NVIDIA AR SDK

Programming Guide (for Windows)

PG-09605-001_v0.7 Beta | April 2021

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 (Beta) detects, localizes, and tracks human faces in images or videos by using bounding boxes.
- Facial landmark detection and tracking (Beta) predicts and tracks the pixel locations of human facial landmark points and head poses in images or videos.

  It can predict 68 and 126 landmark points. The 68 detected facial landmarks follow the *Multi-PIE 68 point mark-ups* information in Facial point annotations. The 126 facial landmark points detector can predict more points on the cheeks, the eyes, and on laugh lines.
- Face 3D mesh and tracking (Beta) reconstructs and tracks a 3D human face and its head pose from the provided facial landmarks.
- 3D Body Pose tracking (Beta) predicts and tracks the 3D human pose from images or videos.

  It predicts 34 keypoints of body pose in 2D and 3D. See Appendix B for the keypoints.

NVIDIA AR SDK can be used in a wide variety of applications, such as augmented reality, beautification, 3D face animation, modeling, and so on. NVIDIA AR SDK provides sample applications that demonstrate 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.

This SDK is designed and optimized for client-side application integration and for local deployment. We **do not** officially support the testing, experimentation, deployment of this SDK to a datacenter/cloud environment.

2.1.1 Hardware Requirements

NVIDIA AR SDK is compatible with GPUs that are based on the NVIDIA Turing™, the NVIDIA Ampere™ architecture, and have 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® 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	455.57 or later

2.2 Installing NVIDIA AR SDK and Associated Software

The SDK is distributed in the following forms:

- The development SDK package.

  The development package includes everything in the SDK including the API headers, runtime dependencies, and sample apps.

- The redistributable SDK package.

  The redistributable package is more convenient if your application only wants to integrate SDK API headers and ask end users to download and install the SDK runtime dependencies.

  To develop applications, because the essential contents in these two packages are the same, you can use either package.

The redistributable SDK package comprises 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 following SDK runtime dependencies:
  - The DLLs
  - The models
  - The SDK dependency libraries

  The install location is the `C:\Program Files\NVIDIA Corporation\NVIDIA AR SDK\` directory.

For an application that is built on the SDK, the developer can package the runtime dependencies into the application or require application users to use the SDK installer.

Note: The source code and sample applications are in the development package and are hosted on GitHub at `https://github.com/NVIDIA/MAXINE-AR-SDK`.

To use the SDK redistributable package, download the source code from GitHub and install the SDK binaries.

Note: If you are using development package, you can ignore this step.

The sample app source code demonstrates how to integrate API headers and call the SDK APIs. The sample app also includes the `nvARProxy.cpp` file that is used to link against the SDK DLL without requiring an import library (.lib) file. With this file, you can compile the compile the open-source code independently of the SDK installer. However, the SDK runtime dependencies are still required to load the runtime dependencies, the DLLs, and models.

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/MAXINE-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`.

If you use the development SDK package, place the `face_model0.nvf` file in the `bin\models` folder. If you use the redistributable SDK package, place the `face_model0.nvf` file in the `%Program Files%\NVIDIA Corporation\NVIDIA AR SDK\models` folder that was 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 121.

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

If you use the development SDK package, place the `face_model0.nvf` file in the `bin\models` folder. If you use the redistributable SDK package, place the `face_model0.nvf` file in the `%Program Files%\NVIDIA Corporation\NVIDIA AR SDK\models` folder that was created by the SDK Installer.

2.4 NVIDIA AR SDK Sample Applications

FaceTrack is a sample Windows application that demonstrates the face tracking, landmark tracking, and 3D mesh tracking features of the NVIDIA AR SDK. Similarly, BodyTrack is a sample Windows application that demonstrates the 3D Body Pose Tracking feature of the SDK.

2.4.1 Building the Sample Applications

The open source repository includes the source code to build the sample applications, 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`.
      a. Use CMake to configure and generate the Visual Studio solution file.
   c). Click **Configure**.
   d). When prompted to confirm that CMake can create the build folder, click **OK**.
   e). Select **Visual Studio** for the generator and **x64** for the platform.
   f). To complete configuring the Visual Studio solution file, click **Finish**.
   g). To generate the Visual Studio Solution file, click **Generate**.
   h). Verify that the build folder contains the `NvAR_SDK.sln` file.
   i). Use Visual Studio to generate the `FaceTrack.exe` and `BodyTrack.exe` files from the `NvAR_SDK.sln` file.
   j). In CMake, to open Visual Studio, click **Open Project**.
   k). In Visual Studio, select **Build** > **Build Solution**.

2.4.2 Running the Sample Applications

The SDK provides two sample apps:

- FaceTrack
- BodyTrack

The Prebuilt `FaceTrack.exe and BodyTrack.exe` files are provided 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. To run a sample application, complete one of the following steps:
   - For the FaceTrack sample application, from the `samples\FaceTrack` folder, under the root folder of the NVIDIA AR SDK, execute the `run.bat` file.
   - To run BodyTrack sample application, from the `samples\BodyTrack` 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. For FaceTrack, it draws a 3D face mesh over the largest detected face. For BodyTrack, it draws a Body Pose skeletal over the detected person.

2.4.3 Command-Line Arguments for the FaceTrack Sample Application

`--model_path=`*path*

Specifies the path to the models.

`--landmarks_126[=(true|false)]`

Specifies whether to set the number of landmark points to 126 or 68.

- `true`: set number of landmarks to 126.
- `false`: set number of landmarks to 68.

`--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 Command-Line Arguments for the BodyTrack Sample Application

`--model_path=`*path*

Specifies the path to the models.

`--mode[=(0|1)]`

Specifies whether to select High Performance mode or High Quality mode.

- `0`: set mode to High Quality.
- `1`: set mode to High Performance.

`--app_mode[=(0|1)]`

Specifies whether to select Body Detection or Body Pose Detection.

- `0`: set mode to Body Detection.
- `1`: set mode to Body Pose Detection.

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

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

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

Uses CUDA Graphs to improve performance. CUDA graph reduces the reduce the overhead of GPU operation submission of 3D body tracking.

`--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.

2.4.5 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.

2.4.6 Keyboard Controls for the FaceTrack Sample Application

The `FaceTrack` 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`.

2.4.7 Keyboard Controls for the BodyTrack Sample Application

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

- 1 selects the "body-tracking-only" mode and shows only the bounding boxes.
- 2 selects the "body and body pose" tracking mode and shows the bounding boxes and body pose keypoints.
- 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, 3D face mesh tracking, and 3D Body Pose tracking features of the SDK.

3.1 Creating an Instance of a Feature Type

The feature type is a predefined structure that is used to access the SDK features. Each feature requires an instantiation of the 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 13 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 13 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:
   - The CUDA Runtime API function `cudaStreamCreate()`
   - `NvAR_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 11 for more information.
   - The key value `NVAR_Parameter_Config(CUDAStream)`.
     See "Key Values in the Properties of a Feature Type" on page 13 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 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()`

3.2.3 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 14 for more information.
- `NvAR_Parameter_Input` indicates an input property.

  See "Input Properties" on page 16 for more information.

- `NvAR_Parameter_Output` indicates an output property.

  See "Output Properties" on page 17 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 13.

3.2.3.1 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 121.

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

`NvAR_Parameter_Config(ShapeEigenValueCount)`

The number of eigenvalues used to describe shape.

String equivalent: `NvAR_Parameter_Config_ShapeEigenValueCount`

Property type: unsigned integer

`NvAR_Parameter_Config(ExpressionCount)`

The number of coefficients used to represent expression.

String equivalent: `NvAR_Parameter_Config_ExpressionCount`

Property type: unsigned integer

`NvAR_Parameter_Config(FocalLength)`

The focal length of the camera used for 3D Body Pose.

String equivalent: `NvAR_Parameter_Config_FocalLength`

Property type: float

`NvAR_Parameter_Config(UseCudaGraph)`

Flag to enable CUDA Graph optimization. The CUDA graph reduces the overhead of GPU operation submission of 3D body tracking.

String equivalent: `NvAR_Parameter_Config_UseCudaGraph`

Property type: bool

`NvAR_Parameter_Config(NVAR_MODE)`

Flag to select the mode for 3D Body Pose Tracking.

String equivalent: `NvAR_Parameter_Config_NVAR_MODE`

Property type: unsigned integer

`NvAR_Parameter_Config(NVAR_MODE)`

Mode to select High Performance or High Quality for 3D Body Pose.

String equivalent: `NvAR_Parameter_Config_NVAR_MODE`

Property type: unsigned int

`NvAR_Parameter_Config(FocalLength)`

Focal Length of the camera for 3D Body.

String equivalent: `NvAR_Parameter_Config_FocalLength`

Property type: float

`NvAR_Parameter_Config(UseCudaGraph)`

Flag to select Cuda Graph for 3D Body Pose.

String equivalent: `NvAR_Parameter_Config_UseCudaGraph`

Property type: unsigned integer

`NvAR_Parameter_Config(ReferencePose)`

CPU buffer of type `NvAR_Point3f` to hold the Reference Pose for Joint Rotations for 3D Body Pose.

String equivalent: `NvAR_Parameter_Config_ReferencePose`

Property type: object (`void*`)

3.2.3.2 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*`)

3.2.3.3 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, and the 126 points cover more points along the cheeks, the eyes, and the laugh lines.

