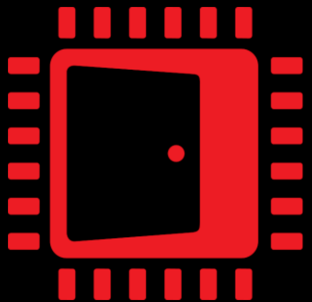
FidelityFX Super Resolution 3

FIDELITYFX API INTEGRATION OVERVIEW

VERSION 3.1.0

(RELEASE API VERSION)



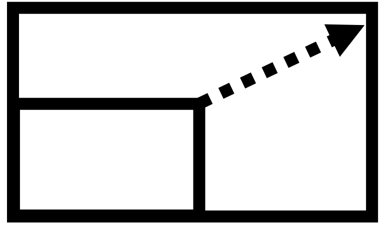
Last revision: 9th July 2024

FidelityFX Super Resolution 3

OVERVIEW

WHAT IS AMD FIDELITYFX™ SUPER RESOLUTION 3.1?

- FidelityFX Super Resolution 3.1 technology combines **resolution upscaling** with **frame generation**

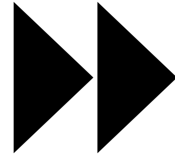


Resolution upscaling

Image quality similar or better than Native Rendering

Support for any area scale factor between 1X and 9X (including DRS)

New FSR "Native AA" mode

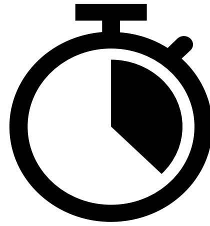


Frame Generation

Insert interpolated frames for smoother results

Works in both GPU and CPU-limited situations

Now with 3rd Party Upscaler support!



Frame Pacing

Keep latency down as much as possible by following guidelines

Frame pacing for regular frame rates

Variable Refresh Rate support

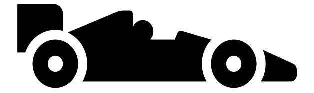
Ease of integration

New FidelityFX API created for DLL compatibility with new versions moving forward.

Open source

Source code provided on GPUOpen under an MIT license.

While source is available, **it is recommended to ship the Prebuilt & Signed FidelityFX DLL with your title**.



Highly optimized

Hand-optimized for great performance across mid to high-range GPUs

Total performance increase of up to **4x** compared to Native Rendering!

FidelityFX
Super Resolution 3

API INTEGRATION GUIDE

FSR 3.1 INTEGRATION GUIDE – OVERVIEW FOR DESKTOP

- FSR 3.1 includes functionality for
  - Upscaling
  - Frame generation
  - Frame pacing

- Supported graphics APIs are DX12 and Vulkan. UE5 plugin also available.

- Integration is via new FidelityFX API which interacts with a single FidelityFX DLL, which is prebuilt and signed by AMD.
  - **Ship any game builds using the prebuilt, signed DLL only.**
  - **Signed DLL with associated lib is available in *_/PrebuiltSignedDLL_***

- FidelityFX API introduces benefits for investigating integration issues, as well as enabling upgrade paths.

REQUISITE FOR SUCCESSFUL FSR 3 INTEGRATION

- Ensure that your game has a high-quality FSR 3 upscale-only implementation first!
  - Correct use of jittering pattern
  - Correct placement of post-process operations
  - Correct use of Reactive mask
  - Correct use of Transparency & composition mask
  - Correct setting of mip-bias for samplers
- For Frame Generation
  - Double buffer where required
  - Use the UI callbacks for UI composition

- **A sub-optimal integration of the upscale component will carry over any upscaling artefacts to interpolated frames!**

FSR 3.1 INTEGRATION STEPS (FIDELITYFX API)

FIDELITYFX API

- The FidelityFX API is simplified for ABI compatibility moving forward.

- There are 5 functions in the API that can be called with different arguments to provide functionality

  - ffxCreateContext
  - ffxDestroyContext
  - ffxDispatch
  - ffxQuery
  - ffxConfigure

- Further information can be provided via .pNext pointer chaining in the argument struct headers.

FIDELITYFX API USE

- The DLL can be linked via the .lib or loaded and function pointers obtained via getprocaddress
  - The templated C++ helpers require the lib at present.

- **Headers are available in *ffx-api/include***

- **Signed DLL with associated lib is available in */PrebuiltSignedDLL***

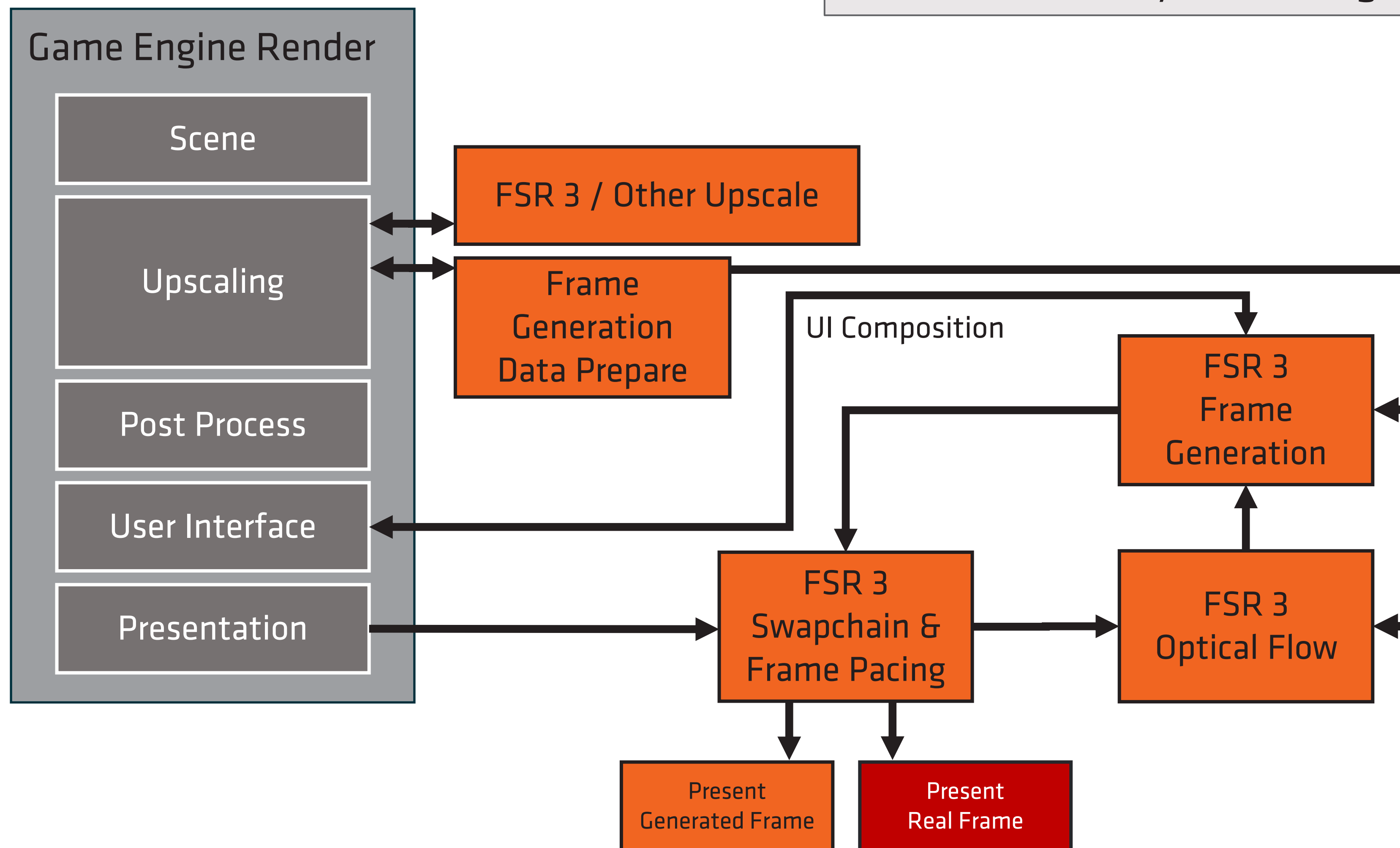
- **When providing game builds, only use this signed DLL.**

FSR3 CONTEXTS

- FSR3 consists of 3 separate systems:
  - Upscale
  - Frame Generation
  - Frame Pacing/Swapchain
- With the FidelityFX API, these systems need created separately and held in a context.
- Creating the context, configuring them, and dispatching them will result in the effect providing desired results
- Individual contexts will have rules guiding where and when in the graphics pipeline configure and dispatch operations are valid.

FSR 3.1 INTEGRATION POINTS

FSR 3.1 Interfaces with Upscaling, User Interface and Presentation systems of a game engine.



UPSCALING CONTEXT

API Function	Struct Type	Associated types
ffxCreateContext	ffxCreateContextDescUpscale	FfxApiCreateContextUpscaleFlags
ffxDispatch	ffxDispatchDescUpscale	FfxApiDispatchFsrUpscaleFlags
ffxDispatch	ffxDispatchDescUpscaleGenerateReactiveMask	FfxApiDispatchUpscaleAutoreactiveFlags
ffxConfigure	ffxConfigureDescUpscaleKeyValue	
ffxQuery	ffxQueryDescUpscaleGetJitterOffset	
ffxQuery	ffxQueryDescUpscaleGetJitterPhaseCount	
ffxQuery	ffxQueryDescUpscaleGetRenderResolutionFromQualityMode	FfxApiUpscaleQualityMode
ffxQuery	ffxQueryDescUpscaleGetUpscaleRatioFromQualityMode	FfxApiUpscaleQualityMode

- ffx_upscale.h

UPSCALING CONTEXT CREATE

API Function	Struct Type	Required Extension Chains
ffxCreateContext	ffxCreateContextDescUpscale	ffxCreateBackendDX12Desc or ffxCreateBackendVKDesc

```
#define FFX_API_CREATE_CONTEXT_DESC_TYPE_UPSCALE 0x00010000u
struct ffxCreateContextDescUpscale
{
    ffxCreateContextDescHeader header;
    uint32_t                   flags;          ///< Zero or a combination of values from FfxApiCreateContextFsrFlags.
    struct FfxApiDimensions2D  maxRenderSize;  ///< The maximum size that rendering will be performed at.
    struct FfxApiDimensions2D  maxUpscaleSize; ///< The size of the presentation resolution targeted by the upscaling process.
    ffxApiMessage              fpMessage;      ///< A pointer to a function that can receive messages from the runtime. May be null.
};
```

When running in **DX12**, pass an instance of this struct in header.pNext:

```
#define FFX_API_CREATE_CONTEXT_DESC_TYPE_BACKEND_DX12 0x0000002u
struct ffxCreateBackendDX12Desc
{
    ffxCreateContextDescHeader header;
    ID3D12Device              *device;  ///< Device on which the backend will run.
};
```

When running in **Vulkan**, pass an instance of this struct in header.pNext:

```
#define FFX_API_CREATE_CONTEXT_DESC_TYPE_BACKEND_VK 0x0000003u
struct ffxCreateBackendVKDesc
{
    ffxCreateContextDescHeader header;
    VkDevice                   vkDevice;
    VkPhysicalDevice           vkPhysicalDevice;
    PFN_vkGetDeviceProcAddr    vkDeviceProcAddr;
};
```

UPSCALING CONTEXT CREATE

API Function	Struct Type	Required Extension Chains
ffxCreateContext	ffxCreateContextDescUpscale	ffxCreateBackendDX12Desc or ffxCreateBackendVKDesc

Example from FSR API sample, which leverages C++ helpers to chain extensions.

```
ffx::CreateBackendDX12Desc backendDesc{};
backendDesc.header.type = FFX_API_CREATE_CONTEXT_DESC_TYPE_BACKEND_DX12;
backendDesc.device      = GetDevice()->GetImpl()->DX12Device();

ffx::CreateContextDescUpscale createFsr{};

createFsr.maxUpscaleSize = {resInfo.DisplayWidth, resInfo.DisplayHeight};
createFsr.maxRenderSize  = {resInfo.DisplayWidth, resInfo.DisplayHeight};
createFsr.flags          = FFX_UPSCALE_ENABLE_AUTO_EXPOSURE;
if (s_InvertedDepth)
{
    createFsr.flags |= FFX_UPSCALE_ENABLE_DEPTH_INVERTED | FFX_UPSCALE_ENABLE_DEPTH_INFINITE;
}
createFsr.flags |= FFX_UPSCALE_ENABLE_HIGH_DYNAMIC_RANGE;

// Do eror checking in debug
#if defined(_DEBUG)
createFsr.flags |= FFX_UPSCALE_ENABLE_DEBUG_CHECKING;
createFsr.fpMessage = &FSRRenderModule::FfxMsgCallback;
#endif  // #if defined(_DEBUG)

ffx::ReturnCode retCode = ffx::CreateContext(m_UpscalingContext, nullptr, createFsr, backendDesc);
```

UPSCALING CONTEXT DISPATCH

API Function	Struct Type	Required Extension Chains
ffxDispatch	ffxDispatchDescUpscale	

```
#define FFX_API_DISPATCH_DESC_TYPE_UPSCALE 0x00010001u
struct ffxDispatchDescUpscale
{
    ffxDispatchDescHeader      header;
    void*                      commandList;                ///< Command list to record upscaling rendering commands into.
    struct FfxApiResource      color;                      ///< Color buffer for the current frame (at render resolution).
    struct FfxApiResource      depth;                      ///< 32bit depth values for the current frame (at render resolution).
    struct FfxApiResource      motionVectors;              ///< 2-dimensional motion vectors (at render resolution if FFX_FSR_ENABLE_DISPLAY_RESOLUTION_MOTION_VECTORS is not set).
    struct FfxApiResource      exposure;                   ///< Optional resource containing a 1x1 exposure value.
    struct FfxApiResource      reactive;                   ///< Optional resource containing alpha value of reactive objects in the scene.
    struct FfxApiResource      transparencyAndComposition; ///< Optional resource containing alpha value of special objects in the scene.
    struct FfxApiResource      output;                     ///< Output color buffer for the current frame (at presentation resolution).
    struct FfxApiFloatCoords2D jitterOffset;               ///< The subpixel jitter offset applied to the camera.
    struct FfxApiFloatCoords2D motionVectorScale;          ///< The scale factor to apply to motion vectors.
    struct FfxApiDimensions2D  renderSize;                 ///< The resolution that was used for rendering the input resources.
    struct FfxApiDimensions2D  upscaleSize;                ///< The resolution that the upscaler will upscale to (optional, assumed maxUpscaleSize otherwise).
    bool                       enableSharpening;           ///< Enable an additional sharpening pass.
    float                      sharpness;                  ///< The sharpness value between 0 and 1, where 0 is no additional sharpness and 1 is maximum additional sharpness.
    float                      frameTimeDelta;             ///< The time elapsed since the last frame (expressed in milliseconds).
    float                      preExposure;                ///< The pre exposure value (must be > 0.0f)
    bool                       reset;                      ///< A boolean value which when set to true, indicates the camera has moved discontinuously.
    float                      cameraNear;                 ///< The distance to the near plane of the camera.
    float                      cameraFar;                  ///< The distance to the far plane of the camera.
    float                      cameraFovAngleVertical;     ///< The camera angle field of view in the vertical direction (expressed in radians).
    float                      viewSpaceToMetersFactor;    ///< The scale factor to convert view space units to meters
    uint32_t                   flags;                      ///< Zero or a combination of values from FfxApiDispatchFsrUpscaleFlags.
};
```

