NVIDIA AR SDK

Programming Guide
Table of Contents

Chapter 1. NVIDIA AR SDK API Architecture.................................................................1
  1.1. Using the NVIDIA AR SDK in Applications.........................................................1
  1.2. Creating an Instance of a Feature Type...............................................................1
  1.3. Getting and Setting Properties for a Feature Type.............................................2
    1.3.1. Setting Up the CUDA Stream.................................................................2
    1.3.2. Summary of NVIDIA AR SDK Accessor Functions.................................3
    1.3.3. Key Values in the Properties of a Feature Type.........................................3
      1.3.3.1. Configuration Properties...............................................................4
      1.3.3.2. Input Properties...............................................................................6
      1.3.3.3. Output Properties..........................................................................6
    1.3.4. Getting the Value of a Property of a Feature.............................................9
    1.3.5. Setting a Property for a Feature..............................................................9
    1.3.6. Loading a Feature Instance.....................................................................10
    1.3.7. Running a Feature Instance.....................................................................10
    1.3.8. Destroying a Feature Instance..................................................................10
  1.4. Working with Image Frames on GPU or CPU Buffers.........................................10
    1.4.1. Converting Image Representations to NvCVImage Objects.......................11
      1.4.1.1. Converting OpenCV Images to NvCVImage Objects..........................11
      1.4.1.2. Converting Other Image Representations to NvCVImage Objects........11
      1.4.1.3. Converting Decoded Frames from the NvDecoder to NvCVImage Objects..12
      1.4.1.4. Converting an NvCVImage Object to a Buffer that can be Encoded by NvEncoder12
    1.4.2. Allocating an NvCVImage Object Buffer..................................................13
      1.4.2.1. Using the NvCVImage Allocation Constructor to Allocate a Buffer..........13
      1.4.2.2. Using Image Functions to Allocate a Buffer......................................14
    1.4.3. Transferring Images Between CPU and GPU Buffers.................................14
      1.4.3.1. Transferring Input Images from a CPU Buffer to a GPU Buffer...........15
      1.4.3.2. Transferring Output Images from a GPU Buffer to a CPU Buffer.........15
  1.5. List of Properties for the AR SDK Features.......................................................16
    1.5.1. Face Tracking Property Values..................................................................16
    1.5.2. Landmark Tracking Property Values.......................................................17
    1.5.3. Face 3D Mesh Tracking Property Values................................................19
    1.5.4. Body Detection Property Values............................................................23
    1.5.5. 3D Body Pose Keypoint Tracking Property Values.................................24
  1.6. Using the AR Features.........................................................................................26
    1.6.1. Face Detection and Tracking....................................................................26
      1.6.1.1. Face Detection for Static Frames (Images)......................................26
2.2.8. NvAR_GetF32.................................................................................................................... 47
2.2.9. NvAR_GetF64.................................................................................................................... 48
2.2.10. NvAR_GetF32Array......................................................................................................... 49
2.2.11. NvAR_GetObject.............................................................................................................. 50
2.2.12. NvAR_GetS32.................................................................................................................. 51
2.2.13. NvAR_GetString.............................................................................................................. 52
2.2.14. NvAR_GetU32.................................................................................................................. 52
2.2.15. NvAR_GetU64.................................................................................................................. 53
2.2.16. NvAR_SetCudaStream.................................................................................................... 54
2.2.17. NvAR_SetF32................................................................................................................... 55
2.2.18. NvAR_SetF64................................................................................................................... 56
2.2.19. NvAR_SetF32Array.......................................................................................................... 57
2.2.20. NvAR_SetObject.............................................................................................................. 58
2.2.21. NvAR_SetS32.................................................................................................................. 59
2.2.22. NvAR_SetString............................................................................................................... 60
2.2.23. NvAR_SetU32.................................................................................................................. 61
2.2.24. NvAR_SetU64.................................................................................................................. 62
2.3. Return Codes........................................................................................................................... 63

Appendix A. NVIDIA 3DMM File Format............................................................................. 65
A.1. Header..................................................................................................................................... 65
A.2. Model Object........................................................................................................................... 66
A.3. IBUG Mappings Object............................................................................................................ 66
A.4. Blend Shapes Object.............................................................................................................. 67
A.5. Model Contours Object........................................................................................................... 67
A.6. Topology Object....................................................................................................................... 68
A.7. Table of Contents Object........................................................................................................ 68

Appendix B. 3D Body Pose Keypoint Format......................................................................... 69
B.1. 34 Keypoints of Body Pose Tracking..................................................................................... 69
B.2. NvAR_Parameter_Output(KeyPoints) Order.......................................................................... 69
Chapter 1. NVIDIA AR SDK API Architecture

This section provides information about the NVIDIA® AR SDK API architecture.

1.1. Using the 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.

1.2. 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.

