvCVImage *im
  unsigned width,
  unsigned height,
  NvCVImage_PixelFormat format,
  NvCVImage_ComponentType type,
  unsigned isPlanar,
  unsigned onGPU,
  unsigned alignment
);
```

4.5.2.1 Parameters

`im [in,out]`

Type: `NvCVImage *`

The image to initialize.

`width [in]`

Type: `unsigned`

The width in pixels of the image.

`height [in]`

Type: `unsigned`

The height in pixels of the image.

`format [in]`

Type: `NvCVImage_PixelFormat`

The format of the pixels.

`type [in]`

Type: `NvCVImage_ComponentType`

The type of the components of the pixels.

`isPlanar [in]`

Type: `unsigned`

The organization of the components of the pixels in the image. See, "Pixel Organizations, on page 44" for more information.

`onGPU [in]`

Type: `unsigned`

The type of memory in which the image data buffers are to be stored. See, "Memory Types," on page 69 for more information.

`alignment [in]`

Type: `unsigned`

The row byte alignment, which specifies the alignment of the first pixel in each scan line. Set this parameter to 0 or a power of 2.

- 1: Specifies no gap between scan lines.
  A byte alignment of 1 is required by all GPU buffers used by the video effect filters.
- 0: Specifies the default alignment, which depends on the type of memory in which the image data buffers are stored:
  - CPU memory: Specifies an alignment of 4 bytes,
  - GPU memory: Specifies the alignment set by `cudaMallocPitch`.

2 or greater: Specifies any other alignment, such as a cache line size of 16 or 32 bytes.

> **Note**: If the product of the `width` and the `pixelBytes` member of `NvCVImage` is a whole-number multiple of `alignment`, the gap between scan lines is 0 bytes, regardless of the `alignment` value.

4.5.2.2 Return Value

- `NVFVX_SUCCESS` on success
- `NVCV_ERR_PIXELFORMAT`, if the pixel format is not supported.
- `NVCV_ERR_MEMORY`, if the buffer requires more memory than is available.

4.5.2.3 Remarks

This function does the following:

- Allocates the memory for, and initializes, an image.
- Assumes that the image data structure has nothing meaningful in it.

This is function is called by the C++ `NvCVImage` constructors, and you can also call this function from C code to allocate memory for (and initialize) an empty image.

4.5.3 NvCVImage_ComponentOffsets

```
void NvCVImage_ComponentOffsets(
  NvCVImage_PixelFormat format,
  int *rOff,
  int *gOff,
  int *bOff,
  int *aOff,
  int *yOff
);
```

4.5.3.1 Parameters

`format [in]`

Type: `NvCVImage_PixelFormat`

The pixel format whose component offsets are to be retrieved.

`rOff [out]`

Type: `int *`

The location where you can store the offset for the red channel (can be `NULL`).

`gOff [out]`

Type: `int *`

The location where you can store the offset for the green channel (can be `NULL`).

`bOff [out]`

Type: `int *`

The location where you can store the offset for the blue channel (can be `NULL`).

`aOff [out]`

Type: `int *`

The location where you can store store the offset for the alpha channel (can be `NULL`).

`yOff [out]`

Type: `int *`

The location where you can store the offset for the luminance channel (can be `NULL`).

4.5.3.2 Return Values

Does not return a value.

4.5.3.3 Remarks

This function gets offsets for the components of a pixel format. These offsets are not byte offsets but are component offsets. For interleaved pixels, to obtain the byte offset, a component offset must be multiplied by the `componentBytes` member of `NvCVImage`.

4.5.4 NvCVImage_Composite

```
NvCV_Status NvCVImage_Composite(
  const NvCVImage *src,
  const NvCVImage *mat,
  NvCVImage *dst
);
```

4.5.4.1 Parameters

`src [in]`

Type: const `NvCVImage *`

The source BGRu8 or RGBu8 image.

`mat [in]`

Type: `const NvCVImage *`

The matte Yu8 or Au8 image, indicating where the source image should come through.

`dst [out]`

Type: `NvCVImage *`

The destination BGRu8 or RGBu8 image.

4.5.4.2 Return Value

- `NVFVX_SUCCESS` on success.
- `NVCV_ERR_PIXELFORMAT`, if the pixel format is not supported.

4.5.4.3 Remarks

This function uses the specified matte image to composite a BGRu8 or RGNU8 image over another image.

4.5.5 NvCVImage_CompositeOverConstant