UPSCALING CONTEXT DISPATCH

API Function	Struct Type	Associated types
ffxDispatch	ffxDispatchDescUpscale	

Example from FSR API sample, which leverages C++ helpers.

```
ffx::DispatchDescUpscale dispatchUpscale{};
dispatchUpscale.commandList = pCmdList->GetImpl()->DX12CmdList();
dispatchUpscale.color         = SDKWrapper::ffxGetResourceApi(m_pTempTexture->GetResource(), FFX_API_RESOURCE_STATE_PIXEL_COMPUTE_READ);
dispatchUpscale.depth         = SDKWrapper::ffxGetResourceApi(m_pDepthTarget->GetResource(), FFX_API_RESOURCE_STATE_PIXEL_COMPUTE_READ);
dispatchUpscale.motionVectors = SDKWrapper::ffxGetResourceApi(m_pMotionVectors->GetResource(), FFX_API_RESOURCE_STATE_PIXEL_COMPUTE_READ);
dispatchUpscale.exposure      = SDKWrapper::ffxGetResourceApi(nullptr, FFX_API_RESOURCE_STATE_PIXEL_COMPUTE_READ);
dispatchUpscale.output        = SDKWrapper::ffxGetResourceApi(m_pColorTarget->GetResource(), FFX_API_RESOURCE_STATE_PIXEL_COMPUTE_READ);
dispatchUpscale.reactive      = SDKWrapper::ffxGetResourceApi(m_pReactiveMask->GetResource(), FFX_API_RESOURCE_STATE_PIXEL_COMPUTE_READ);
dispatchUpscale.transparencyAndComposition = SDKWrapper::ffxGetResourceApi(m_pCompositionMask->GetResource(), FFX_API_RESOURCE_STATE_PIXEL_COMPUTE_READ);
dispatchUpscale.jitterOffset.x      = -m_JitterX;
dispatchUpscale.jitterOffset.y      = -m_JitterY;
dispatchUpscale.motionVectorScale.x = resInfo.fRenderWidth();
dispatchUpscale.motionVectorScale.y = resInfo.fRenderHeight();
dispatchUpscale.reset               = m_ResetUpscale;
dispatchUpscale.enableSharpening    = m_RCASSharpen;
dispatchUpscale.sharpness           = m_Sharpness;

dispatchUpscale.frameTimeDelta = static_cast<float>(deltaTime * 1000.f);// FSR expects milliseconds

dispatchUpscale.preExposure        = GetScene()->GetSceneExposure();
dispatchUpscale.renderSize.width   = resInfo.RenderWidth;
dispatchUpscale.renderSize.height  = resInfo.RenderHeight;
dispatchUpscale.upscaleSize.width  = resInfo.UpscaleWidth;
dispatchUpscale.upscaleSize.height = resInfo.UpscaleHeight;

dispatchUpscale.cameraFovAngleVertical = pCamera->GetFovY();
dispatchUpscale.cameraFar  = pCamera->GetFarPlane();
dispatchUpscale.cameraNear = pCamera->GetNearPlane();
dispatchUpscale.flags = m_DrawUpscalerDebugView ? FFX_UPSCALE_FLAG_DRAW_DEBUG_VIEW : 0;

ffx::ReturnCode retCode = ffx::Dispatch(m_UpscalingContext, dispatchUpscale);
```

UPSCALING CONTEXT DISPATCH RECOMMENDATIONS

API Function	Struct Type	Associated types
`ffxDispatch`	`ffxDispatchDescUpscale`	

- Wherever possible, upscale before post-process effects which impact upon the mapping between motion vectors and input colors.
- Motion vectors should be FP16.
- Motion vectors should be given for all pixels in the input.
- Favor inverse and infinite depth.
- The input and output of the FSR dispatch must be separate resources.
- Use the T&C mask for animated texture regions and items like glass.
- Use the reactive mask to indicate areas of transparency without motion vectors or depth data, such as fog particles.
- Always set reset = true for one frame on scene changes, such as entering and exiting menus or cinematics.

UPSCALING CONTEXT QUERIES

API Function	Struct Type	Associated types
ffxQuery	ffxQueryDescUpscaleGetJitterOffset	
ffxQuery	ffxQueryDescUpscaleGetJitterPhaseCount	

```
#define FFX_API_QUERY_DESC_TYPE_UPSCALE_GETJITTERPHASECOUNT 0x00010004u
struct ffxQueryDescUpscaleGetJitterPhaseCount
{
    ffxQueryDescHeader header;
    uint32_t           renderWidth;    ///< The render resolution width.
    uint32_t           displayWidth;   ///< The output resolution width.
    uint32_t*          pOutPhaseCount; ///< A pointer to a <c>int32_t</c> which will hold the jitter phase count for the scaling factor between <c><i>renderWidth</i></c> and
<c><i>displayWidth</i></c>.
};

#define FFX_API_QUERY_DESC_TYPE_UPSCALE_GETJITTEROFFSET 0x00010005u
struct ffxQueryDescUpscaleGetJitterOffset
{
    ffxQueryDescHeader header;
    int32_t            index;      ///< The index within the jitter sequence.
    int32_t            phaseCount; ///< The length of jitter phase. See <c><i>ffxQueryDescFsrGetJitterPhaseCount</i></c>.
    float*             pOutX;      ///< A pointer to a <c>float</c> which will contain the subpixel jitter offset for the x dimension.
    float*             pOutY;      ///< A pointer to a <c>float</c> which will contain the subpixel jitter offset for the y dimension.
};
```

UPSCALING CONTEXT QUERIES

API Function	Struct Type	Associated types
ffxQuery	ffxQueryDescUpscaleGetRenderResolutionFromQualityMode	FfxApiUpscaleQualityMode
ffxQuery	ffxQueryDescUpscaleGetUpscaleRatioFromQualityMode	FfxApiUpscaleQualityMode

```
#define FFX_API_QUERY_DESC_TYPE_UPSCALE_GETUPSCALERATIOFROMQUALITYMODE 0x00010002u
struct ffxQueryDescUpscaleGetUpscaleRatioFromQualityMode
{
    ffxQueryDescHeader header;
    uint32_t           qualityMode;      ///< The desired quality mode for FSR upscaling.
    float*             pOutUpscaleRatio; ///< A pointer to a <c>float</c> which will hold the upscaling the per-dimension upscaling ratio.
};

#define FFX_API_QUERY_DESC_TYPE_UPSCALE_GETRENDERRESOLUTIONFROMQUALITYMODE 0x00010003u
struct ffxQueryDescUpscaleGetRenderResolutionFromQualityMode
{
    ffxQueryDescHeader header;
    uint32_t           displayWidth;     ///< The target display resolution width.
    uint32_t           displayHeight;    ///< The target display resolution height.
    uint32_t           qualityMode;      ///< The desired quality mode for FSR upscaling.
    uint32_t*          pOutRenderWidth;  ///< A pointer to a <c>uint32_t</c> which will hold the calculated render resolution width.
    uint32_t*          pOutRenderHeight; ///< A pointer to a <c>uint32_t</c> which will hold the calculated render resolution height.
};
```

UPSCALING CONTEXT QUERY EXAMPLE

API Function	Struct Type	Associated types
ffxQuery	ffxQueryDescUpscaleGetJitterOffset	
ffxQuery	ffxQueryDescUpscaleGetJitterPhaseCount	

Example from FSR API sample, which leverages C++ helpers.

```
++m_JitterIndex;

const ResolutionInfo& resInfo = GetFramework()->GetResolutionInfo();

ffx::ReturnCode                         retCode;
int32_t                                 jitterPhaseCount;
ffx::QueryDescUpscaleGetJitterPhaseCount getJitterPhaseDesc{};
getJitterPhaseDesc.displayWidth   = resInfo.DisplayWidth;
getJitterPhaseDesc.renderWidth    = resInfo.RenderWidth;
getJitterPhaseDesc.pOutPhaseCount = &jitterPhaseCount;

retCode = ffx::Query(m_UpscalingContext, getJitterPhaseDesc);

ffx::QueryDescUpscaleGetJitterOffset getJitterOffsetDesc{};
getJitterOffsetDesc.index      = m_JitterIndex;
getJitterOffsetDesc.phaseCount = jitterPhaseCount;
getJitterOffsetDesc.pOutX      = &m_JitterX;
getJitterOffsetDesc.pOutY      = &m_JitterY;

retCode = ffx::Query(m_UpscalingContext, getJitterOffsetDesc);
```

Some can also be called without a context for setup purposes.
If it’s not supported, return code will indicate failure:

```
ffx::QueryDescUpscaleGetRenderResolutionFromQualityMode query;
uint32_t testWidth, testHeight;
query.displayWidth = 3840;
query.displayHeight = 2160;
query.qualityMode = FFX_UPSCALE_QUALITY_MODE_NATIVEAA;
query.pOutRenderWidth = &testWidth;
query.pOutRenderHeight = &testHeight;
auto retCode = ffx::Query(query);
CauldronAssert(ASSERT_ERROR, retCode == ffx::ReturnCode::Ok, L"Query error!")
```

If a context is available, always provide it – as there is a performance penalty to call with context == nullptr.

FRAME GENERATION CONTEXT

API Function	Struct Type	Associated types
ffxCreateContext	ffxCreateContextDescFrameGeneration	FfxApiCreateContextFramegenerationFlags
	ffxCallbackDescFrameGenerationPresent	Callback Function Pointer: FfxApiPresentCallbackFunc
ffxDispatch	ffxDispatchDescFrameGeneration	Callback Function Pointer: FfxApiFrameGenerationDispatchFunc
ffxConfigure	ffxConfigureDescFrameGeneration	FfxApiDispatchFramegenerationFlags
ffxDispatch	ffxDispatchDescFrameGenerationPrepare	
ffxConfigure	ffxConfigureDescFrameGenerationKeyValue	

- ffx_framegeneration.h

FRAME GENERATION CONTEXT CREATE

API Function	Struct Type	Required Extension Chains
ffxCreateContext	ffxCreateContextDescFrameGeneration	ffxCreateBackendDX12Desc or ffxCreateBackendVKDesc

```
#define FFX_API_CREATE_CONTEXT_DESC_TYPE_FRAMEGENERATION 0x00020001u
struct ffxCreateContextDescFrameGeneration
{
    ffxCreateContextDescHeader header;
    uint32_t flags;                          ///< A combination of zero or more values from FfxApiCreateContextFrameGenerationFlags.
    struct FfxApiDimensions2D displaySize;   ///< The resolution at which both rendered and generated frames will be displayed.
    struct FfxApiDimensions2D maxRenderSize; ///< The maximum rendering resolution.
    uint32_t backBufferFormat;               ///< The surface format for the backbuffer. One of the values from FfxApiSurfaceFormat.
};
```

When running in **DX12**, pass an instance of this struct in header.pNext:

```
#define FFX_API_CREATE_CONTEXT_DESC_TYPE_BACKEND_DX12 0x0000002u
struct ffxCreateBackendDX12Desc
{
    ffxCreateContextDescHeader header;
    ID3D12Device              *device;  ///< Device on which the backend will run.
};
```

When running in **Vulkan**, pass an instance of this struct in header.pNext:

```
#define FFX_API_CREATE_CONTEXT_DESC_TYPE_BACKEND_VK 0x0000003u
struct ffxCreateBackendVKDesc
{
    ffxCreateContextDescHeader header;
    VkDevice                   vkDevice;
    VkPhysicalDevice           vkPhysicalDevice;
    PFN_vkGetDeviceProcAddr    vkDeviceProcAddr;
};
```

FRAME GENERATION CONTEXT CREATE

API Function	Struct Type	Required Extension Chains
ffxCreateContext	ffxCreateContextDescFrameGeneration	ffxCreateBackendDX12Desc or ffxCreateBackendVKDesc

Example from FSR API sample, which leverages C++ helpers to chain extensions.

```
ffx::CreateBackendDX12Desc backendDesc{};
backendDesc.header.type = FFX_API_CREATE_CONTEXT_DESC_TYPE_BACKEND_DX12;
backendDesc.device      = GetDevice()->GetImpl()->DX12Device();

ffx::CreateContextDescFrameGeneration createFg{};
createFg.displaySize   = {resInfo.DisplayWidth, resInfo.DisplayHeight};
createFg.maxRenderSize = {resInfo.RenderWidth, resInfo.RenderHeight};

if (s_InvertedDepth)
{
    createFg.flags |= FFX_FRAMEGENERATION_ENABLE_DEPTH_INVERTED | FFX_FRAMEGENERATION_ENABLE_DEPTH_INFINITE;
}
createFg.flags |= FFX_FRAMEGENERATION_ENABLE_HIGH_DYNAMIC_RANGE;

if (m_EnableAsyncCompute)
{
    createFg.flags |= FFX_FRAMEGENERATION_ENABLE_ASYNC_WORKLOAD_SUPPORT;
}

createFg.backBufferFormat = SDKWrapper::GetFfxSurfaceFormat(GetFramework()->GetSwapChain()->GetSwapChainFormat());

ffx::ReturnCode retCode = ffx::CreateContext(m_FrameGenContext, nullptr, createFg, backendDesc);
```

FRAME GENERATION CONFIGURE

API Function	Struct Type	Associated types
ffxConfigure	ffxConfigureDescFrameGeneration	FfxApiDispatchFramegenerationFlags

```
typedef ffxReturnCode_t(*FfxApiPresentCallbackFunc)(ffxCallbackDescFrameGenerationPresent* params, void* pUserCtx);
typedef ffxReturnCode_t(*FfxApiFrameGenerationDispatchFunc)(ffxDispatchDescFrameGeneration* params, void* pUserCtx);

#define FFX_API_CONFIGURE_DESC_TYPE_FRAMEGENERATION 0x00020002u
struct ffxConfigureDescFrameGeneration
{
    ffxConfigureDescHeader header;
    void* swapChain;                                          ///< The swapchain to use with frame generation.
    FfxApiPresentCallbackFunc presentCallback;                ///< A UI composition callback to call when finalizing the frame image.
    void* presentCallbackUserContext;                         ///< A pointer to be passed to the UI composition callback.
    FfxApiFrameGenerationDispatchFunc frameGenerationCallback; ///< The frame generation callback to use to generate a frame.
    void* frameGenerationCallbackUserContext;                 ///< A pointer to be passed to the frame generation callback.
    bool frameGenerationEnabled;                              ///< Sets the state of frame generation. Set to false to disable frame generation.
    bool allowAsyncWorkloads;                                 ///< Sets the state of async workloads. Set to true to enable generation work on async compute.
    struct FfxApiResource HUDLessColor;                       ///< The hudless back buffer image to use for UI extraction from backbuffer resource. May be empty.
    uint32_t flags;                                           ///< Zero or combination of flags from FfxApiDispatchFgFlags.
    bool onlyPresentGenerated;                                ///< Set to true to only present generated frames.
    struct FfxApiRect2D generationRect;                       ///< The area of the backbuffer that should be used for generation in case only a part of the screen is used e.g. due to r
bars
    uint64_t frameID;                                         ///< Identifier used to select internal resources when async support is enabled. Must increment by exactly one (1) for eac
Any non-exactly-one difference will reset the frame generation logic.
};
```

