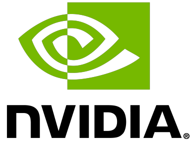


NVIDIA AR SDK

Programming Guide

PG-10463-001_v0.8.2 | December 2023

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.... 7
1.3.4. Getting the Value of a Property of a Feature....9
1.3.5. Setting a Property for a Feature.... 10
1.3.6. Loading a Feature Instance....10
1.3.7. Running a Feature Instance.... 11
1.3.8. Destroying a Feature Instance.... 11
1.4. Working with Image Frames on GPU or CPU Buffers....11
1.4.1. Converting Image Representations to NvCVImage Objects.... 11
1.4.1.1. Converting OpenCV Images to NvCVImage Objects....12
1.4.1.2. Converting Other Image Representations to NvCVImage Objects.... 12
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 NvEncoder13
1.4.2. Allocating an NvCVImage Object Buffer.... 14
1.4.2.1. Using the NvCVImage Allocation Constructor to Allocate a Buffer....14
1.4.2.2. Using Image Functions to Allocate a Buffer.... 15
1.4.3. Transferring Images Between CPU and GPU Buffers.... 15
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....16
1.5. List of Properties for the AR SDK Features....16
1.5.1. Face Detection and Tracking Property Values.... 17
1.5.2. Landmark Tracking Property Values.... 18
1.5.3. Face 3D Mesh Tracking Property Values....20
1.5.4. Eye Contact Property Values.... 25
1.5.5. Body Detection Property Values....28
1.5.6. 3D Body Pose Keypoint Tracking Property Values.... 29
1.5.7. Facial Expression Estimation Property Values....33
1.6. Using the AR Features.... 36

1.6.1. Face Detection and Tracking....36
1.6.1.1. Face Detection for Static Frames (Images)....36
1.6.1.2. Face Tracking for Temporal Frames (Videos)....37
1.6.2. Landmark Detection and Tracking....37
1.6.2.1. Landmark Detection for Static Frames (Images)....37
1.6.2.2. Alternative Usage of Landmark Detection....37
1.6.2.3. Landmark Tracking for Temporal Frames (Videos)....38
1.6.3. Face 3D Mesh and Tracking....38
1.6.3.1. Face 3D Mesh for Static Frames (Images)....38
1.6.3.2. Alternative Usage of the Face 3D Mesh Feature....39
1.6.3.3. Face 3D Mesh Tracking for Temporal Frames (Videos)....39
1.6.4. Eye Contact....40
1.6.4.1. Gaze Estimation....40
1.6.4.2. Gaze Redirection....41
1.6.5. 3D Body Pose Tracking....41
1.6.5.1. 3D Body Pose Tracking for Static Frames (Images)....42
1.6.5.2. 3D Body Pose Tracking for Temporal Frames (Videos)....43
1.6.5.3. Multi-Person Tracking for 3D Body Pose Tracking (Windows Only)....44
1.6.6. Facial Expression Estimation....45
1.6.6.1. Facial Expression Estimation for Static Frames (Images)....45
1.6.6.2. Alternative Usage of the Facial Expression Estimation Feature....46
1.6.6.3. Facial Expression Estimation Tracking for Temporal Frames (Videos)....47
1.7. Using Multiple GPUs....47
1.7.1. Default Behavior in Multi-GPU Environments....48
1.7.2. Selecting the GPU for AR SDK Processing in a Multi-GPU Environment....48
1.7.3. Selecting Different GPUs for Different Tasks....48
1.7.4. Using Multi-Instance GPU (Linux Only)....49
Chapter 2. AR SDK API Reference....51
2.1. Structures....51
2.1.1. NvAR_BBoxes....51
2.1.2. NvAR_TrackingBBox....52
2.1.3. NvAR_TrackingBBoxes....52
2.1.4. NvAR_FaceMesh....53
2.1.5. NvAR_Frustum....54
2.1.6. NvAR_FeatureHandle....54
2.1.7. NvAR_Point2f....55
2.1.8. NvAR_Point3f....55
2.1.9. NvAR_Quaternion....56

2.1.10. NvAR_Rect.......... 56
2.1.11. NvAR_RenderingParams.......... 57
2.1.12. NvAR_Vector2f.......... 58
2.1.13. NvAR_Vector3f.......... 58
2.1.14. NvAR_Vector3u16.......... 59
2.2. Functions.......... 59
2.2.1. NvAR_Create.......... 59
2.2.2. NvAR_Destroy.......... 60
2.2.3. NvAR_Load.......... 60
2.2.4. NvAR_Run.......... 61
2.2.5. NvAR_GetCudaStream.......... 62
2.2.6. NvAR_CudaStreamCreate.......... 63
2.2.7. NvAR_CudaStreamDestroy.......... 63
2.2.8. NvAR_GetF32.......... 64
2.2.9. NvAR_GetF64.......... 65
2.2.10. NvAR_GetF32Array.......... 66
2.2.11. NvAR_GetObject.......... 67
2.2.12. NvAR_GetS32.......... 68
2.2.13. NvAR_GetString.......... 69
2.2.14. NvAR_GetU32.......... 69
2.2.15. NvAR_GetU64.......... 70
2.2.16. NvAR_SetCudaStream.......... 71
2.2.17. NvAR_SetF32.......... 72
2.2.18. NvAR_SetF64.......... 73
2.2.19. NvAR_SetF32Array.......... 74
2.2.20. NvAR_SetObject.......... 75
2.2.21. NvAR_SetS32.......... 76
2.2.22. NvAR_SetString.......... 77
2.2.23. NvAR_SetU32.......... 78
2.2.24. NvAR_SetU64.......... 79
2.3. Return Codes.......... 80
Appendix A. NVIDIA 3DMM File Format.......... 83
A.1. Header.......... 83
A.2. Model Object.......... 84
A.3. IBUG Mappings Object.......... 84
A.4. Blend Shapes Object.......... 85
A.5. Model Contours Object.......... 85
A.6. Topology Object.......... 86

A.7. NVIDIA Landmarks Object....86
A.8. Partition Object....86
A.9. Table of Contents Object....87
Appendix B. 3D Body Pose Keypoint Format....88
B.1. 34 Keypoints of Body Pose Tracking....88
B.2. NvAR_Parameter_Output(KeyPoints) Order....89
Appendix C. 3DMM Versions....90
C.1. Face Expression List....90
Appendix D. Coordinate Systems....103
D.1. NvAR World 3D Space....103
D.2. NvAR Model 3D Space....104
D.3. NvAR Camera 3D Space....105
D.4. NvAR Image 2D Space....106

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 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 are provided at runtime when instances of the feature type are run.

Refer to 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. Refer to 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. Refer to 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.

     Refer to Creating an Instance of a Feature Type for more information.
   - The key value `NVAR_Parameter_Config(CUDAStream)`.

     Refer Key Values in the Properties of a Feature Type for more information.
   - The CUDA stream that you created in the previous step.

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

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

1.3.2. Summary of NVIDIA AR SDK Accessor Functions

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

Table 1. AR SDK Accessor Functions

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

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.

  Refer to Configuration Properties for more information.
- `NvAR_Parameter_Input` indicates an input property.

  Refer to Input Properties for more information.

- `NvAR_Parameter_Output` indicates an output property.

  Refer to 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(CUDAStream)

The CUDA stream in which to run the feature.

String equivalent: `NvAR_Parameter_Config_CUDAStream`

Property type: CUDA stream (`CUstream`)

NvAR_Parameter_Config(ModelDir)

The path to the directory that contains the TensorRT model files that will be used to run inference for face detection or landmark detection, and the .nvf file that contains the 3D Face model, excluding the model file name. For details about the format of the .nvf file, Refer to 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.

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. In the supplied `face_model2.nvf`, there are 100 shapes (also known as identity) eigenvalues, but the `ShapeEigenValueCount` should be queried when you allocate an array to receive the eigenvalues.

String equivalent: `NvAR_Parameter_Config_ShapeEigenValueCount`

Property type: `unsigned integer`

NvAR_Parameter_Config(ExpressionCount)

The number of coefficients used to represent expression. In the supplied `face_model2.nvf`, there are 53 expression blendshape coefficients, but the `ExpressionCount` should be queried when allocating an array to receive the coefficients.

String equivalent: `NvAR_Parameter_Config_ExpressionCount`

Property type: `unsigned integer`

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 or Facial Landmark Detection.

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*`)

NvAR_Parameter_Config(TrackPeople) (Windows only)

Flag to select Multi-Person Tracking for 3D Body Pose Tracking.

String equivalent: `NvAR_Parameter_Config_TrackPeople`

Property type: object (`unsigned integer`)

NvAR_Parameter_Config(ShadowTrackingAge)(Windows only)

The age after which the multi-person tracker no longer tracks the object in shadow mode. This property is measured in the number of frames.

Flag to select Multi-Person Tracking for 3D Body Pose Tracking.

String equivalent: `NvAR_Parameter_Config_ShadowTrackingAge`

Property type: object (`unsigned integer`)

NvAR_Parameter_Config(ProbationAge)(Windows only)

The age after which the multi-person tracker marks the object valid and assigns an ID for tracking. This property is measured in the number of frames.

String equivalent: `NvAR_Parameter_Config_ProbationAge`

Property type: object (`unsigned integer`)

NvAR_Parameter_Config(MaxTargetsTracked)(Windows only)

The maximum number of targets to be tracked by the multi-person tracker. After the new maximum target tracked limit is met, any new targets will be discarded.

String equivalent: `NvAR_Parameter_Config_MaxTargetsTracked`

Property type: object (`unsigned integer`)

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_InputBoundingBoxes`

Property type: object (`void*`)

NvAR_Parameter_Input(FocalLength)

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

String equivalent: `NvAR_Parameter_Input_FocalLength`

Property type: object (`float`)

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(TrackingBoundingBoxes)(Windows only)

CPU output tracking bounding boxes of type NvAR_TrackingBBoxes.

String equivalent: `NvAR_Parameter_Output_TrackingBBoxes`

Property type: object (`object (void*)`)

NvAR_Parameter_Output(BoundingBoxesConfidence)

Float array of confidence values for each returned bounding box.

String equivalent: `NvAR_Parameter_Output_BoundingBoxesConfidence`

Property type: floating point array

NvAR_Parameter_Output(Landmarks)

CPU output buffer of type `NvAR_Point2f` to hold the output detected landmark key points. Refer to Facial point annotations for more information. The order of the points in the CPU buffer follows the order in MultiPIE 68-point markups, and the 126 points cover more points along the cheeks, the eyes, and the laugh lines.

String equivalent: `NvAR_Parameter_Output_Landmarks`

Property type: object (`void*`)

NvAR_Parameter_Output(LandmarksConfidence)

Float array of confidence values for each detected landmark point.

String equivalent: `NvAR_Parameter_Output_LandmarksConfidence`

Property type: floating point array

NvAR_Parameter_Output(Pose)

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

String equivalent: `NvAR_Parameter_Output_Pose`

Property type: object (`void*`)

NvAR_Parameter_Output(FaceMesh)

CPU 3D face Mesh of type `NvAR_FaceMesh`.

String equivalent: `NvAR_Parameter_Output_FaceMesh`

Property type: object (`void*`)

NvAR_Parameter_Output(RenderingParams)

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

String equivalent: `NvAR_Parameter_Output_RenderingParams`

Property type: object (`void*`)

NvAR_Parameter_Output(ShapeEigenValues)

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

String equivalent: `NvAR_Parameter_Output_ShapeEigenValues`

Property type: const floating point array

NvAR_Parameter_Output(ExpressionCoefficients)

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

String equivalent: `NvAR_Parameter_Output_ExpressionCoefficients`

Property type: const floating point array

NvAR_Parameter_Output(KeyPoints)

CPU output buffer of type `NvAR_Point2f` to hold the output detected 2D Keypoints for Body Pose. Refer to 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. Refer to 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 joint angles in axis-angle format for the Keypoints for Body Pose.

String equivalent: `NvAR_Parameter_Output_JointAngles`

Property type: object (`void*`)

NvAR_Parameter_Output(KeyPointsConfidence)

Float array of confidence values for each detected keypoints.

String equivalent: `NvAR_Parameter_Output_KeyPointsConfidence`

Property type: floating point array

NvAR_Parameter_Output(OutputHeadTranslation)

Float array of three values that represent the x, y and z values of head translation with respect to the camera for Eye Contact.

String equivalent: `NvAR_Parameter_Output_OutputHeadTranslation`

Property type: floating point array

NvAR_Parameter_Output(OutputGazeVector)

Float array of two values that represent the yaw and pitch angles of the estimated gaze for Eye Contact.

String equivalent: `NvAR_Parameter_Output_OutputGazeVector`

Property type: floating point array

NvAR_Parameter_Output(HeadPose)

CPU array of type NvAR_Quaternion to hold the output-detected head pose as an XYZW quaternion in Eye Contact. This is an alternative to the head pose that was obtained from the facial landmarks feature. This head pose is obtained using the PnP algorithm over the landmarks.

String equivalent: `NvAR_Parameter_Output_HeadPose`

Property type: `object (void*)`

NvAR_Parameter_Output(GazeDirection)

Float array of two values that represent the yaw and pitch angles of the estimated gaze for Eye Contact.

String equivalent: `NvAR_Parameter_Output_GazeDirection`

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:

```
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:

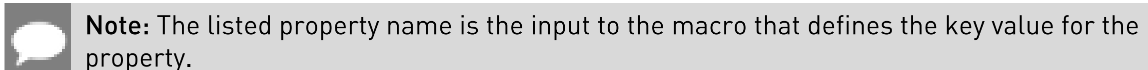
   ```
   const char *modelPath = "file/path/to/model";
   NvAR_SetString(landmarkDetectHandle, NvAR_Parameter_Config(ModelDir), modelPath);
   ```

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

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

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

   Refer to 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. Refer to Creating an Instance of a Feature Type for more information.

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

```
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. Refer to Creating an Instance of a Feature Type for more information.

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

```
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. Refer to Creating an Instance of a Feature Type for more information.

```
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, refer to 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.

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

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

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

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

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

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

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

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

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 eight 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-nvencoderThe following sample shows a frame
 that is encoded in the BGRA pixel format.
//BGRA frame is 4-channel, u8 buffer residing on the GPU
NvCVImage BGRA_frame;
NvCVImage_Alloc(&BGRA_frame, dec.GetWidth(), dec.GetHeight(), NVCV_BGRA, NVCV_U8,
 NVCV_CHUNKY, NVCV_GPU, 1);
//Initialize encoder with a BGRA output pixel format
using NvEncCudaPtr = std::unique_ptr<NvEncoderCuda,
 std::function<void(NvEncoderCuda*)>>;
NvEncCudaPtr pEnc(new NvEncoderCuda(cuContext, dec.GetWidth(), dec.GetHeight(),
 NV_ENC_BUFFER_FORMAT_ARGB));
pEnc->CreateEncoder(&initializeParams);
//...

std::vector<std::vector<uint8_t>> vPacket;
```

```
//Get the address of the next input frame from the encoder
const NvEncInputFrame* encoderInputFrame = pEnc->GetNextInputFrame();


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

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. Refer to 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 image, you can defer buffer allocation.