3. To create an instance of the face detection feature type, run the following example:

   ```
   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.
1.3. Getting and Setting Properties for a Feature Type

To prepare to load and run an instance of a feature type, you need to set the properties that the instance requires.

Here are some of the properties:

- 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](#) 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](#) for a complete list of get and set functions.

1.3.1. 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 the second 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.
   - The key value `NVAR_Parameter_Config(CUDAStream)`.

   See [Creating an Instance of a Feature Type](#) for more information.

   - The CUDA stream that you created in the previous step.

   See [Key Values in the Properties of a Feature Type](#) for more information.

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

```c
CUstream stream;
nvErr = NvAR_CudaStreamCreate (&stream);
nvErr = NvAR_SetCudaStream(featureHandle, NVAR_Parameter_Config(CUDAStream), stream);
```
1.3.2. **Summary of NVIDIA AR SDK Accessor Functions**

The following table provides the details about the SDK accessor functions.

<table>
<thead>
<tr>
<th>Property Type</th>
<th>Data Type</th>
<th>Set and Get Accessor Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit unsigned integer</td>
<td>unsigned int</td>
<td>NvAR_SetU32()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NvAR_GetU32()</td>
</tr>
<tr>
<td>32-bit signed integer</td>
<td>int</td>
<td>NvAR_SetS32()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NvAR_GetS32()</td>
</tr>
<tr>
<td>Single-precision [32-bit] floating-point number</td>
<td>float</td>
<td>NvAR_SetF32()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NvAR_GetF32()</td>
</tr>
<tr>
<td>Double-precision [64-bit] floating point number</td>
<td>double</td>
<td>NvAR_SetF64()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NvAR_GetF64()</td>
</tr>
<tr>
<td>64-bit unsigned integer</td>
<td>unsigned long long</td>
<td>NvAR_SetU64()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NvAR_GetU64()</td>
</tr>
<tr>
<td>Floating-point array</td>
<td>float*</td>
<td>NvAR_SetFloatArray()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NvAR_GetFloatArray()</td>
</tr>
<tr>
<td>Object</td>
<td>void*</td>
<td>NvAR_SetObject()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NvAR_GetObject()</td>
</tr>
<tr>
<td>Character string</td>
<td>const char*</td>
<td>NvAR_SetString()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NvAR_GetString()</td>
</tr>
<tr>
<td>CUDA stream</td>
<td>CUstream</td>
<td>NvAR_SetCudaStream()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NvAR_GetCudaStream()</td>
</tr>
</tbody>
</table>

1.3.3. **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](#) for more information.
- **NvAR_Parameter_Input** indicates an input property.
  See [Input Properties](#) for more information.
NvAR_Parameter_Output indicates an output property. See Output Properties 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 the Summary of NVIDIA AR SDK Accessor Functions table.

1.3.3.1. Configuration Properties
Here are the configuration properties in the AR SDK:

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(CUDAStruct)
- 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.
- 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.
NvAR_Parameter_Config(LandmarksConfidence_Size)
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(Mode)
Mode to select High Performance or High Quality for 3D Body Pose.
String equivalent: NvAR_Parameter_Config_Mode
Property type: unsigned int

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*)
1.3.3.2. Input Properties
Here are the input properties in the AR SDK:

**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_Input_BoundingBoxes
- Property type: object (void*)

1.3.3.3. Output Properties
Here are the output properties in the AR SDK:

**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. See 3D Body Pose Keypoint Format 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. See 3D Body Pose Keypoint Format 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 output detected 3D 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 keypoints for 3D Body Pose. See 3D Body Pose Keypoint Format for information. The order of the points in the CPU buffer follows the order mentioned in 3D Body Pose Keypoint Format.

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 3D Body Pose. See 3D Body Pose Keypoint Format for information. The order of the points in the CPU buffer follows the order mentioned in 3D Body Pose Keypoint Format.

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

1.3.4. 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:

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

1.3.5. Setting a Property for a Feature

The following steps explain how 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:

```c
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:

```c
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));
```

Note: It sets up an 8-bit chunky/interleaved BGR array.
See List of Properties for AR Features 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.

### 1.3.6. 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 for more information.

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

```c
NvAR_Load(faceDetectHandle);
```

### 1.3.7. 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 for more information.

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

```c
NvAR_Run(faceDetectHandle);
```

### 1.3.8. 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 for more information.

```c
NvAR_Destroy(faceDetectHandle);
```

### 1.4. 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. The AR SDK
provides functions for converting an image representation to NvCVImage and transferring images between CPU and GPU buffers.

For more information about NvCVImage, see the NvCVImage API Guide. This section provides a synopsis of the most frequently used functions with the AR SDK.

1.4.1. Converting Image Representations to NvCVImage Objects

The 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.

1.4.1.1. Converting OpenCV Images to NvCVImage Objects

You can use the wrapper functions that the AR SDK provides specifically for RGB OpenCV images.

Note: The 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.

```cpp
// 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.