```
NvCV_Status NvCVImage_CompositeOverConstant(
  const NvCVImage *src,
  const NvCVImage *mat,
  const unsigned char bgColor[3],
  NvCVImage *dst
);
```

4.5.5.1 Parameters

`src [in]`

Type: `const NvCVImage *`

The source BGRu8 or RGBu8 image.

`mat [in]`

Type: `const NvCVImage *`

The matte Yu8 or Au8 image, which indicates where the source image should come through.

`[in] bgColor`

Type: `const unsigned char`

A three-element array of characters that define the color field over which the source image is to be composited. This color field must have the same component ordering as the source and destination images.

`dst [out]`

Type: `NvCVImage *`

The destination BGRu8 or RGBu8 image. The destination image might be the same image as the source image.

4.5.5.2 Return Value

- `NVFVX_SUCCESS` on success
- `NVCV_ERR_PIXELFORMAT` if the pixel format is not supported

4.5.5.3 Remarks

This function uses the specified matte image to composite a BGRu8 or RGNU8 image over a constant color field.

4.5.6 NvCVImage_Create

```
NvCV_Status NvCVImage_Create(
  unsigned width,
  unsigned height,
  NvCVImage_PixelFormat format,
  NvCVImage_ComponentType type,
  unsigned isPlanar,
  unsigned onGPU,
  unsigned alignment,
  NvCVImage **out
);
```

4.5.6.1 Parameters

`width [in]`

Type: `unsigned`

The width of the image in pixels.

`height [in]`

Type: `unsigned`

The height of the image in pixels.

`format [in]`

Type: `NvCVImage_PixelFormat`

The format of the pixels.

`type [in]`

Type: `NvCVImage_ComponentType`

The type of the components of the pixels.

`isPlanar [in]`

Type: `unsigned`

The organization of the components of the pixels in the image. See "Pixel Organizations" on page 44 for more information.

`onGPU [in]`

Type: `unsigned`

The type of memory in which the image data buffers are to be stored. See "Memory Types" on page 69 for more information.

`alignment [in]`

Type: `unsigned`

The row byte alignment, which specifies the alignment of the first pixel in each scan line. Set this parameter to 0 or a power of 2.

- 1: Specifies no gap between scan lines.
  A byte alignment of 1 is required by all GPU buffers that are used by the video effect filters.
- 0: Specifies the default alignment, which depends on the type of memory in which the image data buffers are stored:
  - CPU memory: Specifies an alignment of 4 bytes,
  - GPU memory: Specifies the alignment set by `cudaMallocPitch`.
- 2 or greater: Specifies any other alignment, such as a cache line size of 16 or 32 bytes.

> **Note**: If the product of `width` and the `pixelBytes` member of `NvCVImage` is a whole-number multiple of `alignment`, the gap between scan lines is 0 bytes, regardless of the `alignment` value.

`out [out]`

Type: `NvCVImage **`

Pointer to the location where the newly allocated image will be stored. The image descriptor and the pixel buffer are stored so that they are deallocated when `NvCVImage_Destroy()` is called.

4.5.6.2 Return Value

- `NVFVX_SUCCESS` on success.
- `NVCV_ERR_PIXELFORMAT`, if the pixel format is not supported.
- `NVCV_ERR_MEMORY`, if the buffer requires more memory than is available.

4.5.6.3 Remarks

This function creates an image and allocates an image buffer that will be provided as input to an effect filter and allocates storage for the new image. This function is a C-style constructor for an instance of the `NvCVImage` structure (equivalent to `new NvCVImage` in C++).

4.5.7 NvCVImage_Dealloc