1. Declare an empty `NvCVImage` object.

```
NvCVImage xfr;
```

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

     Allocate a buffer this way when the image is part of a state structure, where you will not know the size 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.

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

2. Create a staging buffer in one of the following ways:

    - To avoid allocating memory in a video pipeline, create a GPU buffer that has the same dimensions and format as required for input to the video effect filter.

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

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

    ```
    NvCVImage srcGpuStaging;
    ```

    An appropriate buffer will be allocated or reallocated as needed.

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

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

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 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 destination CPU buffer.

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

    For more information about `NvCVImage`, refer to 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.

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

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

    ```
    NvCVImage dstGpuStaging;
    ```

    An appropriately sized buffer will be allocated as needed.

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

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

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 Detection and Tracking Property Values

The following tables list the values for the configuration, input, and output properties for Face Detection and Tracking.

Table 2. Configuration Properties for Face Detection and Tracking

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

Table 3. Input Properties for Face Detection and Tracking

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

Table 4. Output Properties for Face Detection and Tracking

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

Property Name	Value
BoundingBoxesConfidence	An array of single-precision (32-bit) floating-point numbers that contain the confidence values for each detected face box. To be allocated by the user.

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

Property Name	Value
FeatureDescription	String that describes the feature.
CUDAStream	The CUDA stream. Set by the user.
ModelDir	String that contains the path to the folder that contains the TensorRT package files. Set by the user.
BatchSize	The number of inferences to be run at one time on the GPU. The maximum value is 8. Temporal optimization of landmark detection is only supported for `BatchSize=1`.
Landmarks_Size	Unsigned integer, 68 or 126. Specifies the number of landmark points (X and Y values) to be returned. Set by the user.
LandmarksConfidence_Size	Unsigned integer, 68 or 126. Specifies the number of landmark confidence values for the detected keypoints to be returned. Set by the user.
Temporal	Unsigned integer, 1/0 to enable/disable the temporal optimization of landmark detection. If

Property Name	Value
	enabled, only one input bounding box is supported as the input. Refer to Landmark Detection and Tracking for more information. Set by the user.
`Mode`	(**Optional**) Unsigned integer. Set 0 to enable Performance mode (default) or 1 to enable Quality mode for Landmark detection. Set by the user.

Table 6. Input Properties for Landmark Tracking

Property Name	Value
`Image`	Interleaved (or chunky) 8-bit BGR input image in a CUDA buffer of type `NvCVImage`. To be allocated and set by the user.
`BoundingBoxes`	`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. Refer to Landmark Detection and Tracking for more information. To be allocated by the user.

Table 7. Output Properties for Landmark Tracking

Property Name	Value
`Landmarks`	`NvAR_Point2f` array, which must be large enough to hold the number of points given by the product of `NvAR_Parameter_Config(BatchSize)` and `NvAR_Parameter_Config(Landmarks_Size)`. To be allocated by the user.
`Pose`	`NvAR_Quaternion` array, which must be large enough to hold the number of quaternions equal to `NvAR_Parameter_Config(BatchSize)`.

Property Name	Value
	The coordinate convention is followed as per OpenGL standards. For example, when seen from the camera, X is right, Y is up , and Z is back/ towards the camera. To be allocated by the user.
`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. Refer to 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

Property Name	Value
`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. Refer to Alternative Usage of the Face 3D Mesh Feature for more information. Set by the user.
`CUDAStream`	(**Optional**) The CUDA stream. Refer to Alternative Usage of the Face 3D Mesh Feature for more information.

Property Name	Value
	Set by the user.
`Temporal`	(**Optional**) Unsigned integer, 1/0 to enable/disable the temporal optimization of face and landmark detection. Refer to Alternative Usage of the Face 3D Mesh Feature for more information. Set by the user.
`Mode`	(**Optional**) Unsigned integer. Set 0 to enable Performance mode (default) or 1 to enable Quality mode for Landmark detection. Set by the user.
`Landmarks_Size`	Unsigned integer, 68 or 126. If landmark detection is run internally, the confidence values for the detected key points are returned. Refer to Alternative Usage of the Face 3D Mesh Feature for more information.
`ShapeEigenValueCount`	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. This property is read-only.
`ExpressionCount`	The number of expressions available in the chosen model. Query this to determine how big the expression coefficient array should be, if that is the desired output. This property is read-only.
`VertexCount`	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. This property is read-only.
`TriangleCount`	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.

Property Name	Value
	This property is read-only.
`GazeMode`	Flag to toggle gaze mode. The default value is 0. If the value is 1, gaze estimation will be explicit.

Table 9. Input Properties for Face 3D Mesh Tracking

Property Name	Value
`Width`	The width of the input image buffer that contains the face to which the face model will be fitted. Set by the user.
`Height`	The height of the input image buffer that contains the face to which the face model will be fitted. Set by the user.
`Landmarks`	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. Refer to Alternative Usage of the Face 3D Mesh Feature for more information. To be allocated by the user.
`Image`	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. Refer to Alternative Usage of the Face 3D Mesh Feature for more information. To be allocated by the user.

Table 10. Output Properties for Face 3D Mesh Tracking

Property Name	Value
`FaceMesh`	`NvAR_FaceMesh` structure that contains the output face mesh. To be allocated by the user. Query `VertexCount` and `TriangleCount` to determine how much memory to allocate.
`RenderingParams`	`NvAR_RenderingParams` structure that contains the rendering parameters for drawing the face mesh that is returned by this feature. To be allocated by the user.
`Landmarks`	An `NvAR_Point2f` array, which must be large enough to hold the number of points of size `NvAR_Parameter_Config(Landmarks_Size)`. Refer to Alternative Usage of the Face 3D Mesh Feature for more information. To be allocated by the user.
`Pose`	`NvAR_Quaternion` array pointer, to hold one quaternion. See `Alternative Usage of the Face 3D Mesh Feature` for more information. The coordinate convention is followed as per OpenGL standards. For example, when seen from the camera, X is right, Y is up , and Z is back/ towards the camera. To be allocated by the user.
`LandmarksConfidence`	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)`. Refer to Alternative Usage of the Face 3D Mesh Feature for more information. To be allocated by the user.
`BoundingBoxes`	`NvAR_BBoxes` structure that contains the detected face that is determined internally. Refer to Alternative Usage of the Face 3D Mesh Feature for more information.

Property Name	Value
	To be allocated by the user.
`BoundingBoxesConfidence`	An array of single-precision (32-bit) floating-point numbers that contain the confidence values for each detected face box. Refer to Alternative Usage of the Face 3D Mesh Feature for more information. To be allocated by the user.
`ShapeEigenValues`	**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.
`ExpressionCoefficients`	**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. The corresponding expression shapes for face_model2.nvf are in the following order: `BrowDown_L`, `BrowDown_R`, `BrowInnerUp_L`, `BrowInnerUp_R`, `BrowOuterUp_L`, `BrowOuterUp_R`, `CheekPuff_L`, `CheekPuff_R`, `CheekSquint_L`, `CheekSquint_R`, `EyeBlink_L`, `EyeBlink_R`, `EyeLookDown_L`, `EyeLookDown_R`, `EyeLookIn_L`, `EyeLookIn_R`, `EyeLookOut_L`, `EyeLookOut_R`, `EyeLookUp_L`, `EyeLookUp_R`, `EyeSquint_L`, `EyeSquint_R`, `EyeWide_L`, `EyeWide_R`, `JawForward`, `JawLeft`, `JawOpen`, `JawRight`, `MouthClose`, `MouthDimple_L`, `MouthDimple_R`, `MouthFrown_L`, `MouthFrown_R`, `MouthFunnel`, `MouthLeft`, `MouthLowerDown_L`, `MouthLowerDown_R`, `MouthPress_L`, `MouthPress_R`, `MouthPucker`, `MouthRight`, `MouthRollLower`, `MouthRollUpper`, `MouthShrugLower`, `MouthShrugUpper`, `MouthSmile_L`, `MouthSmile_R`, `MouthStretch_L`, `MouthStretch_R`, `MouthUpperUp_L`, `MouthUpperUp_R`, `NoseSneer_L`, `NoseSneer_R`

1.5.4. Eye Contact Property Values

The following tables list the values for gaze redirection.

Table 11. Configuration Properties for Eye Contact

Property Name	Value
`FeatureDescription`	String that describes the feature.
`CUDAStream`	The CUDA stream. Set by the user.
`ModelDir`	String that contains the path to the folder that contains the TensorRT package files. Set by the user.
`BatchSize`	The number of inferences to be run at one time on the GPU. The maximum value is 1.
`Landmarks_Size`	Unsigned integer, 68 or 126. Specifies the number of landmark points (X and Y values) to be returned. Set by the user.
`LandmarksConfidence_Size`	Unsigned integer, 68 or 126. Specifies the number of landmark confidence values for the detected keypoints to be returned. Set by the user.
`GazeRedirect`	Flag to enable or disable gaze redirection. When enabled, the gaze is estimated, and the redirected image is set as the output. When disabled, the gaze is estimated, but redirection does not occur.
`Temporal`	Unsigned integer and 1/0 to enable/disable the temporal optimization of landmark detection. Set by the user.

Property Name	Value
DetectClosure	Flag to toggle detection of eye closure and occlusion on/off. Default - ON
EyeSizeSensitivity	Unsigned integer in the range of 2-5 to increase the sensitivity of the algorithm to the redirected eye size. 2 uses a smaller eye region and 5 uses a larger eye size.
UseCudaGraph	Bool, True or False. Default is False Flag to use CUDA Graphs for optimization. Set by the user.

Table 12. Input Properties for Eye Contact

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

Table 13. Output Properties for Eye Contact

Property Name	Value
`Landmarks`	`NvAR_Point2f` array, which must be large enough to hold the number of points given by the product of the following: ▶ `NvAR_Parameter_Config(BatchSize)` ▶ `NvAR_Parameter_Config(Landmarks_Size)` To be allocated by the user.
`HeadPose`	(**Optional**) NvAR_Quaternion array, which must be large enough to hold the number of quaternions equal to NvAR_Parameter_Config(BatchSize). The OpenGL standards coordinate convention is used, which is when you look up from a camera, the coordinates are camera X - right, Y - up , and Z - back/towards the camera. To be allocated by the user.
`LandmarksConfidence`	**Optional**: An array of single-precision (32-bit) floating-point numbers, which must be large enough to hold the number of confidence values given by the product of the following: ▶ `NvAR_Parameter_Config(BatchSize)` ▶ `NvAR_Parameter_Config(LandmarksConfidence_Size)` To be allocated by the user.
`BoundingBoxes`	Optional: NvAR_BBoxes structure that contains the detected face through face detection performed by the landmark detection feature. Refer to Landmark Detection and Tracking for more information. To be allocated by the user.
`OutputGazeVector`	Float array, which must be large enough to hold two values (pitch and yaw) for the gaze angle in radians per image. For batch sizes larger than 1, it should hold `NvAR_Parameter_Config(BatchSize)x 2` float values.

Property Name	Value
	To be allocated by the user.
`OutputHeadTranslation`	**Optional**: Float array, which must be large enough to hold the (x,y,z) head translations per image. For batch sizes larger than 1 it should hold `NvAR_Parameter_Config(BatchSize) x 3` float values. To be allocated by the user.
`GazeDirection`	`Optional`: NvAR_Point3f array that is large enough to hold as many elements as `NvAR_Parameter_Config(BatchSize)`. Each element contains two `NvAR_Point3f` points. One point represents the center point (cx,cy,cz) between the eyes, and the other point represents a unit vector (ux, uy, uz) in the gaze direction for visualization. For batch sizes larger than 1 it should hold `NvAR_Parameter_Config(BatchSize)x 2` `NvAR_Point3f` points. To be allocated by the user.

1.5.5. Body Detection Property Values

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

Table 14. Configuration Properties for Body Detection Tracking

Property Name	Name
`FeatureDescription`	String is free-form text that describes the feature. The string is set by the SDK and cannot be modified by the user.
`CUDAStream`	The CUDA stream, which is set by the user.
`ModelDir`	String that contains the path to the folder that contains the TensorRT package files. Set by the user.
`Temporal`	Unsigned integer, 1/0 to enable/disable the temporal optimization of Body Pose Tracking.

Property Name	Name
	Set by the user.

Table 15. Input Properties for Body Detection

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

Table 16. Output Properties for Body Detection

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

1.5.6. 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 17. Configuration Properties for 3D Body Pose Keypoint Tracking

Property Name	Value
`FeatureDescription`	`FeatureDescription` String that describes the feature.
`CUDAStream`	The CUDA stream. Set by the user.
`ModelDir`	String that contains the path to the folder that contains the TensorRT package files.

Property Name	Value
	Set by the user.
`BatchSize`	The number of inferences to be run at one time on the GPU. The maximum value is 1.
`Mode`	Unsigned integer, 0 or 1. Default is 1. Selects the High Performance (1) mode or High Quality (0) mode Set by the user.
`UseCudaGraph`	Bool, True or False. Default is True Flag to use CUDA Graphs for optimization. Set by the user.
`Temporal`	Unsigned integer and 1/0 to enable/disable the temporal optimization of Body Pose tracking. Set by the user.
`NumKeyPoints`	Unsigned integer. Specifies the number of keypoints available, which is currently 34.
`ReferencePose`	`NvAR_Point3f` array, which contains the reference pose for each of the 34 keypoints. Specifies the Reference Pose used to compute the joint angles.
`TrackPeople`	Unsigned integer and 1/0 to enable/disable multi-person tracking in Body Pose. Set by the user. Available only on Windows.
`ShadowTrackingAge`	Unsigned integer. Specifies the period after which the multi-person tracker stops tracking the object in shadow mode. This value is measured in the number of frames. Set by the user, and the default value is 90. Available only on Windows.

Property Name	Value
ProbationAge	Unsigned integer. Specifies the period after which the multi-person tracker marks the object valid and assigns an ID for tracking. This value is measured in the number of frames. Set by the user, and the default value is 10. Available only on Windows.
MaxTargetsTracked	Unsigned integer. Specifies the maximum number of targets to be tracked by the multi-person tracker. After the tracking is complete, the new targets are discarded. Set by the user, and the default value is 30. Available only on Windows.

Table 18. Input Properties for 3D Body Pose Keypoint Tracking

Property Name	Value
Image	Interleaved (or chunky) 8-bit BGR input image in a CUDA buffer of type `NvCVImage`. To be allocated and set by the user.
FocalLength	Float. Default is 800.79041 Specifies the focal length of the camera to be used for 3D Body Pose. Set by the user.
BoundingBoxes	`NvAR_BBoxes` structure that contains the number of bounding boxes that are equal to BatchSize on which to run 3D Body Pose detection. If not specified as an input property, body detection is automatically run on the input image. To be allocated by the user.