FRAME GENERATION CONFIGURE

API Function	Struct Type	Associated types
ffxConfigure	ffxConfigureDescFrameGeneration	FfxApiDispatchFramegenerationFlags

Example from FSR API sample, which leverages C++ helpers to chain extensions.

```
ffx::ConfigureDescFrameGeneration m_FrameGenerationConfig{};
m_FrameGenerationConfig.frameGenerationEnabled = m_FrameInterpolation;
m_FrameGenerationConfig.flags                   = 0;
m_FrameGenerationConfig.flags                  |= m_DrawFrameGenerationDebugTearLines ? FFX_FRAMEGENERATION_FLAG_DRAW_DEBUG_TEAR_LINES : 0;
m_FrameGenerationConfig.flags                  |= m_DrawFrameGenerationDebugResetIndicators ? FFX_FRAMEGENERATION_FLAG_DRAW_DEBUG_RESET_INDICATORS : 0;
m_FrameGenerationConfig.flags                  |= m_DrawFrameGenerationDebugView ? FFX_FRAMEGENERATION_FLAG_DRAW_DEBUG_VIEW : 0;
m_FrameGenerationConfig.HUDLessColor            = (s_uiRenderMode == SAMPLE_UI_HUDLESS_MODE) ? hudLessResource : FfxApiResource({});
m_FrameGenerationConfig.allowAsyncWorkloads     = m_AllowAsyncCompute && m_EnableAsyncCompute;
m_FrameGenerationConfig.generationRect.left     = (resInfo.DisplayWidth - resInfo.UpscaleWidth) / 2;
m_FrameGenerationConfig.generationRect.top      = (resInfo.DisplayHeight - resInfo.UpscaleHeight) / 2;
m_FrameGenerationConfig.generationRect.width    = resInfo.UpscaleWidth;
m_FrameGenerationConfig.generationRect.height   = resInfo.UpscaleHeight;
if (m_UseCallback)
{
    m_FrameGenerationConfig.frameGenerationCallback = [](ffxDispatchDescFrameGeneration* params, void* pUserCtx) -> ffxReturnCode_t {
        return ffxDispatch(reinterpret_cast<ffxContext*>(pUserCtx), &params->header);
    };
    m_FrameGenerationConfig.frameGenerationCallbackUserContext = &m_FrameGenContext;
}
if (s_uiRenderMode == SAMPLE_UI_CALLBACK_MODE)
{
    m_FrameGenerationConfig.presentCallback = [](ffxCallbackDescFrameGenerationPresent* params, void* self) -> auto {
                                                       return reinterpret_cast<FSRRenderModule*>(self)->UiCompositionCallback(params); };
   m_FrameGenerationConfig.presentCallbackUserContext = this;
}

m_FrameGenerationConfig.onlyPresentGenerated    = m_PresentInterpolatedOnly;
m_FrameGenerationConfig.frameID                 = m_FrameID;

void* ffxSwapChain = GetSwapChain()->GetImpl()->DX12SwapChain();
m_FrameGenerationConfig.swapChain = ffxSwapChain;
```

FRAME GENERATION CONFIGURE RECOMMENDATIONS

API Function	Struct Type	Associated types
`ffxConfigure`	`ffxConfigureDescFrameGeneration`	`FfxApiDispatchFramegenerationFlags`

- **To be called once a frame, before Frame Generation Prepare dispatch.**
- **Ensure frameID is used.**
- Includes enable bit for frame Generation
- Provide the swapchain
- Provide the Frame Generation Callback, if using it (manually dispatch frame generation if not)
  - `frameGenerationCallback`
- Provide any UI composition resources/functions
  - "Hudless Scene" resource **`HUDLessColor`**
  - UI Composition callback function **`presentCallback`**

FRAME GENERATION PREPARE

API Function	Struct Type	Associated types
ffxDispatch	ffxDispatchDescFrameGenerationPrepare	

```
#define FFX_API_DISPATCH_DESC_TYPE_FRAMEGENERATION_PREPARE 0x00020004u
struct ffxDispatchDescFrameGenerationPrepare
{
    ffxDispatchDescHeader      header;
    uint64_t                   frameID;             ///< Identifier used to select internal resources when async support is enabled. Must increment by exactly one (1) for each frame.
Any non-exactly-one difference will reset the frame generation logic.
    uint32_t                   flags;               ///< Zero or combination of values from FfxApiDispatchFgFlags.
    void*                      commandList;         ///< A command list to record frame generation commands into.
    struct FfxApiDimensions2D  renderSize;          ///< The dimensions used to render game content, dilatedDepth, dilatedMotionVectors are expected to be of ths size.
    struct FfxApiFloatCoords2D jitterOffset;        ///< The subpixel jitter offset applied to the camera.
    struct FfxApiFloatCoords2D motionVectorScale;   ///< The scale factor to apply to motion vectors.

    float                 frameTimeDelta;           ///< Time elapsed in milliseconds since the last frame.
    bool                  reset_unused;             ///< A (currently unused) boolean value which when set to true, indicates FrameGeneration will be called in reset mode
    float                 cameraNear;               ///< The distance to the near plane of the camera.
    float                 cameraFar;                ///< The distance to the far plane of the camera. This is used only used in case of non infinite depth.
    float                 cameraFovAngleVertical;   ///< The camera angle field of view in the vertical direction (expressed in radians).
    float                 viewSpaceToMetersFactor;  ///< The scale factor to convert view space units to meters
    struct FfxApiResource depth;                    ///< The depth buffer data
    struct FfxApiResource motionVectors;            ///< The motion vector data
};
```

FRAME GENERATION PREPARE

API Function	Struct Type	Associated types
ffxDispatch	ffxDispatchDescFrameGenerationPrepare	

Example from FSR API sample, which leverages C++ helpers to chain extensions.

```
ffx::DispatchDescFrameGenerationPrepare dispatchFgPrep{};

dispatchFgPrep.commandList = pCmdList->GetImpl()->DX12CmdList();
dispatchFgPrep.depth         = SDKWrapper::ffxGetResourceApi(m_pDepthTarget->GetResource(), FFX_API_RESOURCE_STATE_PIXEL_COMPUTE_READ);
dispatchFgPrep.motionVectors = SDKWrapper::ffxGetResourceApi(m_pMotionVectors->GetResource(), FFX_API_RESOURCE_STATE_PIXEL_COMPUTE_READ);
dispatchFgPrep.flags         = 0;
dispatchFgPrep.jitterOffset.x      = -m_JitterX;
dispatchFgPrep.jitterOffset.y      = -m_JitterY;
dispatchFgPrep.motionVectorScale.x = resInfo.fRenderWidth();
dispatchFgPrep.motionVectorScale.y = resInfo.fRenderHeight();


// FSR expects milliseconds
dispatchFgPrep.frameTimeDelta = static_cast<float>(deltaTime * 1000.f);

dispatchFgPrep.renderSize.width       = resInfo.RenderWidth;
dispatchFgPrep.renderSize.height      = resInfo.RenderHeight;
dispatchFgPrep.cameraFovAngleVertical = pCamera->GetFovY();
dispatchFgPrep.cameraFar   = pCamera->GetFarPlane();
dispatchFgPrep.cameraNear = pCamera->GetNearPlane();

dispatchFgPrep.viewSpaceToMetersFactor = 0.f;
dispatchFgPrep.frameID                 = m_FrameID;
dispatchFgPrep.flags = m_FrameGenerationConfig.flags;

retCode = ffx::Dispatch(m_FrameGenContext, dispatchFgPrep);
```

FRAME GENERATION PREPARE RECOMMENDATIONS

API Function	Struct Type	Associated types
`ffxDispatch`	`ffxDispatchDescFrameGenerationPrepare`	

Recommendations

- As Per Upscale pass.
- Wherever possible, prepare should be called before post-process effects which impact upon the mapping between motion vectors and input colors.
- Motion vectors should be FP16.
- Motion vectors should be given for all pixels in the input.
- Favor inverse and infinite depth.
- Always set reset = true for one frame on scene changes, such as entering and exiting menus or cinematics.

`ffxDispatchDescFrameGenerationPrepare` **is not thread safe** and should be prevented from being called at same time as `ffxDispatchDescFrameGeneration`

FRAME GENERATION DISPATCH

API Function	Struct Type	Associated types
ffxDispatch	ffxDispatchDescFrameGeneration	

```
 #define FFX_API_DISPATCH_DESC_TYPE_FRAMEGENERATION 0x00020003u
struct ffxDispatchDescFrameGeneration
{
    ffxDispatchDescHeader header;
    void*                 commandList;                 ///< The command list on which to register render commands.
    struct FfxApiResource presentColor;                ///< The current presentation color, this will be used as source data.
    struct FfxApiResource outputs[4];                  ///< Destination targets (1 for each frame in numGeneratedFrames).
    uint32_t              numGeneratedFrames;          ///< The number of frames to generate from the passed in color target.
    bool                  reset;                       ///< A boolean value which when set to true, indicates the camera has moved discontinuously.
    uint32_t              backbufferTransferFunction;  ///< The transfer function use to convert frame generation source color data to linear RGB. One of the values from
FfxApiBackbufferTransferFunction.
    float                 minMaxLuminance[2];          ///< Min and max luminance values, used when converting HDR colors to linear RGB.
    struct FfxApiRect2D   generationRect;              ///< The area of the backbuffer that should be used for generation in case only a part of the screen is used e.g. due to movie
bars.
    uint64_t              frameID;                     ///< Identifier used to select internal resources when async support is enabled. Must increment by exactly one (1) for each frame.
Any non-exactly-one difference will reset the frame generation logic.
};
```

FRAME GENERATION DISPATCH

API Function	Struct Type	Associated types
ffxDispatch	ffxDispatchDescFrameGeneration	Swapchain query functions.

Example from FSR API sample, which leverages C++ helpers to chain extensions. Needs to know command lists and outputs, queryable from swapchain.

If using the Frame Generation Callback, you do not need to dispatch frame generation manually like below. The Sample shows both methods.

```
ffx::DispatchDescFrameGeneration dispatchFg{};

dispatchFg.presentColor       = backbuffer;
dispatchFg.numGeneratedFrames = 1;

dispatchFg.generationRect.left   = (resInfo.DisplayWidth - resInfo.UpscaleWidth) / 2;
dispatchFg.generationRect.top    = (resInfo.DisplayHeight - resInfo.UpscaleHeight) / 2;
dispatchFg.generationRect.width  = resInfo.UpscaleWidth;
dispatchFg.generationRect.height = resInfo.UpscaleHeight;

ffx::QueryDescFrameGenerationSwapChainInterpolationCommandListDX12 queryCmdList{};
queryCmdList.pOutCommandList = &dispatchFg.commandList;
ffx::Query(m_SwapChainContext, queryCmdList);

ffx::QueryDescFrameGenerationSwapChainInterpolationTextureDX12 queryFiTexture{};
queryFiTexture.pOutTexture = &dispatchFg.outputs[0];
ffx::Query(m_SwapChainContext, queryFiTexture);


dispatchFg.frameID = m_FrameID;

retCode = ffx::Dispatch(m_FrameGenContext, dispatchFg);
```

FRAME GENERATION DISPATCH RECOMMENDATIONS

API Function	Struct Type	Associated types
`ffxDispatch`	`ffxDispatchDescFrameGeneration`	`Swapchain query functions.`

Recommendations

`ffxDispatchDescFrameGeneration` **is not thread safe** and should be prevented from being called at same time as `ffxDispatchDescFrameGenerationPrepare`

When using the frame generation callback, the swapchain will queue the frame generation dispatch automatically. There will be no need to manually call `ffxFsr3DispatchFrameGeneration`. **`Using the Frame Generation Callback is the recommended integration path.`**

FRAME GENERATION SWAPCHAIN DX12 CONTEXT

API Function	Struct Type	Associated types
ffxCreateContext	ffxCreateContextDescFrameGenerationSwapChainWrapDX12	
ffxCreateContext	ffxCreateContextDescFrameGenerationSwapChainNewDX12	
ffxCreateContext	ffxCreateContextDescFrameGenerationSwapChainForHwndDX12	
ffxConfigure	ffxConfigureDescFrameGenerationSwapChainRegisterUiResourceDX12	FfxApiUiCompositionFlags
ffxDispatch	ffxDispatchDescFrameGenerationSwapChainWaitForPresentsDX12	
ffxQuery	ffxQueryDescFrameGenerationSwapChainInterpolationTextureDX12	
ffxQuery	ffxQueryDescFrameGenerationSwapChainInterpolationCommandListDX12	
ffxConfigure	ffxConfigureDescFrameGenerationSwapChainKeyValueDX12	

- dx12/ffx_api_dx12.h

FRAME GENERATION SWAPCHAIN DX12 CONTEXT CREATE

API Function	Struct Type
ffxCreateContext	ffxCreateContextDescFrameGenerationSwapChainWrapDX12
ffxCreateContext	ffxCreateContextDescFrameGenerationSwapChainNewDX12
ffxCreateContext	ffxCreateContextDescFrameGenerationSwapChainForHwndDX12

```
#define FFX_API_CREATE_CONTEXT_DESC_TYPE_FRAMEGENERATIONSWAPCHAIN_WRAP_DX12 0x30001u
struct ffxCreateContextDescFrameGenerationSwapChainWrapDX12
{
    ffxCreateContextDescHeader header;
    IDXGISwapChain4** swapchain;        ///< Input swap chain to wrap, output frame interpolation swapchain.
    ID3D12CommandQueue* gameQueue;      ///< Input command queue to be used for presentation.
};
#define FFX_API_CREATE_CONTEXT_DESC_TYPE_FRAMEGENERATIONSWAPCHAIN_NEW_DX12 0x30005u
struct ffxCreateContextDescFrameGenerationSwapChainNewDX12
{
    ffxCreateContextDescHeader header;
    IDXGISwapChain4** swapchain;        ///< Output frame interpolation swapchain.
    DXGI_SWAP_CHAIN_DESC* desc;         ///< Swap chain creation parameters.
    IDXGIFactory* dxgiFactory;          ///< IDXGIFactory to use for DX12 swapchain creation.
    ID3D12CommandQueue* gameQueue;      ///< Input command queue to be used for presentation.
};

#define FFX_API_CREATE_CONTEXT_DESC_TYPE_FRAMEGENERATIONSWAPCHAIN_FOR_HWND_DX12 0x30006u
struct ffxCreateContextDescFrameGenerationSwapChainForHwndDX12
{
    ffxCreateContextDescHeader header;
    IDXGISwapChain4** swapchain;                        ///< Output frame interpolation swapchain.
    HWND hwnd;                                          ///< HWND handle for the calling application;
    DXGI_SWAP_CHAIN_DESC1* desc;                        ///< Swap chain creation parameters.
    DXGI_SWAP_CHAIN_FULLSCREEN_DESC* fullscreenDesc;    ///< Fullscreen swap chain creation parameters.
    IDXGIFactory* dxgiFactory;                          ///< IDXGIFactory to use for DX12 swapchain creation.
    ID3D12CommandQueue* gameQueue;                      ///< Input command queue to be used for presentation.
};
```