```
void NvCVImage_Dealloc(
  NvCVImage *im
);
```

4.5.7.1 Parameters

`im [in,out]`

Type: `NvCVImage *`

Pointer to the image whose image buffer is to be freed.

4.5.7.2 Return Value

Does not return.

4.5.7.3 Remarks

This function frees the image buffer from the specified `NvCVImage` structure and sets the contents of the `NvCVImage` structure to 0.

4.5.8 NvCVImage_Destroy

```
void NvCVImage_Destroy(
  NvCVImage *im
);
```

4.5.8.1 Parameters

`im`

Type: `NvCVImage *`

Pointer to the image that is to be destroyed.

4.5.8.2 Return Value

Does not return a value.

4.5.8.3 Remarks

This function destroys an image that was created with the `NvCVImage_Create()` function and frees the allocated resources and memory. This function is a C-style destructor for an instance of the `NvCVImage` structure (equivalent to `delete im` in C++).

4.5.9 NvCVImage_Init

```
NvCV_Status NvCVImage_Init(
  NvCVImage *im,
  unsigned width,
  unsigned height,
  unsigned pitch,
  void *pixels,
  NvCVImage_PixelFormat format,
  NvCVImage_ComponentType type,
  unsigned isPlanar,
  unsigned onGPU
);
```

4.5.9.1 Parameters

`im [in,out]`

Type: `NvCVImage *`

Pointer to the image that will be initialized.

`width [in]`

Type: `unsigned`

The width of the image in pixels.

`height [in]`

Type: `unsigned`

The height of the image in pixels.

`pitch [in]`

Type: `unsigned`

The vertical byte stride between pixels.

`pixels [in]`

Type: `void`

Pointer to the pixel buffer that will be attached to the `NvCVImage` object.

`format`

Type: `NvCVImage_PixelFormat`

The format of the pixels in the image.

`type`

Type: `NvCVImage_ComponentType`

The data type used to represent each component of the image.

`isPlanar [in]`

Type: `unsigned`

The organization of the components of the pixels in the image. See "Pixel Organizations" on page 44.

`onGPU [in]`

Type: `unsigned`

The type of memory in which the image data buffers will be stored. See "Memory Types" on page 47.

4.5.9.2 Return Value

- `NVFVX_SUCCESS` on success.
- `NVCV_ERR_PIXELFORMAT`, if the pixel format is not supported.

4.5.9.3 Remarks

This function initializes an `NvCVImage` structure from a raw buffer pointer, which is useful for wrapping an existing pixel buffer in an `NvCVImage` image descriptor.

This function is called by functions that initialize an `NvCVImage` object's data structure, for example:

- C++ constructors
- `NvCVImage_Alloc()`
- `NvCVImage_Realloc()`
- `NvCVImage_InitView()`

Call this function to initialize an `NvCVImage` object instead of setting the fields directly.

4.5.10 NvCVImage_InitView

```
void NvCVImage_InitView(
  NvCVImage *subImg,
  NvCVImage *fullImg,
  int x,
  int y,
  unsigned width,
  unsigned height
);
```

4.5.10.1 Parameters

`subImg [in]`

Type: `NvCVImage *`

Pointer to the existing image that will be initialized with the view.

`fullImg [in]`

Type: `NvCVImage *`

Pointer to the existing image from which the view of a specified rectangle in the image will be taken.

`x [in]`

Type: `int`

The x coordinate of the left edge of the view to be taken.

`y [in]`

Type: `int`

The y coordinate of the top edge of the view to be taken.

`width [in]`

Type: `unsigned`

The width of the view to be taken, in pixels.

`height [in]`

Type: `unsigned`

The height of the view to be taken, in pixels.

4.5.10.2 Return Value

Does not return a value.

4.5.10.3 Remarks

This function takes a view of the specified rectangle in an image and initializes another existing image descriptor with the view. No memory is allocated because the buffer of the image that is being initialized with the view (specified by the parameter `fullImg`) is used instead.

4.5.11 NvCVImage_Realloc