Table 19. Output Properties for 3D Body Pose Keypoint Tracking

Property Name	Value
Keypoints	NvAR_Point2f array, which must be large enough to hold the 34 points given by the product of NvAR_Parameter_Config(BatchSize) and 34. To be allocated by the user.
Keypoints3D	NvAR_Point3f array, which must be large enough to hold the 34 points given by the product of NvAR_Parameter_Config(BatchSize) and 34. To be allocated by the user.
JointAngles	NvAR_Quaternion array, which must be large enough to hold the 34 joints given by the product of NvAR_Parameter_Config(BatchSize) and 34. They represent the local rotation (in Quaternion) of each joint with reference to the ReferencePose. To be allocated by the user.
KeyPointsConfidence	An array of single-precision (32-bit) floating-point numbers, which must be large enough to hold the number of confidence values given by the product of the following: ▶ NvAR_Parameter_Config(BatchSize) ▶ 34 To be allocated by the user.
BoundingBoxes	NvAR_BBoxes structure that contains the detected body through body detection performed by the 3D Body Pose feature. To be allocated by the user.
TrackingBoundingBoxes	The NvAR_TrackingBBoxes structure that contains the detected body through body detection and that is completed by the 3D Body Pose feature and the tracking ID assigned by multi-person tracking. To be allocated by the user. Available only on Windows.

1.5.7. Facial Expression Estimation Property Values

The following tables list the values for the configuration, input, and output properties for Facial Expression Estimation.

Table 20. Configuration Properties for Facial Expression Estimation

Property Name	Value
`FeatureDescription`	String that describes the feature. This property is read-only.
`ModelDir`	String that contains the path to the folder that contains the TensorRT package files. Set by the user.
`CUDAStream`	(**Optional**) The CUDA stream. Set by the user.
`Temporal`	(**Optional**) Bitfield to control temporal filtering. ▶ 0x001 - filter face detection ▶ 0x002 - filter facial landmarks ▶ 0x004 - filter rotational pose ▶ 0x010 - filter facial expressions ▶ 0x020 - filter gaze expressions ▶ 0x100 - enhance eye and mouth closure ▶ Default - 0x037 (all on except 0x100) Set by the user.
`Landmarks_Size`	Unsigned integer, 68 or 126. Required array size of detected facial landmark points. To accommodate the {x,y} location of each of the detected points, the length of the array must be 126.
`ExpresionCount`	Unsigned integer. The number of expressions in the face model.
`PoseMode`	Determines how to compute pose. 0=3DOF implicit (default), 1=6DOF explicit. 6DOF (6

Property Name	Value
	degrees of freedom) is required for 3D translation output.
`Mode`	Flag to toggle landmark mode. Set 0 to enable Performance model for landmark detection. Set 1 to enable Quality model for landmark detection for higher accuracy. Default - 1.
`EnableCheekPuff`	(Experimental) Enables cheek puff blendshapes

Table 21. Input Properties for Facial Expression Estimation

Property Name	Value
`Landmarks`	(**Optional**) An `NvAR_Point2f` array that contains the landmark points of size `NvAR_Parameter_Config(Landmarks_Size)` that is returned by the landmark detection feature. If landmarks are not provided to this feature, an input image must be provided. To be allocated by the user.
`Image`	(**Optional**) An interleaved (or chunky) 8-bit BGR input image in a CUDA buffer of type NvCVImage. If an input image is not provided as input, the landmark points must be provided to this feature as input. To be allocated by the user.
`CameraIntrinsicParams`	**Optional:** Camera intrinsic parameters. Three element float array with elements corresponding to focal length, cx, cy intrinsics respectively, of an ideal perspective camera. Any barrel or fisheye distortion should be removed or considered negligible. Only used if PoseMode = 1.

Table 22. Output Properties for Facial Expression Estimation

Property Name	Value
`Landmarks`	(**Optional**) An `NvAR_Point2f` array, which must be large enough to hold the number of points of size `NvAR_Parameter_Config(Landmarks_Size)`.
`Pose`	(**Optional**) `NvAR_Quaternion` Pose rotation quaternion. Coordinate frame is NvAR Camera 3D Space. To be allocated by the user.
`PoseTranslation`	**Optional:** `NvAR_Point3f` Pose 3D Translation. Only computed if PoseMode = 1. Translation coordinates are in NvAR Camera 3D Space coordinates, where the units are centimeters. To be allocated by the user.
`LandmarksConfidence`	(**Optional**) An array of single-precision (32-bit) floating-point numbers, which must be large enough to hold the number of confidence values of size `NvAR_Parameter_Config(LandmarksConfidence_Size)`. To be allocated by the user.
`BoundingBoxes`	(**Optional**) `NvAR_BBoxes` structure that contains the detected face that is determined internally. To be allocated by the user.
`BoundingBoxesConfidence`	(**Optional**) An array of single-precision (32-bit) floating-point numbers that contain the confidence values for each detected face box. To be allocated by the user.
`ExpressionCoefficients`	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. The corresponding expression shapes are in the following order:

Property Name	Value
	BrowDown_L, BrowDown_R, BrowInnerUp_L, BrowInnerUp_R, BrowOuterUp_L, BrowOuterUp_R, CheekPuff_L, CheekPuff_R, CheekSquint_L, CheekSquint_R, EyeBlink_L, EyeBlink_R, EyeLookDown_L, EyeLookDown_R, EyeLookIn_L, EyeLookIn_R, EyeLookOut_L, EyeLookOut_R, EyeLookUp_L, EyeLookUp_R, EyeSquint_L, EyeSquint_R, EyeWide_L, EyeWide_R, JawForward, JawLeft, JawOpen, JawRight, MouthClose, MouthDimple_L, MouthDimple_R, MouthFrown_L, MouthFrown_R, MouthFunnel, MouthLeft, MouthLowerDown_L, MouthLowerDown_R, MouthPress_L, MouthPress_R, MouthPucker, MouthRight, MouthRollLower, MouthRollUpper, MouthShrugLower, MouthShrugUpper, MouthSmile_L, MouthSmile_R, MouthStretch_L, MouthStretch_R, MouthUpperUp_L, MouthUpperUp_R, NoseSneer_L, NoseSneer_R

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:

```
//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);
```

1.6.1.2. 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 the following frames.

However, explicitly calling the face detection feature is not the only way to obtain a bounding box that denotes detected faces. Refer to 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:

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

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

//Set landmark detection mode: Performance[0] (Default) or Quality[1]
unsigned int mode = 0; // Choose performance mode
NvAR_SetU32(landmarkDetectHandle, NvAR_Parameter_Config(Mode), mode);

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

1.6.2.2. Alternative Usage of Landmark Detection

As described in the *Configuration Properties for Landmark Tracking* table 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:

```
//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));

//Set landmark detection mode: Performance[0] (Default) or Quality[1]
unsigned int mode = 0; // Choose performance mode
NvAR_SetU32(landmarkDetectHandle, NvAR_Parameter_Config(Mode), mode);

//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:

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

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

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(faceFitHandle, NvAR_Parameter_Input(Image), &inputImageBuffer,
 sizeof(NvCVImage));

//Set output memory for face mesh
NvAR_FaceMesh face_mesh = new NvAR_FaceMesh();
unsigned int n;
err = NvAR_GetU32(faceFitHandle, NvAR_Parameter_Config(VertexCount), &n);
```

```
face_mesh->num_vertices = n;
err = NvAR_GetU32(faceFitHandle, NvAR_Parameter_Config(TriangleCount), &n);
face_mesh->num_triangles = n;
face_mesh->vertices = new NvAR_Vector3f[face_mesh->num_vertices];
face_mesh->tvi = new NvAR_Vector3u16[face_mesh->num_triangles];
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. Eye Contact

This feature estimates the gaze of a person from an eye patch that was extracted using landmarks and redirects the eyes to make the person look at the camera in a permissible range of eye and head angles.

The feature also supports a mode where the estimation can be obtained without redirection. The eye contact feature can be invoked by using the GazeRedirection feature ID.

Eye contact feature has the following options:

- Gaze Estimation
- Gaze Redirection

In this release, gaze estimation and redirection of only one face in the frame is supported.

1.6.4.1. Gaze Estimation

The estimation of gaze requires face detection and landmarks as input. The inputs to the gaze estimator are an input image buffer and buffers to hold facial landmarks and confidence scores. The output of gaze estimation is the gaze vector (pitch, yaw) values in radians. A float array needs to be set as the output buffer to hold estimated gaze. The `GazeRedirect` parameter must be set to `false`.

This example runs the Gaze Estimation with an input image buffer and output memory to hold the estimated gaze vector:

```
bool bGazeRedirect=false
NvAR_SetU32(gazeRedirectHandle, NvAR_Parameter_Config(GazeRedirect), bGazeRedirect);
//Set input image buffer
NvAR_SetObject(gazeRedirectHandle, NvAR_Parameter_Input(Image), &inputImageBuffer,
 sizeof(NvCVImage));
//Set output memory for gaze vector
float gaze_angles_vector[2];
NvvAR_SetF32Array(gazeRedirectHandle, NvAR_Parameter_Output(OutputGazeVector),
 gaze_angles_vector, batchSize * 2);

//OPTIONAL – Set output memory for landmarks, head pose, head translation and gaze
 direction if desired
```

```
std::vector<NvAR_Point2f> facial_landmarks;
facial_landmarks.assign(batchSize * OUTPUT_SIZE_KPTS, {0.f, 0.f});
NvAR_SetObject(gazeRedirectHandle, NvAR_Parameter_Output(Landmarks),
 facial_landmarks.data(),sizeof(NvAR_Point2f));

NvAR_Quaternion head_pose;
NvAR_SetObject(gazeRedirectHandle, NvAR_Parameter_Output(HeadPose), &head_pose,
 sizeof(NvAR_Quaternion));

float head_translation[3] = {0.f};
NvAR_SetF32Array(gazeRedirectHandle, NvAR_Parameter_Output(OutputHeadTranslation),
 head_translation,
                           batchSize * 3);

NvAR_Run(gazeRedirectHandle);
```

1.6.4.2. Gaze Redirection

Gaze redirection takes identical inputs as the gaze estimation. In addition to the outputs of gaze estimation, to store the gaze redirected image, an output image buffer of the same size as the input image buffer needs to be set.

The gaze will be redirected to look at the camera within a certain range of gaze angles and head poses. Outside this range, the feature disengages. Head pose, head translation, and gaze direction can be optionally set as outputs. The GazeRedirect parameter must be set to `true`.

```
bool bGazeRedirect=true;
NvAR_SetU32(gazeRedirectHandle, NvAR_Parameter_Config(GazeRedirect), bGazeRedirect);
//Set input image buffer
NvAR_SetObject(gazeRedirectHandle, NvAR_Parameter_Input(Image), &inputImageBuffer,
 sizeof(NvCVImage));
//Set output memory for gaze vector
float gaze_angles_vector[2];
NvvAR_SetF32Array(gazeRedirectHandle, NvAR_Parameter_Output(OutputGazeVector),
 gaze_angles_vector, batchSize * 2);
//Set output image buffer
NvAR_SetObject(gazeRedirectHandle, NvAR_Parameter_Output(Image), &outputImageBuffer,
 sizeof(NvCVImage));
NvAR_Run(gazeRedirectHandle);
```

1.6.5. 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. The feature will still work if a part of the body, such as an arm or a foot, is occluded/truncated.

1.6.5.1. 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:

```
//Set input image buffer
NvAR_SetObject(bodyDetectHandle, NvAR_Parameter_Input(Image), &inputImageBuffer,
 sizeof(NvCVImage));
 //Set output memory for bounding boxes

NvAR_BBoxes = output_boxes{};
output_bboxes.boxes = new NvAR_Rect[25];
output_bboxes.max_boxes = 25;
NvAR_SetObject(bodyDetectHandle, NvAR_Parameter_Output(BoundingBoxes),
 &output_bboxes, sizeof(NvAR_BBoxes));

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

NvAR_Run(bodyDetectHandle);
```

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

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

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

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

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

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

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

//Set output memory for bounding boxes
NvAR_BBoxes = output_boxes{};
```

```
output_bboxes.boxes = new NvAR_Rect[25];
output_bboxes.max_boxes = 25;
NvAR_SetObject(keyPointDetectHandle, NvAR_Parameter_Output(BoundingBoxes),
 &output_bboxes, sizeof(NvAR_BBoxes));

NvAR_Run(keyPointDetectHandle);
```

1.6.5.2. 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:

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

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

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

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

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

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

NvAR_Run(keyPointDetectHandle);
```

1.6.5.3. Multi-Person Tracking for 3D Body Pose Tracking (Windows Only)

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.

The feature provides the ability to track multiple people in the following ways:

- In the scene across different frames.
- When they leave the scene and enter the scene again.
- When they are completely occluded by an object or another person and reappear (controlled using Shadow Tracking Age).
- **Shadow Tracking Age**

  This option represents the period of time where a target is still being tracked in the background even when the target is not associated with a detector object. When a target is not associated with a detector object for a time frame, shadowTrackingAge, an internal variable of the target, is incremented. After the target is associated with a detector object, shadowTrackingAge will be reset to zero. When the target age reaches the shadow tracking age, the target is discarded and is no longer tracked. This is measured by the number of frames, and the default is 90.
- **Probation Age**

  This option is the length of probationary period. After an object reaches this age, it is considered to be valid and is appointed an ID. This will help with false positives, where false objects are detected only for a few frames. This is measured by the number of frames, and the default is 10.
- **Maximum Targets Tracked**

  This option is the maximum number of targets to be tracked, which can be composed of the targets that are active in the frame and ones in shadow tracking mode. When you select this value, keep the active and inactive targets in mind. The minimum is 1 and the default is 30.

Note: Currently, we only actively track eight people in the scene. There can be more than eight people throughout the video but only a maximum of eight people in a given frame. Temporal mode is not supported for Multi-Person Tracking. The batch size should be 8 when Multi-Person Tracking is enabled. This feature is currently Windows only.

This example uses the 3D Body Pose Tracking AR feature to enable multi-person tracking and object the tracking ID for each person:

```
//Set input image buffer
NvAR_SetObject(keypointDetectHandle, NvAR_Parameter_Input(Image), &inputImageBuffer,
 sizeof(NvCVImage));
// Enable Multi-Person Tracking
NvAR_SetU32(keyPointDetectHandle, NvAR_Parameter_Config(TrackPeople),
 bEnablePeopleTracking);
// Set Shadow Tracking Age
NvAR_SetU32(keyPointDetectHandle, NvAR_Parameter_Config(ShadowTrackingAge),
 shadowTrackingAge);
// Set Probation Age
```