FRAME GENERATION SWAPCHAIN DX12 CONTEXT CREATE

API Function	Struct Type
ffxCreateContext	ffxCreateContextDescFrameGenerationSwapChainForHwndDX12

Example from FSR API sample, which leverages C++ helpers to chain extensions. Needs to know command lists and outputs, queryable from swapchain.

```
IDXGISwapChain4* dxgiSwapchain = GetSwapChain()->GetImpl()->DX12SwapChain();
dxgiSwapchain->AddRef();
cauldron::GetSwapChain()->GetImpl()->SetDXGISwapChain(nullptr);

ffx::CreateContextDescFrameGenerationSwapChainForHwndDX12 createSwapChainDesc{};
dxgiSwapchain->GetHwnd(&createSwapChainDesc.hwnd);
DXGI_SWAP_CHAIN_DESC1 desc1; dxgiSwapchain->GetDesc1(&desc1); createSwapChainDesc.desc = &desc1;
DXGI_SWAP_CHAIN_FULLSCREEN_DESC fullscreenDesc; dxgiSwapchain->GetFullscreenDesc(&fullscreenDesc); createSwapChainDesc.fullscreenDesc = &fullscreenDesc;
dxgiSwapchain->GetParent(IID_PPV_ARGS(&createSwapChainDesc.dxgiFactory));
createSwapChainDesc.gameQueue = GetDevice()->GetImpl()->DX12CmdQueue(cauldron::CommandQueue::Graphics);
dxgiSwapchain->Release(); dxgiSwapchain = nullptr;
createSwapChainDesc.swapchain = &dxgiSwapchain;
ffx::ReturnCode retCode = ffx::CreateContext(m_SwapChainContext, nullptr, createSwapChainDesc);
CauldronAssert(retCode == ffx::ReturnCode::Ok, L"Couldn't create the ffxapi fg swapchain (dx12): %d", (uint32_t)retCode);
createSwapChainDesc.dxgiFactory->Release();
cauldron::GetSwapChain()->GetImpl()->SetDXGISwapChain(dxgiSwapchain);

// In case the app is handling Alt-Enter manually we need to update the window association after creating a different swapchain
IDXGIFactory7* factory = nullptr;
if (SUCCEEDED(dxgiSwapchain->GetParent(IID_PPV_ARGS(&factory))))
{
    factory->MakeWindowAssociation(cauldron::GetFramework()->GetImpl()->GetHWND(), DXGI_MWA_NO_WINDOW_CHANGES);
    factory->Release();
}

dxgiSwapchain->Release();

// Lets do the same for HDR as well as it will need to be re initialized since swapchain was re created
cauldron::GetSwapChain()->SetHDRMetadataAndColorspace();
```

FRAME GENERATION SWAPCHAIN DX12 CONFIGURE

API Function	Struct Type	Associated types
ffxConfigure	ffxConfigureDescFrameGenerationSwapChainRegisterUiResourceDX12	FfxApiUiCompositionFlags

```
#define FFX_API_CONFIGURE_DESC_TYPE_FRAMEGENERATIONSWAPCHAIN_REGISTERUIRESOURCE_DX12 0x30002u
struct ffxConfigureDescFrameGenerationSwapChainRegisterUiResourceDX12
{
    ffxConfigureDescHeader header;
    struct FfxApiResource  uiResource;   ///< Resource containing user interface for composition. May be empty.
    uint32_t               flags;        ///< Zero or combination of values from FfxApiUiCompositionFlags.
};
```

Example from FSR API sample, which leverages C++ helpers.

```
ffx::ConfigureDescFrameGenerationSwapChainRegisterUiResourceDX12 uiConfig{};
uiConfig.uiResource = {};
uiConfig.flags = 0;
ffx::Configure(m_SwapChainContext, uiConfig);
```

FRAME GENERATION SWAPCHAIN DX12 DISPATCH

API Function	Struct Type
ffxDispatch	ffxDispatchDescFrameGenerationSwapChainWaitForPresentsDX12

```
#define FFX_API_DISPATCH_DESC_TYPE_FRAMEGENERATIONSWAPCHAIN_WAIT_FOR_PRESENTS_DX12 0x30007u
struct ffxDispatchDescFrameGenerationSwapChainWaitForPresentsDX12
{
    ffxDispatchDescHeader header;
};
```

FRAME GENERATION SWAPCHAIN DX12 QUERIES

API Function	Struct Type	Associated types
ffxQuery	ffxQueryDescFrameGenerationSwapChainInterpolationTextureDX12	
ffxQuery	ffxQueryDescFrameGenerationSwapChainInterpolationCommandListDX12	

```
#define FFX_API_QUERY_DESC_TYPE_FRAMEGENERATIONSWAPCHAIN_INTERPOLATIONCOMMANDLIST_DX12 0x30003u
struct ffxQueryDescFrameGenerationSwapChainInterpolationCommandListDX12
{
    ffxQueryDescHeader header;
    void** pOutCommandList;             ///< Output command list (ID3D12GraphicsCommandList) to be used for frame generation dispatch.
};

#define FFX_API_QUERY_DESC_TYPE_FRAMEGENERATIONSWAPCHAIN_INTERPOLATIONTEXTURE_DX12 0x30004u
struct ffxQueryDescFrameGenerationSwapChainInterpolationTextureDX12
{
    ffxQueryDescHeader header;
    struct FfxApiResource *pOutTexture; ///< Output resource in which the frame interpolation result should be placed.
};
```

FRAME GENERATION SWAPCHAIN DX12 QUERIES

API Function	Struct Type	Associated types
ffxQuery	ffxQueryDescFrameGenerationSwapChainInterpolationTextureDX12	
ffxQuery	ffxQueryDescFrameGenerationSwapChainInterpolationCommandListDX12	

Example from FSR API sample, which leverages C++ helpers.

```
ffx::DispatchDescFrameGeneration dispatchFg{};

ffx::QueryDescFrameGenerationSwapChainInterpolationCommandListDX12 queryCmdList{};
queryCmdList.pOutCommandList = &dispatchFg.commandList;
ffx::Query(m_SwapChainContext, queryCmdList);

ffx::QueryDescFrameGenerationSwapChainInterpolationTextureDX12 queryFiTexture{};
queryFiTexture.pOutTexture = &dispatchFg.outputs[0];
ffx::Query(m_SwapChainContext, queryFiTexture);
```

FRAME GENERATION SWAPCHAIN VULKAN CONTEXT

API Function	Struct Type	Associated types
ffxCreateContext	ffxCreateContextDescFrameGenerationSwapChainVK	VkFrameInterpolationInfoFFX
ffxConfigure	ffxConfigureDescFrameGenerationSwapChainRegisterUiResourceVK	FfxApiUiCompositionFlags
ffxDispatch	ffxDispatchDescFrameGenerationSwapChainWaitForPresentsVK	
ffxQuery	ffxQueryDescFrameGenerationSwapChainInterpolationTextureVK	
ffxQuery	ffxQueryDescFrameGenerationSwapChainInterpolationCommandListVK	
ffxQuery	ffxQueryDescSwapchainReplacementFunctionsVK	
ffxConfigure	ffxConfigureDescFrameGenerationSwapChainKeyValueVK	

- vk/ffx_api_vk.h

FRAME GENERATION SWAPCHAIN VULKAN CONTEXT CREATE

API Function	Struct Type
ffxCreateContext	ffxCreateContextDescFrameGenerationSwapChainVK

```
#define FFX_API_CREATE_CONTEXT_DESC_TYPE_FGSWAPCHAIN_VK 0x40001u
struct ffxCreateContextDescFrameGenerationSwapChainVK
{
    ffxCreateContextDescHeader header;
    VkDevice                   device;     ///< the logical device used by the program.
    VkSwapchainKHR*            swapchain;  ///< the current swapchain to be replaced. Will be destroyed when the context is created. This can be VK_NULL_HANDLE.
Will contain the new swapchain on return.
    VkQueue                    gameQueue;  ///< the graphics queue used for presenting.
    VkAllocationCallbacks*     allocator;  ///< optional allocation callbacks.
    VkSwapchainCreateInfoKHR   createInfo; ///< the description of the desired swapchain. If its VkSwapchainCreateInfoKHR::oldSwapchain field isn't
VK_NULL_HANDLE, it should be the same as the ffxCreateContextDescFrameGenerationSwapChainVK::swapchain field above.
};
```

FRAME GENERATION SWAPCHAIN VULKAN CONTEXT CREATE

API Function	Struct Type
ffxCreateContext	ffxCreateContextDescFrameGenerationSwapChainVK

Example from FSR API sample, which leverages C++ helpers to chain extensions. Needs to know command lists and outputs, queryable from swapchain.

```
cauldron::SwapChain* pSwapchain       = cauldron::GetFramework()->GetSwapChain();
VkSwapchainKHR       currentSwapchain = pSwapchain->GetImpl()->VKSwapChain();
ffx::CreateContextDescFrameGenerationSwapChainVK createSwapChainDesc{};
createSwapChainDesc.device     = cauldron::GetDevice()->GetImpl()->VKDevice();
createSwapChainDesc.swapchain  = &currentSwapchain;
createSwapChainDesc.gameQueue  = cauldron::GetDevice()->GetImpl()->VKCmdQueue(cauldron::CommandQueue::Graphics);
createSwapChainDesc.createInfo = *cauldron::GetFramework()->GetSwapChain()->GetImpl()->GetCreateInfo();

cauldron::GetFramework()->GetSwapChain()->GetImpl()->SetVKSwapChain(VK_NULL_HANDLE);

VkFrameInterpolationInfoFFX frameInterpolationInfo   = {};
frameInterpolationInfo.sType                         = VK_STRUCTURE_TYPE_FRAME_INTERPOLATION_INFO_FFX;
frameInterpolationInfo.pNext                         = nullptr;
frameInterpolationInfo.device                        = cauldron::GetDevice()->GetImpl()->VKDevice();
frameInterpolationInfo.physicalDevice                = cauldron::GetDevice()->GetImpl()->VKPhysicalDevice();
frameInterpolationInfo.pAllocator                    = nullptr;
frameInterpolationInfo.gameQueue.queue               = cauldron::GetDevice()->GetImpl()->VKCmdQueue(cauldron::CommandQueue::Graphics);
frameInterpolationInfo.gameQueue.familyIndex         = cauldron::GetDevice()->GetImpl()->GetQueueFamilies().familyIndices[cauldron::RequestedQueue::Graphics];
frameInterpolationInfo.gameQueue.submitFunc          = nullptr;  // this queue is only used in vkQueuePresentKHR, hence doesn't need a callback
frameInterpolationInfo.asyncComputeQueue.queue       = cauldron::GetDevice()->GetImpl()->GetFIAsyncComputeQueue()->queue;
frameInterpolationInfo.asyncComputeQueue.familyIndex = cauldron::GetDevice()->GetImpl()->GetQueueFamilies().familyIndices[cauldron::RequestedQueue::FIAsyncCompute];
frameInterpolationInfo.asyncComputeQueue.submitFunc  = nullptr;
frameInterpolationInfo.presentQueue.queue            = cauldron::GetDevice()->GetImpl()->GetFIPresentQueue()->queue;
frameInterpolationInfo.presentQueue.familyIndex      = cauldron::GetDevice()->GetImpl()->GetQueueFamilies().familyIndices[cauldron::RequestedQueue::FIPresent];
frameInterpolationInfo.presentQueue.submitFunc       = nullptr;
frameInterpolationInfo.imageAcquireQueue.queue       = cauldron::GetDevice()->GetImpl()->GetFIImageAcquireQueue()->queue;
frameInterpolationInfo.imageAcquireQueue.familyIndex = cauldron::GetDevice()->GetImpl()->GetQueueFamilies().familyIndices[cauldron::RequestedQueue::FIImageAcquire];
frameInterpolationInfo.imageAcquireQueue.submitFunc  = nullptr;
```

FRAME GENERATION SWAPCHAIN VULKAN CONTEXT CREATE

API Function	Struct Type
`ffxCreateContext`	`ffxCreateContextDescFrameGenerationSwapChainVK`

In Vulkan, it is necessary to pass specific queues when creating the context. The context needs 4 distinct queues, which 3 of them are only used by the FSR context.
For each queue, pass the queue, its family (for queue family ownership transfer purposes) and an optional Submit function if you want to control concurrent submissions in case the queue is used elsewhere in your engine.
However, it is recommended to make sure that each queue is only used by the FSR3 context and to leave the Submit function at nullptr.

The queues are:

- game queue: the queue where the replacement of vkQueuePresentKHR is called. This queue should have Graphics and Compute capabilities (Transfer is implied as per Vulkan specification). It can be shared with the engine. No Submit function is necessary. The code assumes that the UI texture is owned by that queue family when present is called.
- async compute queue: optional queue with Compute capability (Transfer is implied as per Vulkan specification). If used by the engine, prefer not to enable the async compute path of FSR3 Frame interpolation.
- present queue: queue with Graphics, Compute or Transfer capability, and Present support. This queue **cannot** be used by the engine. Otherwise, some deadlock can occur.
- image acquire queue: this one doesn't need any capability. Strongly prefer a queue not used by the engine. The main graphics queue can work too but it might delay the signaling of the semaphore/fence when acquiring a new image, negatively impacting the performance.

Because Vulkan requires queues to be requested at VkDevice creation time, the way to select those queues is the responsibility of the integrator. The cauldron sample however shows *one* way to do it.