```
NvCV_Status NvCVImage_Realloc(
  NvCVImage *im,
  unsigned width,
  unsigned height,
  NvCVImage_PixelFormat format,
  NvCVImage_ComponentType type,
  unsigned isPlanar,
  unsigned onGPU,
  unsigned alignment
);
```

4.5.11.1 Parameters

`im [in,out]`

Type: `NvCVImage *`

The image to initialize.

`width [in]`

Type: `unsigned`

The width of the image in pixels.

`height [in]`

The height of the image in pixels.

`format [in]`

Type: `NvCVImage_PixelFormat`

The format of the pixels.

`type [in]`

Type: `NvCVImage_ComponentType`

The type of the components of the pixels.

`isPlanar [in]`

Type: `unsigned`

The organization of the components of the pixels in the image. See "Pixel Organizations" on page 44.

`[in] onGPU`

Type: `unsigned`

The type of memory in which the image data buffers are to be stored. See "Memory Types" on page 47.

`alignment [in]`

Type: `unsigned`

The row byte alignment, which specifies the alignment of the first pixel in each scan line. Set this parameter to 0 or a power of 2.

- 1: Specifies no gap between scan lines.
  A byte alignment of 1 is required by all GPU buffers that are used by the video effect filters.
- 0: Specifies the default alignment, which depends on the type of memory in which the image data buffers are stored:
  - CPU memory: Specifies an alignment of 4 bytes,
  - GPU memory: Specifies the alignment set by `cudaMallocPitch`.
- 2 or greater: Specifies any other alignment, such as a cache line size of 16 or 32 bytes.

Note: If the product of `width` and the `pixelBytes` member of `NvCVImage` is a whole-number multiple of `alignment`, the gap between scan lines is 0 bytes, regardless of the `alignment` value.

4.5.11.2 Return Value

- `NVFVX_SUCCESS` on success.
- `NVCV_ERR_PIXELFORMAT` if the pixel format is not supported.
- `NVCV_ERR_MEMORY` if the buffer requires more memory than is available.

4.5.11.3 Remarks

This function reallocates memory for, and initializes, an image and assumes that the image is valid.

The function checks the `bufferBytes` member of `NvCVImage` to determine whether enough memory is already available:

- If sufficient memory is already available, the function reshapes, instead of reallocating, the memory.
- If sufficient memory is not available, the function frees the memory for the existing buffer and allocates memory for a new buffer.

4.5.12 NvCVImage_Transfer

```
NvCV_Status NvCVImage_Transfer(
  const NvCVImage *src,
  NvCVImage *dst,
  float scale,
  CUstream stream,
  NvCVImage *tmp
);
```

4.5.12.1 Parameters

`src [in]`

Type: `const NvCVImage *`

Pointer to the source image that will be transferred.

`dst [out]`

Type: `NvCVImage *`

Pointer to the destination image to which the source image will be transferred.

scale [in]

Type: float

A scale factor that can be applied if the component type of the source or destination image is a floating-point. The scale has an effect only when the component type of the source or destination image is floating-point.

Here are the typical values:

- 1.0f
- 255.0f
- 1.0f/255.0f

If the component type of all images is the same (all integer or all floating-point), this parameter is ignored.

stream [in]

Type: CUstream

The CUDA stream on which to transfer the image. If the memory type of both the source and destination images is CPU, this parameter is ignored.

tmp [in,out]

Type: NvCVImage *

Pointer to a temporary buffer in the GPU memory that is required only when the source image is being converted **and** when the memory types of the source and destination images are different. The buffer has the same characteristics as the CPU image, but buffer resides on the GPU.

If necessary, the temporary GPU buffer is reshaped to suit the needs of the transfer, for example, to match the characteristics of the CPU image. Therefore, for best performance, you can supply an empty image as the temporary GPU buffer. If necessary, NvCVImage_Transfer() allocates an appropriately sized buffer. The same temporary GPU buffer can be used in subsequent calls to NvCVImage_Transfer(), regardless of the shape, format, or component type, because the buffer will grow to accommodate the largest memory requirement.

If a temporary GPU buffer is not needed, no buffer is allocated. If a temporary GPU buffer is not required, tmp might be NULL. However, if tmp is NULL and a temporary GPU buffer is required, an ephemeral buffer is allocated, with a resultant performance degradation for image sequences.

4.5.12.2 Return Value

- NVFVX_SUCCESS on success.
- NVCV_ERR_CUDA, if a CUDA error occurs.
- NVCV_ERR_PIXELFORMAT, if the pixel format of the source or destination image is not supported.
- NVCV_ERR_GENERAL, if an unspecified error occurs.

4.5.12.3 Remarks

This function transfers one image to another image, performs some conversions on the image, and uses the GPU to perform the conversions.

The function supports the following conversions:

- Between pixel formats:
  - From RGBu8 to RGBu8, where the RGB components are in any order.
  - Between RGBu8 and RGBf32 in either direction, where the RGB components are in any order, for example, BGR.
  - From RGBu8 to GRAYf32, where the RGB components are in any order.
  - From RGBf32 to RGBAu8 by setting A to 255, where the RGB and RGBA components are in any order.
  - From RGBAu8 to RGBu8 by removing the alpha component.
  - From GRAYf32 to GRAYu8.
  - From GRAYu8 to GRAYu8, where the RGB components are in any order (also works with ALPHAu8).
  - From ALPHAu8 to RGBAu8 (insertion) without touching the RGB components.
  - From GRAYu8 to RGBu8 by replicating gray into all RGB components.
- Between chunky and planar pixel organizations in either direction.
- Between CPU and GPU memory types in either direction.

If both images reside on the CPU, the transfer occurs synchronously. However, if either image resides on the GPU, the transfer might occur asynchronously. A chain of asynchronous calls on the same CUDA stream is automatically sequenced as expected, but if synchronize needs to occur, the `cudaStreamSynchronize()` function can be called.

If the format of the source and destination images are the same, all formats are accommodated on the CPU and the GPU. This function can now be used as a replacement for the `cudaMemcpy2DAsync()` function.

4.5.13 NVWrapperForCVMat