```
NvAR_SetU32(keyPointDetectHandle, NvAR_Parameter_Config(ProbationAge),
 probationAge);
// Set Maximum Targets to be tracked
NvAR_SetU32(keyPointDetectHandle, NvAR_Parameter_Config(MaxTargetsTracked),
 maxTargetsTracked);
//Set output buffer to hold detected keypoints
std::vector<NvAR_Point2f> keypoints;
std::vector<NvAR_Point3f> keypoints3D;
std::vector<NvAR_Point3f> jointAngles;
std::vector<float> keypoints_confidence;
// Get the number of keypoints
unsigned int numKeyPoints;
NvAR_GetU32(keyPointDetectHandle, NvAR_Parameter_Config(NumKeyPoints),
 &numKeyPoints);
keypoints.assign(batchSize * numKeyPoints , {0.f, 0.f});
keypoints3D.assign(batchSize * numKeyPoints , {0.f, 0.f, 0.f});
jointAngles.assign(batchSize * numKeyPoints , {0.f, 0.f, 0.f});
NvAR_SetObject(keyPointDetectHandle, NvAR_Parameter_Output(KeyPoints),
 keypoints.data(), sizeof(NvAR_Point2f));
NvAR_SetObject(keyPointDetectHandle, NvAR_Parameter_Output(KeyPoints3D),
 keypoints3D.data(), sizeof(NvAR_Point3f));
NvAR_SetF32Array(keyPointDetectHandle, NvAR_Parameter_Output(KeyPointsConfidence),
 keypoints_confidence.data(), batchSize * numKeyPoints);
NvAR_SetObject(keyPointDetectHandle, NvAR_Parameter_Output(JointAngles),
 jointAngles.data(), sizeof(NvAR_Point3f));
//Set output memory for tracking bounding boxes
NvAR_TrackingBBoxes output_tracking_bboxes{};
std::vector<NvAR_TrackingBBox> output_tracking_bbox_data;
output_tracking_bbox_data.assign(maxTargetsTracked, { 0.f, 0.f, 0.f, 0.f, 0 });
output_tracking_bboxes.boxes = output_tracking_bbox_data.data();
output_tracking_bboxes.max_boxes = (uint8_t)output_tracking_bbox_size;

NvAR_SetObject(keyPointDetectHandle, NvAR_Parameter_Output(TrackingBoundingBoxes),
 &output_tracking_bboxes, sizeof(NvAR_TrackingBBoxes));
NvAR_Run(keyPointDetectHandle);
```

1.6.6. Facial Expression Estimation

This section provides information about how to use the Facial Expression Estimation feature.

1.6.6.1. Facial Expression Estimation for Static Frames (Images)

Typically, the input to the Facial Expression Estimation feature is an input image and a set of detected landmark points corresponding to the face on which we want to estimate face expression coefficients.

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

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

//Set output memory for expression coefficients
unsigned int expressionCount;
err = NvAR_GetU32(faceExpressionHandle, NvAR_Parameter_Config(ExpressionCount),
 &expressionCount);
float expressionCoeffs = new float[expressionCount];
err = NvAR_SetF32Array(faceExpressionHandle,
 NvAR_Parameter_Output(ExpressionCoefficients), expressionCoeffs, expressionCount);

//Set output memory for pose rotation quaternion
```

```
NvAR_Quaternion pose = new NvAR_Quaternion();
err = NvAR_SetObject(faceExpressionHandle, NvAR_Parameter_Output(Pose), pose,
 sizeof(NvAR_Quaternion));

//OPTIONAL – Set output memory for bounding boxes, and their confidences if desired
err = NvAR_Run(faceExpressionHandle);
```

1.6.6.2. Alternative Usage of the Facial Expression Estimation Feature

Similar to the alternative usage of the Landmark detection feature and the Face 3D Mesh Feature, the Facial Expression Estimation 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 expression estimation. 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.

ExpressionCoefficients and Pose 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.

The expression coefficients can be used to drive the expressions of an avatar.

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 Facial Expression Estimation feature to obtain the face expression coefficients directly from the image, without explicitly running Landmark Detection or Face Detection:

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

//Set output memory for expression coefficients
unsigned int expressionCount;
err = NvAR_GetU32(faceExpressionHandle, NvAR_Parameter_Config(ExpressionCount),
 &expressionCount);
float expressionCoeffs = new float[expressionCount];
err = NvAR_SetF32Array(faceExpressionHandle,
 NvAR_Parameter_Output(ExpressionCoefficients), expressionCoeffs, expressionCount);

//Set output memory for pose rotation quaternion
NvAR_Quaternion pose = new NvAR_Quaternion();
err = NvAR_SetObject(faceExpressionHandle, NvAR_Parameter_Output(Pose), pose,
 sizeof(NvAR_Quaternion));

//OPTIONAL - Set facial keypoints as an output
NvAR_SetObject(faceExpressionHandle, 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(faceExpressionHandle);
```

1.6.6.3. Facial Expression Estimation 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. If the Face Detection, and Landmark Detection features are used explicitly, they need their own Temporal flags to be set, however, the temporal flag also affects the Facial Expression Estimation feature through the `NVAR_TEMPORAL_FILTER_FACIAL_EXPRESSIONS`, `NVAR_TEMPORAL_FILTER_FACIAL_GAZE`, and `NVAR_TEMPORAL_FILTER_ENHANCE_EXPRESSIONS` bits.

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.

```
// 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.

   ```
   // 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 is 7.5, 8.6, or 8.9, which denotes a GPU that is based on Turing, the Ampere architecture, or the Ada architecture respectively.

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

3. In the loop to complete the application's tasks, select the best GPU for each task before performing the task.
   a). Call `cudaSetDevice()` to select the GPU for the task.
   b). Make all the function calls required to perform the task.

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

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

1.7.4. Using Multi-Instance GPU (Linux Only)

Applications that are developed with the AR SDK can be deployed on Multi-Instance GPU (MIG) on supported devices, such as NVIDIA DGX™ A100.

MIG allows you to partition a device into up to seven multiple GPU instances, each with separate streaming multiprocessors, separate slices of the GPU memory, and separate pathways to the memory. This process ensures that heavy resource usage by an application on one partition does not impact the performance of the applications running on other partitions.

To run an application on a MIG partition, you do not have to call any additional SDK API in your application. You can specify which MIG instance to use for execution during invocation of your application.

You can select the MIG instance using one of the following options:

- The bare-metal method of using `CUDA_VISIBLE_DEVICES` environment variable.
- The container method by using the NVIDIA Container Toolkit.

  MIG is supported only on Linux.

Refer to the NVIDIA Multi-Instance GPU User Guide for more information about the MIG and its usage.

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_defs.h`

2.1.2. NvAR_TrackingBBox

Here is detailed information about the `NvAR_TrackingBBox` structure.

```
struct NvAR_TrackingBBox {
  NvAR_Rect bbox;
  uint16_t tracking_id;
};
```

Members

bbox

Type: `NvAR_Rect`

Bounding box that is allocated by the user.

tracking_id

Type: `uint16_t`

The Tracking ID assigned to the bounding box by Multi-Person Tracking.

Remarks

This structure is returned as the output of the body pose feature when multi-person tracking is enabled.

Defined in: `nvAR_defs.h`

2.1.3. NvAR_TrackingBBoxes

Here is detailed information about the `NvAR_TrackingBBoxes` structure.

```
struct NvAR_TrackingBBoxes {
  NvAR_TrackingBBox *boxes;
  uint8_t num_boxes;
  uint8_t max_boxes;
};
```

Members

boxes

Type: `NvAR_TrackingBBox *`

Pointer to an array of tracking 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 body pose feature when multi-person tracking is enabled.

Defined in: `nvAR_defs.h`

2.1.4. NvAR_FaceMesh

Here is detailed information about the `NvAR_FaceMesh` structure.

```
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_triangles

Type: `size_t`

The number of mesh triangles.

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_defs.h`

2.1.5. NvAR_Frustum

Here is detailed information about the `NvAR_Frustum` structure.

```
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`

2.1.6. NvAR_FeatureHandle

Here is detailed information about the `NvAR_FeatureHandle` structure.

```
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`

2.1.7. NvAR_Point2f

Here is detailed information about the `NvAR_Point2f` structure.

```
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`

2.1.8. NvAR_Point3f

Here is detailed information about the NvAR_Point3f structure.

```
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`

2.1.9. NvAR_Quaternion

Here is detailed information about the `NvAR_Quaternion` structure.

```
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`

2.1.10. NvAR_Rect

Here is detailed information about the `NvAR_Rect` structure.

```
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`

2.1.11. NvAR_RenderingParams

Here is detailed information about the `NvAR_RenderingParams` structure.

```
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`

2.1.12. NvAR_Vector2f

Here is detailed information about the `NvAR_Vector2f` structure.

```
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`

2.1.13. NvAR_Vector3f

Here is detailed information about the `NvAR_Vector3f` structure.

```
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`

2.1.14. NvAR_Vector3u16

Here is detailed information about the `NvAR_Vector3u16` structure.

```
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.

```
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
- `NVCV_ERR_FEATURENOTFOUND`
- `NVCV_ERR_INITIALIZATION`

Remarks

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

2.2.2. NvAR_Destroy

Here is detailed information about the `NvAR_Destroy` structure.

```
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.

```
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.

```
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. Refer to Summary of NVIDIA AR SDK Accessor Functions for more information.

2.2.5. NvAR_GetCudaStream

Here is detailed information about the `NvAR_GetCudaStream` structure.

```
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.

```
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.

```
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.

```
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.

```
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 *`

Refer to 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.

```
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 *`

Refer to 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.

```
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 *`

Refer to 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,
```

```
   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.

```
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.

```
NvAR_SetCudaStream(
   NvAR_FeatureHandle handle,
   const char *name,
   CUStream 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.

```
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.

```
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.

```
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 *`

Refer to 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.

```
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 *

Refer to 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.

```
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.

```
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 *`

Refer to 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.

```
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.

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

NVCV_SUCCESS = 0
: Successful execution.

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

NVCV_ERR_UNIMPLEMENTED
: The requested feature is not implemented.

NVCV_ERR_MEMORY
: The requested operation requires more memory than is available.

NVCV_ERR_EFFECT
: An invalid effect handle has been supplied.

NVCV_ERR_SELECTOR
: The specified selector is not valid for this effect filter.

NVCV_ERR_BUFFER
: No image buffer has been specified.

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

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

NVCV_ERR_PIXELFORMAT
: The specified pixel format is not supported.

NVCV_ERR_MODEL
: An error occurred while the TRT model was being loaded.

NVCV_ERR_LIBRARY
: An error while the dynamic library was being loaded.

NVCV_ERR_INITIALIZATION
: The effect has not been properly initialized.

NVCV_ERR_FILE
: The specified file could not be found.

NVCV_ERR_FEATURENOTFOUND
: The requested feature was not found.

NVCV_ERR_MISSINGINPUT
: A required parameter was not set.

NVCV_ERR_RESOLUTION
: The specified image resolution is not supported.

NVCV_ERR_UNSUPPORTEDGPU
: The GPU is not supported.

NVCV_ERR_WRONGGPU
: The current GPU is not the one selected.

NVCV_ERR_UNSUPPORTEDDRIVER
The currently installed graphics driver is not supported.
NVCV_ERR_MODELDEPENDENCIES
There is no model with dependencies that match this system.
NVCV_ERR_PARSE
There has been a parsing or syntax error while reading a file.
NVCV_ERR_MODELSUBSTITUTION
The specified model does not exist and has been substituted.
NVCV_ERR_READ
An error occurred while reading a file.
NVCV_ERR_WRITE
An error occurred while writing a file.
NVCV_ERR_PARAMREADONLY
The selected parameter is read-only.
NVCV_ERR_TRT_ENQUEUE
TensorRT enqueue failed.
NVCV_ERR_TRT_BINDINGS
Unexpected TensorRT bindings.
NVCV_ERR_TRT_CONTEXT
An error occurred while creating a TensorRT context.
NVCV_ERR_TRT_INFER
There was a problem creating the inference engine.
NVCV_ERR_TRT_ENGINE
There was a problem deserializing the inference runtime engine.
NVCV_ERR_NPP
An error has occurred in the NPP library.
NVCV_ERR_CONFIG
No suitable model exists for the specified parameter configuration.
NVCV_ERR_TOOSMALL
The supplied parameter or buffer is not large enough.
NVCV_ERR_TOOBIG
The supplied parameter is too big.
NVCV_ERR_WRONGSIZE
The supplied parameter is not the expected size.
NVCV_ERR_OBJECTNOTFOUND
The specified object was not found.
NVCV_ERR_SINGULAR
A mathematical singularity has been encountered.
NVCV_ERR_NOTHINGRENDERED
Nothing was rendered in the specified region.
NVCV_ERR_OPENGL
An OpenGL error has occurred.
NVCV_ERR_DIRECT3D
A Direct3D error has occurred.
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 many other CUDA-related errors that are not listed here. However, the function `NvCV_GetErrorStringFromCode()` will turn the error code into a string to help you debug.

Appendix A. NVIDIA 3DMM File Format

The NVIDIA 3DMM file format is based on encapsulated objects that are scoped by a FOURCC tag and a 32-bit size. The header must appear first in the file. The objects and their subobjects can appear in any order. In this guide, they are listed in the default order.

A.1. Header

The header contains the following information:

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

NFAC				
size				
	endian	sizeBits	indexBits	zero
	TOC loc			

A.2. Model Object

The model object contains a shape component and an optional color component.

Both objects contain the following information:

- A mean shape.
- A set of shape modes.
- The eigenvalues for the modes.
- A triangle list.

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

A.3. IBUG Mappings Object

Here is some information about the IBUG mappings object.

The IBUG mappings object contains the following information:

- Landmarks

- Right contour
- Left contour

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

A.4. Blend Shapes Object

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

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

A.5. Model Contours Object

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

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

A.6. Topology Object

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

TOPO		
size		
	AJFC	
	size	
		adjacent faces
	AJVX	
	size	
		adjacent vertices

A.7. NVIDIA Landmarks Object

NVIDIA expands the number of landmarks from 68 to 126, including more detailed contours on the left and right.

NVLM		
size		
	LMRK	
	size	
		Landmarks
	RCTR	
	size	
		Right contour
	LCTR	
	size	
		Left contour

A.8. Partition Object

This partitions the mesh into coherent submeshes of the same material, used for rendering.

PRTS			
size			
	PART		
	size		
		Partition index	
		Face index	

		Number of faces	
		Vertex index	
		Number of vertices	
		Smoothing group	
		NAME	
		size	
			Partition name string
		MTRL	
	PART	size	
			Material name string
		(any number of additional partitions)	
	...		