```cpp
// 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);
```

1.4.1.2. Converting Other Image Representations to NvCVImage Objects

To convert other image representations, call the NvCVImage_Init() function to place a wrapper around an existing buffer [srcPixelBuffer].

NvCVImage src_gpu;
1.4.1.3. Converting Decoded Frames from the NvDecoder to NvCVImage Objects

To convert 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 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.

1.4.1.4. 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.
convert-nvcvimage-obj-buffer-encoded-nvencoder

The following sample shows a frame that is encoded in the BGRA pixel format.

```cpp
//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);
```

1.4.2. 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.

1.4.2.1. 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 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
);

1.4.2.2. Using Image Functions to Allocate a Buffer

By declaring an empty NvCVImage object, 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 of 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().

1.4.3. 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.
1.4.3.1. Transferring Input Images from a CPU Buffer to a GPU Buffer

Here are the steps to transfer 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.

   ```c
   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.
     ```c
     NvCVImage srcGpuStaging(inWidth, inHeight, srcCPUImg.pixelFormat, srcCPUImg.componentType, srcCPUImg.planar, NVCV_GPU)
     ```
   - To simplify your application program code, declare an empty staging buffer.
     ```c
     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.

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

1.4.3.2. Transferring Output Images from a GPU Buffer to a CPU Buffer

Here are the steps to transfer 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.

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

   For more information about `NvCVImage`, see the `NvCVImage API Guide`.

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.
     ```c
     NvCVImage dstGpuStaging(outWidth, outHeight, dstCPUImg.pixelFormat, dstCPUImg.componentType, dstCPUImg.planar, NVCV_GPU)
     ```
   - To simplify your application program code, declare an empty staging buffer:
     ```c
     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.

   ```c
   //Retrieve the image from the GPU to CPU, perhaps with conversion.
   NvCVImage_Transfer(&dstGpuPlanar, &dstCPUImg, 1.0f, stream, &dstGpuStaging);
   ```
1.5. List of Properties for the AR SDK Features

This section provides the properties and their values for the features in the AR SDK.

1.5.1. Face Tracking Property Values

The following tables list the values for the configuration, input, and output properties for face tracking.

### Table 2. Configuration Properties for Face Tracking

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FeatureDescription</td>
<td>String is free-form text that describes the feature. The string is set by the SDK and cannot be modified by the user.</td>
</tr>
<tr>
<td>CUDAStream</td>
<td>The CUDA stream, which is set by the user.</td>
</tr>
<tr>
<td>ModelDir</td>
<td>String that contains the path to the folder that contains the TensorRT package files. Set by the user.</td>
</tr>
<tr>
<td>Temporal</td>
<td>Unsigned integer, 1/0 to enable/disable the temporal optimization of face detection. If enabled, only one face is returned. See <a href="#">Face Detection and Tracking</a> for more information. Set by the user.</td>
</tr>
</tbody>
</table>

### Table 3. Input Properties for Face Tracking

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>Interleaved (or chunky) 8-bit BGR input image in a CUDA buffer of type NvCVImage. To be allocated and set by the user.</td>
</tr>
</tbody>
</table>
### Table 4. Output Properties for Face Tracking

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoundingBoxes</td>
<td>NvAR_BBoxes structure that holds the detected face boxes.</td>
</tr>
<tr>
<td></td>
<td>To be allocated by the user.</td>
</tr>
<tr>
<td>BoundingBoxesConfidence</td>
<td>An array of single-precision (32-bit) floating-point numbers that contains the confidence values for each detected face box.</td>
</tr>
<tr>
<td></td>
<td>To be allocated by the user.</td>
</tr>
</tbody>
</table>

### 1.5.2. Landmark Tracking Property Values

The following tables list the values for the configuration, input, and output properties for landmark tracking.

### Table 5. Configuration Properties for Landmark Tracking

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FeatureDescription</td>
<td>String that describes the feature.</td>
</tr>
<tr>
<td>CUDAStream</td>
<td>The CUDA stream.</td>
</tr>
<tr>
<td></td>
<td>Set by the user.</td>
</tr>
<tr>
<td>ModelDir</td>
<td>String that contains the path to the folder that contains the TensorRT package files.</td>
</tr>
<tr>
<td></td>
<td>Set by the user.</td>
</tr>
<tr>
<td>BatchSize</td>
<td>The number of inferences to be run at one time on the GPU.</td>
</tr>
<tr>
<td></td>
<td>The maximum value is 1.</td>
</tr>
<tr>
<td>Landmarks_Size</td>
<td>Unsigned integer, 68 or 126.</td>
</tr>
<tr>
<td></td>
<td>Specifies the number of landmark points (X and Y values) to be returned.</td>
</tr>
<tr>
<td></td>
<td>Set by the user.</td>
</tr>
<tr>
<td>LandmarksConfidence_Size</td>
<td>Unsigned integer, 68 or 126.</td>
</tr>
<tr>
<td></td>
<td>Specifies the number of landmark confidence values for the detected keypoints to be returned.</td>
</tr>
</tbody>
</table>
### Table 6. Input Properties for Landmark Tracking