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*`)

`NvAR_Parameter_Output(ShapeEigenValues)`

Float array of shape eigenvalues. Get NvAR_Parameter_Config(ShapeEigenValueCount) to determine how many eigenvalues there are.

String equivalent: `NvAR_Parameter_Output_ShapeEigenValues`

Property type: const floating point array

`NvAR_Parameter_Output(ExpressionCoefficients)`

Float array of expression coefficients. Get NvAR_Parameter_Config(ExpressionCount) to determine how many coefficients there are.

String equivalent: `NvAR_Parameter_Output_ExpressionCoefficients`

Property type: const floating point array

`NvAR_Parameter_Output(KeyPoints)`

CPU output buffer of type `NvAR_Point2f` to hold the output detected 2D Keypoints for Body Pose. Refer to Appendix B for information about the Keypoint names and the order of Keypoint output.

String equivalent: `NvAR_Parameter_Output_KeyPoints`

Property type: object (`void*`)

`NvAR_Parameter_Output(KeyPoints3D)`

CPU output buffer of type `NvAR_Point3f` to hold the output detected 3D Keypoints for Body Pose. Refer to "3D Body Pose Keypoint Format" on page 126 for information about the Keypoint names and the order of Keypoint output

String equivalent: `NvAR_Parameter_Output_KeyPoints3D`

Property type: object (`void*`)

`NvAR_Parameter_Output(JointAngles)`

CPU output buffer of type `NvAR_Point3f` to hold the joint angles in axis-angle format for the Keypoints for Body Pose

String equivalent: `NvAR_Parameter_Output_JointAngles`

Property type: object (`void*`)

`NvAR_Parameter_Output(KeyPointsConfidence)`

Float array of confidence values for each detected keypoints.

String equivalent: `NvAR_Parameter_Output_` KeyPointsConfidence

Property type: floating point array

`NvAR_Parameter_Output(KeyPoints)`

CPU output buffer of type `NvAR_Point2f` to hold the output detected 2D key points for 3D Body Pose. Refer to Appendix B for more information. The order of the points in the CPU buffer follows the order mentioned in Appendix B.

String equivalent: `NvAR_Parameter_Output_KeyPoints`

Property type: object (`void*`)

`NvAR_Parameter_Output(KeyPoints3D)`

CPU output buffer of type `NvAR_Point3f` to hold the output detected 3D key points for 3D Body Pose. Refer to Appendix B for more information. The order of the points in the CPU buffer follows the order mentioned in Appendix B.

String equivalent: `NvAR_Parameter_Output_KeyPoints3D`

Property type: object (`void*`)

`NvAR_Parameter_Output(JointAngles)`

CPU output buffer of type `NvAR_Quaternion` to hold the output the joint rotations for 3D Body Pose.

String equivalent: `NvAR_Parameter_Output_JointAngles`

Property type: object (`void*`)

`NvAR_Parameter_Output(KeyPointsConfidence)`

Float array of confidence values for each detected keypoint for 3D Body Pose.

String equivalent: `NvAR_Parameter_Output_KeyPointsConfidence`

Property type: floating point array

3.2.4 Getting the Value of a Property of a Feature

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);
```

3.2.5 Setting a Property for a Feature

To set a property for a feature:

1. Allocate memory for all inputs and outputs that are required by the feature 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:

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 28 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 11 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 11 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 11 for more information.

```
NvAR_Destroy(faceDetectHandle);
```

3.6 Working with Image Frames on GPU or CPU Buffers

Effect filters 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 . NVIDIA AR SDK provides functions for converting an image representation to `NvCVImage` and transferring images between CPU and GPU buffers.

3.6.1 Converting Image Representations to NvCVImage Objects

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

3.6.1.1 Converting OpenCV Images to NvCVImage Objects

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

Note: NVIDIA AR 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 to NvCVImage Objects

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

```
NvCVImage src_gpu;
vfxErr = NvCVImage_Init(&src_gpu, 640, 480, 1920, srcPixelBuffer, NVCV_BGR,
NVCV_U8, NVCV_INTERLEAVED, NVCV_GPU);

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

3.6.1.3 Converting Decoded Frames from the NvDecoder to NvCVImage Objects

Call the `NvCVImage_Transfer()` function to convert the decoded frame that is provided by the `NvDecoder` from the decoded pixel format to the format that is required by a feature of the NVIDIA AR SDK. The following sample shows a decoded frame that was converted from the NV12 to the BGRA pixel format.

```
NvCVImage decoded_frame, BGRA_frame, stagingBuffer;
NvDecoder dec;
```

```
//Initialize decoder...
//Assuming dec.GetOutputFormat() == cudaVideoSurfaceFormat_NV12

//Initialize memory for decoded frame
NvCVImage_Init(&decoded_frame, dec.GetWidth(), dec.GetHeight(),
dec.GetDeviceFramePitch(), NULL, NVCV_YUV420, NVCV_U8, NVCV_NV12, NVCV_GPU,
1);
decoded_frame.colorSpace = NVCV_709 | NVCV_VIDEO_RANGE |
NVCV_CHROMA_COSITED;

//Allocate memory for BGRA frame, and set alpha opaque
NvCVImage_Alloc(&BGRA_frame, dec.GetWidth(), dec.GetHeight(), NVCV_BGRA,
NVCV_U8, NVCV_CHUNKY, NVCV_GPU, 1);
cudaMemset(BGRA_frame.pixels, -1, BGRA_frame.pitch * BGRA_frame.height);

decoded_frame.pixels = (void*)dec.GetFrame();

//Convert from decoded frame format(NV12) to desired format(BGRA)
NvCVImage_Transfer(&decoded_frame, &BGRA_frame, 1.f, stream, &
stagingBuffer);
```

Note: The sample above assumes the typical colorspace specification for HD content. SD typically uses NVCV_601. There are 8 possible combinations, and you should use the one that matches your video as described in the video header or proceed by trial and error:

Here is some additional information:

- If the colors are incorrect, swap 709<->601.
- If they are washed out or blown out, swap VIDEO<->FULL.
- If the colors are shifted horizontally, swap INTSTITIAL<->COSITED

3.6.1.4 Converting an NvCVImage Object to a Buffer that can be Encoded by NvEncoder

To convert the `NvCVImage` to the pixel format that is used during encoding via NvEncoder, if necessary, call the `NvCVImage_Transfer()` function. The following sample shows a frame that is encoded in the BGRA pixel format.

```
//BGRA frame is 4-channel, u8 buffer residing on the GPU
NvCVImage BGRA_frame;
NvCVImage_Alloc(&BGRA_frame, dec.GetWidth(), dec.GetHeight(), NVCV_BGRA,
NVCV_U8, NVCV_CHUNKY, NVCV_GPU, 1);

//Initialize encoder with a BGRA output pixel format
using NvEncCudaPtr = std::unique_ptr<NvEncoderCuda,
std::function<void(NvEncoderCuda*)>>;
NvEncCudaPtr pEnc(new NvEncoderCuda(cuContext, dec.GetWidth(),
dec.GetHeight(), NV_ENC_BUFFER_FORMAT_ARGB));
pEnc->CreateEncoder(&initializeParams);
```

```
//...

std::vector<std::vector<uint8_t>> vPacket;
//Get the address of the next input frame from the encoder
const NvEncInputFrame* encoderInputFrame = pEnc->GetNextInputFrame();


//Copy the pixel data from BGRA_frame into the input frame address obtained
above
NvEncoderCuda::CopyToDeviceFrame(cuContext,
                        BGRA_frame.pixels,
                        BGRA_frame.pitch,
                        (CUdeviceptr)encoderInputFrame->inputPtr,
                        encoderInputFrame->pitch,
                        pEnc->GetEncodeWidth(),
                        pEnc->GetEncodeHeight(),
                        CU_MEMORYTYPE_DEVICE,
                        encoderInputFrame->bufferFormat,
                        encoderInputFrame->chromaOffsets,
                        encoderInputFrame->numChromaPlanes);
pEnc->EncodeFrame(vPacket);
```

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. In both 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 114 for more information.

The final three optional parameters of the allocation constructor 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 final three optional parameters of the allocation constructor to determine the properties of the `NvCVImage` object.

This example creates an object without setting the final three optional parameters of the allocation constructor. In this object, the blue, green, and red components 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 setting the final three optional parameters explicitly. 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 the default, that is, 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.

     Allocate a buffer this way when the image is part of a state structure, where you will not know the size the image until later.
   - To reallocate a buffer, call `NvCVImage_Realloc()`.

     This function checks for an allocated buffer and reshapes the buffer if it is big enough, 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 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 to use as a staging GPU buffer 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, which is 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 and Tracking" on page 37 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.

Property Name	Value
	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 or 126. Specifies the number of landmark points (X and Y values) to be returned. Set by the user.
`LandmarksConfidence_Size`	Unsigned integer, 68 or 126. Specifies the number of landmark confidence values for the detected keypoints to be returned. Set by the user.
`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 and Tracking" on page 38 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 and Tracking" on page 38 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 the following: • `NvAR_Parameter_Config(BatchSize)` • `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 and Tracking" on page 38 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 the Face 3D Mesh Feature" on page 40 for more information. Set by the user.
`CUDAStream`	**Optional:** The CUDA stream. See "Alternative Usage of the Face 3D Mesh Feature" on page 40 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 the Face 3D Mesh Feature" on page 40 for more information. Set by the user.
`LandmarksConfidence_Size`	Unsigned integer, 68 or 126. If landmark detection is run internally, the confidence values for the detected key points are returned. See "Alternative Usage of the Face 3D Mesh Feature" on page 40 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 will be fitted. Set by the user.
`Height`	The height of the input image buffer that contains the face to which the face model will 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 the Face 3D Mesh Feature" on page 40 for more information.