A.9. Table of Contents Object

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

TOC0		
size		
	record size	
	tag	
	offset	
	tag	
	offset	
	...	

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.

Here is a list of the skeletal structures:

Keypoint	Parent
Pelvis	None, as this is root.
Left hip	Pelvis
Right hip	Pelvis
Torso	Pelvis
Left knee	Left hip
Right knee	Right hip
Neck	Torso
Left ankle	Left knee
Right ankle	Right knee
Left big toe	Left ankle
Right big toe	Right ankle
Left small toe	Left ankle
Right small toe	Right ankle
Left heel	Left ankle
Right heel	Right ankle
Nose	Neck

Keypoint	Parent
Left eye	Nose
Right eye	Nose
Left ear	Nose
Right ear	Nose
Left shoulder	Neck
Right shoulder	Neck
Left elbow	Left shoulder
Right elbow	Right shoulder
Left wrist	Left elbow
Right wrist	Right elbow
Left pinky knuckle	Left wrist
Right pinky knuckle	Right wrist
Left middle tip	Left wrist
Right middle tip	Right wrist
Left index knuckle	Left wrist
Right index knuckle	Right wrist
Left thumb tip	Left wrist
Right thumb tip	Right wrist

B.2. NvAR_Parameter_Output(KeyPoints) Order

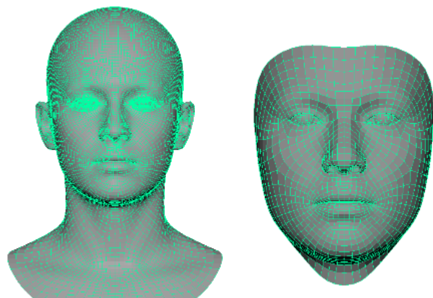
Here is some information about the order of the keypoints.

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

Appendix C. 3DMM Versions

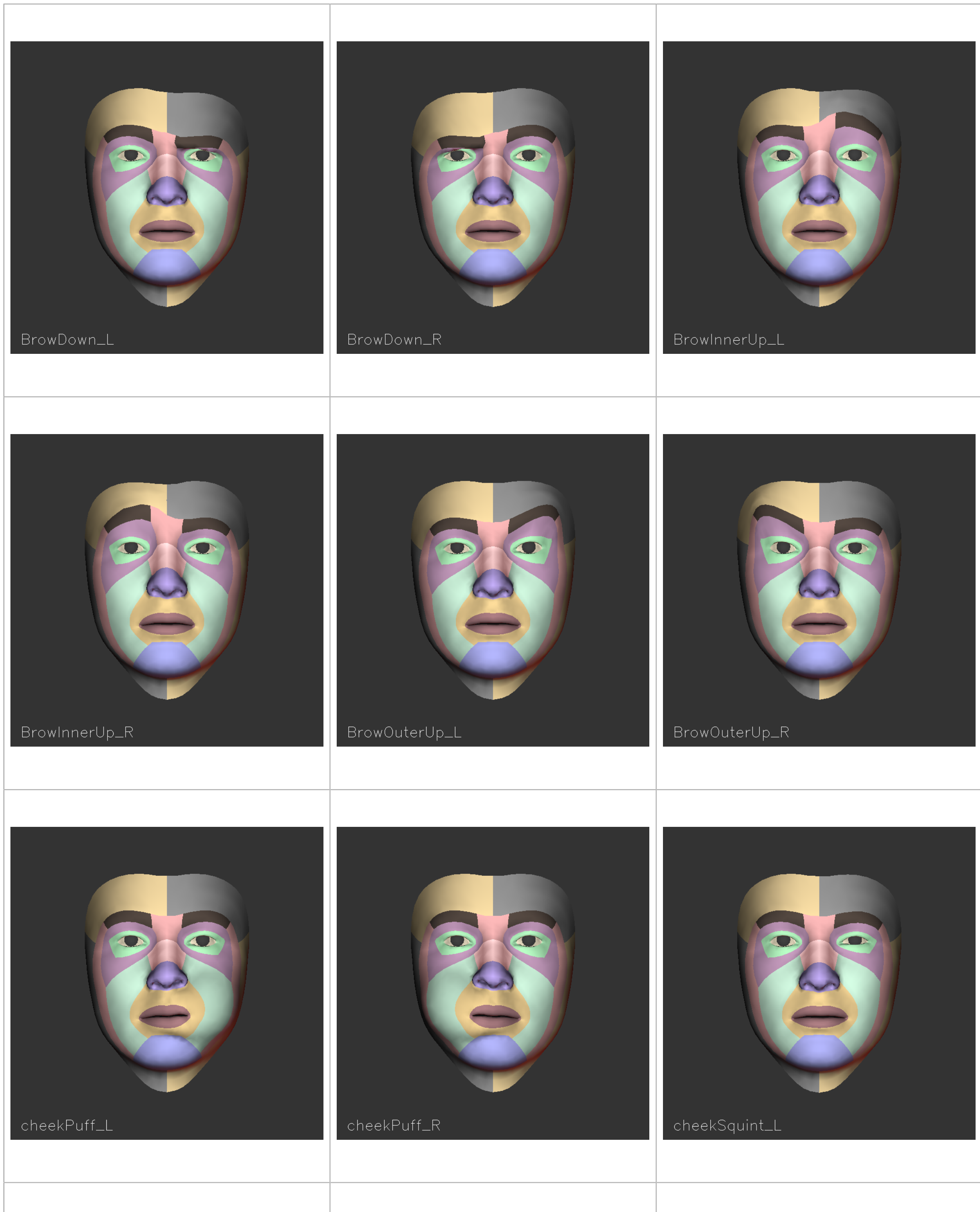
With the SDK comes a default face model used by the face fitting feature. This model is a modification of the ICT Face Model https://github.com/ICT-VGL/ICT-FaceKit. The modified version of the face model is optimized for real time face fitting applications and is therefore of lower resolution. This model is the version `face_model2.nvf`. In addition to the blendshapes provided by the ICT Model, it uses linear blendshapes for eye gaze expressions, which enables it to be used in implicit gaze tracking.

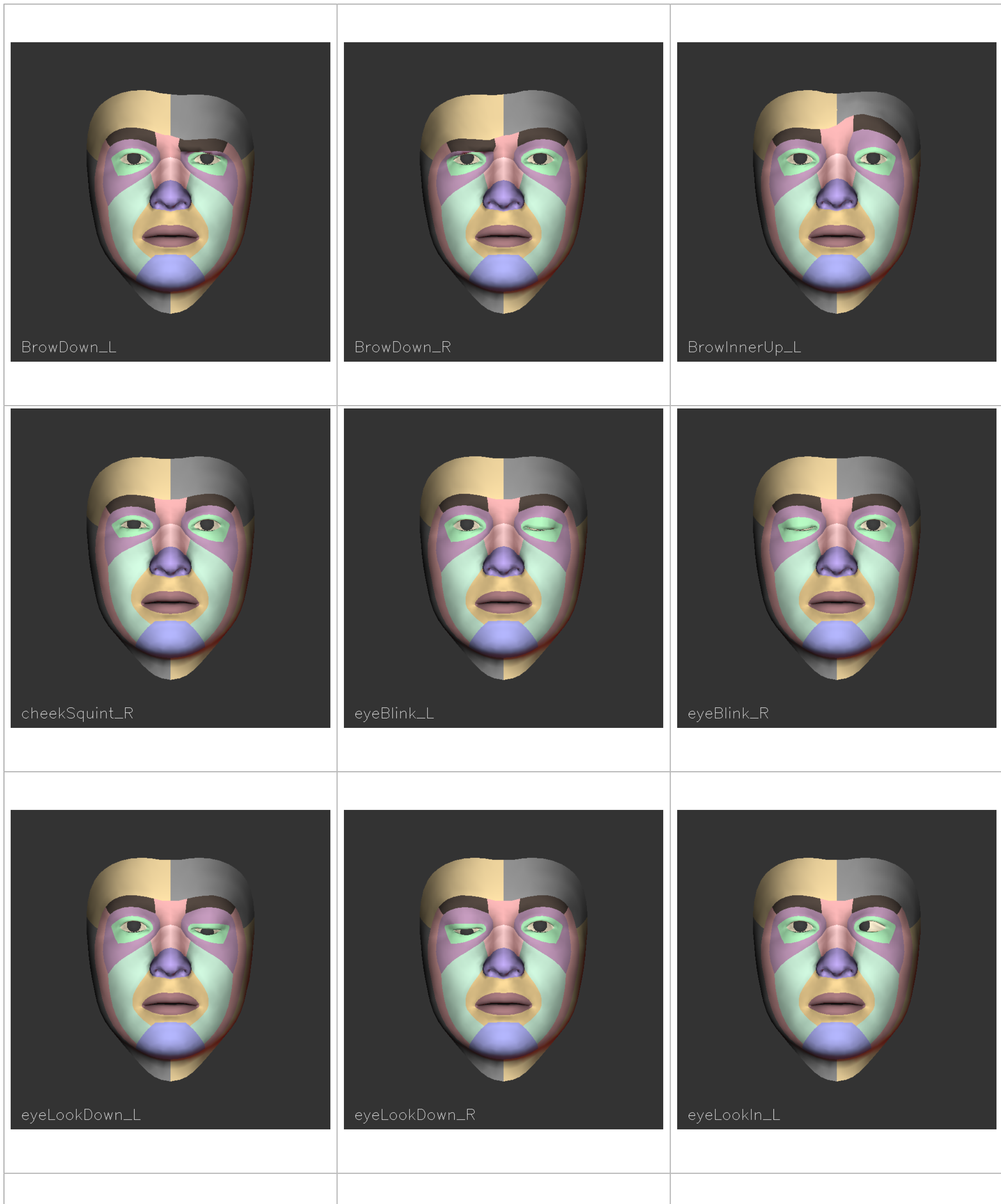
In the following graphic, on the left is the original ICT face model topology, and on the right is the modified `face_model2.nvf` face model topology.

