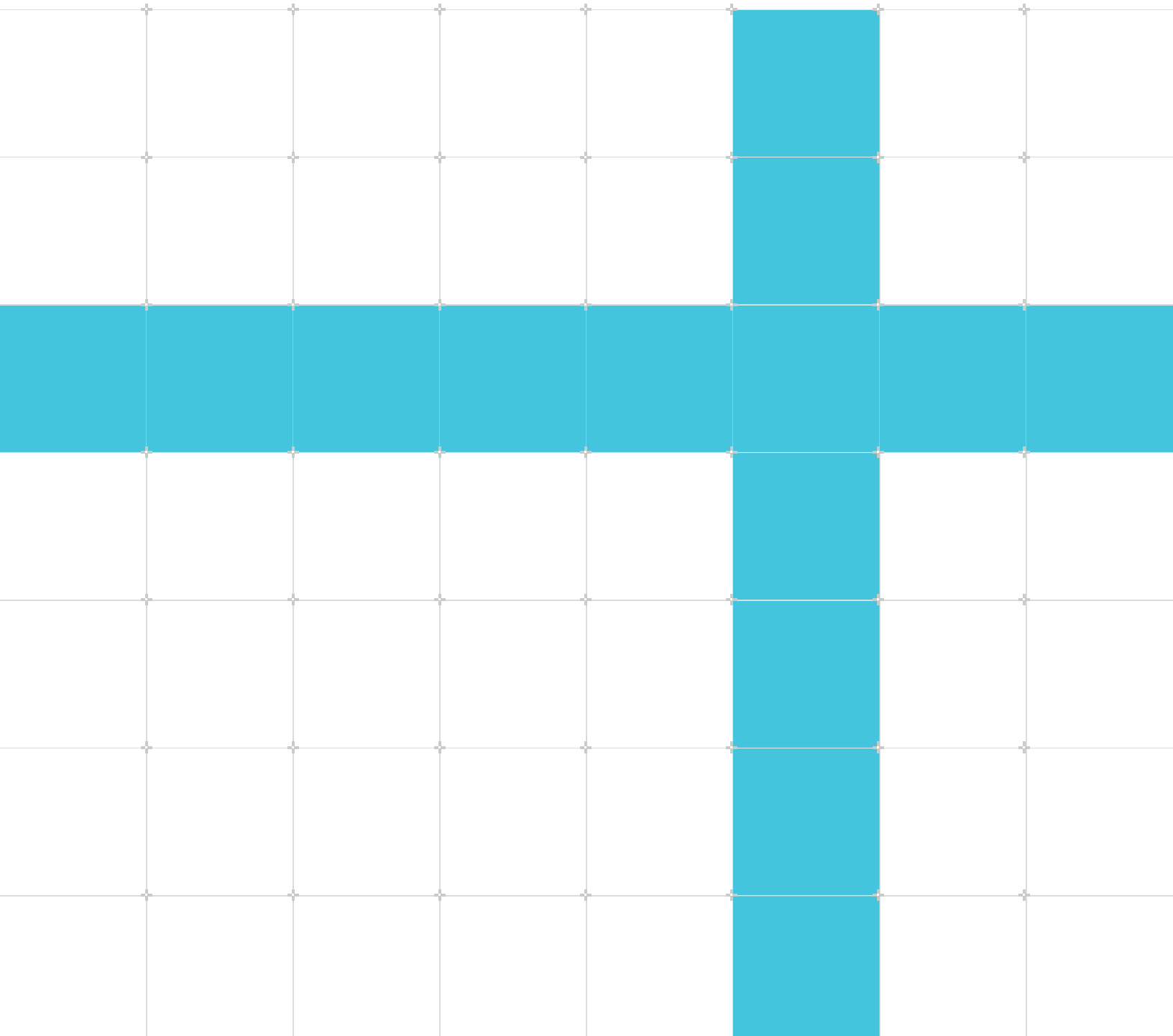
Arm® Accuracy Super Resolution™ for Generic Library

Version 2.0

Tutorial

Non-Confidential

Issue 02
110404_0200_02_en



Arm® Accuracy Super Resolution™ for Generic Library Tutorial

This document (110404_0200_02_en) was issued on 2025-05-01. There might be a later issue at https://developer.arm.com/documentation/110404

The product version is 2.0.