FRAME GENERATION SWAPCHAIN VULKAN CONTEXT CREATE

API Function	Struct Type
ffxCreateContext	ffxCreateContextDescFrameGenerationSwapChainVK

```
frameInterpolationInfo.pNext          = createSwapChainDesc.createInfo.pNext;
createSwapChainDesc.createInfo.pNext = &frameInterpolationInfo;

ffx::ReturnCode retCode = ffx::CreateContext(m_SwapChainContext, nullptr, createSwapChainDesc);

ffx::QueryDescSwapchainReplacementFunctionsVK replacementFunctions{};
ffx::Query(m_SwapChainContext, replacementFunctions);
cauldron::GetDevice()->GetImpl()->SetSwapchainMethodsAndContext(nullptr,
                                                                nullptr,
                                                                replacementFunctions.pOutGetSwapchainImagesKHR,
                                                                replacementFunctions.pOutAcquireNextImageKHR,
                                                                replacementFunctions.pOutQueuePresentKHR,
                                                                replacementFunctions.pOutSetHdrMetadataEXT,
                                                                replacementFunctions.pOutCreateSwapchainFFX,
                                                                replacementFunctions.pOutDestroySwapchainFFX,
                                                                replacementFunctions.pOutGetLastPresentCountFFX,
                                                                m_SwapChainContext);

// Set frameinterpolation swapchain to engine
cauldron::GetFramework()->GetSwapChain()->GetImpl()->SetVKSwapChain(currentSwapchain, &frameInterpolationInfo);
```

Queries of replacement functions will be explained in the next slides

FRAME GENERATION SWAPCHAIN VULKAN CONFIGURE

API Function	Struct Type	Associated types
ffxConfigure	ffxConfigureDescFrameGenerationSwapChainRegisterUiResourceVK	FfxApiUiCompositionFlags

```
#define FFX_API_CONFIGURE_DESC_TYPE_FRAMEGENERATIONSWAPCHAIN_REGISTERUIRESOURCE_VK 0x40002u
struct ffxConfigureDescFrameGenerationSwapChainRegisterUiResourceDX12
{
    ffxConfigureDescHeader header;
    struct FfxApiResource  uiResource;    ///< Resource containing user interface for composition. May be empty.
    uint32_t               flags;         ///< Zero or combination of values from FfxApiUiCompositionFlags.
};
```

Example from FSR API sample, which leverages C++ helpers.

```
ffx::ConfigureDescFrameGenerationSwapChainRegisterUiResourceVK uiConfig{};
uiConfig.uiResource = {};
uiConfig.flags = 0;
ffx::Configure(m_SwapChainContext, uiConfig);
```

FRAME GENERATION SWAPCHAIN VULKAN DISPATCH

API Function	Struct Type
ffxDispatch	ffxDispatchDescFrameGenerationSwapChainWaitForPresentsVK

```
#define FFX_API_DISPATCH_DESC_TYPE_FRAMEGENERATIONSWAPCHAIN_WAIT_FOR_PRESENTS_VK 0x40007u
struct ffxDispatchDescFrameGenerationSwapChainWaitForPresentsVK
{
    ffxDispatchDescHeader header;
};
```

FRAME GENERATION SWAPCHAIN VULKAN QUERIES

API Function	Struct Type	Associated types
ffxQuery	ffxQueryDescFrameGenerationSwapChainInterpolationCommandListVK	
ffxQuery	ffxQueryDescFrameGenerationSwapChainInterpolationTextureVK	

```
#define FFX_API_QUERY_DESC_TYPE_FGSWAPCHAIN_INTERPOLATIONCOMMANDLIST_VK 0x40003u
struct ffxQueryDescFrameGenerationSwapChainInterpolationCommandListVK
{
    ffxQueryDescHeader header;
    void**             pOutCommandList;  ///< Output command nuffer (VkCommandBuffer) to be used for frame generation dispatch.
};

#define FFX_API_QUERY_DESC_TYPE_FGSWAPCHAIN_INTERPOLATIONTEXTURE_VK 0x40004u
struct ffxQueryDescFrameGenerationSwapChainInterpolationTextureVK
{
    ffxQueryDescHeader     header;
    struct FfxApiResource* pOutTexture;  ///< Output resource in which the frame interpolation result should be placed.
};
```

FRAME GENERATION SWAPCHAIN VULKAN QUERIES

API Function	Struct Type	Associated types
ffxQuery	ffxQueryDescFrameGenerationSwapChainInterpolationTextureVK	
ffxQuery	ffxQueryDescFrameGenerationSwapChainInterpolationCommandListVK	

Example from FSR API sample, which leverages C++ helpers.

```
ffx::DispatchDescFrameGeneration dispatchFg{};

ffx::QueryDescFrameGenerationSwapChainInterpolationCommandListVK queryCmdList{};
queryCmdList.pOutCommandList = &dispatchFg.commandList;
ffx::Query(m_SwapChainContext, queryCmdList);

ffx::QueryDescFrameGenerationSwapChainInterpolationTextureVK queryFiTexture{};
queryFiTexture.pOutTexture = &dispatchFg.outputs[0];
ffx::Query(m_SwapChainContext, queryFiTexture);
```

FRAME GENERATION SWAPCHAIN VULKAN QUERIES

API Function	Struct Type	Associated types
ffxQuery	ffxQueryDescSwapchainReplacementFunctionsVK	

```
/// Function to get the number of presents. This is useful when using frame interpolation
typedef uint64_t (*PFN_getLastPresentCountFFX)(VkSwapchainKHR);

/// FFX API specific functions to create and destroy a swapchain
typedef VkResult(*PFN_vkCreateSwapchainFFX)(VkDevice device, const VkSwapchainCreateInfoKHR* pCreateInfo, const VkAllocationCallbacks* pAllocator, VkSwapchainKHR* pSwapchain,
                                            void* pContext);
typedef void(*PFN_vkDestroySwapchainFFX)(VkDevice device, VkSwapchainKHR swapchain, const VkAllocationCallbacks* pAllocator, void* pContext);

#define FFX_API_QUERY_DESC_TYPE_FGSWAPCHAIN_FUNCTIONS_VK 0x40005u
struct ffxQueryDescSwapchainReplacementFunctionsVK
{
    ffxQueryDescHeader header;
    PFN_vkCreateSwapchainFFX    pOutCreateSwapchainFFX;      ///< Replacement of vkCreateSwapchainKHR. Can be called when swapchain is recreated but swapchain context isn't (for
example when toggling vsync).
    PFN_vkDestroySwapchainFFX   pOutDestroySwapchainFFX;     ///< Replacement of vkDestroySwapchainKHR. Can be called when swapchain is destroyed but swapchain context isn't.
    PFN_vkGetSwapchainImagesKHR pOutGetSwapchainImagesKHR;   ///< Replacement of vkGetSwapchainImagesKHR.
    PFN_vkAcquireNextImageKHR   pOutAcquireNextImageKHR;     ///< Replacement of vkAcquireNextImageKHR.
    PFN_vkQueuePresentKHR       pOutQueuePresentKHR;         ///< Replacement of vkQueuePresentKHR.
    PFN_vkSetHdrMetadataEXT     pOutSetHdrMetadataEXT;       ///< Replacement of vkSetHdrMetadataEXT.
    PFN_getLastPresentCountFFX  pOutGetLastPresentCountFFX;  ///< Additional function to get the number of times present has been called since the swapchain creation.
};
```

Note that functions to create and destroy swapchain have a different signature than the usual vulkan ones. They need the swapchain context.
The other functions can be used as if they were the usual Vulkan ones.

FRAME GENERATION SWAPCHAIN VULKAN QUERIES

API Function	Struct Type	Associated types
ffxQuery	ffxQueryDescSwapchainReplacementFunctionsVK	

Example from FSR API sample, showing how to query the new replacement functions

```
ffx::QueryDescSwapchainReplacementFunctionsVK replacementFunctions{};
ffx::Query(m_SwapChainContext, replacementFunctions);
cauldron::GetDevice()->GetImpl()->SetSwapchainMethodsAndContext(nullptr,
                                                                 nullptr,
                                                                 replacementFunctions.pOutGetSwapchainImagesKHR,
                                                                 replacementFunctions.pOutAcquireNextImageKHR,
                                                                 replacementFunctions.pOutQueuePresentKHR,
                                                                 replacementFunctions.pOutSetHdrMetadataEXT,
                                                                 replacementFunctions.pOutCreateSwapchainFFX,
                                                                 replacementFunctions.pOutDestroySwapchainFFX,
                                                                 replacementFunctions.pOutGetLastPresentCountFFX,
                                                                 m_SwapChainContext);
```

OTHER VULKAN CONSIDERATIONS

- When using the frame interpolation swapchain:
  - the frame interpolation queried swapchain image aren't real swapchain images. They are standard VkImage objects. As a result, they shouldn't be transitioned into a VK_IMAGE_LAYOUT_PRESENT_SRC_KHR layout as it isn't allowed by the Vulkan specification.
  - Before calling the replacement of vkQueuePresentKHR, the swapchain image should be in a VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL layout.

FSR 3.1 INTEGRATION STEPS (DIRECTX® 12 VERSION)

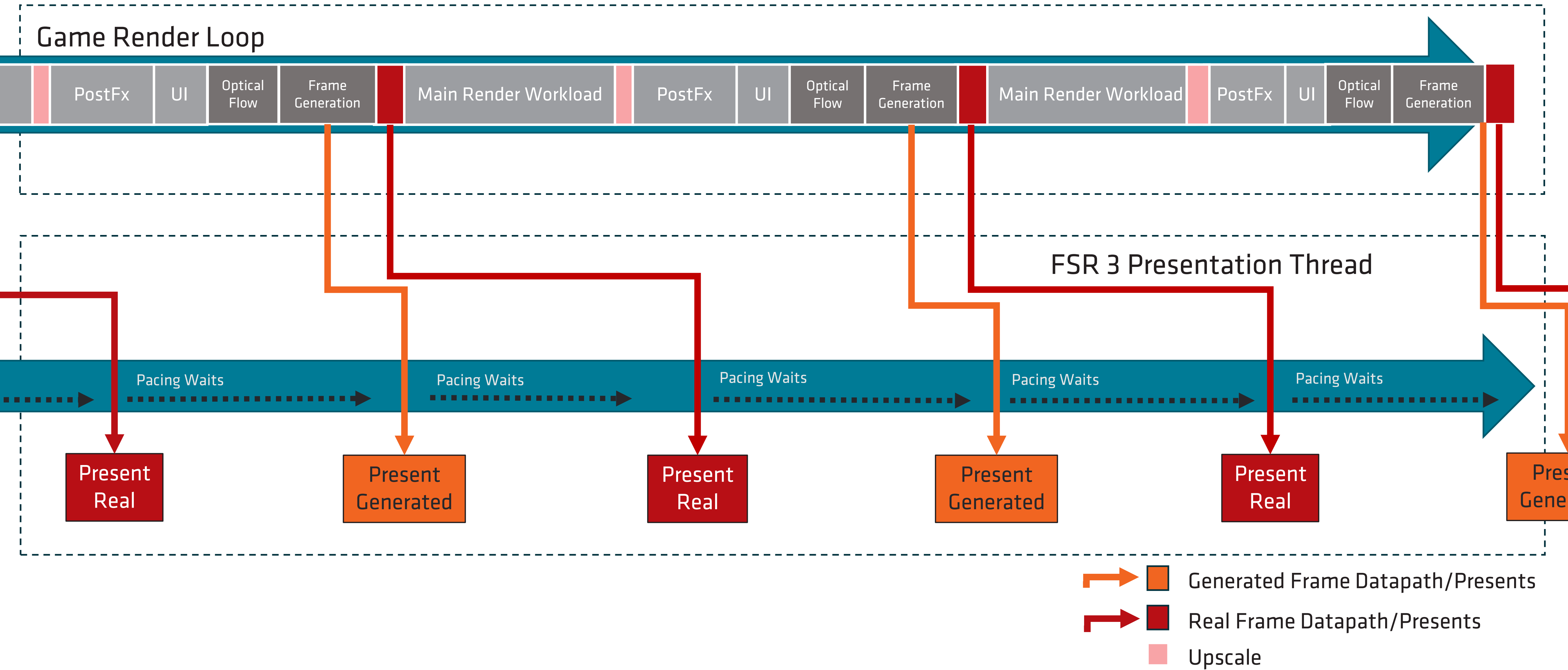
FSR 3.1 DX12 INTEGRATION – BASIC STEPS

- Step 1 - Upscaling
  - FSR 3 integration is similar to FSR 2 for upscaling
    - If FSR 2 already is supported: Replace FSR 2 upscale with FSR 3 upscale
- Step 2 - Swapchain
  - Replace or create your swap chain using Frame Generation Swapchain context
    - Implementing IDXGISwapChain interface
    - Assisting frame generation, frame pacing and present
  - This will internally create
    - CPU threads for frame pacing calculations and real present/UI callbacks
    - A GPU queue to handle presents and another for interpolation
- Step 3 – Frame Generation
  - FSR 3 frame generation requires some additional input on setup and dispatch, and swapchain handling.
    - Start off with Async disabled to validate integration quality.
    - Ensure the FrameGenerationPreparation dispatch is done with upscale input resources.
    - Call Configure before FrameGenerationPreparation
- Step 4 – UI Handling
  - Three different UI composition options: (note: this step is REQUIRED for integration)
    - **UI texture**: UI is stored in another RT on top of generated frames.
    - **Callback**: User calls an FSR 3 function where UI can rendered on top of the generated frame.
    - **HUDLess**: Identify UI by comparing the game frame without UI to the game frame with UI.

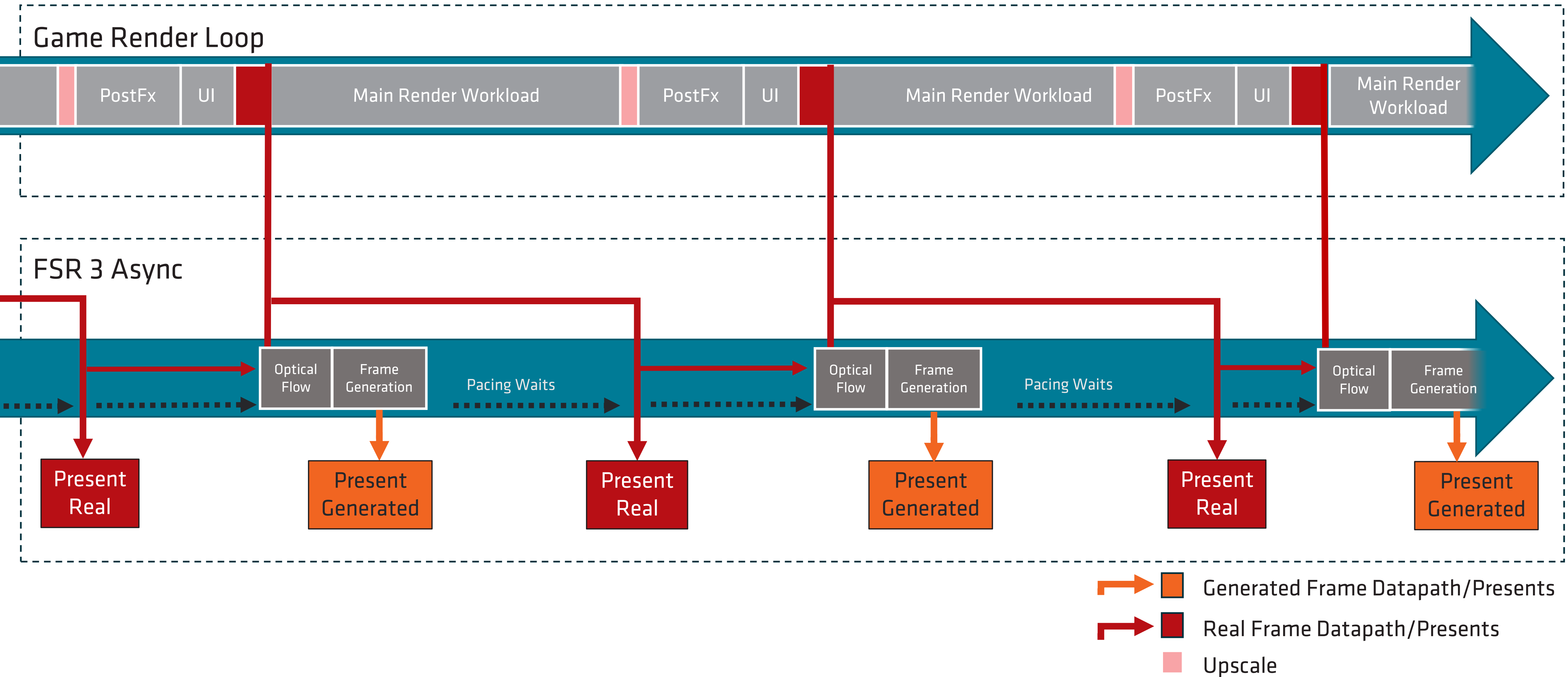
FSR 3 OPTICAL FLOW AND FRAME GENERATION WORKLOADS

- The Optical Flow and Frame Generation workloads can be either run on the **Presentation** Queue provided by the game, or an **Async** Queue provided by the FrameInterpolationSwapchain.
  - To enable Async queue the Frame Generation context must be created with flag FFX_FSR3_ENABLE_ASYNC_WORKLOAD_SUPPORT, and ffxConfigureFrameGeneration called with allowAsyncWorkloads = TRUE
- We recommend integrating with Presentation Queue use first, and then testing Async Queue use.
  - Async can introduce sync issues with frames that call Upscale, but do not Present(), which means accurate tracking and disabling/enabling frame generation is essential. Presentation Queue use can minimise these issues.
  - If a fence is required to indicate any HUDLess UI resource is consumed, this can be done on the Presentation Queue, but not on the Async Queue.
- Async Queue use can increase performance if the workload can be overlapped with early next frame rendering.
- Async Queue use will increase memory usage, as some resources need to get double-buffered so Frame Generation can execute in parallel to the next frame being rendered