Property Name	Value
	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 the Face 3D Mesh Feature" on page 40 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 the Face 3D Mesh Feature" on page 40 for more information. To be allocated by the user.
`Pose`	**Optional:** `NvAR_Quaternion` array pointer, to hold one quaternion. See "Alternative Usage of the Face 3D Mesh Feature" on page 40 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 the Face 3D Mesh Feature" on page 40 for more information. To be allocated by the user.

Property Name	Value
`BoundingBoxes`	**Optional:** `NvAR_BBoxes` structure that contains the detected face that is determined internally. See "Alternative Usage of the Face 3D Mesh Feature" on page 40 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 the Face 3D Mesh Feature" on page 40 for more information. To be allocated by the user.

3.7.4 Body Detection Property Values

Table 3-11: Configuration Properties for Body Detection

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, which is 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 body detection. Set by the user.

Table 3-12: Input Properties for Body Detection

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 3-13: Output Properties for Body Detection

Property Name	Value
`BoundingBoxes`	`NvAR_BBoxes` structure that holds the detected body 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 body box. To be allocated by the user.

3.7.5 3D Body Pose Keypoint Tracking Property Values

Table 3-14: Configuration Properties for 3D Body Pose Keypoint 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.
`NVAR_MODE`	Unsigned integer, 0 or 1. Default is 1. Selects the High Performance (1) mode or High Quality (0) mode Set by the user.
`UseCudaGraph`	Bool, True or False. Default is True Flag to use CUDA Graphs for optimization. Set by the user.

Property Name	Value
`FocalLength`	Float. Default is 800.79041 Specifies the focal length of the camera to be used for 3D Body Pose. Set by the user.
`Temporal`	Unsigned integer, 1/0 to enable/disable the temporal optimization of Body Pose tracking. Set by the user.
`NumKeyPoints`	Unsigned integer. Specifies the number of keypoints available, which is currently 34.
`ReferencePose`	Vector<NvAR_Point3f>. Specifies the Reference Pose used to compute the joint angles.

Table 3-15: Input Properties for 3D Body Pose Keypoint 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 3D Body Pose detection. If not specified as an input property, body detection is automatically run on the input image. To be allocated by the user.

Table 3-16: Output Properties for 3D Body Pose Keypoint Tracking

Property Name	Value
`Keypoints`	`NvAR_Point2f` array, which must be large enough to hold the 34 points given by the product of `NvAR_Parameter_Config(BatchSize)` and `34`. To be allocated by the user.
`Keypoints3D`	`NvAR_Point3f` array, which must be large enough to hold the 34 points given by the product of `NvAR_Parameter_Config(BatchSize)` and `34`. To be allocated by the user.
`JointAngles`	`NvAR_Quaternion` array, which must be large enough to hold the 34 joints given by the product of `NvAR_Parameter_Config(BatchSize)` and `34`. They represent the local rotation (in Quaternion) of each joint with reference to the ReferencePose. To be allocated by the user.
`KeyPointsConfidence`	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 the following: • `NvAR_Parameter_Config(BatchSize)` • `34` To be allocated by the user.
`BoundingBoxes`	`NvAR_BBoxes` structure that contains the detected body through body detection performed by the 3D Body Pose feature. 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 38 and "Face 3D Mesh and Tracking" on page 40 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 Table 3-5, 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 the 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 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);
```

3.8.4 3D Body Pose Tracking

3D Body Pose Tracking consists of the following parts:

- Body Detection
- 3D Keypoint Detection

In this release, we support only one person in the frame, and when the full body (head to toe) is visible. However, the feature will still work if a part of the body, such as an arm or a foot, is occluded/truncated.

This feature relies on temporal information to track the person in scene, where the keypoints information from the previous frame is used to estimate the keypoints of the next frame.

3.8.4.1 3D Body Pose Tracking for Static Frames (Images)

You can obtain the bounding boxes that encapsulate the people in the scene. To obtain detected bounding boxes, you can explicitly instantiate and run body detection as shown in the example below and pass image buffer as input.

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

```
//Set input image buffer
NvAR_SetObject(bodyDetectHandle, 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(bodyDetectHandle, NvAR_Parameter_Output(BoundingBoxes),
&output_bboxes, sizeof(NvAR_BBoxes));

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

NvAR_Run(bodyDetectHandle);
```

The input to 3D Body Keypoint Detection is an input image. It outputs the 2D Keypoints, 3D Keypoints, Keypoints confidence scores, and bounding box encapsulating the person.

This example runs the 3D Body Pose Detection AR feature:

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

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

```
//Set output buffer to hold detected keypoints
std::vector<NvAR_Point2f> keypoints;
std::vector<NvAR_Point3f> keypoints3D;
std::vector<NvAR_Point3f> jointAngles;
std::vector<float> keypoints_confidence;

// Get the number of keypoints
unsigned int numKeyPoints;
NvAR_GetU32(keyPointDetectHandle, NvAR_Parameter_Config(NumKeyPoints),
&numKeyPoints);

keypoints.assign(batchSize * numKeyPoints , {0.f, 0.f});
keypoints3D.assign(batchSize * numKeyPoints , {0.f, 0.f, 0.f});
jointAngles.assign(batchSize * numKeyPoints , {0.f, 0.f, 0.f});
NvAR_SetObject(keyPointDetectHandle, NvAR_Parameter_Output(KeyPoints),
keypoints.data(), sizeof(NvAR_Point2f));
NvAR_SetObject(keyPointDetectHandle, NvAR_Parameter_Output(KeyPoints3D),
keypoints3D.data(), sizeof(NvAR_Point3f));
NvAR_SetF32Array(keyPointDetectHandle,
NvAR_Parameter_Output(KeyPointsConfidence), keypoints_confidence.data(),
batchSize * numKeyPoints);
NvAR_SetObject(keyPointDetectHandle, NvAR_Parameter_Output(JointAngles),
jointAngles.data(), sizeof(NvAR_Point3f));

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

NvAR_Run(keyPointDetectHandle);
```

3.8.4.2 3D Body Pose Tracking for Temporal Frames (Videos)

The feature relies on temporal information to track the person in scene. The keypoints information from the previous frame is used to estimate the keypoints of the next frame. This example uses the 3D Body Pose Tracking AR feature to obtain 3D Body Pose Keypoints directly from the image:

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

//Pass output bounding boxes from body detection as an input on which
//landmark detection is to be run
```

```
NvAR_SetObject(keypointDetectHandle, NvAR_Parameter_Input(BoundingBoxes),
&output_bboxes, sizeof(NvAR_BBoxes));

//Set output buffer to hold detected keypoints
std::vector<NvAR_Point2f> keypoints;
std::vector<NvAR_Point3f> keypoints3D;
std::vector<NvAR_Point3f> jointAngles;
std::vector<float> keypoints_confidence;

// Get the number of keypoints
unsigned int numKeyPoints;
NvAR_GetU32(keyPointDetectHandle, NvAR_Parameter_Config(NumKeyPoints),
&numKeyPoints);

keypoints.assign(batchSize * numKeyPoints , {0.f, 0.f});
keypoints3D.assign(batchSize * numKeyPoints , {0.f, 0.f, 0.f});
jointAngles.assign(batchSize * numKeyPoints , {0.f, 0.f, 0.f});
NvAR_SetObject(keyPointDetectHandle, NvAR_Parameter_Output(KeyPoints),
keypoints.data(), sizeof(NvAR_Point2f));
NvAR_SetObject(keyPointDetectHandle, NvAR_Parameter_Output(KeyPoints3D),
keypoints3D.data(), sizeof(NvAR_Point3f));
NvAR_SetF32Array(keyPointDetectHandle,
NvAR_Parameter_Output(KeyPointsConfidence), keypoints_confidence.data(),
batchSize * numKeyPoints);
NvAR_SetObject(keyPointDetectHandle, NvAR_Parameter_Output(JointAngles),
jointAngles.data(), sizeof(NvAR_Point3f));

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

NvAR_Run(keyPointDetectHandle);
```

3.9 Using Multiple GPUs

Applications that are developed with the NVIDIA AR SDK can be used with multiple GPUs. By default, the SDK determines which GPU to use based on the capability of the currently selected GPU: If the currently selected GPU supports the NVIDIA AR SDK, the SDK uses it. Otherwise, the SDK selects the best GPU.