See also: Proprietary notice | Product and document information | Useful resources

Start reading

If you prefer, you can skip to the start of the content.

Intended audience

This document is for software developers who want to use Arm ASR using a generic library implementation.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com.

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Contents

1. Introduction

This tutorial shows you how to get started with Arm® Accuracy Super Resolution™ (Arm ASR) using a generic library implementation.

Arm ASR is a mobile-optimized temporal upscaling technique derived from AMD's FidelityFX™ Super Resolution 2 v2.2.2.

The main features of Arm ASR are:

- It has multiple optimizations on top of the original FidelityFX Super Resolution 2 to make the technique suited for the more resource-constrained environment of mobile gaming.
- Uses temporal accumulation to reconstruct high-resolution images while maintaining fine geometric and texture details compared to native rendering.
- Enables practical performance for costly render operations, for example, hardware ray tracing.

This tutorial is aimed at game developers who want to apply upscaling techniques to their projects. You will learn how to install Arm ASR and some of the common tasks that you might encounter when setting up Arm ASR for the first time.

1.1 Contact

You can reach out to us using the following email address arm-asr-support@arm.com.

1.2 Before you begin

You must have Git installed.

You must also clone the Generic Library repository:

```
git clone https://github.com/arm/accuracy-super-resolution-generic-library
```

2. Integration

You must use a quick integration or a tight integration to implement Arm ASR in your custom engine.

2.1 Quick integration

If you want to quickly integrate Arm ASR, you can use the built-in standalone backend.

Procedure

1. Copy the `Arm_ASR` directory into your project. If you want to use the prebuilt shaders, add `Arm_ASR/src/backends/shared/blob_accessors/prebuilt_shaders` to the include path.
2. Include the following header files in your codebase where you wish to interact with the technique:

```
$ARMASR_DIR/include/host/ffxm_fsr2.h#L1
$ARMASR_DIR/include/host/backends/vk/ffxm_vk.h#L1
```

3. Create a Vulkan backend:
   a. Allocate a Vulkan scratch buffer. The size of the buffer must be the same size as returned by `ffxmGetScratchMemorySizeVK`.
   b. Use `ffxmGetDeviceVK` to create `FfxmDevice`.
   c. Create `FfxmInterface` by calling `ffxmGetInterfaceVK`.
4. Create a context by calling `ffxmFsr2ContextCreate`. The parameters structure must match the configuration of your application. For more information, see Integration guidelines.
5. To record the workload of the technique, each frame calls `ffxmFsr2ContextDispatch` via `$ARMASR_DIR/include/host/ffxm_fsr2.h#L337`. You should fill out the parameters structure to match the configuration of your application. For more information, see Integration guidelines.
6. When your application is terminating, or if you want to destroy the context, call `ffxmFsr2ContextDestroy`.


Note

The GPU must be idle before calling the `ffxmFsr2ContextDestroy` function.

7. You must add sub-pixel jittering to the projection matrix of your application. You should add sub-pixel jittering during the main rendering of your application. To compute the precise jitter offsets, use the `ffxmFsr2GetJitterOffset` function. For more information, see Camera jitter.
8. When texturing, you should add a global mip bias. For more information, see Mipmap biasing.

9. For the best upscaling quality, we strongly advise that you populate the `Reactive mask`. As a starting point, you can also use `ffxmFsr2ContextGenerateReactiveMask`. For more information, see Reactive mask.
10. Link the two built libraries, `Arm_ASR_api` and `Arm_ASR_backend`.

Results

You are now ready to use Arm ASR in your game engine projects using the built-in standalone backend.

2.2 Tight integration

If you want to use your own backend or renderer, a tight integration with your engine is required. This is a similar process to the Quick integration but with the added requirement to fill the `FfxmInterface` accessed via `$ARMASR_DIR/include/host/ffxm_interface.h` with functions implemented on your end.