```
void NVWrapperForCVMat(
  const cv::Mat *cvIm,
  NvCVImage *vIm
);
```

4.5.13.1 Parameters

`cvIm [in]`

Type: `const cv::Mat *`

Pointer to an allocated OpenCV image.

vfxIm [out]

Type: NvCVImage *

Pointer to an empty NvCVImage object that has been appropriately initialized by this function to access the buffer of the OpenCV image. An empty NvCVImage object is created by the default (no-argument) NvCVImage() constructor.

4.5.13.2 Return Value

Does not return a value.

4.5.13.3 Remarks

This function creates an NvCVImage object wrapper for an OpenCV image.

4.6 Image Functions for C++ Only

The image API provides constructors and a destructor for C++ and some additional functions that are accessible only to C++.

4.6.1 NvCVImage Constructors

4.6.1.1 Default Constructor

```
NvCVImage();
```

The default constructor creates an empty image with no buffer.

4.6.1.2 Allocation Constructor

```
NvCVImage(
  unsigned width,
  unsigned height,
  NvCVImage_PixelFormat format,
  NvCVImage_ComponentType type,
  unsigned isPlanar,
  unsigned onGPU,
  unsigned alignment
);
```

The allocation constructor creates an image to which memory has been allocated and that has been initialized.

`width [in]`

Type: `unsigned`

The width of the image in pixels.

`height [in]`

The height of the image in pixels.

`format [in]`

Type: `NvCVImage_PixelFormat`

The format of the pixels.

`type [in]`

Type: `NvCVImage_ComponentType`

The type of the components of the pixels.

`isPlanar [in]`

Type: `unsigned`

The organization of the components of the pixels in the image. See “Pixel Organizations” on page 44.

`onGPU [in]`

Type: `unsigned`

The type of memory in which the image data buffers are to be stored. See “Memory Types” on page 69.

`alignment [in]`

Type: `unsigned`

The row byte alignment, which specifies the alignment of the first pixel in each scan line. Set this parameter to 0 or a power of 2.

- 1: Specifies no gap between scan lines. A byte alignment of 1 is required by all GPU buffers used by the video effect filters.
- 0: Specifies the default alignment, which depends on the type of memory in which the image data buffers are stored:
  - CPU memory: Specifies an alignment of 4 bytes,
  - GPU memory: Specifies the alignment set by `cudaMallocPitch`.
- 2 or greater: Specifies any other alignment, such as a cache line size of 16 or 32 bytes.

> Note: If the product of `width` and the `pixelBytes` member of `NvCVImage` is a whole-number multiple of `alignment`, the gap between scan lines is 0 bytes, regardless of the `alignment` value.

4.6.1.3 Subimage Constructor