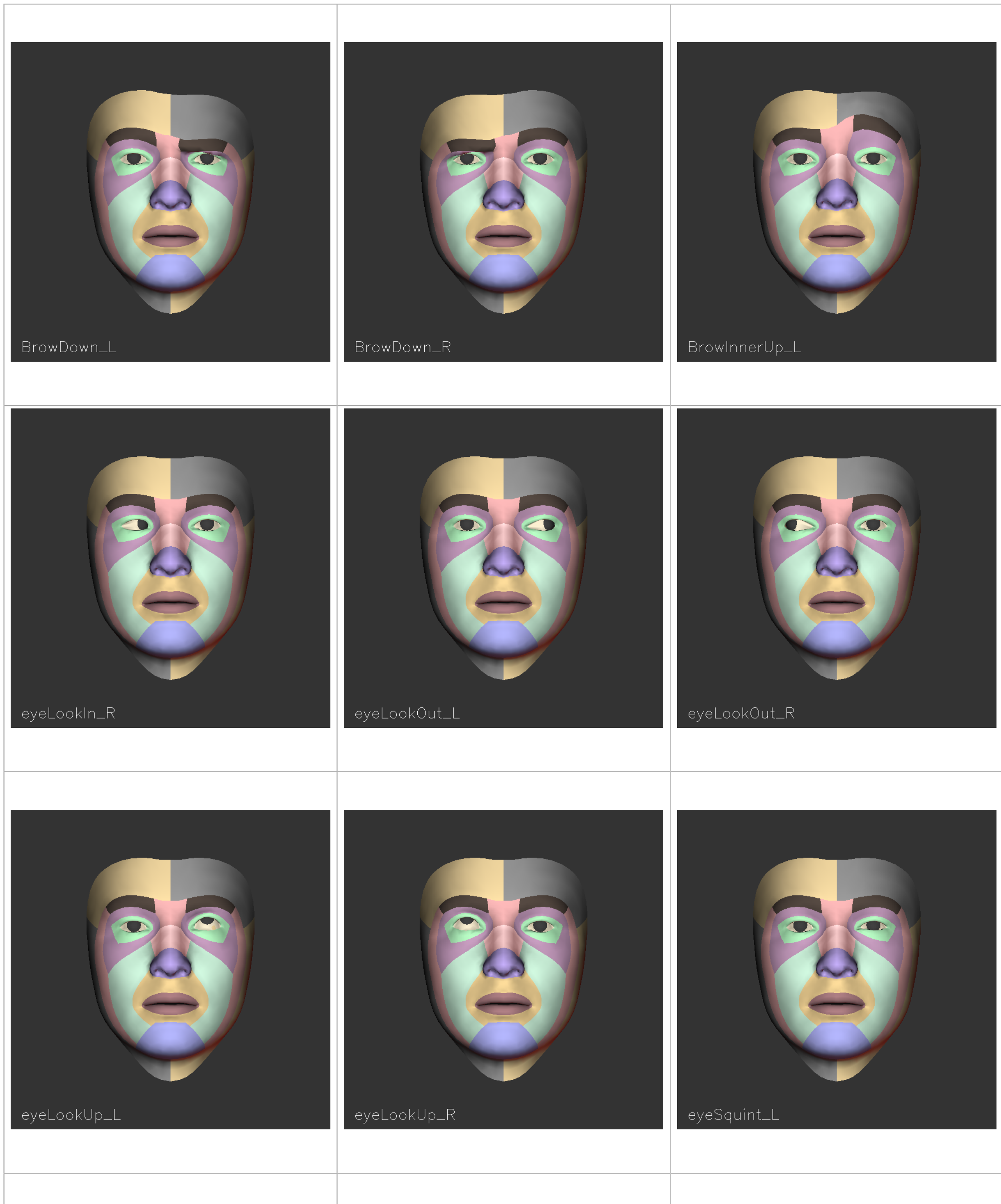


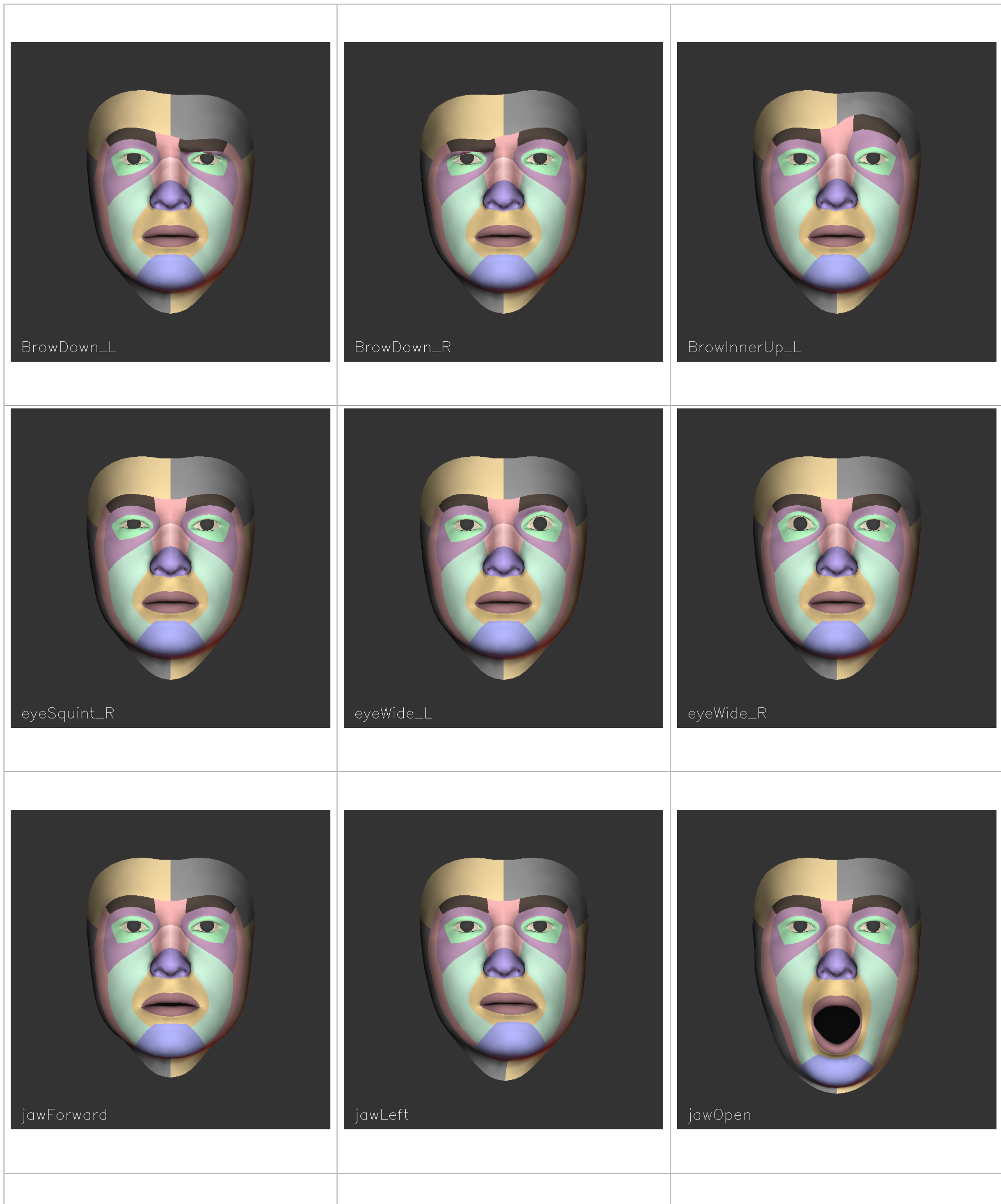
C.1. Face Expression List

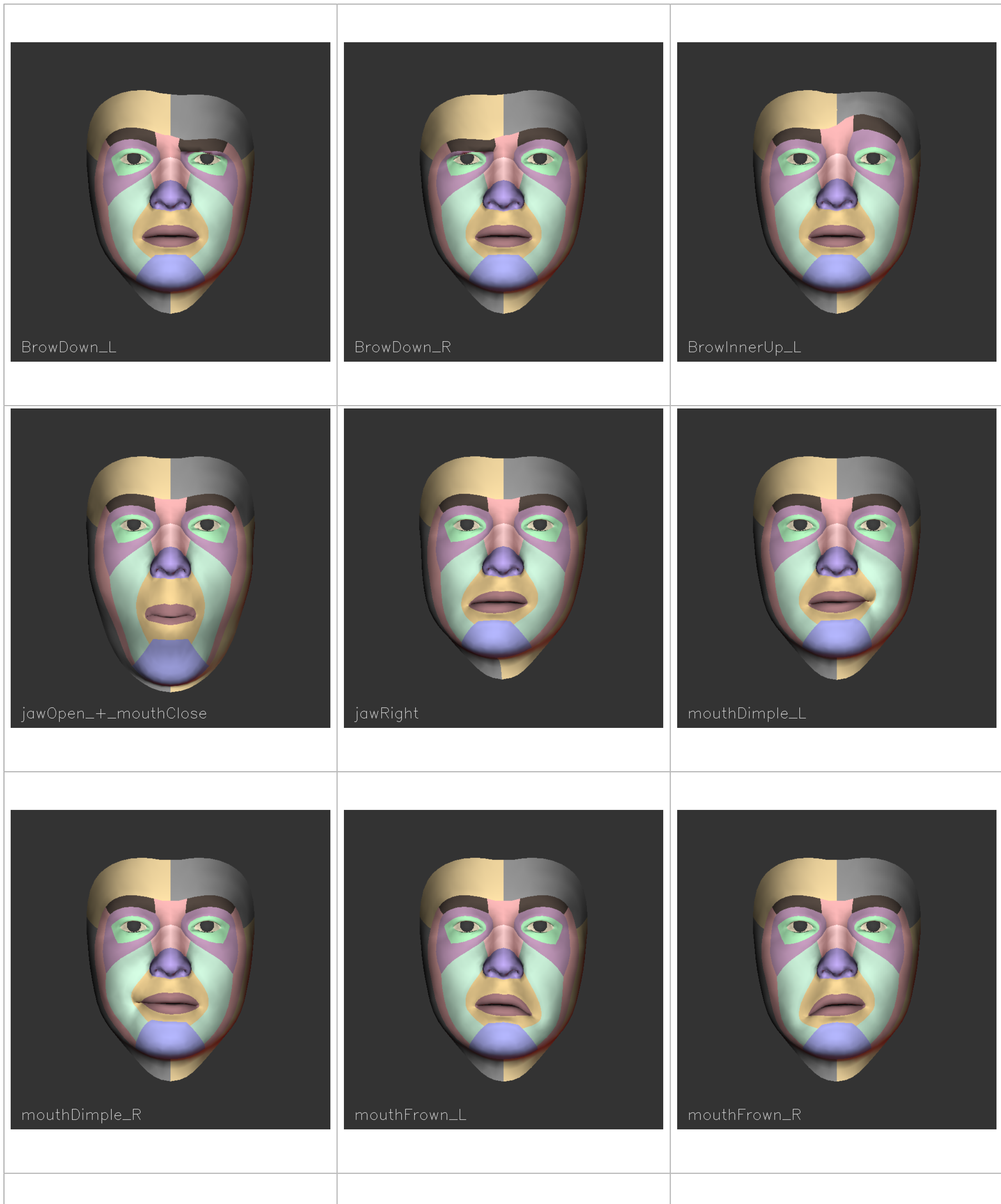
Here is a graphic visualization of all expression blendshapes used in the face expression estimation feature.

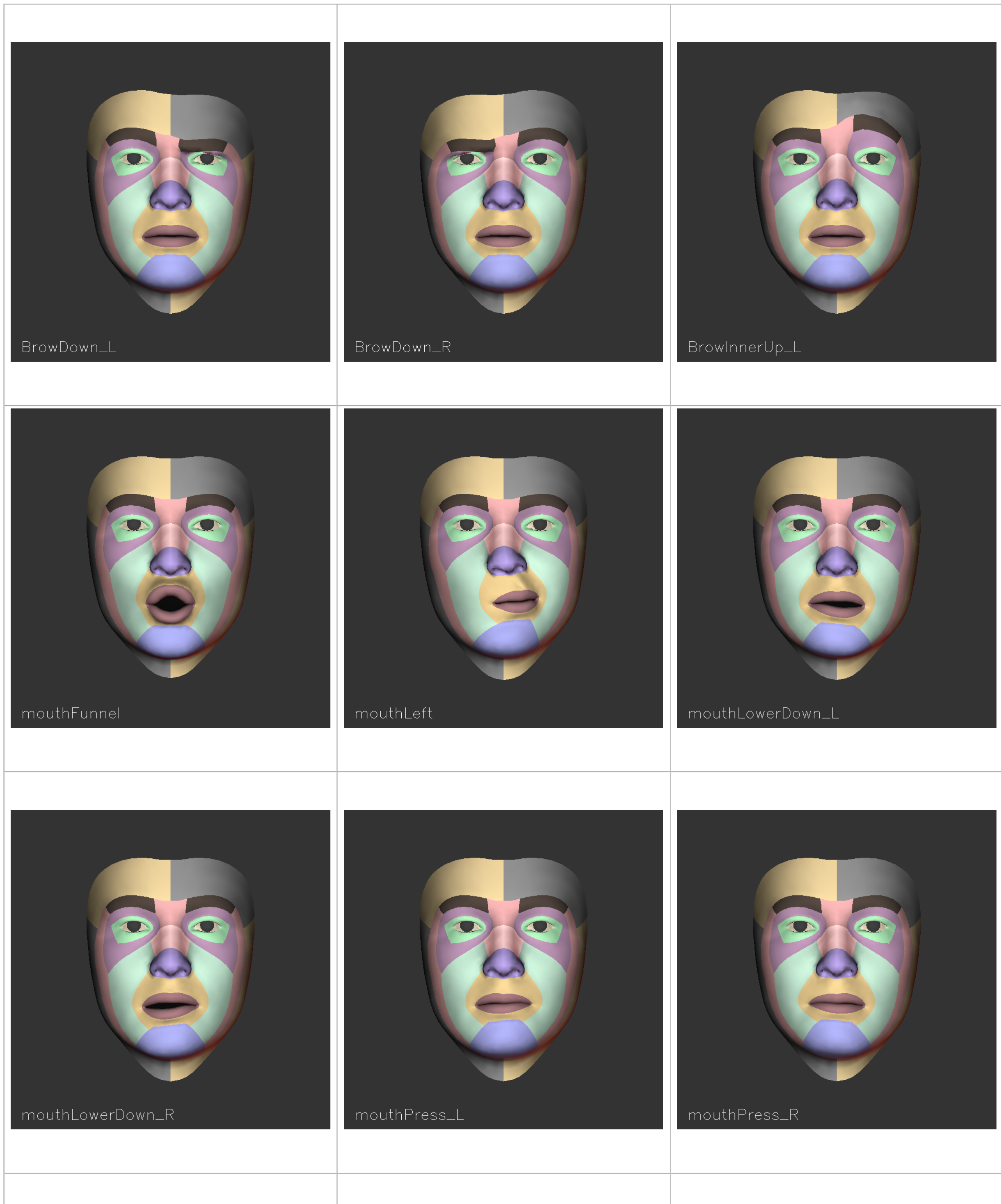

BrowDown_L
BrowDown_R
BrowInnerUp_L
BrowInnerUp_R
BrowOuterUp_L
BrowOuterUp_R
cheekPuff_L
cheekPuff_R
cheekSquint_L

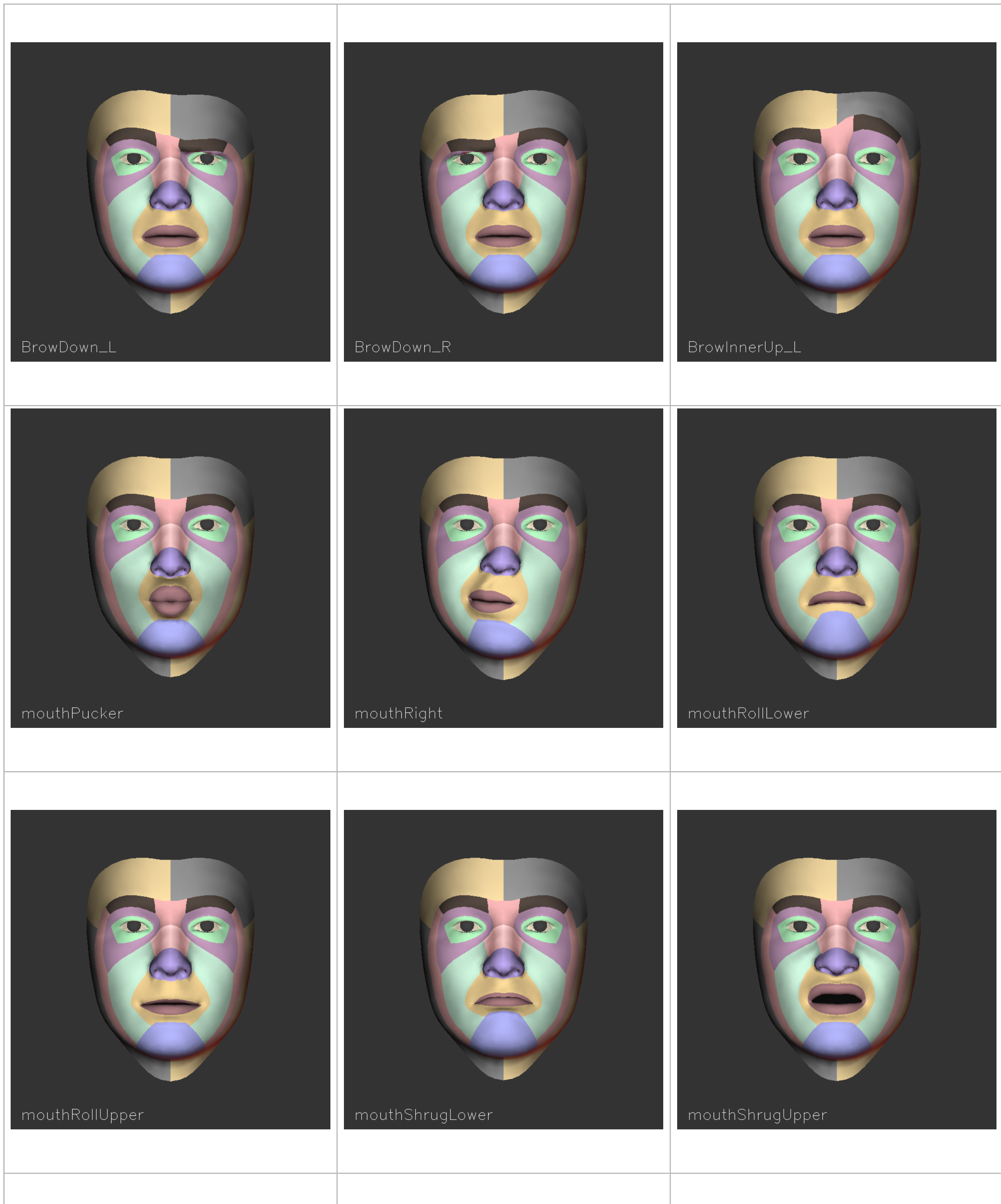

BrowDown_L
BrowDown_R
BrowInnerUp_L
cheekSquint_R
eyeBlink_L
eyeBlink_R
eyeLookDown_L
eyeLookDown_R
eyeLookIn_L