About this task

In a tight integration, we expect the engine to build the shaders. Arm ASR shaders have been micro-optimized to use explicit 16-bit floating-point types. Therefore, we advise that you build the shaders using such types. For example, `min16float` is used in High-Level Shader Language (HLSL) and `float16_t` in OpenGL Shading Language (GLSL). If you are using HLSL, define the following symbol with a value of `1`:

```
#define FFXM_HLSL_6_2 1
```

The `FFXM_HALF` symbol is enabled by default in the provided shader sources.

Procedure

1. Add the `ffxm_interface.h` header file to your codebase.
2. To link Arm ASR with the engine renderer, implement your own functions, for example `xxxGetInterfacexxx` and `xxxGetScratchMemorySizexxx` and callbacks in `FfxmInterface`.
3. Create your own backend by calling `xxxGetInterfacexxx`. Allocate a scratch buffer of the size returned by calling `xxxGetScratchMemorySizexxx`. The pointer to the scratch buffer must be passed to `xxxGetInterfacexxx`.
4. The remainder of the steps are the same as listed in Quick integration from creating an Arm ASR context (step 4) except you only need to link to `Arm_ASR_api`.

HLSL-based workflows

In an HLSL-based workflow using a DirectX Shader Compiler to cross-compile to SPIR-V, use the following flags when building:

```
-fspv-target-env=vulkan1.1spirv1.4 -enable-16bit-types
```

Use the `VK_KHR_shader_float16_int8` extension at runtime.

Results

You are now ready to use Arm ASR in your game engine projects.

3. Integration guidelines

This section describes the features of Arm ASR and how to implement the features correctly.

3.1 Quality presets

Arm ASR provides a number of shader quality presets. You can use the presets to select a more efficient version of the technique to apply quality and performance tradeoffs.

The presets are:

Quality

This preset is an extremely optimized version of Fsr2 that maintains the same image quality as the original technique.

Balanced

This preset gives a significant improvement in both bandwidth savings and performance uplift while maintaining close image quality to the Quality preset.

Performance

This preset is more aggressive and gives you the highest performance but with some quality sacrifices.

When creating a context, you must provide a `FfxmFsr2ShaderQualityMode` as part of the input settings in `FfxmFsr2ContextDescription`.

3.2 Upscaling ratios

To improve the flexbility when working with the technique, we have separated the shader quality presets and the upscaling ratio. Therefore, you can choose any combination of `FfxmFsr2ShaderQualityMode` and `FfxmFsr2UpscalingRatio` to fine tune the quality and performance of your applications.

We provide some utilities to get the upscaled output resolution from the frame. The `ffxmFsr2GetRenderResolutionFromUpscalingRatio` helper function calculates the rendering resolution from a target resolution and the desired upscaling ratio (`FfxmFsr2UpscalingRatio`).

```
FfxErrorCode ffxmFsr2GetRenderResolutionFromUpscalingRatio(
    uint32_t* renderWidth,
    uint32_t* renderHeight,
    uint32_t displayWidth,
    uint32_t displayHeight,
    FfxmFsr2UpscalingRatio upscalingRatio)
```

3.3 Shader variants and extensions

Unless you are using the prebuilt shaders with the standalone Vulkan backend, when doing the integration of the Arm ASR shaders you must be aware of some defines.

The defines you must be aware of are:

FFXM_GPU

Needs to be defined globally when including the shader headers.

FFXM_HLSL

If defined, the logic will fallback to use the High-Level Shader Language (HLSL) specific syntax.

FFXM_GLSL

If defined, the logic will fallback to use the OpenGL Shading Language (GLSL) specific syntax.

The following table lists the available shader mutators. You must define each mutator with a value of `0` or `1`. Which shader variant to use is guided internally by `getPipelinePermutationFlags` based on your flags and shader quality.