You can control which GPU is used in a multi-GPU environment by using the `cudaSetDevice(int whichGPU)` and `cudaGetDevice(int *whichGPU)` NVIDIA CUDA Toolkit functions and the `NvAR_SetS32(NULL, NvAR_Parameter_Config(GPU),` `whichGPU)` AR SDK Set function. The `Set()` call is called only once for the AR SDK, before any effects are created. Since it is impossible to transparently pass images that are allocated on one GPU to another GPU, you must ensure that the same GPU is used for all AR features.

```
NvCV_Status err;
int chosenGPU = 0; // or whatever GPU you want to use
err = NvAR_SetS32(NULL, NvAR_Parameter_Config(GPU), chosenGPU);
if (NVCV_SUCCESS != err) {
    printf("Error choosing GPU %d: %s\n", chosenGPU,
            NvCV_GetErrorStringFromCode(err));
}
cudaSetDevice(chosenGPU);
NvCVImage dst = new NvCVImage(…);
NvAR_Handle eff;
err = NvAR_API NvAR_CreateEffect(code, &eff);
…
err = NvAR_API NvAR_Load(eff);
err = NvAR_API NvAR_Run(eff, true);
// switch GPU for other task, then switch back for next frame
```

Buffers need to be allocated on the selected GPU, so **before** you allocate images on the GPU, call `cudaSetDevice()`. Neural networks need to be loaded on the selected GPU, so before `NvAR_Load()` is called, set this GPU as the current device .

To use the buffers and models, **before** you call `NvAR_Run()` and set the GPU device as the current device. A previous call to `NvAR_SetS32(NULL, NvAR_Parameter_Config(GPU), whichGPU)` helps enforce this requirement.

For performance concerns, switching to the appropriate GPU is the responsibility of the application.

3.9.1 Default Behavior in Multi-GPU Environments

The `NvAR_Load()` function internally calls `cudaGetDevice()` to identify the currently selected GPU. The function checks the compute capability of the currently selected GPU (default 0) to determine whether the GPU architecture supports the NVIDIA AR SDK and completes one of the following tasks:

- If the SDK is supported, `NvAR_Load()` uses the GPU.
- If the SDK is not supported, `NvAR_Load()` searches for the most powerful GPU that supports the NVIDIA AR SDK and calls `cudaSetDevice()` to set that GPU as the current GPU.

If you do not require your application to use a specific GPU in a multi-GPU environment, the default behavior should suffice.

3.9.2 Selecting the GPU for AR SDK Processing in a Multi-GPU Environment

Your application might be designed to only perform the task of applying an AR filter by using in a specific GPU in multi-GPU environment. In this situation, ensure that the NVIDIA AR SDK does not override your choice of GPU for applying the video effect filter.

```
// Initialization
cudaGetDevice(&beforeGPU);
err = NvAR_Load(eff);
if (NVCV_SUCCESS != err) { printf("Cannot load ARSDK: %s\n",
   NvCV_GetErrorStringFromCode(err)); exit(-1); }
cudaGetDevice(&arsdkGPU);
if (beforeGPU != arsdkGPU) {
  printf("GPU #%d cannot run AR SDK, so GPU #%d was chosen instead\n",
    beforeGPU, arsdkGPU);
}
...
```

3.9.3 Selecting Different GPUs for Different Tasks

Your application might be designed to perform multiple tasks in a multi-GPU environment such as, for example, rendering a game and applying an ARfilter. In this situation, select the best GPU for each task before calling `NvAR_Load()`.

1. Call `cudaGetDeviceCount()` to determine the number of GPUs in your environment.

   ```
   // Get the number of GPUs
   cuErr = cudaGetDeviceCount(&deviceCount);
   ```

2. Get the properties of each GPU and determine whether it is the best GPU for each task by performing the following operations for each GPU in a loop:
   a). Call `cudaSetDevice()` to set the current GPU.
   b). Call `cudaGetDeviceProperties()` to get the properties of the current GPU.
   c). To determine whether the GPU is the best GPU for each specific task, use a custom code in your application to analyze the properties that were retrieved by `cudaGetDeviceProperties()`.

This example uses the compute capability to determine whether a GPU's properties should be analyzed and determine whether the current GPU is the best GPU on which to apply a video effect filter. A GPU's properties are analyzed only when the compute capability is 7.5 or 8.6, which denotes a GPU that is based on the NVIDIA Turing GPU architecture or NVIDIA Ampere architecture, respectively.

```
  // Loop through the GPUs to get the properties of each GPU and
  //determine if it is the best GPU for each task based on the
  //properties obtained.
  for (int dev = 0; dev < deviceCount; ++dev) {
    cudaSetDevice(dev);
    cudaGetDeviceProperties(&deviceProp, dev);
    if (DeviceIsBestForARSDK(&deviceProp))  gpuARSDK = dev;
    if (DeviceIsBestForGame(&deviceProp)) gpuGame = dev;
    ...
  }
  cudaSetDevice(gpuARSDK);
  err = NvAR_Set...; // set parameters
  err = NvAR_Load(eff);
```

3. In the loop to complete the application's tasks, select the best GPU for each task before performing the task.

   a). Call `cudaSetDevice()` to select the GPU for the task.

   b). Make all the function calls required to perform the task.

   In this way, you select the best GPU for each task only once without setting the GPU for every function call.

   This example selects the best GPU for rendering a game and uses custom code to render the game. It then selects the best GPU for applying a video effect filter before calling the `NvCVImage_Transfer()` and `NvAR_Run()` functions to apply the filter, avoiding the need to save and restore the GPU for every NVIDIA AR SDK API call.

```
// Select the best GPU for each task and perform the task.
while (!done) {
  ...
  cudaSetDevice(gpuGame);
  RenderGame();
  cudaSetDevice(gpuARSDK);
  err = NvAR_Run(eff, 1);
  ...
}
```

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_Point3f

```
typedef struct NvAR_Point3f {
  float x, y, z;
} NvAR_Point3f;
```

4.1.6.1 Members

`x`

Type: `float`

The X coordinate of the point in pixels.

`y`

Type: `float`

The Y coordinate of the point in pixels.

`z`

Type: `float`

The Z coordinate of the point in pixels.

4.1.6.2 Remarks

This structure represents the X, Y, Z coordinates of one point in 3D space.

Defined in: `nvAR_defs.h`.

4.1.7 NvAR_Quaternion

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

4.1.7.1 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.

4.1.7.2 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`.

4.1.8 NvAR_Rect

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

4.1.8.1 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.

4.1.8.2 Remarks

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

Defined in: `nvAR_defs.h`.

4.1.9 NvAR_RenderingParams

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

4.1.9.1 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.

4.1.9.2 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`.

4.1.10 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 NvAR_Vector3f

```
typedef struct NvAR_Vector3f {
  float vec[3];
} NvAR_Vector3f;
```

4.1.11.1 Members

`vec`

Type: `float` array of size 3

A vector of size 3.

4.1.11.2 Remarks

This structure represents a 3D vector.

Defined in: `nvAR_defs.h`.

4.1.12 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              reserved[2];
  void                       *pixels;
  void                       *deletePtr;
  void                       (*deleteProc)(void *p);
  unsigned long long         bufferBytes;

} NvCVImage;
```

4.1.12.1 Members

`width`

Type: `unsigned int`

The width, in pixels, of the image.

`height`

Type: `unsigned int`

The height, in pixels, of the image.

`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
- 2: pinned CPU 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;
```

See "YUV Color Spaces" on page 62 for more information about the type definitions.

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

`reserved`

Type: `unsigned char[2]`

Reserved for padding and future capabilities. Set this parameter to 0.

`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 that is available through pixels.

4.1.12.2 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 `nvCVImage.h` header file.

4.2.1 NvCVImage_ComponentType

This enumeration defines the data type that is 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_YUV444`

Luminance and full bandwidth 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 together.
- **Semi-planar** pixels are a mix between **interleaved** and **planar** components.

  These types of pixels are found in the video world, where the [Y] component sits in one plane, and the [UV] components are interleaved in another plane.

Typically, pixels are interleaved. However, many neural networks perform better with planar pixels.

In the descriptions of the pixel organizations, 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 Y, U, and V components of all pixels are grouped.
- [Y][UV] indicates that groups of Y components and groups of U and V components are interleaved.

Refer to YUV pixel formats for more information about YUV pixel formats.

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.

```
NVCV_UYVY            2
```

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

Pixels are arranged in [VYUY] groups.

```
NVCV_YUYV               6
NVCV_YUY2               6
```

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

Pixels are arranged in [YUYV] groups.

```
NVCV_YVYU               8
```

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

Pixels are arranged in [YVYU] groups.

NVCV_CYUV 10

This type specifies the interleaved (chunky) YUV 4:4:4 pixels.

Pixels are arranged in [YUV] groups.

NVCV_CYVU 12

This type specifies interleaved (chunky) YVU 4:4:4 pixels.

Pixels are arranged in [YVU] groups.

```
NVCV_YUV                3
NVCV_I420               3      (used with NVCV_YUV420)
NVCV_IYUV               3      (used with NVCV_YUV420)
```

NVCV_I444 3 (used with NVCV_YUV444)

NVCV_YM24 3 (used with NVCV_YUV444)

```
Each of these types specifies one of the following planar YUV
arrangements:
```

- YUV 4:2:2
- YUV 4:2:0
- YUV 4:4:4

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

```
NVCV_YVU                5
NVCV_YV12               5      (used with NVCV_YUV420)
```

NVCV_YM42 5 (used with NVCV_YUV444)

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

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

```
NVCV_YCUV               7
NVCV_NV12               7      (used with NVCV_YUV420)
```

NVCV_NV24 7 (used with NVCV_YUV444)

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

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

```
NVCV_YCVU               9
NVCV_NV21               9       (used with NVCV_YUV420)
```

`NVCV_NV42` 9 (used with NVCV_YUV444)

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

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

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

4.3.2 YUV Color Spaces

The NVIDIA AR SDK API defines the following 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            0
```

Each of these types specifies a color space in which the chroma is sampled horizontally in the same location as the luma samples. Most video formats use this sampling scheme.