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>Interleaved (or chunky) 8-bit BGR input image in a CUDA buffer of type NvCVImage. To be allocated and set by the user.</td>
</tr>
<tr>
<td>BoundingBoxes</td>
<td>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 for more information. To be allocated by the user.</td>
</tr>
</tbody>
</table>

### Table 7. Output Properties for Landmark Tracking

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Landmarks</td>
<td>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.</td>
</tr>
<tr>
<td>Pose</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Property Name | Value
---|---
LandmarksConfidence | 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 | `NvAR_BBoxes` structure that contains the detected face through face detection performed by the landmark detection feature. See Landmark Detection and Tracking for more information.
To be allocated by the user.

#### 1.5.3. Face 3D Mesh Tracking Property Values

The following tables list the values for the configuration, input, and output properties for Face 3D Mesh tracking.

**Table 8. Configuration Properties for Face 3D Mesh Tracking**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Value</th>
</tr>
</thead>
</table>
| FeatureDescription | String that describes the feature.  
This property is read-only.                                      |
| ModelDir        | String that contains the path to the face model, and the TensorRT package files. See Alternative Usage of the Face 3D Mesh Feature for more information.  
Set by the user.                                            |
| CUDAStream      | The CUDA stream.                                                      
See Alternative Usage of the Face 3D Mesh Feature for more information.  
Set by the user.                                            |
<p>| Temporal        | Unsigned integer, 1/0 to enable/disable the temporal optimization of face and landmark detection. See Alternative Usage of the Face 3D Mesh Feature for more information. |</p>
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set by the user.</td>
<td></td>
</tr>
<tr>
<td>LandmarksConfidence_Size</td>
<td>Unsigned integer, 68 or 126.</td>
</tr>
<tr>
<td></td>
<td>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 for more information.</td>
</tr>
<tr>
<td>ShapeEigenValueCount</td>
<td>The number of eigenvalues that describe the identity shape. Query this to determine how big the eigenvalue array should be, if that is a desired output.</td>
</tr>
<tr>
<td></td>
<td>This property is read-only.</td>
</tr>
<tr>
<td>ExpressionCount</td>
<td>The number of expressions available in the chosen model. Query this to determine how big the expression coefficient array should be, if that is a desired output.</td>
</tr>
<tr>
<td></td>
<td>This property is read-only.</td>
</tr>
<tr>
<td>VertexCount</td>
<td>The number of vertices in the chosen model. Query this property to determine how big the vertex array should be, where VertexCount is the number of vertices.</td>
</tr>
<tr>
<td></td>
<td>This property is read-only.</td>
</tr>
<tr>
<td>TriangleCount</td>
<td>The number of triangles in the chosen model. Query this property to determine how big the triangle array should be, where TriangleCount is the number of triangles.</td>
</tr>
<tr>
<td></td>
<td>This property is read-only.</td>
</tr>
</tbody>
</table>

Table 9. Input Properties for Face 3D Mesh Tracking

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width</td>
<td>The width of the input image buffer that contains the face to which the face model will be fitted. Set by the user.</td>
</tr>
<tr>
<td>Property Name</td>
<td>Value</td>
</tr>
<tr>
<td>---------------</td>
<td>-------</td>
</tr>
<tr>
<td>Height</td>
<td>The height of the input image buffer that contains the face to which the face model will be fitted. Set by the user.</td>
</tr>
<tr>
<td>Landmarks</td>
<td>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 for more information. To be allocated by the user.</td>
</tr>
<tr>
<td>Image</td>
<td>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 for more information. To be allocated by the user.</td>
</tr>
</tbody>
</table>

Table 10. Output Properties for Face 3D Mesh Tracking

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FaceMesh</td>
<td>NvAR_FaceMesh structure that contains the output face mesh. To be allocated by the user.</td>
</tr>
<tr>
<td>RenderingParams</td>
<td>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.</td>
</tr>
<tr>
<td>Landmarks</td>
<td>An NvAR_Point2f array, which must be large enough to hold the number of points of size NvAR_Parameter_Config(Landmarks_Size).</td>
</tr>
<tr>
<td>Property Name</td>
<td>Value</td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Pose</td>
<td>NVAR_Quaternion array pointer, to hold one quaternion. See Alternative Usage of the Face 3D Mesh Feature for more information. To be allocated by the user.</td>
</tr>
<tr>
<td>LandmarksConfidence</td>
<td>An array of single-precision [32-bit] floating-point numbers, which must be large enough to hold the number of confidence values of size NVAR_Parameter_Config(LandmarksConfidence_Size). See Alternative Usage of the Face 3D Mesh Feature for more information. To be allocated by the user.</td>
</tr>
<tr>
<td>BoundingBoxes</td>
<td>NVAR_BBoxes structure that contains the detected face that is determined internally. See Alternative Usage of the Face 3D Mesh Feature for more information. To be allocated by the user.</td>
</tr>
<tr>
<td>BoundingBoxesConfidence</td>
<td>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 for more information. To be allocated by the user.</td>
</tr>
<tr>
<td>ShapeEigenValues</td>
<td>Optional: The array into which the shape eigenvalues will be placed, if desired. Query ShapeEigenValueCount to determine how big this array should be. To be allocated by the user.</td>
</tr>
<tr>
<td>ExpressionCoefficients</td>
<td>Optional: The array into which the expression coefficients will be placed, if desired. Query ExpressionCount to determine how big this array should be. To be allocated by the user.</td>
</tr>
</tbody>
</table>
1.5.4. Body Detection Property Values