Define	Description
FFXM_FSR2_OPTION_HDR_COLOR_INPUT	If 1, assumes that the input color is in linear RGB.
FXM_FSR2_OPTION_LOW_RESOLUTION_MOTION_VECTORS	If 1, assumes that the input motion vectors texture is in low resolution.
FFXM_FSR2_OPTION_JITTERED_MOTION_VECTORS	If 1, assumes the jittered motion vectors use the same jitter offsets as the input color and depth.
FFXM_FSR2_OPTION_INVERTED_DEPTH	If 1, assumes the input depth containing reversed depth values (far == 0.0f).
FFXM_FSR2_OPTION_APPLY_SHARPENING	If 1, informs the shaders that RCAS (sharpening) pass is used.
FFXM_FSR2_OPTION_SHADER_OPT_BALANCED	If 1, enables a batch of optimizations when the Balanced quality preset is selected.
FFXM_FSR2_OPTION_SHADER_OPT_PERFORMANCE	If 1, enables a batch of optimizations when the Performance quality preset is selected. When enabled, `FFXM_FSR2_OPTION_SHADER_OPT_BALANCED` is also enabled.

When you are using an HLSL-based workflow, we recommend you use the `FFXM_HLSL_6_2` global define to improve performance. If defined with a value of `1`, this define enables the use of explicit 16-bit types instead of relying on `half` precision.

The `VK_KHR_shader_float16_int8` extension is required on Vulkan.

3.4 Input resources

Arm ASR is a temporal algorithm, and therefore requires access to data from both the current and previous frame.

The following is a list of all external inputs required by Arm ASR. The resolution entry indicates if the data should be at rendered resolution or presentation resolution:

- Rendered resolution indicates that the resource should match the resolution that the application is performing rendering.
- Presentation resolution indicates that the resolution of the target should match the resolution presented to the user.

All resources are from the current rendered frame. For Vulkan applications, all input resources should be transitioned to `VK_ACCESS_SHADER_READ_BIT` respectively before calling `ffxmFsr2ContextDispatch`.

Color buffer

The render resolution color buffer for the current frame provided by the application. If the contents of the color buffer are in High Dynamic Range (HDR), then set the `FFXM_FSR2_ENABLE_HIGH_DYNAMIC_RANGE` flag in the flags field of the `FfxmFsr2ContextDescription` structure.

Resolution: Render

Format: APPLICATION SPECIFIED

Type: Texture

Depth buffer

The render resolution depth buffer for the current frame provided by the application. The data should be provided as a single floating point value, the application controls the precision of the single floating point value. The configuration of the depth should be communicated to Arm ASR via the flags field of the `FfxmFsr2ContextDescription` structure when creating the `FfxmFsr2Context`. You should set the `FFXM_FSR2_ENABLE_DEPTH_INVERTED` flag if your depth buffer is inverted (that is [1..0] range). You should also set the `FFXM_FSR2_ENABLE_DEPTH_INFINITE` flag if your depth buffer has an infinite far plane. If the application provides the depth buffer in D32S8 format, then it ignores the stencil component of the buffer, and creates an R32_FLOAT resource to address the depth buffer.

Resolution: Render

Format: APPLICATION SPECIFIED (1x FLOAT)

Type: Texture

Motion vectors

The 2D motion vectors for the current frame are provided by the application in <-width, -height> ... <width, height> range. If your application renders motion vectors with a different range, you can use the `motionVectorScale` field of the `FfxmFsr2DispatchDescription` structure to adjust them to match the expected range for Arm ASR. Internally, Arm ASR uses 16-bit quantities to represent motion vectors in many cases. This means that while motion vectors with greater precision can be provided, Arm ASR does not benefit from the increased precision. The resolution of the motion vector buffer should be equal to the render resolution, unless the `FFXM_FSR2_ENABLE_DISPLAY_RESOLUTION_MOTION_VECTORS` flag is set in the flags field of the `FfxmFsr2ContextDescription` structure when creating `FfxmFsr2Context`. In which case, the resolution should be equal to the presentation resolution.