```
NVCV_CHROMA_INTSTITIAL       8
NVCV_CHROMA_MPEG1            8
```

Each of these types specifies a color space in which the chroma is sampled horizontally midway between luma samples. This sampling scheme is used for JPEG.

Example: Creating an HD NV12 CUDA buffer

```
NvCVImage *imp = new NvCVImage(1920, 1080, NVCV_YUV420, NVCV_U8,
                               NVCV_NV12, NVCV_CUDA, 0);
imp->colorSpace = NVCV_709 | NVCV_VIDEO_RANGE | NVCV_CHROMA_COSITED;
```

Example: Wrapping an NvCVImage descriptor around an existing HD NV12 CUDA buffer

```
NvCVImage img;
NvCVImage_Init(&img, 1920, 1080, 1920, existingBuffer, NVCV_YUV420, NVCV_U8,
               NVCV_NV12, NVCV_CUDA);
img.colorSpace = NVCV_709 | NVCV_VIDEO_RANGE | NVCV_CHROMA_COSITED;
```

These are particularly useful and performant for interfacing to the NVDEC video decoder.

4.3.3 Memory Types

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

`NVCV_CPU`

The buffer is stored in normal CPU memory.

`NVCV_CPU_PINNED`

The buffer is stored in pinned CPU memory; this can yield higher transfer rates (115%-200%) between the CPU and GPU but should be used sparingly.

`NVCV_GPU`

`NVCV_CUDA`

The 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 13 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

- `NVCV_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 13.

`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 13.

`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 13 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 13 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 48.

`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 13.

`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 13 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 13.

`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 13.

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 13.

`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 13.

`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 "Key Values in the Properties of a Feature Type" on page 13 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 13 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 48.

`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.

4.4.20.2 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 13.

`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 13 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 13.

`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` and in "Key Values in the Properties of a Feature Type" on page 13.

`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 .

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 to 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, appropriately initialized to access the buffer of the `NvCVImage` object. An empty OpenCV image is created by the default the `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 layout,
  unsigned memSpace,
  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.

`layout [in]`

Type: `unsigned`

The organization of the components of the pixels in the image. See “Pixel Organizations” on page 60 for more information.

`memSpace [in]`

Type: `unsigned`

The type of memory in which the image data buffers are to be stored. See “Memory Types” on page 63 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.
- 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, irrespective of the value of `alignment`.

4.5.2.2 Return Value

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

4.5.2.3 Remarks

This function allocates memory for, and initializes, an image. This function assumes that the image data structure has nothing meaningful in it.

This function is called by the C++ `NvCVImage` constructors. You can call this function from C code to allocate memory for, and to 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 will be retrieved.

`rOff [out]`

Type: `int *`

The location in which to store the offset for the red channel (can be `NULL`).

`gOff [out]`

Type: `int *`

The location in which to store the offset for the green channel (can be `NULL`).

`bOff [out]`

Type: `int *`

The location in which to store the offset for the blue channel (can be `NULL`).

`aOff [out]`

Type: `int *`

The location in which to store the offset for the alpha channel (can be `NULL`).

`yOff [out]`

Type: `int *`

The location in which to 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 component, and not byte, offsets. For interleaved pixels, a component offset must be multiplied by the `componentBytes` member of `NvCVImage` to obtain the byte offset.

4.5.4 NvCVImage_Composite

```
NvCV_Status NvCVImage_Composite(
  const NvCVImage *fg,
  const NvCVImage *bg,
  const NvCVImage *mat,
  NvCVImage *dst,
  CUstream stream
);
```

4.5.4.1 Parameters

`fg [in]`

Type: const `NvCVImage *`

The foreground source, which is an RGB or BGR image with u8 or f32 components.

`bg [in]`

Type: const `NvCVImage *`

The background source, which is an RGB or BGR image with u8 or f32 components.

`mat [in]`

Type: `const NvCVImage *`

The matte Y or A image with u8 or f32 components, which indicates where the source image should come through.

`dst [out]`

Type: `NvCVImage *`

The destination image, which can be the same as the `fg` foreground or `bg` background image, or a totally unrelated image.

`stream [out]`

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.

4.5.4.2 Return Value

- `NVCV_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 foreground image over a background image. The `fg`, `bg`, `mat`, and `dst` images must be of the same component type.

4.5.5 NvCVImage_CompositeRect

```
NvCV_Status NvCVImage_CompositeRect(
  const NvCVImage *fg,  const NvCVPoint2i *fgOrg,
  const NvCVImage *bg,  const NvCVPoint2i *bgOrg,
  const NvCVImage *mat, unsigned int mode,
  NvCVImage *dst, const NvCVPoint2i *dstOrg,
);
```

4.5.5.1 Parameters

`fg [in]`

Type: const `NvCVImage *`

The foreground source, which is a RGB or BGR image with u8 or f32 components.

`fgOrg [in]`

Type: `const NvCVPoint2i *`

Pointer to the foreground image upper-left origin from which the image will be transferred.

```
typedef struct NvCVPoint2i { int x, y; } NvCVPoint2i;
```

If this is NULL, the image is transferred from (0,0).

`bg [in]`

Type: const `NvCVImage *`

The background source, which is a RGB or BGR image with u8 or f32 components.

`bgOrg [in]`

Type: `const NvCVPoint2i *`

Pointer to the background image upper-left origin

```
typedef struct NvCVPoint2i { int x, y; } NvCVPoint2i;
```

from which the image will be transferred. If this is NULL, the image is transferred from (0,0).

`mat [in]`

Type: `const NvCVImage *`

The matte Y or A image with u8 or f32 components, which indicates where the source image should come through. The dimensions of the matte determine size of the area to be composited.

`mode [in]`

Type: `unsigned int`

The compositional mode selection. Currently only mode 0 (normal, over) is implemented, and other values return a parameter error.

`dst [out]`

Type: `NvCVImage *`

The destination image, which can be the same as the `fg` foreground or `bg` background image, or a totally unrelated image.

`dstOrg [in]`

Type: `const NvCVPoint2i *`

Pointer to the destination image upper-left origin

```
typedef struct NvCVPoint2i { int x, y; } NvCVPoint2i;
```

to which the image will be transferred. If this is NULL, the image is transferred to (0,0).

stream [out]

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.

4.5.5.2 Return Value

- NVCV_SUCCESS on success
- NVCV_ERR_PIXELFORMAT if the pixel format is not supported.
- NVCV_ERR_PARAMETER if a mode other than 0 is selected.

4.5.5.3 Remarks

This function uses the specified matte image to composite a foreground image over a background image. The fg, bg, mat, and dst images must be of the same component type.