FSR 3 – PRESENT QUEUE UPSCALING AND FRAME GENERATION PIPELINE



FSR 3 – ASYNC UPSCALING AND FRAME GENERATION PIPELINE



FSR 3 NATIVE DX12 INTEGRATION - SWAPCHAIN

- Swapchain handling has multiple possibilities for integration.
- The Cauldron sample shows an example using `CreateContextDescFrameGenerationSwapChainForHwndDX12`
  - This requires games are **not** using Exclusive Fullscreen.
  - It recommended to replace swapchain creation with creation of a FrameInterpolation swapchain in the engine and always use the proxy swapchain, even when frame generation is disabled

- For some titles this may not be ideal. Another option is to always use the `FrameGenerationSwapChain` when FSR 3 Upscaling is enabled.
  - It is acceptable to require a game restart for this to take effect.

- The `FrameGenerationSwapChain` will not perform frame generation until it's registered using `ffxConfigureDescFrameGeneration`.
  - **When frame generation is disabled the `FrameGenerationSwapChain` will still handle UI composition.**

- If another system is in place which also hooks into swap chain, undefined behaviors may mean a game restart is required to enable FSR 3.

FSR 3 NATIVE DX12 INTEGRATION: UI COMPOSITION

- UI composition and Present can happen asynchronously to a game's Frame Generation
  - Note: the game needs to use **multiple command lists** to allow UI composition and presents to be injected on the GPUs graphics queue

- UI composition is a **required** step for FSR 3 integration
  - Failure to implement this step will result in major UI artefacts when Frame Generation is enabled!

- FSR 3 supports three composition modes of UI (User Interface) rendering:
  - **Callback** (recommended)
  - **UI texture**
  - **HUDLess** (compatibility mode, not recommended)

- Pick the best one according to your needs (see next slides)
  - One of them **must** be used. We recommend Callback for quality purposes.

FSR 3 NATIVE DX12 INTEGRATION - UI CALLBACK COMPOSITION MODE

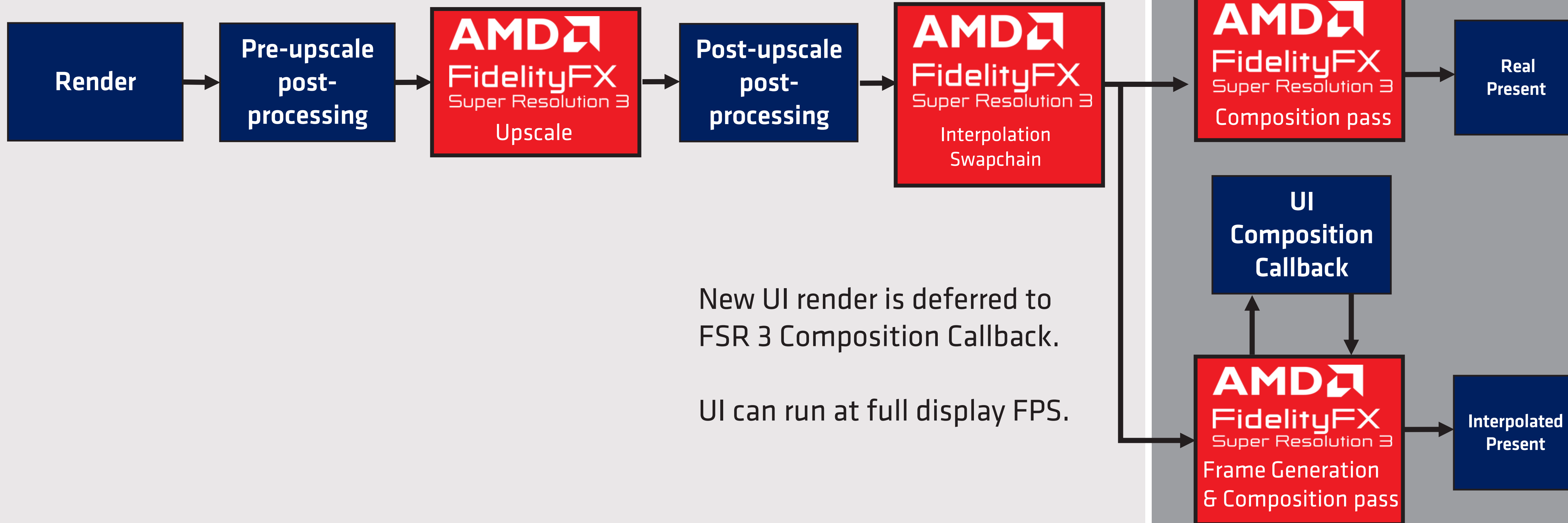
- Overload the UI composition by passing a callback function pointer in the `ffxConfigureDescFrameGeneration.presentCallback` during frame generation configure.
- Function Pointer definition:

```
typedef ffxReturnCode_t(*FfxApiPresentCallbackFunc)(ffxCallbackDescFrameGenerationPresent*
params, void* pUserCtx);
```

- Use this for more complex effects needed for the UI (e.g. blurring the UI background) or when low latency UI is desired
- Using this method the UI will be displayed at **generated** framerate
  - We recommend only basic rendering is performed in this callback!
- UI callback will still be called even when FSR 3 Frame Generation is OFF
  - Will be composited on top of the real frame in this case

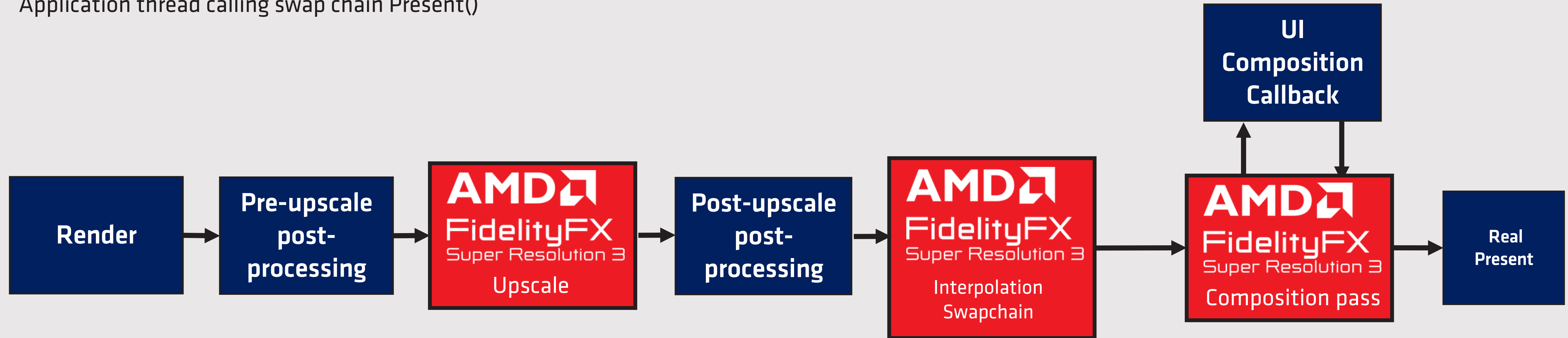
- *Note: UI callback composition mode is the **recommended** mode for UI composition*

FSR 3 NATIVE DX12 INTEGRATION - UI CALLBACK COMPOSITION MODE



FSR 3 NATIVE DX12 INTEGRATION - UI CALLBACK COMPOSITION MODE

WHEN FRAME GENERATION IS DISABLED



New UI render is deferred to FSR 3 Composition Callback.

FSR 3 NATIVE DX12 INTEGRATION – UI TEXTURE COMPOSITION MODE

- Render UI into a dedicated texture

```
ffx::ConfigureDescFrameGenerationSwapChainRegisterUiResourceDX12 uiConfig{};
uiConfig.uiResource = uiColor;
uiConfig.flags      = m_DoublebufferInSwapchain ? FFX_UI_COMPOSITION_FLAG_ENABLE_INTERNAL_UI_DOUBLE_BUFFERING : 0;
ffx::Configure(m_SwapChainContext, uiConfig);
```

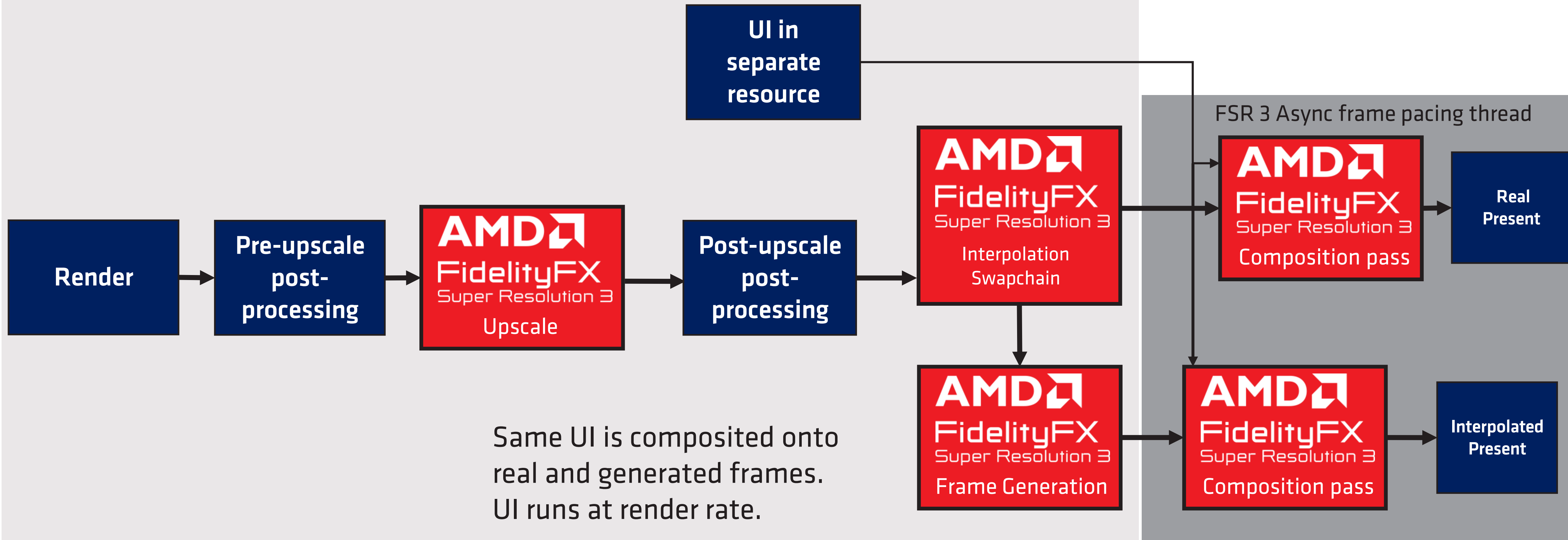
- Provide FSR 3 swapchain and UI resource in NON_PIXEL_SHADER_RESOURCE. Enable internal double buffering if required. If the resource is already double buffered for async use, memory can be saved by not passing this flag.

- FSR 3 will blend the UI onto the back buffer of **both** the interpolated and the real frame using the alpha channel of the UI texture

- Using this method the UI will be displayed at **render** framerate (not interpolated)
  - i.e. the same UI will be displayed for two consecutive frames if Frame Generation is enabled

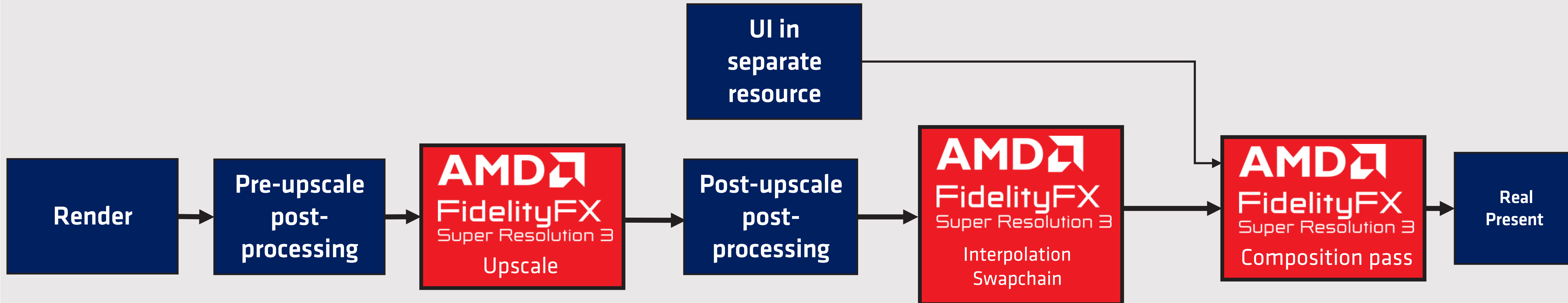
- *Note: If UI includes part of the currently-rendered frame (e.g. blurry background) then the Callback option is required otherwise artefacts will ensue*

FSR 3 NATIVE DX12 INTEGRATION - UI TEXTURE COMPOSITION MODE



FSR 3 NATIVE DX12 INTEGRATION - UI TEXTURE COMPOSITION MODE
WHEN FRAME GENERATION IS DISABLED

Application thread calling swap chain Present()



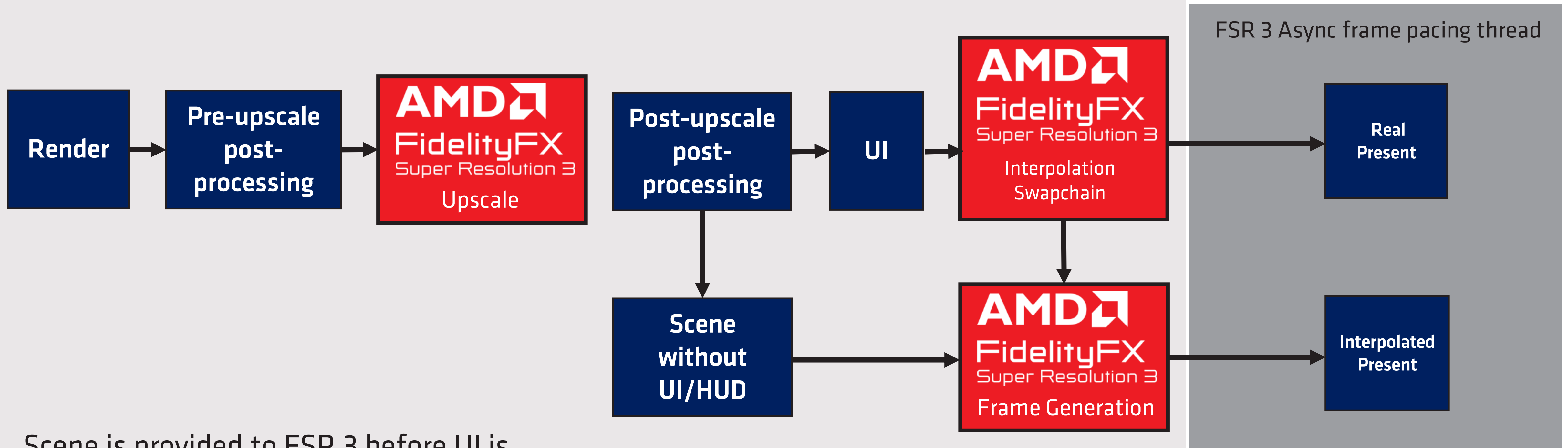
UI is composited onto real frames.