Resolution: Render or presentation

Format: APPLICATION SPECIFIED (2x FLOAT)

Type: Texture

Reactive mask

Some areas of a rendered image do not leave a footprint in the depth buffer or include motion vectors. Therefore Arm ASR provides support for a reactive mask texture which you can use to indicate to the technique where such areas are. Good examples of these are particles, or alpha-blended objects which do not write depth or motion vectors. If this resource is not set, the shading change detection logic of Arm ASR handles these cases as best it can, but for optimal results, this resource should be set. For more information, see Reactive mask.

Resolution: Render

Format: `R8_UNORM`

Type: Texture

Exposure

A 1 x 1 texture containing the exposure value computed for the current frame. This resource is optional. You can omit this resource if the `FFXM_FSR2_ENABLE_AUTO_EXPOSURE` flag is set in the `flags` field of the `FfxmFsr2ContextDescription` structure when creating the `FfxmFsr2Context`.

Resolution: 1 x 1

Format: `R8_UNORM`

Type: Texture

All inputs that are provided at Render Resolution, except for motion vectors, should be rendered with jitter. By default, motion vectors are expected to be unjittered unless the `FFXM_FSR2_ENABLE_MOTION_VECTORS_JITTER_CANCELLATION` flag is present.

3.5 Providing motion vectors

A key part of a temporal algorithm, whether antialiasing or upscaling, is the provision of motion vectors. Motion vectors describe the movement of pixels between consecutive frames.

Space

Arm ASR accepts motion vectors in 2D which encode the motion from a pixel in the current frame to the position of that same pixel in the previous frame. Arm ASR expects that motion vectors are provided by the application in <-width, -height> ... <width, height> range which matches Screen-Space. For example, a motion vector for a pixel in the upper-left corner of the screen with a value of <width, height> would represent a motion that traversed the full width and height of the input surfaces, originating from the bottom-right corner.

If your application computes motion vectors in another space, for example normalized device coordinate space, you can use the `motionVectorScale` field of the `FfxmFsr2DispatchDescription` structure to instruct the technique to adjust the motion vectors to match the expected range.

This code example illustrates how motion vectors can be scaled to screen space. The example HLSL and C++ code illustrates how NDC-space motion vectors can be scaled using the Arm ASR host API.

```
// GPU: Example of application NDC motion vector computation
float2 motionVector = (previousPosition.xy / previousPosition.w) -
 (currentPosition.xy / currentPosition.w);

// CPU: Matching Arm ASR motionVectorScale configuration
dispatchParameters.motionVectorScale.x = (float)renderWidth;
dispatchParameters.motionVectorScale.y = (float)renderHeight;
```

Precision and resolution

Internally, in many cases Arm ASR uses 16-bit quantities to represent motion vectors. This means that while motion vectors with greater precision can be provided, the quality of the motion vectors will not currently benefit from the increased precision.

The resolution of the motion vector buffer should be equal to the render resolution, unless the `FFXM_FSR2_ENABLE_DISPLAY_RESOLUTION_MOTION_VECTORS` flag is set in the `flags` field of the `FfxmFsr2ContextDescription` structure when creating `FfxmFsr2Context`. In which case the resolution of the motion vector buffer should be equal to the presentation resolution.

Coverage

Arm ASR performs better quality upscaling when more objects provide their motion vectors. Therefore we advise that all opaque, alpha-tested, and alpha-blended objects write their motion vectors for all covered pixels. If vertex shader effects are applied, for example scrolling UVs, for best results you should also factor these calculations into the calculation of motion.

For alpha-blended objects, we also strongly advise that the alpha value of each covered pixel is stored to the corresponding pixel in the `reactive mask`. This allows the technique to better handle alpha-blended objects during upscaling. The reactive mask is especially important for alpha-blended objects where writing motion vectors can be prohibitive, for example particles.

3.6 Reactive mask

In the context of Arm ASR, the term “reactivity” refers to how much influence the samples rendered for the current frame have over the production of the final upscaled image.

Typically, samples rendered for the current frame contribute a relatively small amount to the result computed by the algorithm, however there are exceptions. As there is no good way to determine from either color, depth, or motion vectors which pixels have been rendered using alpha blending, Arm ASR performs best when applications explicitly mark such areas.