The following tables list the values for the configuration, input, and output properties for Body Detection tracking.

Table 11. Configuration Properties for Body Detection Tracking

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FeatureDescription</td>
<td>String is free-form text that describes the feature. The string is set by the SDK and cannot be modified by the user.</td>
</tr>
<tr>
<td>CUDAStream</td>
<td>The CUDA stream, which is set by the user.</td>
</tr>
<tr>
<td>ModelDir</td>
<td>String that contains the path to the folder that contains the TensorRT package files. Set by the user.</td>
</tr>
<tr>
<td>Temporal</td>
<td>Unsigned integer, 1/0 to enable/disable the temporal optimization of body detection. Set by the user.</td>
</tr>
</tbody>
</table>

Table 12. Input Properties for Body Detection

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>Interleaved (or chunky) 8-bit BGR input image in a CUDA buffer of type NvCVImage. To be allocated and set by the user.</td>
</tr>
</tbody>
</table>

Table 13. Output Properties for Body Detection

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoundingBoxes</td>
<td>NvAR_BBoxes structure that holds the detected body boxes. To be allocated by the user.</td>
</tr>
<tr>
<td>BoundingBoxesConfidence</td>
<td>An array of single-precision (32-bit) floating-point numbers that contains the confidence values for each detected body box.</td>
</tr>
</tbody>
</table>
1.5.5. 3D Body Pose Keypoint Tracking Property Values

The following tables list the values for the configuration, input, and output properties for 3D Body Pose Keypoint Tracking racking.

Table 14. Configuration Properties for 3D Body Pose Keypoint Tracking

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FeatureDescription</td>
<td>String that describes the feature.</td>
</tr>
<tr>
<td>CUDAStream</td>
<td>The CUDA stream.</td>
</tr>
<tr>
<td>ModelDir</td>
<td>String that contains the path to the folder that contains the TensorRT package files.</td>
</tr>
<tr>
<td>BatchSize</td>
<td>The number of inferences to be run at one time on the GPU. The maximum value is 1.</td>
</tr>
<tr>
<td>Mode</td>
<td>Unsigned integer, 0 or 1. Default is 1. Selects the High Performance (1) mode or High Quality (0) mode Set by the user.</td>
</tr>
<tr>
<td>UseCudaGraph</td>
<td>Bool, True or False. Default is True Flag to use CUDA Graphs for optimization. Set by the user.</td>
</tr>
<tr>
<td>FocalLength</td>
<td>Float. Default is 800.79041 Specifies the focal length of the camera to be used for 3D Body Pose. Set by the user.</td>
</tr>
</tbody>
</table>
### Table 15. Input Properties for 3D Body Pose Keypoint Tracking

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temporal</td>
<td>Unsigned integer and 1/0 to enable/disable the temporal optimization of Body Pose tracking. Set by the user.</td>
</tr>
<tr>
<td>NumKeyPoints</td>
<td>Unsigned integer. Specifies the number of keypoints available, which is currently 34.</td>
</tr>
<tr>
<td>ReferencePose</td>
<td>NvAR_Point3f array, which contains the reference pose for each of the 34 keypoints. Specifies the Reference Pose used to compute the joint angles.</td>
</tr>
</tbody>
</table>

### Table 16. Output Properties for 3D Body Pose Keypoint Tracking

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keypoints</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Property Name | Value
---|---
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.

### 1.6. Using the AR Features
This section provides information about how to use the AR features.

#### 1.6.1. Face Detection and Tracking
This section provides information about how to use the Face Detection and Tracking feature.

##### 1.6.1.1. 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:
1.6.1.2. 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 and Face 3D Mesh and Tracking for more information about how to use the Landmark Detection or Face3D Reconstruction AR features and return a face bounding box.

1.6.2. Landmark Detection and Tracking

This section provides information about how to use the Landmark Detection and Tracking feature.

1.6.2.1. 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. Currently, the maximum value is 1. 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:

```cpp
//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);
```

1.6.2.2. Alternative Usage of Landmark Detection

However, as described in Landmark Tracking Property Values, 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.

### 1.6.2.3. 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:

```c
// 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);
```

### 1.6.3. Face 3D Mesh and Tracking

This section provides information about how to use the Face 3D Mesh and Tracking feature.