FSR 3 NATIVE DX12 INTEGRATION - HUDLESS COMPOSITION MODE

- Provide FSR 3 with scene resource pre UI/HUD render
  - Resource must be of same color and resource format as the swapchain present buffer
- Provide the resource in **HUDLessColor** as part of the `ffxConfigureDescFrameGeneration` struct of frame generation configure operation.
  - NON_PIXEL_SHADER_RESOURCE
- There should not be any additional effects applied after this resource, except basic UI.
- FSR 3 will blend the UI onto the back buffer of **both** the interpolated and the real frame using the alpha channel of the UI surface
- Using this method the UI will be displayed at **render** framerate (not interpolated)
  - i.e. the same UI will be displayed for two consecutive frames if Frame Generation is enabled

- *Note: this mode is considered a compatibility mode and is **not recommended** as it will likely produce visual artefacts in the UI*

FSR 3 NATIVE DX12 INTEGRATION - HUDLESS COMPOSITION MODE



Scene is provided to FSR 3 before UI is rendered. FSR 3 uses this data to generate UI masks. There is a possibility for artefacts on transparent elements.

FSR 3.1 INTEGRATION STEPS (UNREAL ENGINE 5)

FSR 3 UE5 INTEGRATION - INSTALL

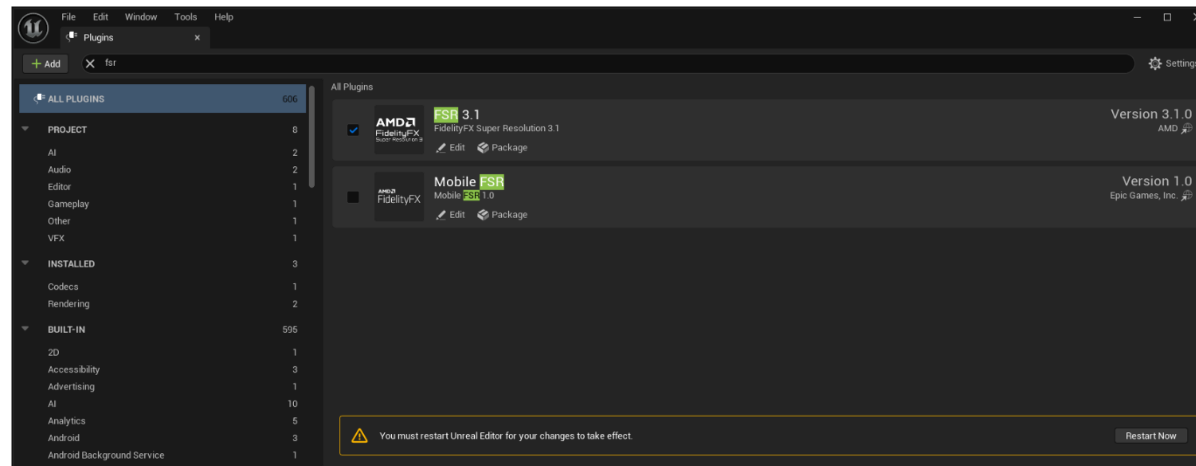
- The FSR 3 and FSR3MovieRenderPipeline plugins are intended for Unreal Engine 5.1.1* or later. If you are not a registered Unreal Engine developer, you will need to follow these instructions and register for access to this link.

- For optimal quality it is necessary to use Unreal Engine from source code and apply source code patches.
  - To improve FSR 3's handling of animated opaque materials:
    - Use: git apply <VERS>-LitReactiveShadingModel.patch
    - Where <VERS> should be the engine-version in use.

- Install the plugin into the Unreal Engine:
  - Locate the *Engine/Plugins* directory of your Unreal Engine installation.
  - Extract the contents of the FSR3 zip file.
  - Select the sub-folder that corresponds to the Unreal Engine version to be used.
  - Place the **FSR3** folder within your Unreal Engine source tree at: *Engine/Plugins/Marketplace*
    - (Optional) Place the **FSR3MovieRenderPipeline** folder within your Unreal Engine source tree at: *Engine/Plugins/Marketplace*.

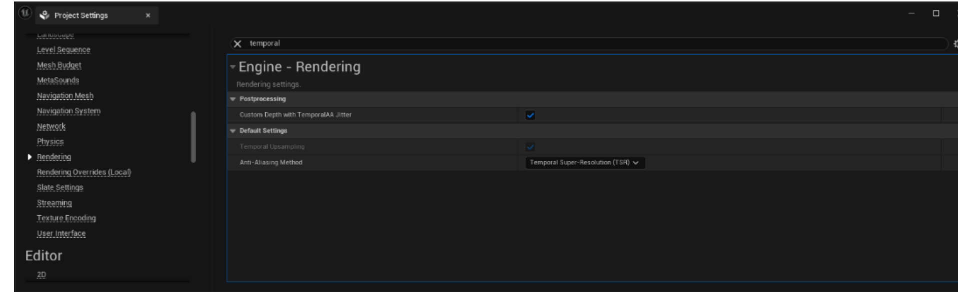
FSR 3 UE5 INTEGRATION – ENABLE PLUGIN

- To enable the plugin once installed:
  - Open your Unreal Engine project in the Editor.
  - Navigate to **Edit > Plugins** in the Unreal Engine toolbar.
  - Within the plugin dialog:
    - Ensure that All is selected on the left side.
    - Type fsr into the search box in the top right corner.
    - Select the **Enabled** checkbox for the **FSR 3.1** plugin.
      - i. (Optional) Select the **Enabled** checkbox for the **FSR3MovieRenderPipeline** plugin.
    - When prompted, click **Restart Now** to apply changes, and restart Unreal Engine.



FSR 3 UE5 INTEGRATION – BASIC USAGE

- **Temporal Upsampling** must be enabled in the **Project Settings > Rendering** window, accessed via **Edit > Project** Settings in the Unreal Engine toolbar or via the Console Variable *`r.TemporalAA.Upsampling`*.



- FSR 3.1 can be enabled or disabled via the **Enabled** option in the **Project Settings > FidelityFX Super Resolution 3.1** settings window, or with the console variable *`r.FidelityFX.FSR3.Enabled`* in the configuration files. The variable can be modified at runtime ***however*** this is not guaranteed to be safe when other third-party upscalers are also enabled.

FSR 3 UE5 INTEGRATION – QUALITY MODES

- The plugin will use specific quality modes specified via `` `r.FidelityFX.FSR3.QualityMode` `` overriding `` `r.ScreenPercentage` ``. The exposed modes are:
  - **Native AA (1.0x)**: *`r.FidelityFX.FSR3.QualityMode 0`*
    - Provides an image quality superior to native rendering with a modest performance cost.
  - **Quality (1.5x)**: *`r.FidelityFX.FSR3.QualityMode 1`*
    - Provides an image quality equal or superior to native rendering with a significant performance gain.
  - **Balanced (1.7x)**: *`r.FidelityFX.FSR3.QualityMode 2`*
    - Offers an ideal compromise between image quality and performance gains.
  - **Performance (2.0x)**: *`r.FidelityFX.FSR3.QualityMode 3`*
    - Provides an image quality similar to native rendering with a major performance gain.
  - **Ultra Performance (3.0x)**: *`r.FidelityFX.FSR3.QualityMode 4`*
    - Provides the highest performance gain while still maintaining an image quality representative of native rendering.

FSR 3 UE5 INTEGRATION

- For more integration instructions, including specific items such as world-position-offset and UI rendering, please see the Unreal Documentation available at https://gpuopen.com/learn/ue-fsr3/

FSR 3 TECHNICAL DETAILS AND GUIDELINES

FSR 3 FRAME GENERATION – MINIMUM FRAME RATE

- FSR 3 Frame Generation runs best when interpolating from a minimum of 60 fps **pre-interpolation** (e.g. after upscale)

- Whilst FSR 3 can roughly double any input frame rate, going below 60 is **not recommended**
  - This is due to interpolation visual artefacts being more prominent at lower frame rates
  - Any fps below 30 fps pre-interpolation should be absolutely avoided

- Altering FSR 3 behaviour based on sub-60 fps detection is **not recommended**
  - Better to educate users about this and let them adjust their own graphics settings as needed

FSR 3 FRAME GENERATION –FRAME RATE DISPLAY

- To calculate fps when FSR 3 frame generation is on, you can use the following function:

  swapchain::GetLastPresentCount()

- Please refer to how the sample outputs "Display FPS" to calculate fps

FSR 3 VARIABLE REFRESH RATE CONSIDERATIONS

- FreeSync, G-Sync and Adaptive Sync are all forms of Variable Refresh Rate technologies
  - With VRR, refresh rate is dictated by frames sent by the GPU to the monitor
- FSR 3 Frame Generation behaves according to the following table

	VRR OFF	VRR ON
V-Sync OFF	Tearing will be seen at all frame rates.	**Recommended setting if frame times are highly variable** Some tearing may appear in some circumstances (e.g. fps close to or above monitor's max refresh). Hardware Accelerated GPU Scheduling disabled may result in more tearing.
V-Sync ON	Tearing-free experience at all frame rates. FPS limited to integer multiple of max refresh when fps is under max refresh. This may cause "judder" due to uneven sync intervals.	**Recommended setting if frame times are stable (e.g. via limiter)** Tearing-free experience at all FPS Frame rate will be capped at monitor's maximum refresh rate. Render rate is implicitly limited to half of the monitor's max refresh rate

- It is highly recommended that games implement a frame limiter to provide options to players who want a steady framerate
- We have seen Streamline add tearing flags when the game requested none. This can directly affect the quality of FSR3 Frame Generation, and we recommend it is avoided when FSR3 used.

VISUAL DEBUG MARKERS

- FSR 3 has a debug view for Frame Generation and screen tearing
  - To enable it, configure with the FFX_FSR3_FRAME_GENERATION_FLAG_DRAW_DEBUG_TEAR_LINES flag

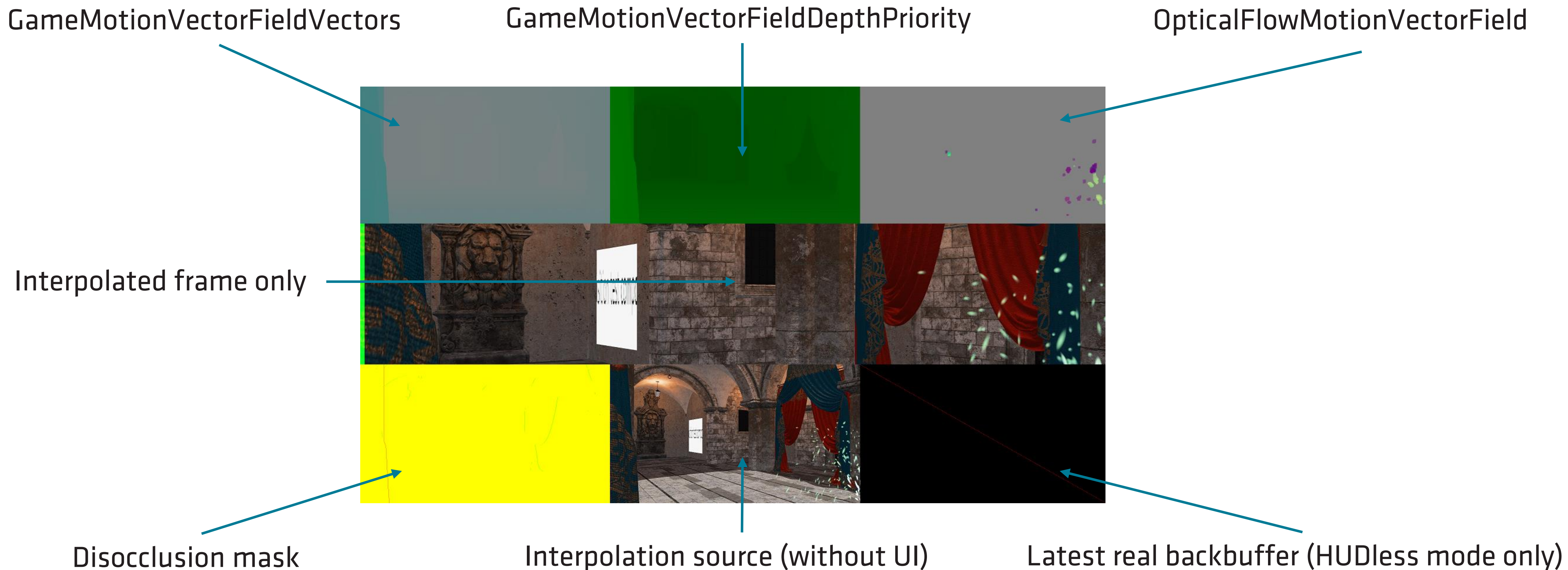
- FSR 3 will then render a solid green bar on the left, and a bar on the right which cycles through colors
  - A large red square in top left indicates Optical Flow has detected the scene has changed significantly.
  - A blue square in top left indicates the RESET passed to FSR 3 Frame Generation is TRUE

- With V-Sync enabled these bars should flicker entirely to show alternative frames being generated in perfect circumstances