Therefore, we strongly advise that applications provide a reactive mask as an input. The reactive mask guides Arm ASR on where it should reduce its reliance on historical information when compositing the current pixel, and instead allow the current frame’s samples to contribute more to the final result. The reactive mask allows the application to provide a value from 0.0 to 1.0, where:

- 0.0 indicates that the pixel is not reactive and should use the default composition strategy.
- 1.0 indicates the pixel is fully reactive. This is a floating point range and can be tailored to different situations.

While there are other applications for the reactive mask, the primary application for the reactive mask is producing better results when upscaling images which include alpha-blended objects. A good proxy for reactiveness is actually the alpha value used when compositing an alpha-blended object into the scene. Therefore, applications should write `alpha` to the reactive mask.

Note that it is unlikely that a reactive value close to 1.0 will produce good results. Therefore, we recommend clamping the maximum reactive value to approximately 0.9.

If a reactive mask is not provided, by setting the `reactive` field of `FfxmFsr2DispatchDescription` to `NULL`, then an internally generated 1 x 1 texture with a cleared reactive value is used.

3.7 Automatically generating reactivity

To help applications generate the reactive mask, we provide an optional utility pass. The API uses a fragment shader to compute these values for each pixel using a luminance-based heuristic.

To generate the reactive mask, applications can call the `ffxmFsr2ContextGenerateReactiveMask` function. The application should also pass two versions of the color buffer, one containing opaque only geometry and the other containing both opaque and alpha-blended objects.

3.8 Exposure

Arm ASR provides two values which control the exposure used when performing upscaling.

The values are:

Pre-exposure

A value by which we divide the input signal to get back to the original signal produced by the game before any packing into lower precision render targets.

Exposure

A value which is multiplied against the result of the pre-exposed color value.

The exposure value should match the value that the application uses during any subsequent tonemapping passes performed by the application. This means Arm ASR operates consistently with what is likely to be visible in the final tonemapped image.

During various stages of the algorithm, the technique computes its own exposure value for internal use. Note that all outputs have this internal tonemapping reversed before the final output is written. This means that Arm ASR returns results in the same domain as the original input signal.

Poorly selected exposure values can have a drastic impact on the final quality of the upscaling performed by Arm ASR. Therefore, we recommended that `FFXM_FSR2_ENABLE_AUTO_EXPOSURE` is used by the application whenever possible. When `FFXM_FSR2_ENABLE_AUTO_EXPOSURE` is set in the `flags` field of the `FfxmFsr2ContextDescription` structure, the exposure calculation in `ComputeAutoExposureFromLavg` is used to compute the exposure value. This calculation matches the exposure response of ISO 100 film.

3.9 Modular backend

The design of the Arm ASR API means that the core implementation of the algorithm is unaware on which rendering API Arm ASR sits. Instead, Arm ASR calls functions provided to the rendering API through an interface, allowing different backends to be used with the technique.

Applications which have their own rendering abstractions can implement their own backend, taking control of all aspects of the underlying function of Arm ASR, including:

- Memory management
- Resource creation
- Shader compilation
- Shader resource bindings
- Submission of the workloads to the graphics device

Out of the box, the Arm ASR API compiles into multiple libraries following the separation already outlined between the core API and the backends. Therefore, if you want to use the backends provided, you should link both the core API library (Arm_ASR_api) and the backend (Arm_ASR_backend) to match your requirements.


Note

Arm ASR only provides a built-in Vulkan backend as it targets Vulkan mobile applications.

3.10 Camera jitter

Arm ASR relies on the application to apply sub-pixel jittering while rendering.

Sub-pixel jittering is typically included in the projection matrix of the camera. To make the application of camera jitter simple, the API provides a set of utility functions which computes the sub-pixel jitter offset for a particular frame within a sequence of separate jitter offsets:

```
int32_t ffxmFsr2GetJitterPhaseCount(int32_t renderWidth, int32_t displayWidth);
FfxErrorCode ffxmFsr2GetJitterOffset(float* outX, float* outY, int32_t jitterPhase,
 int32_t sequenceLength);
```

Internally, these functions implement a Halton[2,3] sequence. The goal of the Halton sequence is to provide spatially separated points which cover the available space.