#### 1.6.3.1. 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:
1.6.3.2. 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.

1.6.3.3. Face 3D Mesh Tracking for Temporal Frames (Videos)

If the Temporal flag is set and face and landmark detection are run internally, these features will be optimized 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);

1.6.4. 3D Body Pose Tracking
This feature relies on temporal information to track the person in the scene, where the keypoints information from the previous frame is used to estimate the keypoints of the next frame.

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.

1.6.4.1. 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 the image buffer as input.

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

```c
NvCVImage inputImageBuffer;
NvAR_SetObject(bodyDetectHandle, NvAR_Parameter_Input(Image), &inputImageBuffer, sizeof(NvCVImage));
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));
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:

```c
//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<float> keypoints_confidence;
std::vector<NvAR_Point3f> jointAngles;

// 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);
```

1.6.4.2. **3D Body Pose Tracking for Temporal Frames (Videos)**

The feature relies on temporal information to track the person in the 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:

```c
//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<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);

1.7. Using Multiple GPUs

Applications that are developed with the 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 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.

### 1.7.1. 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 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 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.

### 1.7.2. 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 a specific GPU in a multi-GPU environment. In this situation, ensure that the AR SDK does not override your choice of GPU for applying the video effect filter.

```c
// 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);
}
```

### 1.7.3. 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 AR filter. 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.

```c
// 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.

   ```c
   // 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 2. AR SDK API Reference

This section provides detailed information about the APIs in the AR SDK.

2.1. Structures

The structures in the 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.

2.1.1. NvAR_BBoxes

Here is detailed information about the `NvAR_BBoxes` structure.

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

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.
Remarks
This structure is returned as the output of the face detection feature.
Defined in: *nvAR_def.h*

### 2.1.2. NvAR_FaceMesh
Here is detailed information about the NvAR_FaceMesh structure.

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

**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.

Remarks
This structure is returned as an output of the Mesh Tracking feature.
Defined in: *nvAR_def.h*

### 2.1.3. NvAR_Frustum
Here is detailed information about the NvAR_Frustum structure.

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

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*

---

### 2.1.4. NvAR_FeatureHandle

Here is detailed information about the `NvAR_FeatureHandle` structure.

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

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*

---

### 2.1.5. NvAR_Point2f

Here is detailed information about the `NvAR_Point2f` structure.

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

x
Type: float
The X coordinate of the point in pixels.

y
Type: float
The Y coordinate of the point in pixels.

Remarks
This structure represents the X and Y coordinates of one point in 2D space.
Defined in: nvAR_defs.h

2.1.6. NvAR_Point3f
Here is detailed information about the NvAR_Point3f structure.

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

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.

Remarks
This structure represents the X, Y, Z coordinates of one point in 3D space.
Defined in: nvAR_defs.h
2.1.7. NvAR_Quaternion

Here is detailed information about the NvAR_Quaternion structure.

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

**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.

**Remarks**

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

Defined in: `nvAR_defs.h`

2.1.8. NvAR_Rect

Here is detailed information about the NvAR_Rect structure.

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

**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.

Remarks
This structure represents the position and size of a rectangular 2D bounding box.
Defined in: nvAR_defs.h

2.1.9. NvAR_RenderingParams
Here is detailed information about the NvAR_RenderingParams structure.

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

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.

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`

2.1.10. **NvAR_Vector2f**

Here is detailed information about the `NvAR_Vector2f` structure.

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

**Members**

`x`

Type: `float`

The X component of the 2D vector.

`y`

Type: `float`

The Y component of the 2D vector.

**Remarks**

This structure represents a 2D vector.

Defined in: `nvAR_defs.h`

2.1.11. **NvAR_Vector3f**

Here is detailed information about the `NvAR_Vector3f` structure.

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

**Members**

`vec`

Type: `float` array of size 3

A vector of size 3.

**Remarks**

This structure represents a 3D vector.

Defined in: `nvAR_defs.h`
2.1.12. NvAR_Vector3u16
Here is detailed information about the NvAR_Vector3u16 structure.

```c
typedef struct NvAR_Vector3u16 {
    unsigned short vec[3];
} NvAR_Vector3u16;
```

**Members**

- **vec**
  - **Type:** unsigned short array of size 3
  - A vector of size 3.

**Remarks**
This structure represents a 3D vector.
Defined in: `nvAR_defs.h`

2.2. Functions
This section provides information about the functions in the AR SDK.

2.2.1. NvAR_Create
Here is detailed information about the NvAR_Create structure.

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

**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.

**Return Value**
Returns one of the following values:

- **NVCV_SUCCESS** on success
2.2.2. NvAR_Destroy

Here is detailed information about the NvAR_Destroy structure.

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

**Parameters**

**handle [in]**

Type: NvAR_FeatureHandle

The handle to the feature instance to be released.