```
NvCVImage_Composite(fg, bg, mat, dst, str);
```

is equal to

```
NvCVImage_CompositeRect(fg, 0, bg, 0, mat, 0, dst, 0, str);
```

4.5.6 NvCVImage_CompositeOverConstant

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

4.5.6.1 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 defines the color field over which the source image will 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.

4.5.6.2 Return Value

- NVCV_SUCCESS on success.
- NVCV_ERR_PIXELFORMAT when the pixel format is not supported.

4.5.6.3 Remarks

This function uses the specified matte image to composite a BGRu8 or RGNU8 image over a constant color field. This function is primarily used for debugging on the CPU and might be discontinued in the future.

4.5.7 NvCVImage_Create

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

4.5.7.1 Parameters

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.

`layout [in]`

Type: `unsigned`

The organization of the components of the pixels in the image. See "Pixel Organizations" on page 60 for more information.

`memSpace [in]`

Type: `unsigned`

The type of memory in which the image data buffers will be stored. See "Memory Types" on page 63 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.
- 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, irrespective of the value of `alignment`.

`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.

4.5.7.2 Return Value

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

4.5.7.3 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++).

4.5.8 NvCVImage_Dealloc

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

4.5.8.1 Parameters

`im [in,out]`

Type: `NvCVImage *`

Pointer to the image whose image buffer will be freed.

4.5.8.2 Return Value

Does not return a value.

4.5.8.3 Remarks

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

4.5.9 NvCVImage_Destroy

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

4.5.9.1 Parameters

`im`

Type: `NvCVImage *`

Pointer to the image that will be destroyed.

4.5.9.2 Return Value

Does not return a value.

4.5.9.3 Remarks

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

4.5.10 NvCVImage_Init

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

4.5.10.1 Parameters

`im [in,out]`

Type: `NvCVImage *`

Pointer to the image that will be initialized.

`width [in]`

Type: `unsigned`

The width, in pixels, of the image.

`height [in]`

Type: `unsigned`

The height, in pixels, of the image.

`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.

`layout [in]`

Type: `unsigned`

The organization of the components of the pixels in the image. See “Pixel Organizations” on page 60 for more information.

`memSpace [in]`

Type: `unsigned`

The type of memory in which the image data buffers are to be stored. See “Memory Types” on page 63 for more information.

4.5.10.2 Return Value

- `NVCV_SUCCESS` on success.
- `NVCV_ERR_PIXELFORMAT` when the pixel format is not supported.

4.5.10.3 Remarks

This function initializes an `NvCVImage` structure from a raw buffer pointer. Initializing an `NvCVImage` object from a raw buffer pointer is useful when you wrap 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 directly setting the fields.

4.5.11 NvCVImage_InitView

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

4.5.11.1 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, in pixels, of the view to be taken.

`height [in]`

Type: `unsigned`

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

4.5.11.2 Return Value

Does not return a value.

4.5.11.3 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.

4.5.12 NvCVImage_Realloc

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

4.5.12.1 Parameters

`im [in,out]`

Type: `NvCVImage *`

The image to initialize.

`width [in]`

Type: `unsigned`

The width, in pixels, of the image.

`height [in]`

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.

`layout [in]`

Type: `unsigned`

The organization of the components of the pixels in the image. See “Pixel Organizations” on page 60 for more information.

`memSpace [in]`

Type: `unsigned`

The type of memory in which the image data buffers are to be stored. See "Memory Types" on page 63 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.
- 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 value of `alignment`.

4.5.12.2 Return Value

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

4.5.12.3 Remarks

This function reallocates memory for, and initializes, an image.

Note: This function assumes that the image is valid.

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

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

4.5.13 NvCVImage_Transfer

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

4.5.13.1 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 when the component type of the source or destination image is 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

This parameter is ignored if the component type of all images is the same (all integer or all floating-point).

`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 GPU memory that is required only if the source image is being converted **and** if the memory types of the source and destination images are different. The buffer has the same characteristics as the CPU image but 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 the 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 as needed 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` can be `NULL`. However, if `tmp` is `NULL`, and a temporary GPU buffer is required, an ephemeral buffer is allocated with a resultant degradation in performance for image sequences.

4.5.13.2 Return Value

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

4.5.13.3 Remarks

This function transfers one image to another image and can perform some conversions on the image. The function uses the GPU to perform the conversions when an image resides on the GPU.

Table 4-1 provides details about the supported conversions between pixel formats.

Note: In each conversion type, the RGB can be in any order.

Table 4-1. Pixel Conversions

	u8→u8	u8→f32	f32→u8	f32→f32
Y→Y	X		X	
Y→A	X		X	
Y→RGB	X	X	X	X
Y→RGBA	X	X	X	X
A→Y	X		X	
A→A	X		X	
A→RGB	X	X	X	X
A→RGBA	X			
RGB→Y	X	X		
RGB→A	X	X		
RGB→RGB	X	X	X	X
RGB→RGBA	X	X	X	X
RGBA→Y	X	X		
RBBA→A		X		
RGBA→RGB	X	X	X	X
RGBA→RGBA	X		X	
YUV420→RGB	X	X		
YUV422→RGB	X	X		
YUV444→RGB	X	X		
RGB→YUV420	X		X	
RGB→YUV422	X		X	
RGB→YUV444	X		X	

- Here is some additional information about these conversions:
- Conversions between chunky and planar pixel organizations occur in either direction.
- Conversions between CPU and GPU memory types can occur in either direction.
- Conversions between different orderings of components occur in either direction, for example, BGR → RGB.
- For RGBA (or BGRA) destinations, most implementations do not change the alpha channel, so we recommend that you set it at the initialization time with

```
[cuda]memset(im.pixels,  -1, im.pitch * im.height) or
[cuda]memset(im.pixels, -1, im.pitch * im.height*
                              im.numComponents)
```

  or chunky and planar u8 images, respectively.

- Other than pitch, if no conversion is necessary, all pixel format transfers are implemented, with `cudaMemcpy2DAsync()`.

  Another restriction in YUV→YUV transfers is that the formats, layouts and colorspaces must match between `src` and `dst`.
- YUV420 and YUV422 and YUV444 sources have several variants.

  The colorspace must be set manually prior to Transfer.

  See "YUV Color Spaces" on page 62 for more information.
- There are also RGBf16→RGBf32 and RGBf32→RGBf16 transfers.
- CPU→CPU transfers are synchronous.
- Additionally, when the `src` and `dst` formats are the same, all formats are accommodated on CPU and GPU, and this can be used as a replacement for `cudaMemcpy2DAsync()` (which it utilizes).
- If the `src` and `dst` have different sizes, the transfer still occurs, but it will be clipped to the smaller size.

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 to synchronize, the `cudaStreamSynchronize()` function can be called.

4.5.14 NvCVImage_TransferRect

```
NvCV_Status NvCVImage_TransferRect(
  const NvCVImage *src,
  const NvCVRect2i *srcRect,
  NvCVImage *dst,
  const NvCVPoint2i *dstPt,
  float scale,
  CUstream stream,
  NvCVImage *tmp
);
```

4.5.14.1 Parameters

`src [in]`

Type: `const NvCVImage *`

Pointer to the source image that will be transferred.

`srcRect [in]`

Type: `const NvCVRect2i *`

Pointer to the source image rectangle that will be transferred.

```
typedef struct NvCVRect2i { int x, y, width, height; } NvCVRect2i;
```

If this is NULL, the entire src image rectangle is used.

`dst [out]`

Type: `NvCVImage *`

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

`dstPt [in]`

Type: `const NvCVPoint2i *`

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

```
typedef struct NvCVPoint2i { int x, y; } NvCVPoint2i;
```

If this is NULL, the image is transferred to (0,0).

`scale [in]`

Type: `float`

A scale factor that can be applied when the component type of the source or destination image is 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

This parameter is ignored if the component type of all images is the same (all integer or all floating-point).

`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 GPU memory that is required only if the source image is being converted **and** if the memory types of the source and destination images are different. The buffer has the same characteristics as the CPU image but 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 the 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 as needed 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` can be `NULL`. However, if `tmp` is `NULL`, and a temporary GPU buffer is required, an ephemeral buffer is allocated with a resultant degradation in performance for image sequences.

4.5.14.2 Return Value

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

4.5.14.3 Remarks

This function is like `NvCVImage_Transfer()`, because they share the same code. A rectangle can be copied by combining `NvCVImage_InitView()` with `NvCVImage_Transfer()`, but this only works for chunky images.

`NvCVImage_TransferRect` works on the chunky, planar, and semi-planar image layouts, and there is no difference in performance.

If you are not careful when you copy YUV rectangles, unexpected clipping will occur:

- YUV420 must have even x, y, width and height.
- YUV422 must have even x and width.

4.5.15 NvCVImage_TransferFromYUV

```
NvCV_Status NvCVImage_TransferFromYUV(
  const void *y,                 int yPixBytes,  int yPitch,
  const void *u, const void *v, int uvPixBytes, int uvPitch,
  NvCVImage_PixelFormat yuvFormat, NvCVImage_ComponentType yuvType,
  unsigned yuvColorSpace, unsigned yuvMemSpace,
  NvCVImage *dst, const NvCVRect2i *dstRect,
  float scale, struct CUstream_st *stream, NvCVImage *tmp
);
```

4.5.15.1 Parameters

`y [in]`

Type: `const void *`

Pointer to pixel(0,0) of the luminance channel.

`yPixBytes [in]`

Type: `int`

`The byte stride between y pixels horizontally.`

`yPitch [in]`

Type: `int`

`The byte stride between y pixels vertically.`

`u [in]`

Type: `const void *`

`Pointer to pixel(0,0) of the u (Cb) chrominance channel.`

`v [in]`

Type: `const void *`

`Pointer to pixel(0,0) of the v (Cr) chrominance channel.`

`uvPixBytes [in]`

Type: `int`

`The byte stride between u or v pixels horizontally.`

`uvPitch [in]`

Type: `int`

`The byte stride between u or v pixels vertically.`

`yuvColorSpace [in]`

Type: unsigned `int`

`The yuv colorspace, which specifies the range, the chromaticities, and the chrominance phase.`

`yuvMemSpace   [in]`

Type: unsigned `int`

`The byte stride between y pixels horizontally.`

`dst [out]`

Type: `NvCVImage *`

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

`dstRect [in]`

Type: `const NvCVRect2i *`

Pointer to the destination image rectangle

```
typedef struct NvCVRect2i { int x, y, width, height; } NvCVRect2i;
```

to which the image will be transferred. This can be NULL, in which case the entire dst image is transferred.

`scale [in]`

Type: `float`

A scale factor that can be applied when the component type of the source or destination image is 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

When 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 GPU memory that is required only if the source image is being converted **and** if the memory types of the source and destination images are different. The buffer has the same characteristics as the CPU image but 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 the 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 as needed 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` can be `NULL`. However, if `tmp` is `NULL`, and a temporary GPU buffer is required, an ephemeral buffer is allocated with a resultant degradation in performance for image sequences.

4.5.15.2 Return Value

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

4.5.15.3 Remarks

This function is like `NvCVImage_TransferRect()`, which can also copy from YUV images. The difference is that `TransferRect` works with images that have a structure, as described in the layout (or planar) parameter, and `NvCVImage_TransferFromYUV` works with images that have no structure that is represented in the taxonomy of the layout parameter. Since the structure is not known, `TransferFromYUV` is also slower than TransferRect when transferring from CPU→GPU.

4.5.16 NvCVImage_TransferToYUV

```
NvCV_Status NvCVImage_TransferToYUV(
  const NvCVImage *src,             const NvCVRect2i *srcRect,
  const void *y,                    int yPixBytes,  int yPitch,
  const void *u, const void *v,     int uvPixBytes, int uvPitch,
  NvCVImage_PixelFormat yuvFormat, NvCVImage_ComponentType yuvType,
  unsigned yuvColorSpace,           unsigned yuvMemSpace,
  float scale,
  CUstream stream,
  NvCVImage *tmp
);
```

4.5.16.1 Parameters

`src [in]`

Type: `const NvCVImage *`

Pointer to the source image that will be transferred.

`srcRect [in]`

Type: `const NvCVRect2i *`

Pointer to the source image rectangle that will be transferred.