It is important to understand that the values returned from the `ffxmFsr2GetJitterOffset` are in unit pixel space. To composite the jitter offset correctly into a projection matrix we must convert the jitter offset into projection offsets. This code shows how to correctly composite the sub-pixel jitter offset value into a projection matrix:

```
const int32_t jitterPhaseCount = ffxmFsr2GetJitterPhaseCount(renderWidth,
 displayWidth);

float jitterX = 0;
float jitterY = 0;
ffxmFsr2GetJitterOffset(&jitterX, &jitterY, index, jitterPhaseCount);

// Calculate the jittered projection matrix.
const float jitterX = 2.0f * jitterX / (float)renderWidth;
const float jitterY = -2.0f * jitterY / (float)renderHeight;
const Matrix4 jitterTranslationMatrix = translateMatrix(Matrix3::identity,
 Vector3(jitterX, jitterY, 0));
const Matrix4 jitteredProjectionMatrix = jitterTranslationMatrix * projectionMatrix;
```

Jitter should be applied to all rendering. This includes opaque, alpha transparent, and raytraced objects. For rasterized objects, the sub-pixel jittering values calculated by the `ffxmFsr2GetJitterOffset` function can be applied to the camera projection matrix which is ultimately used to perform transformations during vertex shading. For raytraced rendering, the sub-pixel jitter should be applied to the origin of the ray which is often the position of the camera.

Whether you use the recommended `ffxmFsr2GetJitterOffset` function or your own sequence generator, you must set the `jitterOffset` field of the `FfxmFsr2DispatchDescription` structure to inform the algorithm of the jitter offset that has been applied to render each frame. If you are not using the recommended `ffxmFsr2GetJitterOffset` function, take care that your jitter sequence does not generates a null vector which is a value of 0 in both the X and Y dimensions.

3.11 Camera jump cuts

Most applications with real-time rendering have a large degree of temporal consistency between any two consecutive frames. However, there are cases where a change to a camera's transformation might cause an abrupt change in what is rendered.

In such cases, Arm ASR is unlikely to be able to reuse any data it has accumulated from previous frames. Arm ASR should clear this data to exclude the data from consideration in the compositing process. To indicate that a jump cut has occurred with the camera, you should set the `reset` field of the `FfxmFsr2DispatchDescription` structure to `true` for the first frame of the discontinuous camera transformation.

When you use the reset flag, the rendering performance can be slightly less than typical frame-to-frame operation because Arm ASR will clear some additional internal resources.

3.12 Mipmap biasing

Applying a negative mipmap bias typically generates an upscaled image with better texture detail.

We recommend applying the following formula to your mipmap bias:

```
mipBias = log2(renderResolution/displayResolution) - 1.0;
```

3.13 Frame time delta input

The Arm ASR API needs the application to provide the `frameTimeDelta` value through the `FfxmFsr2DispatchDescription` structure.

The `frameTimeDelta` value is in milliseconds, but you must use the float 32 value in the code. Therefore, if the application is running at 60 fps, you should use a float 32 value of 16.6f in your Arm ASR implementation. The value is used within the temporal component of the auto-exposure feature. This allows for tuning of the history accumulation for quality purposes.

3.14 HDR support

High Dynamic Range (HDR) images are supported.

To enable HDR, set the `FFXM_FSR2_ENABLE_HIGH_DYNAMIC_RANGE` bit in the `flags` field of the `FfxmFsr2ContextDescription` structure. When using this flag, provide the input color image in linear RGB color space.

3.15 Debug checker

The context description structure can be provided with a callback function for passing textual warnings from the runtime to the underlying application.

About this task

The `fpMessage` variable is a `FfxmFsr2Message` type and is a function pointer for passing types of string messages. To enable the debug feature, you must follow the Procedure.