**Return Value**

Returns one of the following values:

- NVCV_SUCCESS on success
- NVCV_ERR_FEATURENOTFOUND

**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.

2.2.3. NvAR_Load

Here is detailed information about the NvAR_Load structure.

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

**Parameters**

**handle [in]**

Type: NvAR_FeatureHandle

The handle to the feature instance to load.
Return Value

Returns one of the following values:

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

Remarks

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

2.2.4. NvAR_Run

Here is detailed information about the NvAR_Run structure.

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

Parameters

handle[in]

Type: const NvAR_FeatureHandle

The handle to the feature instance to be run.

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

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 for more information.

2.2.5. NvAR_GetCudaStream

Here is detailed information about the NvAR_GetCudaStream structure.

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

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.

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

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 stream parameter.
2.2.6. **NvAR_CudaStreamCreate**

Here is detailed information about the NvAR_CudaStreamCreate structure.

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

**Parameters**

- `stream [out]`
  - Type: `CUstream *`
  - The location in which to store the newly allocated CUDA stream.

**Return Value**

Returns one of the following values:

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

**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.

2.2.7. **NvAR_CudaStreamDestroy**

Here is detailed information about the NvAR_CudaStreamDestroy structure.

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

**Parameters**

- `stream [in]`
  - Type: `CUstream`
  - The CUDA stream to destroy.

**Return Value**

Does not return a value.
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.

2.2.8. NvAR_GetF32

Here is detailed information about the NvAR_GetF32 structure.

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

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.

**val**
- **Type:** float *
- Pointer to the 32-bit floating-point number where the value retrieved is to be written.

Return Value

Returns one of the following values:

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

2.2.9. `NvAR_GetF64`
Here is detailed information about the `NvAR_GetF64` structure.

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

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*.

**val**

Type: `double *`

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

Return Value

Returns one of the following values:

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

2.2.10. NvAR_GetF32Array
Here is detailed information about the NvAR_GetF32Array structure.

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

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 Key Values in the Properties of a Feature Type 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.

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
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.

2.2.11. NvAR_GetObject
Here is detailed information about the NvAR_GetObject structure.

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

Parameters

handle

Type: NvAR_FeatureHandle

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

name

Type: const char *

See Key Values in the Properties of a Feature Type 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.

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.

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
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.

2.2.12. NvAR_GetS32
Here is detailed information about the `NvAR_GetS32` structure.

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

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*.

**val**
- **Type:** `int*`
- Pointer to the 32-bit signed integer where the retrieved value will be written.

Return Value

Returns one of the following values:

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

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.
2.2.13. NvAR_GetString

Here is detailed information about the NvAR_GetString structure.

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

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 Key Values in the Properties of a Feature Type for a complete list of key values.

str

  Type: const char**

  The address where the requested character string pointer is stored.

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

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.

2.2.14. NvAR_GetU32

Here is detailed information about the NvAR_GetU32 structure.

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

**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.

**val**

*Type:* unsigned int*

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

**Return Value**

Returns one of the following values:

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

**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.

### 2.2.15. NvAR_GetU64

Here is detailed information about the NvAR_GetU64 structure.

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

**val**

Type: unsigned long long*

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

Return Values

Returns one of the following values:

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

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.

2.2.16. NvAR_SetCudaStream

Here is detailed information about the NvAR_SetCudaStream structure.

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

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.

**Return Value**

Returns one of the following values:

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

**Remarks**

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

Defined in: nvAR.h

### 2.2.17. **NvAR_SetF32**

Here is detailed information about the `NvAR_SetF32` structure.

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

**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.

**val**

**Type:** float

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

**Return Value**

Returns one of the following values:

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

**Remarks**

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

### 2.2.18. NvAR_SetF64

Here is detailed information about the `NvAR_SetF64` structure.

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

**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.

**val**

**Type:** `double`

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

Returns one of the following values:

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

Remarks

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

2.2.19. **NvAR_SetF32Array**

Here is detailed information about the NvAR_SetF32Array structure.

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

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](#) 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.
Return Value

Returns one of the following values:

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

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.

2.2.20. NvAR_SetObject

Here is detailed information about the `NvAR_SetObject` structure.

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

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 [Key Values in the Properties of a Feature Type](#) 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](#).

**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.
Return Value

Returns one of the following values:

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

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.

2.2.21. NvAR_SetS32

Here is detailed information about the `NvAR_SetS32` structure.

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

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.

- **val**
  
  Type: `int`

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

Return Value

Returns one of the following values:

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

Remarks

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

2.2.22. NvAR_SetString

Here is detailed information about the NvAR_SetString structure.

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

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 for a complete list of key values.

**str**

Type: const char*

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

Return Value

Returns one of the following values:

- NVCV_SUCCESS on success
- NVCV_ERR_PARAMETER
- NVCV_ERR_SELECTOR
- NVCV_ERR_GENERAL
- NVCV_ERR_MISMATCH
Remarks
This function sets the value of the specified character string parameter for the specified feature instance to the `str` parameter.