```
typedef struct NvCVRect2i { int x, y, width, height; } NvCVRect2i;
```

If this is NULL, the entire src image rectangle is used.

`y [out]`

Type: `NvCVImage *`

Pointer to pixel(0,0) of the luminance channel.

`yPixBytes [in]`

Type: `int`

The byte stride between y pixels horizontally.

`yPitch [in]`

Type: `in`

The byte stride between y pixels vertically.

`u [out]`

Type: `NvCVImage *`

Pointer to pixel(0,0) of the u (Cb) chrominance channel.

`v [out]`

Type: `NvCVImage *`

Pointer to pixel(0,0) of the v (Cr) chrominance channel.

`uvPixBytes [in]`

Type: `int`

The byte stride between u or v pixels horizontally.

`uvPitch [in]`

Type: `int`

The byte stride between u or v pixels vertically.

`yuvColorSpace [in]`

Type: unsigned `int`

The yuv colorspace, which specifies the range, the chromaticities, and the chrominance phase.

`yuvMemSpace [in]`

Type: unsigned `int`

The memory space where the pixel buffers reside.

`scale [in]`

Type: `float`

A scale factor that can be applied when the component type of the source or destination image is 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

This parameter is ignored if the component type of all images is the same (all integer or all floating-point).

`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 GPU memory that is required only if the source image is being converted **and** if the memory types of the source and destination images are different. The buffer has the same characteristics as the CPU image but 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 the 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 as needed 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` can be `NULL`. However, if `tmp` is `NULL`, and a temporary GPU buffer is required, an ephemeral buffer is allocated with a resultant degradation in performance for image sequences.

4.5.16.2 Return Value

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

4.5.16.3 Remarks

This function is like `NvCVImage_TransferRect()`, which can also copy to YUV images. The difference is that `TransferRect` works with images that have a structure, as described in the layout (or planar) parameter, and `NvCVImage_TransferToYUV` works with images that have no structure that is represented in the taxonomy of the layout parameter.

4.5.17 NvCVImage_MapResource

```
NvCV_Status NvCVImage_MapResource(
  NvCVImage *im,
  struct CUstream_st *stream
);
```

4.5.17.1 Parameters

`im [in,out]`

Type: `NvCVImage *`

The image to be mapped.

`stream [out]`

Type: `struct CUStream_st *`

The stream on which the mapping is to be performed.

4.5.17.2 Return Value

- `NVCV_SUCCESS` on success.

4.5.17.3 Remarks

Between rendering by a graphics system and Transfer by CUDA, you also need to map the texture resource. This process involves quite a bit of overhead, so its use should be minimized. Every call to `NvCVImage_MapResource()` should be matched by a subsequent call to `NvCVImage_UnmapResource()`.

One way to create an image-wrapped resource on Windows is to call `NVCVImage_InitFromD3DTexture()`.

4.5.18 NvCVImage_UnmapResource

```
NvCV_Status NvCVImage_UnmapResource(
  NvCVImage *im,
  struct CUstream_st *stream
);
```

4.5.18.1 Parameters

`im [in,out]`

Type: `NvCVImage *`

The image to be mapped.

`stream [out]`

Type: `struct CUStream_st *`

The stream on which the mapping is to be performed.

4.5.18.2 Return Value

- `NVCV_SUCCESS` on success.

4.5.18.3 Remarks

Between rendering by a graphics system and Transfer by CUDA, you also need to map the texture resource. This process involves quite a bit of overhead, so its use should be minimized. Every call to `NvCVImage_MapResource()` should be matched by a subsequent call to `NvCVImage_UnmapResource()`.

One way to create an image-wrapped resource on Windows is to call `NvCVImage_InitFromD3DTexture()`.

4.5.19 NvCVImage_InitFromD3DTexture

```
NvCV_Status NvCVImage_InitFromD3DTexture(
  NvCVImage *im,
  struct ID3D11Texture2D *tx
);
```

4.5.19.1 Parameters

`im [in,out]`

Type: `NvCVImage *`

The image to be initialized.

`tx [in]`

Type: `struct ID3D11Texture2D *`

The texture to be used for initialization.

4.5.19.2 Return Value

- `NVCV_SUCCESS` on success.

4.5.19.3 Remarks

You can initialize an `NvCVImage` from a D3D11 texture. The `pixelFormat` and component types are transferred, and a `cudaGraphicsResource` is registered. The `NvCVImage` destructor unregisters the resource.

Note: This is designed to work with `NvCVImage_Transfer()`.

Before you allow the D3D texture to render into the `NvCVImage`, you need to first call `NvCVImage_MapResource()` and `NvCVImage_UnmapResource()`.

4.5.20 NVWrapperForCVMat

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

4.5.20.1 Parameters

`cvIm [in]`

Type: `const cv::Mat *`

Pointer to an allocated OpenCV image.

`vfxIm [out]`

Type: `NvCVImage *`

Pointer to an empty `NvCVImage` object, 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.

4.5.20.2 Return Value

Does not return a value.

4.5.20.3 Remarks

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

4.6 Image Functions for C++ Only

The image API provides constructors, 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 layout,
  unsigned memSpace,
  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, in pixels, of the image.

`height [in]`

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.

`layout [in]`

Type: `unsigned`

The organization of the components of the pixels in the image. See "Pixel Organizations" on page 60 for more information.

`memSpace [in]`

Type: `unsigned`

The type of memory in which the image data buffers are to be stored. See "Memory Types" on page 63 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.
- 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 value of `alignment`.

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 will be taken.

`x [in]`

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

`y [in]`

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

`width [in]`

Type: `unsigned`

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

`height [in]`

Type: `unsigned`

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

4.6.2 NvCVImage Destructor