Procedure

1. Assign the `fpMessage` variable to a function pointer, for example:

```
void Message(FfxmFsr2MsgType type, const wchar_t* message);
```

2. Assign the message function pointer to the `fpMessage` variable:

```
Params.fpMessage = Message;
```

3. The `FfxmFsr2ContextDescription` object has a flags parameter. You can set the debug flag like this:

```
Params.flags |= FFXM_FSR2_ENABLE_DEBUG_CHECKING;
```

We recommend that the debug feature is enabled in debug development builds only.

4. Shader and platform extensions

This section describes some of the additional platforms that Arm ASR can run on, such as GLES 3.2. This section also contains information about how the AMD Shader Compiler was modified to support RenderTargets and how to generate prebuilt shaders.

4.1 Extended ffx_shader_compiler

Most of the workloads in the upscalers have been converted to fragment shaders.

The workflow using the standalone Vulkan backend relies on reflection data generated with the AMD Shader Compiler. To support RenderTargets which are used by Arm ASR, we had to modify the AMD Shader Compiler.

You may want to evolve the algorithm potentially changing the RenderTargets in the process. To do so we provide a `ffx_shader_compiler` diff file with the changes that were applied locally for the latest version of the technique.

4.2 Generate prebuilt shaders

We provide a helper script to generate prebuilt shaders that are used for the standalone backend.

You can run `generate_prebuilt_shaders.py`. The output path is `src/backends/shared/blob_accessors/prebuilt_shaders`.

4.3 Running Arm ASR on GLES 3.2

You can run Arm ASR on GLES.

About this task

To run Arm ASR on GLES, you must use the Tight integration procedure and also apply two changes.

Procedure

1. When creating the context, you must specify the `FFXM_FSR2_OPENGL_ES_3_2` flag in the `flags` field of `FfxmFsr2ContextDescription`. This triggers some changes internally so that Arm ASR is compatible with GLES.
2. The `permutationOptions` provided when creating the pipelines will now include the `FSR2_SHADER_PERMUTATION_PLATFORM_GLES_3_2` permutation option. You must now use the shader variants for the technique with the following symbol defined:

```
#define FFXM_SHADER_PLATFORM_GLES_3_2 1
```

Proprietary Notice

export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Product and document information

Read the information in these sections to understand the release status of the product and documentation, and the conventions used in Arm documents.

Product status

All products and services provided by Arm require deliverables to be prepared and made available at different levels of completeness. The information in this document indicates the appropriate level of completeness for the associated deliverables.

Product completeness status

The information in this document is for a product under development (dev product).

Revision history

These sections can help you understand how the document has changed over time.

Document release information

The Document history table gives the issue number and the released date for each released issue of this document.

Document history

Issue	Date	Confidentiality	Change
0200-02	1 May 2025	Non-Confidential	First release of version 2.0.
0100-01	18 March 2025	Non-Confidential	First release of version 1.0.

Change history

The Change history tables describe the technical changes between released issues of this document in reverse order. Issue numbers match the revision history in Document release information on page 22.

Table 2: Differences between issue 0100-01 and 0200-02

Change	Location
First release of version 2.0	-
Adds a new chapter	Integration guidelines
Adds a new chapter	Shader and platform extensions

Table 3: Issue 0100-01

Change	Location
First release of version 1.0	-

Conventions

The following subsections describe conventions used in Arm documents.

Glossary

The Arm Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm Glossary for more information: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<and>`	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.

We recommend the following. If you do not follow these recommendations your system might not work.

Your system requires the following. If you do not follow these requirements your system will not work.

You are at risk of causing permanent damage to your system or your equipment, or harming yourself.

This information is important and needs your attention.

A useful tip that might make it easier, better or faster to perform a task.

A reminder of something important that relates to the information you are reading.

Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Arm documents are available on developer.arm.com/documentation.

Confidential documents are only available to licensees, when logged in. Each document link in the tables below provides direct access to the online version of the document.

Arm product resources	Document ID	Confidentiality
Arm® Accuracy Super Resolution™ for Unreal Engine	109993	Non-Confidential