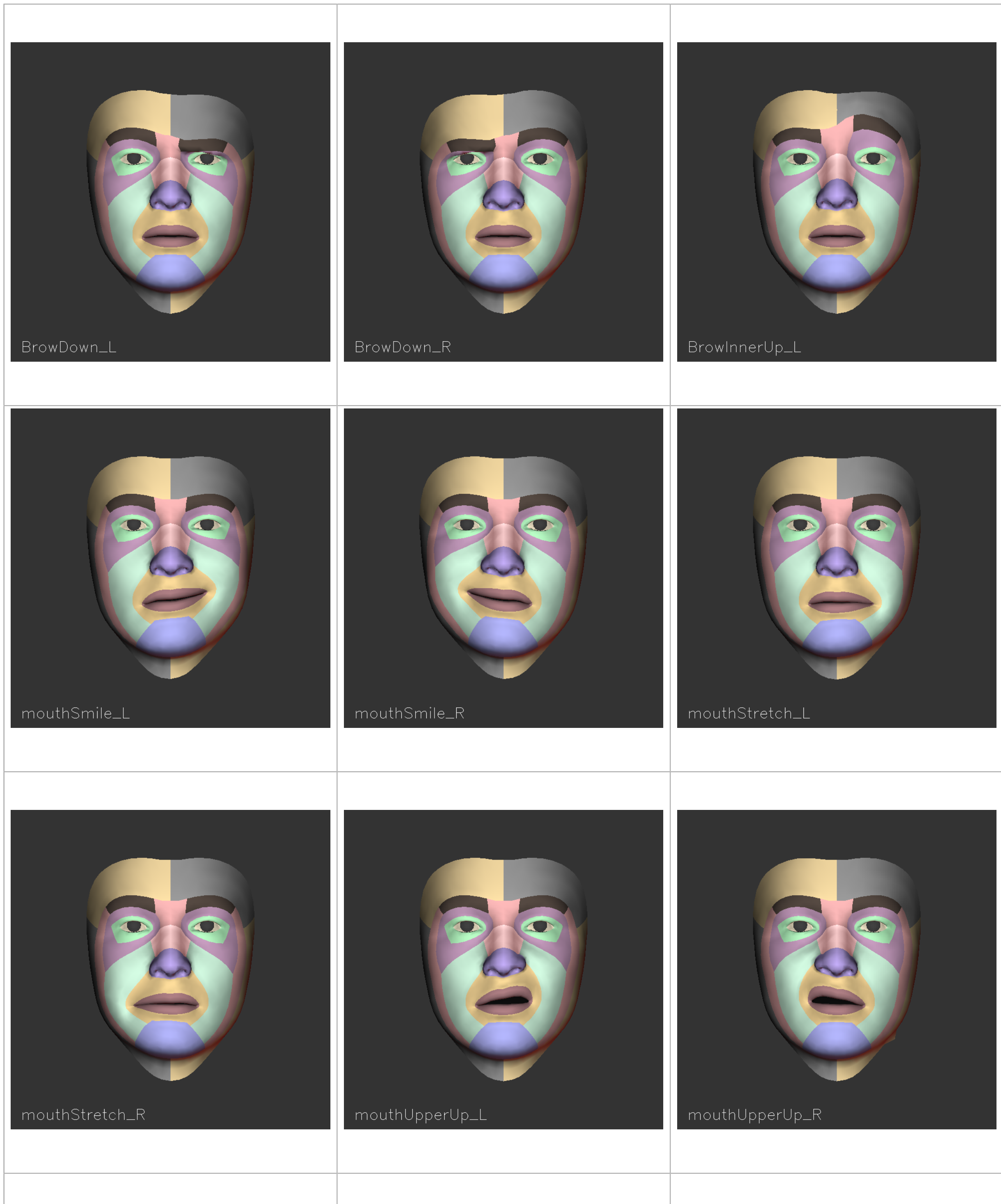

BrowDown_L
BrowDown_R
BrowInnerUp_L
eyeLookIn_R
eyeLookOut_L
eyeLookOut_R
eyeLookUp_L
eyeLookUp_R
eyeSquint_L

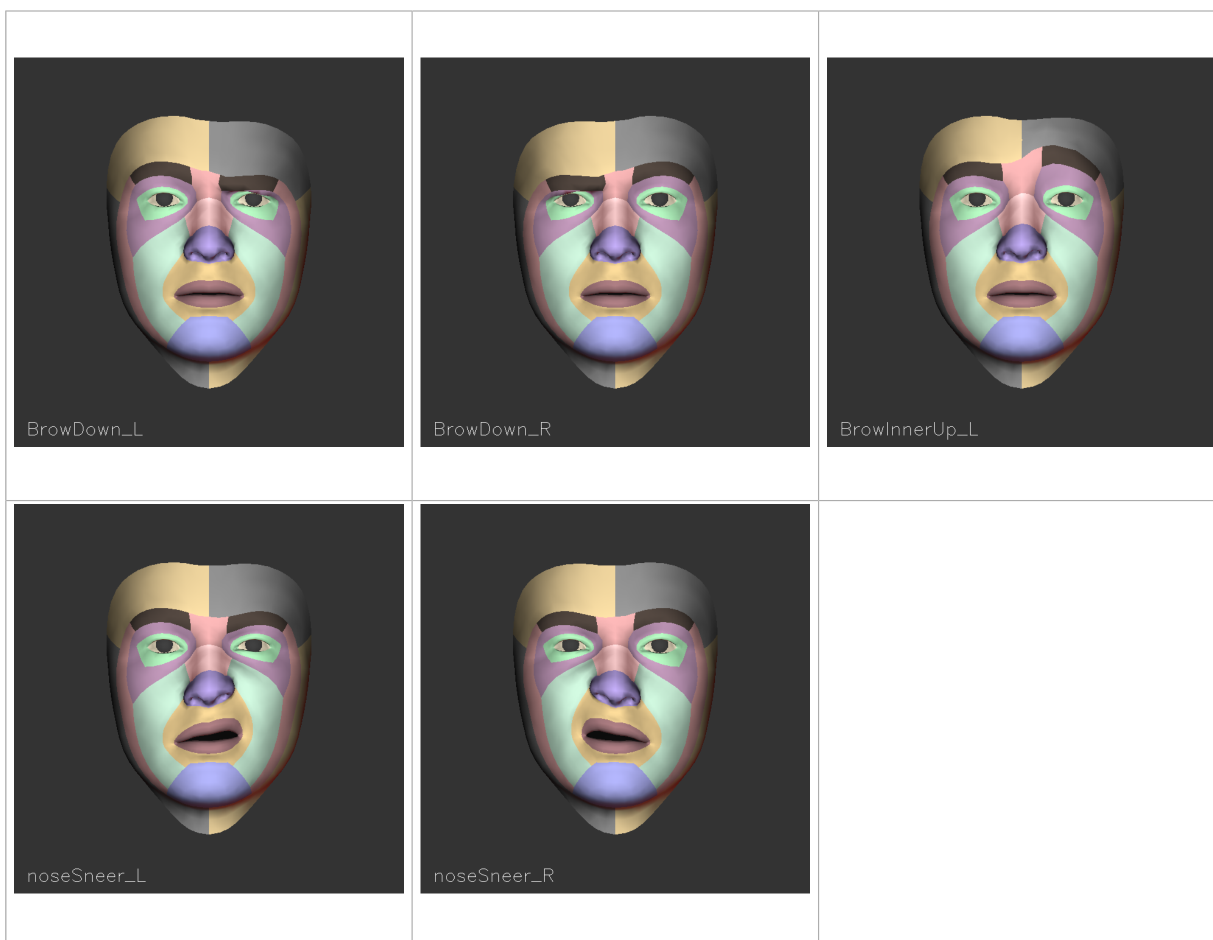

BrowDown_L
BrowDown_R
BrowInnerUp_L
eyeSquint_R
eyeWide_L
eyeWide_R
jawForward
jawLeft
jawOpen


BrowDown_L
BrowDown_R
BrowInnerUp_L
jawOpen_+_mouthClose
jawRight
mouthDimple_L
mouthDimple_R
mouthFrown_L
mouthFrown_R


BrowDown_L
BrowDown_R
BrowInnerUp_L
mouthFunnel
mouthLeft
mouthLowerDown_L
mouthLowerDown_R
mouthPress_L
mouthPress_R


BrowDown_L
BrowDown_R
BrowInnerUp_L
mouthPucker
mouthRight
mouthRollLower
mouthRollUpper
mouthShrugLower
mouthShrugUpper


BrowDown_L
BrowDown_R
BrowInnerUp_L
mouthSmile_L
mouthSmile_R
mouthStretch_L
mouthStretch_R
mouthUpperUp_L
mouthUpperUp_R



Here is the face blendshape expression list that is used in the Face 3D Mesh feature and the Facial Expression Estimation feature:

- 0: BrowDown_L
- 1: BrowDown_R
- 2: BrowInnerUp_L
- 3: BrowInnerUp_R
- 4: BrowOuterUp_L
- 5: BrowOuterUp_R
- 6: cheekPuff_L
- 7: cheekPuff_R
- 8: cheekSquint_L
- 9: cheekSquint_R
- 10: eyeBlink_L
- 11: eyeBlink_R
- 12: eyeLookDown_L

- 13: eyeLookDown_R
- 14: eyeLookIn_L
- 15: eyeLookIn_R
- 16: eyeLookOut_L
- 17: eyeLookOut_R
- 18: eyeLookUp_L
- 19: eyeLookUp_R
- 20: eyeSquint_L
- 21: eyeSquint_R
- 22: eyeWide_L
- 23: eyeWide_R
- 24: jawForward
- 25: jawLeft
- 26: jawOpen
- 27: jawRight
- 28: mouthClose
- 29: mouthDimple_L
- 30: mouthDimple_R
- 31: mouthFrown_L
- 32: mouthFrown_R
- 33: mouthFunnel
- 34: mouthLeft
- 35: mouthLowerDown_L
- 36: mouthLowerDown_R
- 37: mouthPress_L
- 38: mouthPress_R
- 39: mouthPucker
- 40: mouthRight
- 41: mouthRollLower
- 42: mouthRollUpper
- 43: mouthShrugLower
- 44: mouthShrugUpper
- 45: mouthSmile_L
- 46: mouthSmile_R

- 47: mouthStretch_L
- 48: mouthStretch_R
- 49: mouthUpperUp_L
- 50: mouthUpperUp_R
- 51: noseSneer_L
- 52: noseSneer_R

The items in the list above can be mapped to the ARKit blendshapes using the following options:

- A01_Brow_Inner_Up = 0.5 * (browInnerUp_L + browInnerUp_R)
- A02_Brow_Down_Left = browDown_L
- A03_Brow_Down_Right = browDown_R
- A04_Brow_Outer_Up_Left = browOuterUp_L
- A05_Brow_Outer_Up_Right = browOuterUp_R
- A06_Eye_Look_Up_Left = eyeLookUp_L
- A07_Eye_Look_Up_Right = eyeLookUp_R
- A08_Eye_Look_Down_Left = eyeLookDown_L
- A09_Eye_Look_Down_Right = eyeLookDown_R
- A10_Eye_Look_Out_Left = eyeLookOut_L
- A11_Eye_Look_In_Left = eyeLookIn_L
- A12_Eye_Look_In_Right = eyeLookIn_R
- A13_Eye_Look_Out_Right = eyeLookOut_R
- A14_Eye_Blink_Left = eyeBlink_L
- A15_Eye_Blink_Right = eyeBlink_R
- A16_Eye_Squint_Left = eyeSquint_L
- A17_Eye_Squint_Right = eyeSquint_R
- A18_Eye_Wide_Left = eyeWide_L
- A19_Eye_Wide_Right = eyeWide_R
- A20_Cheek_Puff = 0.5 * (cheekPuff_L + cheekPuff_R)
- A21_Cheek_Squint_Left = cheekSquint_L
- A22_Cheek_Squint_Right = cheekSquint_R
- A23_Nose_Sneer_Left = noseSneer_L
- A24_Nose_Sneer_Right = noseSneer_R
- A25_Jaw_Open = jawOpen
- A26_Jaw_Forward = jawForward

- A27_Jaw_Left = jawLeft
- A28_Jaw_Right = jawRight
- A29_Mouth_Funnel = mouthFunnel
- A30_Mouth_Pucker = mouthPucker
- A31_Mouth_Left = mouthLeft
- A32_Mouth_Right = mouthRight
- A33_Mouth_Roll_Upper = mouthRollUpper
- A34_Mouth_Roll_Lower = mouthRollLower
- A35_Mouth_Shrug_Upper = mouthShrugUpper
- A36_Mouth_Shrug_Lower = mouthShrugLower
- A37_Mouth_Close = mouthClose
- A38_Mouth_Smile_Left = mouthSmile_L
- A39_Mouth_Smile_Right = mouthSmile_R
- A40_Mouth_Frown_Left = mouthFrown_L
- A41_Mouth_Frown_Right = mouthFrown_R
- A42_Mouth_Dimple_Left = mouthDimple_L
- A43_Mouth_Dimple_Right = mouthDimple_R
- A44_Mouth_Upper_Up_Left = mouthUpperUp_L
- A45_Mouth_Upper_Up_Right = mouthUpperUp_R
- A46_Mouth_Lower_Down_Left = mouthLowerDown_L
- A47_Mouth_Lower_Down_Right = mouthLowerDown_R
- A48_Mouth_Press_Left = mouthPress_L
- A49_Mouth_Press_Right = mouthPress_R
- A50_Mouth_Stretch_Left = mouthStretch_L
- A51_Mouth_Stretch_Right = mouthStretch_R
- A52_Tongue_Out = 0

Appendix D. Coordinate Systems

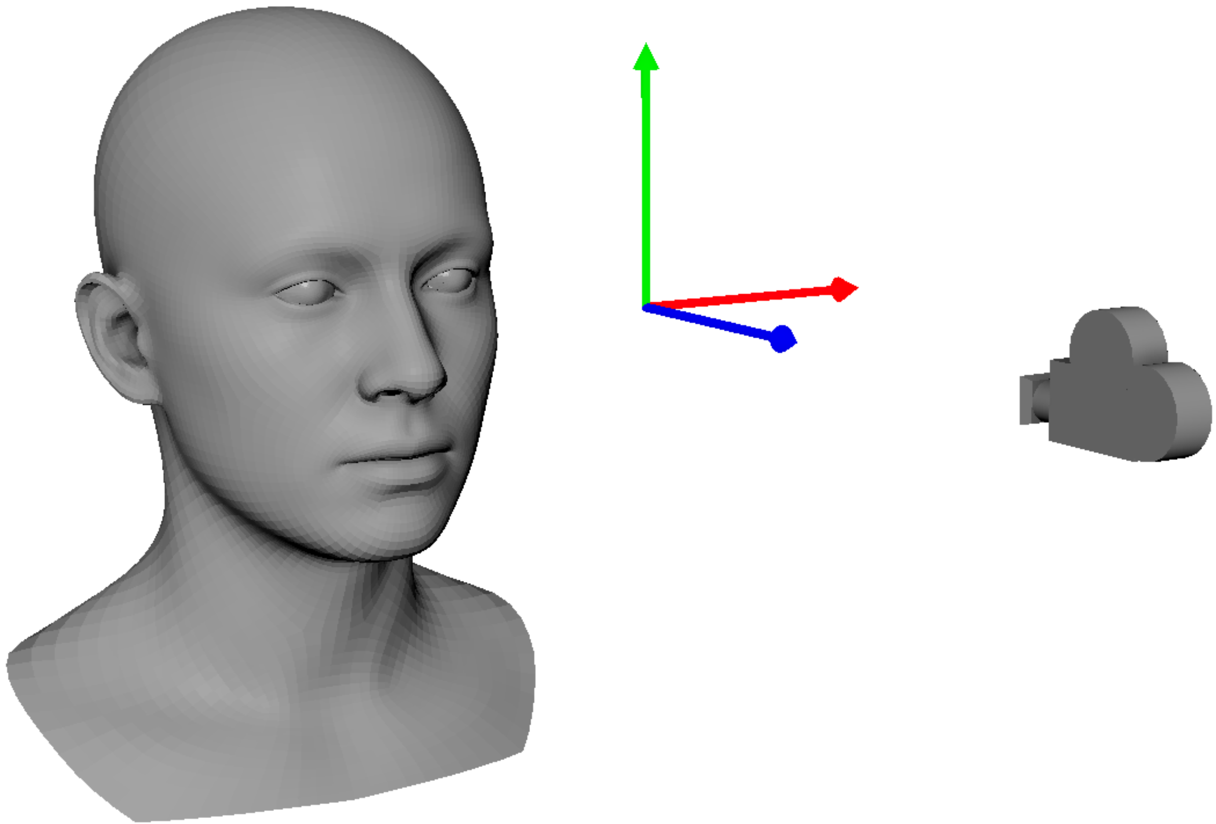
The documentation may refer to different coordinate systems where virtual 3D objects are defined. This appendix defines the different coordinate spaces that can be interfaced with through the SDK.