2.2.23. **NvAR_SetU32**
Here is detailed information about the `NvAR_SetU32` structure.

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

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*.

**val**

Type: `unsigned int`

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

Return Values

Returns one of the following values:

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

Remarks
This function sets the value of the specified 32-bit unsigned integer parameter for the specified feature instance to the `val` parameter.
2.2.24. **NvAR_SetU64**

Here is detailed information about the `NvAR_SetU64` structure.

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

**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.

**val**

*Type:* `unsigned long long`

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

**Return Value**

Returns one of the following values:

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

**Remarks**

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

The \texttt{NvCV\_Status} enumeration defines the following values that the AR SDK functions might return to indicate error or success.

\textbf{NVCV\_SUCCESS = 0}
Successful execution.

\textbf{NVCV\_ERR\_GENERAL}
Generic error code, which indicates that the function failed to execute for an unspecified reason.

\textbf{NVCV\_ERR\_UNIMPLEMENTED}
The requested feature is not implemented.

\textbf{NVCV\_ERR\_MEMORY}
The requested operation requires more memory than is available.

\textbf{NVCV\_ERR\_EFFECT}
An invalid effect handle has been supplied.

\textbf{NVCV\_ERR\_SELECTOR}
The specified selector is not valid for this effect filter.

\textbf{NVCV\_ERR\_BUFFER}
No image buffer has been specified.

\textbf{NVCV\_ERR\_PARAMETER}
An invalid parameter value has been supplied for this combination of effect and selector string.

\textbf{NVCV\_ERR\_MISMATCH}
Some parameters, for example, image formats or image dimensions, are not correctly matched.

\textbf{NVCV\_ERR\_PIXELFORMAT}
The specified pixel format is not supported.

\textbf{NVCV\_ERR\_MODEL}
An error occurred while the TRT model was being loaded.

\textbf{NVCV\_ERR\_LIBRARY}
An error while the dynamic library was being loaded.

\textbf{NVCV\_ERR\_INITIALIZATION}
The effect has not been properly initialized.

\textbf{NVCV\_ERR\_FILE}
The specified file could not be found.

\textbf{NVCV\_ERR\_FEATURENOTFOUND}
The requested feature was not found.

\textbf{NVCV\_ERR\_MISSINGINPUT}
A required parameter was not set.

\textbf{NVCV\_ERR\_RESOLUTION}
The specified image resolution is not supported.

\textbf{NVCV\_ERR\_UNSUPPORTEDGPU}
The GPU is not supported.

\textbf{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 an invalid memory address.
NVCV_ERR_CUDA
   An unspecified CUDA error has occurred.

There are 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.

<table>
<thead>
<tr>
<th>NFAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
</tr>
<tr>
<td>endian</td>
</tr>
<tr>
<td>sizeBits</td>
</tr>
<tr>
<td>indexBits</td>
</tr>
<tr>
<td>zero</td>
</tr>
<tr>
<td>TOC loc</td>
</tr>
</tbody>
</table>
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.

A.3. IBUG Mappings Object

Here is some information about the IBUG mappings object. The IBUG mappings object contains the following information:

- Landmarks
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.

B.1. 34 Keypoints of Body Pose Tracking

Here is a list of the 34 keypoints.

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

Here is some information about the order of the keypoints.

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

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, or in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer’s own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS [TOGETHER AND SEPARATELY, "MATERIALS"] ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS; AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA’s aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

VESA DisplayPort

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

HDMI

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

ARM

ARM, AMBA and ARM Powered are registered trademarks of ARM Limited. Cortex, MPCore and Mali are trademarks of ARM Limited. All other brands or product names are the property of their respective holders. "ARM" is used to represent ARM Holdings plc; its operating company ARM Limited; and the regional subsidiaries ARM Inc.; ARM KK; ARM Korea Limited; ARM Taiwan Limited; ARM France SAS; ARM Consulting (Shanghai) Co. Ltd.; ARM Germany GmbH; ARM Embedded Technologies Pvt. Ltd.; ARM Norway, AS and ARM Sweden AB.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.
Trademarks
NVIDIA, the NVIDIA logo, and cuBLAS, CUDA, CUDA Toolkit, cuDNN, DALI, DIGITS, DGX, DGX-1, DGX-2, DGX Station, DLProf, GPU, JetPack, Jetson, Kepler, Maxwell, NCCL, Nsight Compute, Nsight Systems, NVCache, NVIDIA Ampere GPU architecture, NVIDIA Deep Learning SDK, NVIDIA Developer Program, NVIDIA GPU Cloud, NVLink, NVSHMEM, PerfWorks, Pascal, SDK Manager, T4, Tegra, TensorRT, TensorRT Inference Server, Tesla, TF-TRT, Triton Inference Server, Turing, and Volta are trademarks and/or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright
© 2021-2021 NVIDIA Corporation and affiliates. All rights reserved.