```
NvCVImage(
  NvCVImage *fullImg,
  int x,
  int y,
  unsigned width,
  unsigned height
);
```

The subimage constructor creates an image that is initialized with a view of the specified rectangle in another image. No additional memory is allocated.

`fullImg [in]`

Type: `NvCVImage *`

Pointer to the existing image from which the view of a specified rectangle in the image that will be taken.

`x [in]`

The x coordinate of the left edge of the view that will be taken.

`y [in]`

The y coordinate of the top edge of the view that will be taken.

`width [in]`

Type: `unsigned`

The width of the view that will be taken, in pixels.

`height [in]`

Type: `unsigned`

The height of the view that will be taken, in pixels.

4.6.2 NvCVImage Destructor

```
~NvCVImage();
```

4.6.3 copyFrom

This version copies an entire image to another image, which is functionally identical to `NvCVImage_Transfer(src, this, 1.0f, 0, NULL);`.

```
NvCV_Status copyFrom(
  const NvCVImage *src
);
```

This version copies the specified rectangle in the source image to the destination image.

```
NvCV_Status copyFrom(
  const NvCVImage *src,
  int srcX,
  int srcY,
  int dstX,
  int dstY,
  unsigned width,
  unsigned height
);
```

4.6.3.1 Parameters

`src [in]`

Type: `const NvCVImage *`

Pointer to the existing source image from which the specified rectangle will be copied.

`srcX [in]`

Type: `int`

The x coordinate in the source image of the left edge of the rectangle to be copied.

`srcY [in]`

Type: `int`

The y coordinate in the source image of the top edge of the rectangle to be copied.

`dstX [in]`

Type: `int`

The x coordinate in the destination image of the left edge of the copied rectangle.

`srcY [in]`

Type: `int`

The y coordinate in the destination image of the top edge of the copied rectangle.

`width [in]`

Type: `unsigned`

The width in pixels of the rectangle to be copied.

`height [in]`

Type: `unsigned`

The height in pixels of the rectangle to be copied.

4.6.3.2 Return Value

- `NVFVX_SUCCESS` on success.
- `NVCV_ERR_PIXELFORMAT`, if the pixel format is not supported.
- `NVCV_ERR_MISMATCH`, if the formats of the source and destination images are different.
- `NVCV_ERR_CUDA`, if a CUDA error occurs.

4.6.3.3 Remarks

This overloaded function either copies an entire image to another image or copies the specified rectangle in an image to another image.

This function can copy image data buffers stored in the following memory types:

- From CPU to CPU
- From CPU to GPU
- From GPU to GPU
- From GPU to CPU

> **Note**: For additional use cases, use the `NvCVImage_Transfer()` function.

4.7 The NVIDIA AR SDK Return Codes

The `NvCV_Status` enumeration defines the following values that the NVIDIA AR functions might return to indicate error or success:

> **Note**: These values are defined in the `nvCVStatus.h` file.

`NVCV_SUCESS = 0`

The procedure returned successfully.

`NVCV_ERR_GENERAL`

An otherwise unspecified error has occurred.

`NVCV_ERR_UNIMPLEMENTED`

The requested feature is not yet implemented.

`NVCV_ERR_MEMORY`

There is not enough memory for the requested operation.

`NVCV_ERR_EFFECT`

An invalid effect handle has been supplied.

`NVCV_ERR_SELECTOR`

The given parameter selector is not valid in this effect filter.

`NVCV_ERR_BUFFER`

An image buffer has not been specified.

`NVCV_ERR_PARAMETER`

An invalid parameter value has been supplied for this feature+key.

`NVCV_ERR_MISMATCH`

Some parameters are not appropriately matched.

`NVCV_ERR_PIXELFORMAT`

The specified pixel format is not accommodated.

`NVCV_ERR_MODEL`

Error while loading the TRT model.

`NVCV_ERR_LIBRARY`