D.1. NvAR World 3D Space

Here is the face blendshape expression list that is used in the Face 3D Mesh feature and the Facial Expression Estimation feature:

NvAR World 3D space, or simply **world space** defines the main reference frame used in 3D applications. The reference frame is right handed, and the coordinate units are centimeters. The camera is usually, but not always, placed so that it is looking down the negative *z-axis*, with the *x-axis* pointing right, and the *y-axis* pointing up. The face model is usually, but not always placed so that the positive *z-axis* points out from the nose of the model, the *x-axis* points out from the left ear, and the *y-axis* points up from the head. This is so that no rotation of a face model or a camera model would correspond to the model being in view.

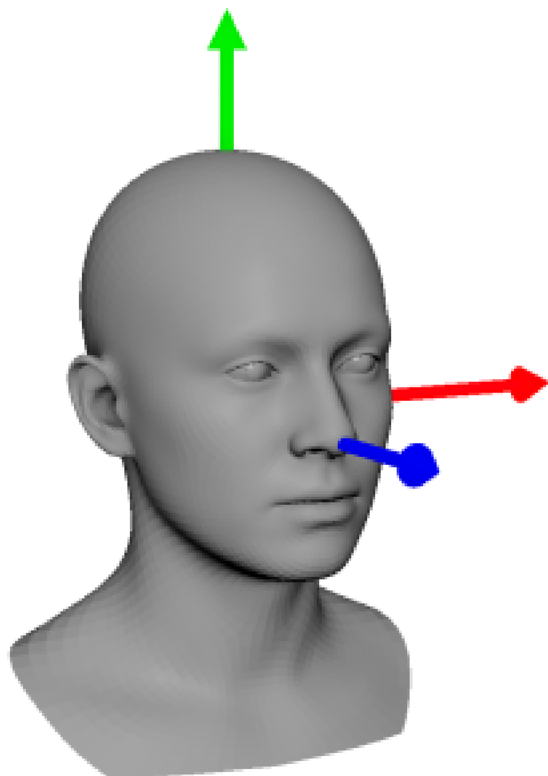
Figure 1. NvAR World 3D Space



D.2. NvAR Model 3D Space

In the **NvAR Model 3D Space**, or simply the **model space**, a face model has its origin in the center of the skull where the first neck joint would be placed. The reference frame is right handed, and the coordinate units are centimeters. The positive *z-axis* points out from the nose of the model, the *x-axis* points out from the left ear, and the *y-axis* points up from the head.

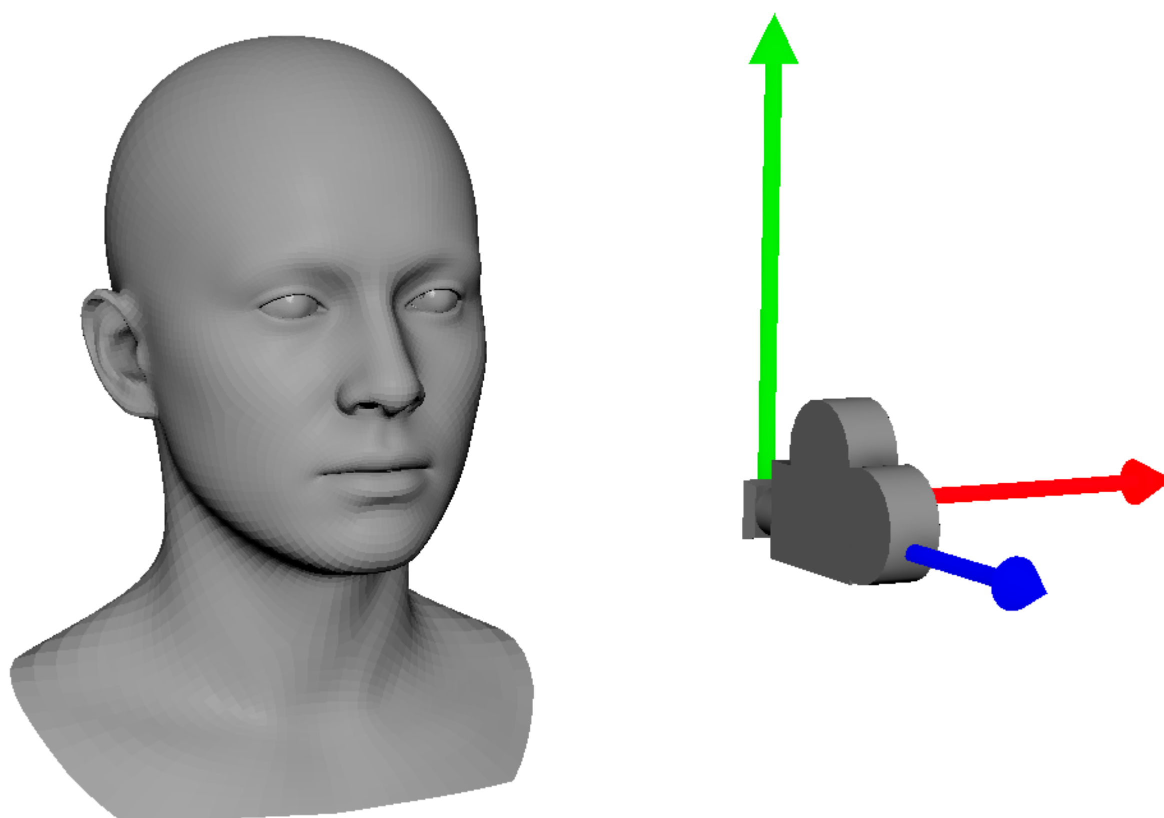
Figure 2. NvAR Model 3D Space



D.3. NvAR Camera 3D Space

In **NvAR Camera 3D Space** or simply the **camera space**, objects are placed in relation to a specific camera. The camera space can in many cases be the same as the world space unless explicit camera extrinsics are used. The reference frame is right handed, and the coordinate units are centimeters. The positive *x-axis* points to the right, the *y-axis* points up and the *z-axis* points back.

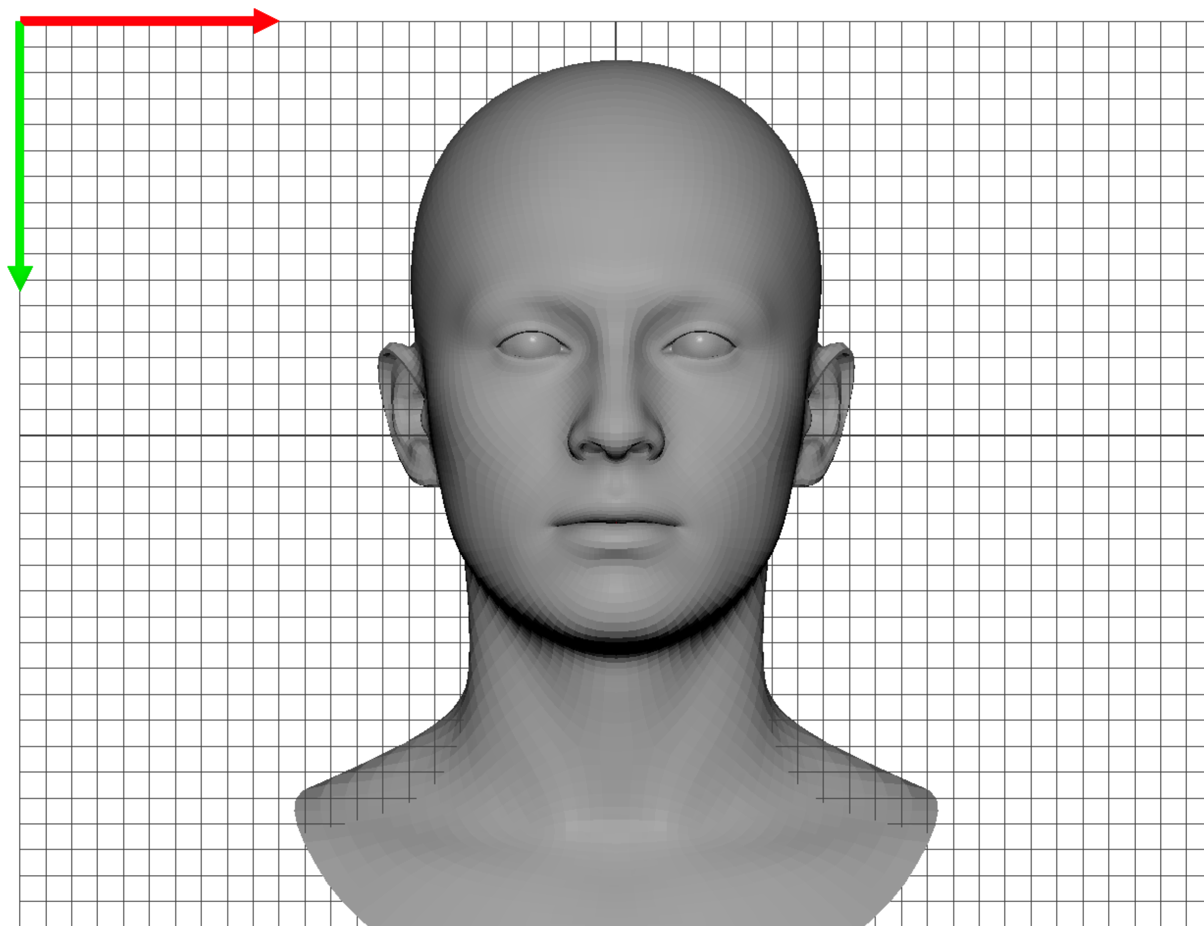
Figure 3. NvAR Camera 3D Space



D.4. NvAR Image 2D Space

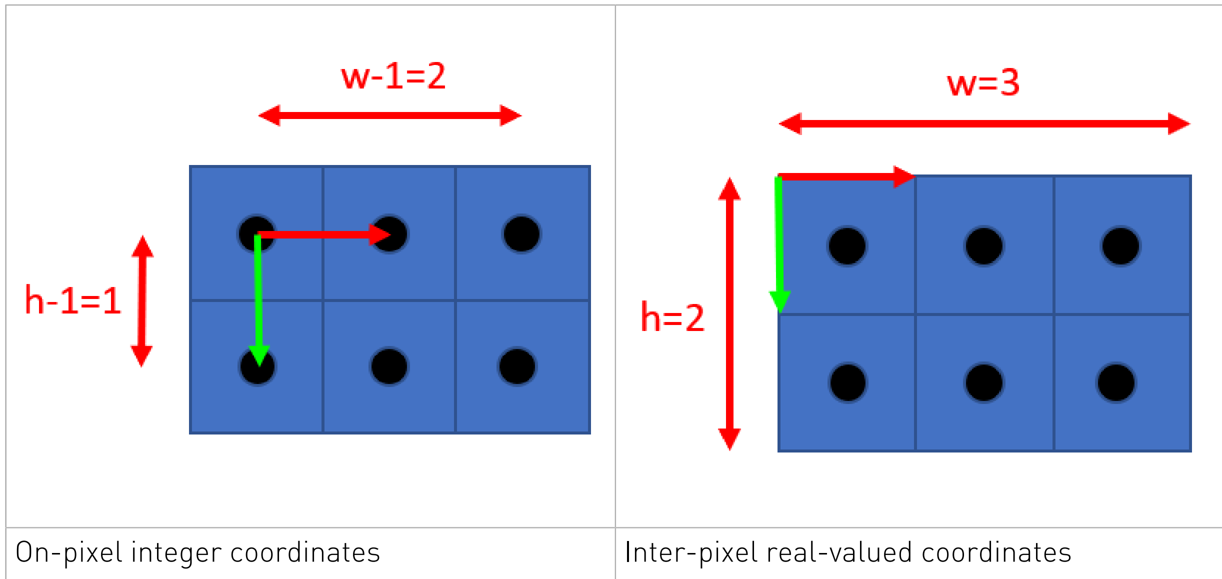
The **NvAR Image 2D Space**, or **screen space** is the main 2D space for screen coordinates. The origin of an image is the location of the uppermost, leftmost pixel; the *x-axis* points to the right and the *y-axis* points down. The unit of this coordinate system is pixels.

Figure 4. NvAR Image 2D Space



There are two different flavors of this coordinate system, either *on-pixel* or *inter-pixel.*

In the on-pixel system, the origin point is located on the center of the pixel. In the *inter-pixel* system, the origin is offset from the center of the pixel by a distance of -0.5 pixels along the *x-axis* and the *y-axis*. The *on-pixel* should be used for integer based coordinates, while the *inter-pixel* should be used for real valued coordinates (usually defined as floating point coordinates).



On-pixel integer coordinates	Inter-pixel real-valued coordinates

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, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on 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.

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
http://www.nvidia.com

NVIDIA

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, NVCaffe, 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

© 2021-2023 NVIDIA Corporation and affiliates. All rights reserved.

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
http://www.nvidia.com