```
~NvCVImage();
```

4.6.3 copyFrom

This version copies an entire image to another image. This version 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 will 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

- `NVCV_SUCCESS` on success.
- `NVCV_ERR_PIXELFORMAT` when the pixel format is not supported.
- `NVCV_ERR_MISMATCH` when 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 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 that are stored in different memory types as follows:

- 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 Return Codes

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

`NVCV_SUCCESS = 0`

Successful execution.

`NVCV_ERR_GENERAL`

Generic error code, which indicates that the function failed to execute for an unspecified reason.

`NVCV_ERR_UNIMPLEMENTED`

The requested feature is not implemented.

`NVCV_ERR_MEMORY`

The requested operation requires more memory than is available.

`NVCV_ERR_EFFECT`

An invalid effect handle has been supplied.

`NVCV_ERR_SELECTOR`

The specified selector is not valid for this effect filter.

`NVCV_ERR_BUFFER`

No image buffer has been specified.

`NVCV_ERR_PARAMETER`

An invalid parameter value has been supplied for this combination of effect and selector string.

`NVCV_ERR_MISMATCH`

Some parameters, for example, image formats or image dimensions, are not correctly matched.

`NVCV_ERR_PIXELFORMAT`

The specified pixel format is not supported.

`NVCV_ERR_MODEL`

An error occurred while the TRT model was being loaded.

`NVCV_ERR_LIBRARY`

An error while the dynamic library was being loaded.

`NVCV_ERR_INITIALIZATION`

The effect has not been properly initialized.

`NVCV_ERR_FILE`

The specified 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_RESOLUTION`

The specified image resolution is not supported.

`NVCV_ERR_UNSUPPORTEDGPU`

The GPU is not supported.

`NVCV_ERR_WRONGGPU`

The current GPU is not the one selected.

`NVCV_ERR_UNSUPPORTEDDRIVER`

The currently installed graphics driver is not supported.

`NVCV_ERR_MODELDEPENDENCIES`

There is no model with dependencies that match this system

`NVCV_ERR_PARSE`

There has been a parsing or syntax error while reading a file

`NVCV_ERR_MODELSUBSTITUTION`

The specified model does not exist and has been substituted.

`NVCV_ERR_READ`

An error occurred while reading a file.

`NVCV_ERR_WRITE`

An error occurred while writing a file.

`NVCV_ERR_PARAMREADONLY`

The selected parameter is read-only.

`NVCV_ERR_TRT_ENQUEUE`

TensorRT enqueue failed.

`NVCV_ERR_TRT_BINDINGS`

Unexpected TensorRT bindings.

`NVCV_ERR_TRT_CONTEXT`

An error occurred while creating a TensorRT context.

`NVCV_ERR_NPP`

An error has occurred in the NPP library.

`NVCV_ERR_CUDA_MEMORY`

The requested operation requires more CUDA memory than is available.

`NVCV_ERR_CUDA_VALUE`

A CUDA parameter is not within its acceptable range.

`NVCV_ERR_CUDA_PITCH`

A CUDA pitch is not within its acceptable range.

`NVCV_ERR_CUDA_INIT`

The CUDA driver and runtime could not be initialized.

`NVCV_ERR_CUDA_LAUNCH`

The CUDA kernel failed to launch.

`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 attempted to load or store on an invalid memory address.

`NVCV_ERR_CUDA`

An unspecified CUDA error has occurred.

There are a some other CUDA-related errors that are not listed here. However, the function `NvCV_GetErrorStringFromCode()` will turn the error code into a string to help you debug.

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:

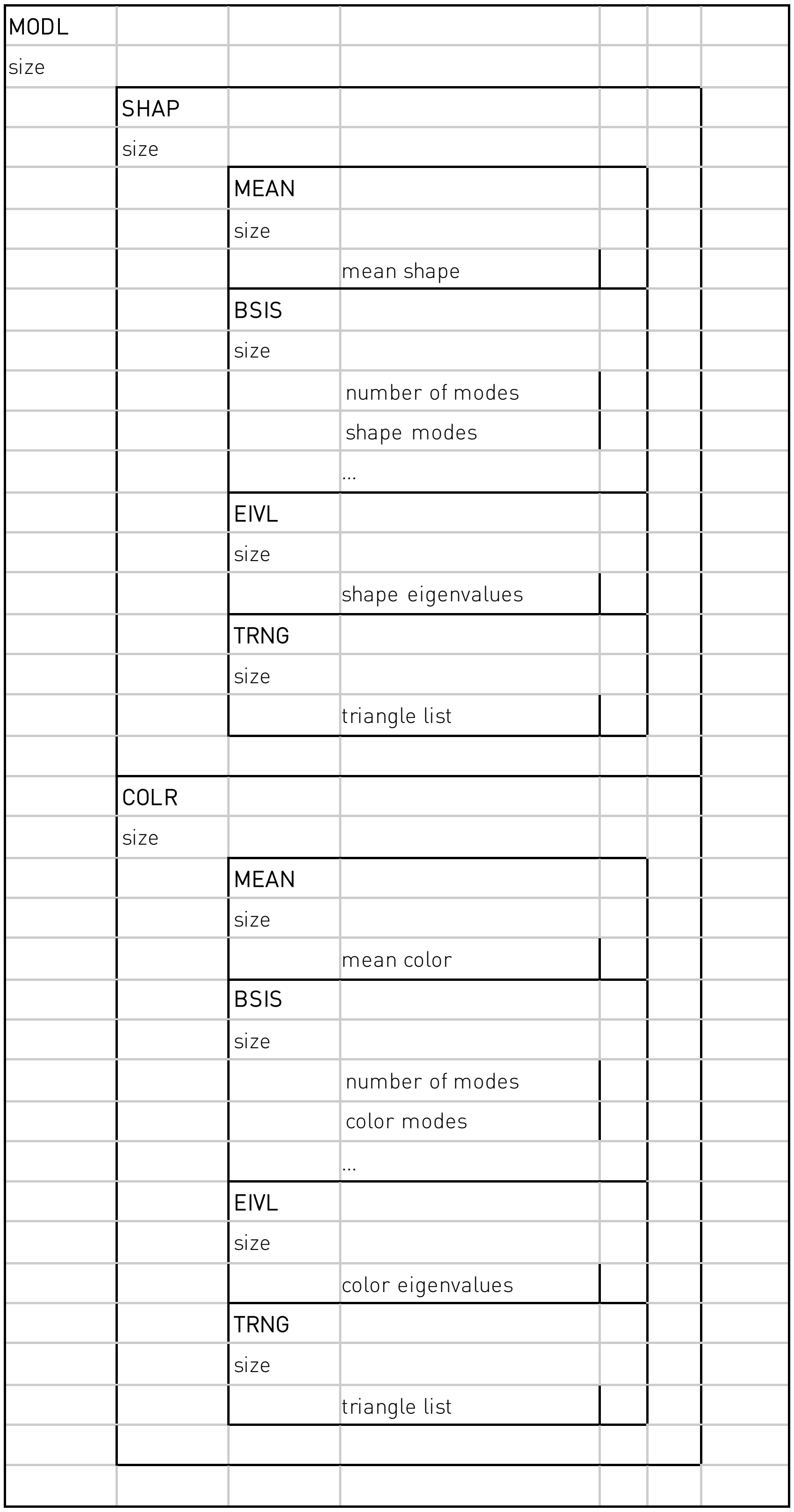
- The name `NFAC`
- `size=8`
- `endian=0xe4` (little endian)
- `sizeBits=32`
- `indexBits=16`
- The offset of the table of contents

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

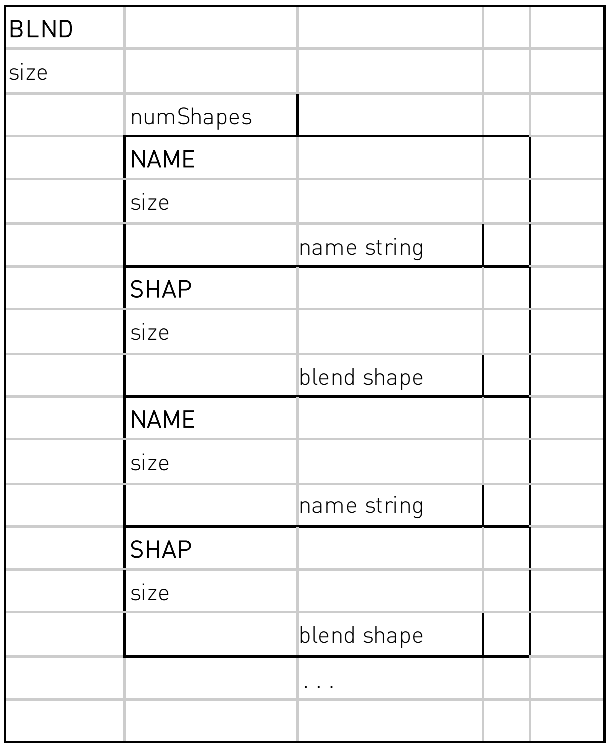
The IBUG mappings object contains the following information:

- Landmarks
- Right contour
- Left contour

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

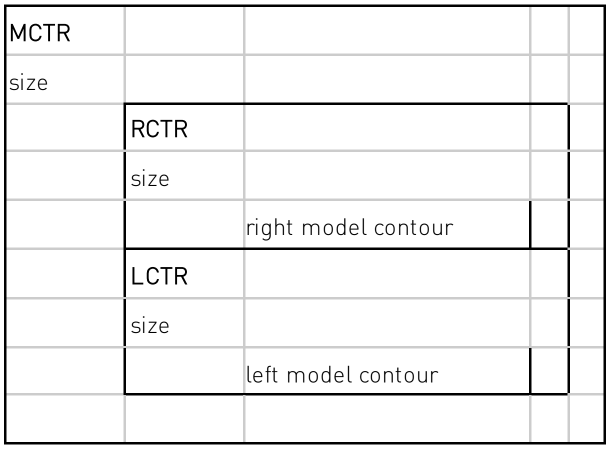
A.4 Blend Shapes Object

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



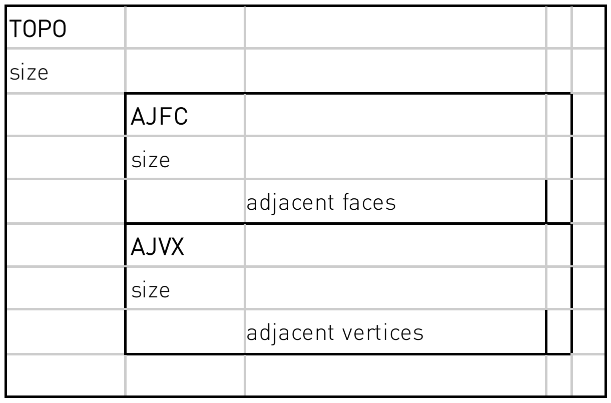
A.5 Model Contours Object

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



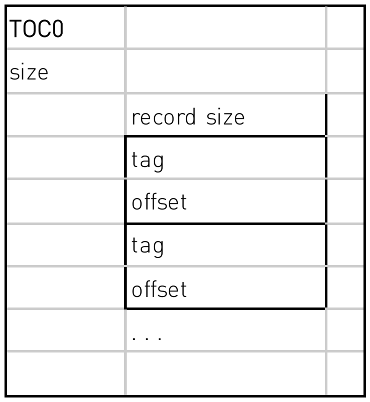
A.6 Topology Object

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



Appendix B. 3D Body Pose Keypoint Format

The 3D Body Pose consists of 34 Keypoints for Body Pose Tracking.

Appendix B.

B.1 34 Keypoints of Body Pose Tracking

The 34 Keypoints of Body Pose tracking are pelvis, left hip, right hip, torso, left knee, right knee, neck, left ankle, right ankle, left big toe, right big toe, left small toe, right small toe, left heel, right heel, nose, left eye, right eye, left ear, right ear, left shoulder, right shoulder, left elbow, right elbow, left wrist, right wrist, left pinky knuckle, right pinky knuckle, left middle tip, right middle tip, left index knuckle, right index knuckle, left thumb tip, right thumb tip.

B.2 NvAR_Parameter_Output(KeyPoints) Order

The Keypoints order of the output from `NvAR_Parameter_Output(KeyPoints)` are same as mentioned in "34 Keypoints of Body Pose Tracking".