Error loading the dynamic library.

`NVCV_ERR_INITIALIZATION`

The effect has not been properly initialized.

`NVCV_ERR_FILE`

The file could not be found.

`NVCV_ERR_FEATURENOTFOUND`

The requested feature was not found

`NVCV_ERR_MISSINGINPUT`

A required parameter was not set

`NVCV_ERR_CUDA_MEMORY`

There is not enough CUDA memory for the requested operation.

`NVCV_ERR_CUDA_VALUE`

A CUDA parameter is not within the acceptable range.

`NVCV_ERR_CUDA_PITCH`

A CUDA pitch is not within the acceptable range.

`NVCV_ERR_CUDA_INIT`

The CUDA driver and runtime could not be initialized.

`NVCV_ERR_CUDA_LAUNCH`

The CUDA kernel launch has failed.

`NVCV_ERR_CUDA_KERNEL`

No suitable kernel image is available for the device.

`NVCV_ERR_CUDA_DRIVER`

The installed NVIDIA CUDA driver is older than the CUDA runtime library.

`NVCV_ERR_CUDA_UNSUPPORTED`

The CUDA operation is not supported on the current system or device.

`NVCV_ERR_CUDA_ILLEGAL_ADDRESS`

CUDA tried to load or store on an invalid memory address.

`NVCV_ERR_CUDA`

An otherwise unspecified CUDA error has been reported.

Appendix A. NVIDIA 3DMM File Format

The NVIDIA 3DMM file format is based on encapsulated objects that are scoped by a FOURCC tag and a 32-bit size.

The header must appear first in the file. The objects and their subobjects can appear in any order. In this guide, they are listed in the default order.

A.1 Header

The header contains the following information:

- The name `NFAC`
- `size=8`
- `endian=0xe4` (little endian)
- `sizeBits=32`
- `indexBits=16`
- The offset of the table of contents

NFAC				
size				
	endian	sizeBits	indexBits	zero
	TOC loc			

A.2 Model Object

The model object contains a shape component and an optional color component. Both objects contain the following information:

- A mean shape
- A set of shape modes
- The eigenvalues for the modes

- A triangle list

MODL						
size						
	SHAP					
	size					
		MEAN				
		size				
			mean shape			
		BSIS				
		size				
			number of modes			
			shape modes			
			...			
		EIVL				
		size				
			shape eigenvalues			
		TRNG				
		size				
			triangle list			
	COLR					
	size					
		MEAN				
		size				
			mean color			
		BSIS				
		size				
			number of modes			
			color modes			
			...			
		EIVL				
		size				
			color eigenvalues			
		TRNG				
		size				
			triangle list			

A.3 IBUG Mappings Object

The IBUG mappings object contains the following information:

- Landmarks
- Right contour
- Left contour

IBUG				
size				
	LMRK			
	size			
		landmarks		
	RCTR			
	size			
		right contour		
	LCTR			
	size			
		left contour		

A.4 Blend Shapes Object

The blend shapes object contains a set of blend shapes, and each blend shape has a name.

BLND				
size				
	numShapes			
	NAME			
	size			
		name string		
	SHAP			
	size			
		blend shape		
	NAME			
	size			
		name string		
	SHAP			
	size			
		blend shape		
		. . .		

A.5 Model Contours Object

The model contours object contains a right contour and a left contour.

MCTR				
size				
	RCTR			
	size			
		right model contour		
	LCTR			
	size			
		left model contour		

A.6 Topology Object

The topology contains a list of pairs of the adjacent faces and vertices.

TOPO		
size		
	AJFC	
	size	
		adjacent faces
	AJVX	
	size	
		adjacent vertices

A.7 Table of Contents Object

The optional table of contents object contains a list of tagged objects and their offsets. This object can be used to randomly access objects. The file is usually read in sequential order.

TOC0	
size	
	record size
	tag
	offset
	tag
	offset
	. . .

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051

http://www.nvidia.com