- With FPS > Monitor Refresh Rate, and tearing enabled, these bars allow developers to see the tear regions clearly and tweak any particular frame pacing issue that may require per-game tuning

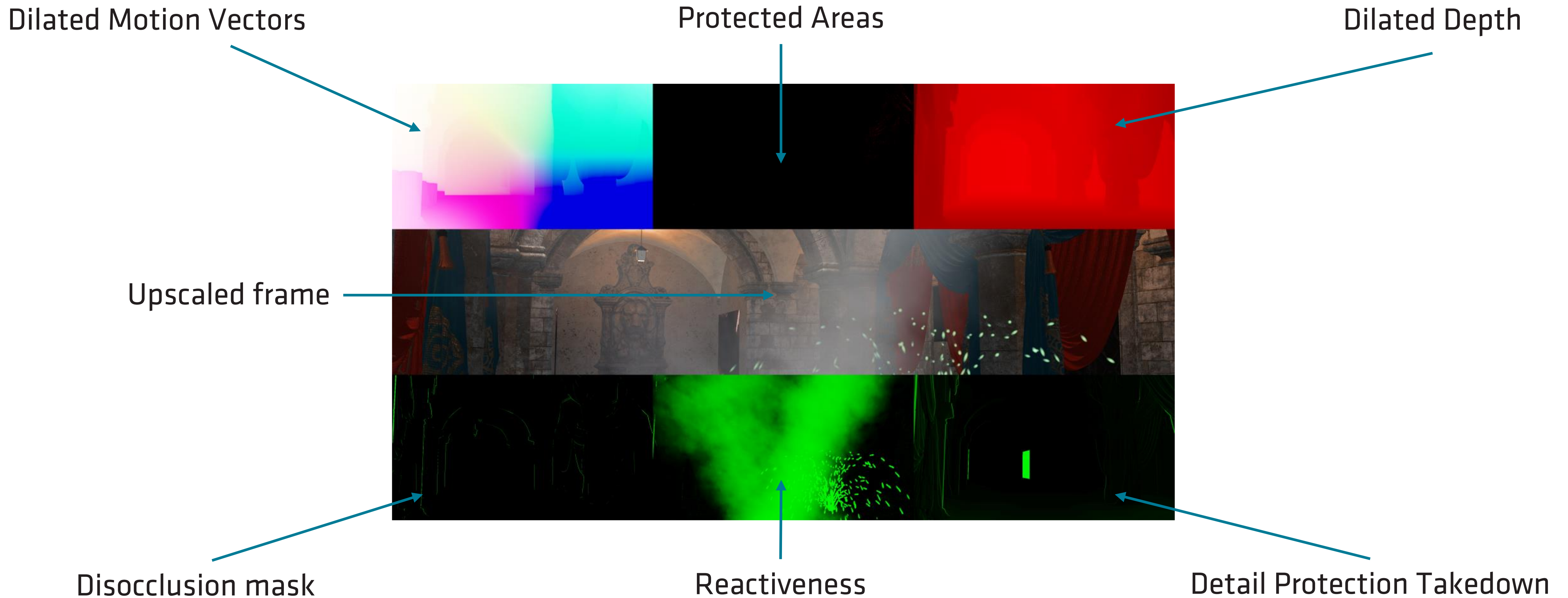
FRAME GENERATION VISUAL DEBUG MARKERS

- FSR Frame Generation has a debug view for integration testing
  - To enable it, configure with the FFX_FSR3_FRAME_GENERATION_FLAG_DRAW_DEBUG_VIEW flag



UPSCALE VISUAL DEBUG MARKERS

- FSR Upscaling has a debug view for integration testing
  - To enable it, configure with the FFX_FSR3UPSCALER_DISPATCH_DRAW_DEBUG_VIEW flag



MOTION VECTORS

- Motion vectors must be provided to the FSR 3 upscale component in the same way as FSR 2
- Motion vectors should be of **minimum 16-bit precision** for quality purposes
- All opaque elements and elements writing depth should have motion vectors:
    - Foliage and all other alpha tested materials
    - Playable characters & NPCs
    - Vehicles and other deformable geometry
- If elements have incorrect or missing motion vectors, ghosting will occur when upscaling
- **If upscaling has poor motion vector inputs, interpolated frames will amplify artefacts!**

FSR 3 CONSIDERATIONS

- Using FSR 3 **Swapchain context** has a slight performance impact even if frame generation is disabled (one extra back buffer copy)
  - When switching the swapchain, the game **must not** be in Exclusive Fullscreen mode

- FSR 3 and other frame generation solutions **should not** be enabled at the same time
  - When creating the FSR 3 **SwapChain,** other frame generation solutions should already be fully disabled

- Shader Model 6 is now required

RESOURCE LIFETIMES

- When UITexture composition mode is used:
  - The UITexture will get used in composition (or the UI callback)
  - If FFX_UI_COMPOSITION_FLAG_ENABLE_INTERNAL_UI_DOUBLE_BUFFERING is set:
    - The UITexture gets copied to an internal resource on the game queue
    - The UITexture may be reused on the GFX queue immediately in the next frame
  - If FFX_UI_COMPOSITION_FLAG_ENABLE_INTERNAL_UI_DOUBLE_BUFFERING is NOT set:
    - The application is responsible to ensure it persists until composition of the real frame is finished
    - This is typically in the middle of the next frame, so the UITexture should not be used during the next frame. The application must ensure double buffering of the UITexture

- When HUDLess composition mode is used:
  - The HUDLess texture will be used during FrameInterpolation
  - The application is responsible to ensure it persists until FrameInterpolation is complete
  - If FfxFrameGenerationConfig::allowAsyncWorkloads is true:
    - Frameinterpolation happens on an async compute queue so the HUDLess texture needs to be double buffered by the application
  - If FfxFrameGenerationConfig:: allowAsyncWorkloads is false:
    - Frameinterpolation happens on the game GFX queue, so the HUDLess texture may be reused in the next frame

NEW FSR 3 QUALITY MODE: NATIVE AA

- FSR 3 introduces a new Quality mode: **Native AA**
- In this mode FSR 3 Upscaling is a pure AA option (no actual upscaling takes place)
- “Native AA” with “Frame Generation” enabled together essentially provide frame generation without upscaling

- Note: Native AA quality mode has the largest performance overhead!

- Note: Native AA quality mode **still requires Reactive and Transparency & Composition masks** to work correctly!

VERSIONS OF FSR

FSR version	Technology description	Quality modes supported	API support	Hardware support[1]
FSR 1	Spatial upscale of input frames	Ultra Quality Quality Balanced Performance	DX12, DX11, Vulkan® UE4, UE5, Unity HDRP & URP	RX 460 and above Xbox Series X and S
FSR 2	Temporal upscale of input frames	Quality Balanced Performance Ultra performance	DX12, DX11[2], Vulkan UE4, UE5, Unity URP	RX Vega Series and above Xbox Series X and S[3]
FSR 3	Temporal upscale of input frames with frame generation	Native AA Quality Balanced Performance Ultra performance	DX12, Vulkan,UE5	AMD RDNA Series and above Xbox Series X and S[3]

[1] AMD FidelityFX Super Resolution is "game dependent" and is only supported if the minimum requirements of the game are met.

[2] On demand only

[3] FSR 3 memory requirements may require special considerations on Xbox Series S

SHARPENING

- FSR 3 Upscale comes with its own optional sharpening pass

- It is strongly recommended that the game exposes a sharpening slider
  - This is a common request from players

- If your game already supports a sharpening option in the UI please connect the value to the FSR parameter
  - This is to avoid a clash with the sharpening feature of FSR 3
  - Depending on the existing sharpening range, this may require some trivial remapping

COMPATIBILITY WITH THIRD-PARTY SOFTWARE OR CODE

- FSR 3 requires unencumbered access to the swap chain for best frame pacing results
- Software libraries that intercept DXGI calls may cause frame pacing issues with FSR 3
- Third-party software that intercept DXGI calls to display an on-screen overlay may be incompatible with FSR 3 Frame Generation
  - It is recommended to disable those for best frame pacing FSR 3 results
  - Benchmarking can still be performed without the real-time overlay

COMPATIBILITY WITH DRIVER FEATURES

- AMD Radeon™ Anti-Lag should be disabled in the AMD Software: Adrenalin Edition Application control panel to avoid any issues with frame pacing smoothness or stuttering.
  - This can be done in the specific game profile in the driver settings. We are working to fully enable AMD anti-lag technologies with AMD FSR 3 frame generation, which we will provide more details on in the future.

- It is not recommended to use AMD FSR 3 and FSR 3.1 frame generation in combination with AMD Fluid Motion Frames.
- Frame Pacing smoothness may also be impacted by using other driver-based technologies and the use of third-party graphics overlays.

HARDWARE ACCELERATED GPU SCHEDULING

- For best results with FSR 3 Frame Generation it is recommended that Hardware Accelerated GPU Scheduling be enabled in your Windows OS
- Currently, Hardware Accelerated GPU scheduling is supported on the following AMD GPUs under Windows® 11:
  - Radeon RX 7900XTX
  - Radeon RX 7900XT
  - Radeon RX 7800XT
  - Radeon RX 7700XT

FidelityFX Super Resolution 3

USER INTERFACE GUIDE

FSR UI GUIDE - OVERVIEW

- The FSR 3 UI should be composed of two elements labelled as such:
  - **AMD FSR Upscaling**: a selection between all supported **FSR quality modes**
  - **AMD FSR Frame Generation**: either **On** or **Off**

- Default FSR API usage requires:
  - **AMD FSR** to be set to one of the **FSR quality modes** when **AMD FSR Frame Generation** is on
  - If **AMD FSR** is set to **Off** then **AMD FSR Frame Generation** must be automatically set to **Off** and **grayed out**

- **AMD FSR Frame generation** can be enabled with **any FSR quality modes**
  - Native AA, Quality, Balanced, Performance (and optionally Ultra Performance)

- **AMD FSR Frame generation can also be enabled with third party upscalers.**

- Ensure your game exposes an option to limit fps in order to provide steadier frame inputs to FSR.

- Due to interactions with similar technologies a game restart may be an acceptable solution to enable FSR
  - However the toggling of the Frame Generation feature of FSR should not require a restart

FIDELITYFX API – FSR VERSIONING

- The FidelityFX API has the ability to enumerate supported versions of FSR for each of the Upscaling, Frame Generation or Swapchain contexts.
  - The context created can then be overridden to request the specific version.

- By default, the FFX API DLL will always provide the latest version of FSR able to provide the respective functionality of the context.

- Developers can provide a menu to select what version of Upscale, Frame Generation and Swapchain they would like.

- If AMD drivers provide an updated version – this would appear in the enumerated list automatically.

- UI elements should not force “FSR 3.1” but use the version number provided by enumerating and querying the API.

FSR 3 UI GUIDE – VISUAL EXAMPLE 1

AMD FSR **Upscale** is set to **Quality** with Frame Generation **on**
(1.5x upscaling with frame interpolation enabled)

AMD FSR Upscale	Off	Native AA	**Quality**	Balanced	Performance	Ultra Performance
AMD FSR Frame Generation	Off	**On**				

FSR 3 UI GUIDE – VISUAL EXAMPLE 2

AMD FSR **Upscale** is set to **Performance** with Frame Generation **off**
(2.0x upscaling with Frame Generation disabled)

AMD FSR Upscale	Off	Native AA	Quality	Balanced	**Performance**	Ultra Performance
AMD FSR Frame Generation	**Off**	On				

FSR 3 UI GUIDE – VISUAL EXAMPLE 3

AMD FSR **Upscale** is set to **Native AA** with Frame Generation **on**
(No upscaling with Frame Generation enabled)

AMD FSR Upscale	Off	**Native AA**	Quality	Balanced	Performance	Ultra Performance
AMD FSR Frame Generation	Off	**On**				

FSR 3 UI GUIDE – VISUAL EXAMPLE 4

AMD FSR **Upscale** is set to **Off** with Frame Generation **on**

Third party upscaling is enabled.

(Upscaling via 3rd party method with FSR Frame Generation enabled)

AMD FSR Upscale	**Off**	Native AA	Quality	Balanced	Performance	Ultra Performance
AMD FSR Frame Generation	Off	**On**				
3rd Party Upscale	Off	?	?	?	**Performance**	

FSR 3 UI GUIDE – VISUAL EXAMPLE 5

AMD FSR Upscale and Frame Generation disabled

AMD FSR Upscale	Off	Native AA	Quality	Balanced	Performance	Ultra Performance
AMD FSR Frame Generation	Off	On				

FSR QUALITY MODES

FSR 3 quality mode	Description	Scale factor	Input resolution	Output resolution
Native AA	**Native AA** mode provides an image quality superior to native rendering with a modest performance cost.	1.0x per dimension (1.0x area scale) (100% screen resolution)	1920 x 1080 2560 x 1440 3440 x 1440 3840 x 2160	1920 x 1080 2560 x 1440 3440 x 1440 3840 x 2160
Quality	**Quality** mode provides an image quality equal or superior to native rendering with a significant performance gain.	1.5x per dimension (2.25x area scale) (67% screen resolution)	1280 x 720 1706 x 960 2293 x 960 2560 x 1440	1920 x 1080 2560 x 1440 3440 x 1440 3840 x 2160
Balanced	**Balanced** mode offers an ideal compromise between image quality and performance gains.	1.7x per dimension (2.89x area scale) (59% screen resolution)	1129 x 635 1506 x 847 2024 x 847 2259 x 1270	1920 x 1080 2560 x 1440 3440 x 1440 3840 x 2160
Performance	**Performance** mode provides an image quality similar to native rendering with a major performance gain.	2.0x per dimension (4x area scale) (50% screen resolution)	960 x 540 1280 x 720 1720 x 720 1920 x 1080	1920 x 1080 2560 x 1440 3440 x 1440 3840 x 2160
Ultra Performance*	**Ultra Performance** mode provides the highest performance gain while still maintaining an image quality representative of native rendering.	3.0x per dimension (9x area scale) (33% screen resolution)	640 x 360 854 x 480 1147 x 480 1280 x 720	1920 x 1080 2560 x 1440 3440 x 1440 3840 x 2160

**Optional mode to expose.*

FSR 3 UI REQUIREMENTS – DESCRIPTION OF UI ELEMENTS

- Desired UI description for **AMD FSR**:
  - "AMD FidelityFX Super Resolution combines high-quality image upscaling with frame generation technologies to generate high resolution frames at fast frame rates"
- Desired UI description for **AMD FSR Frame Generation**:
  - "AMD FidelityFX Super Resolution Frame Generation increases frame rate by creating additional frames computed from existing inputs."
- Desired UI descriptions for **FSR quality modes**:
  - Please use the descriptions mentioned on the previous slide.

- If a version must be indicated, display the version provided back via the API when overriding version.

- Localization strings are available on the GPUOpen website at https://gpuopen.com/fidelityfx-naming-guidelines/

SUMMARY

- FSR 3 combines upscaling and frame generation
  - Compatible with VRR monitors
- A successful integration requires a quality upscaling implementation first
- Integrate FSR 3 Frame Generation using the Presentation queue first to ensure correctness
  - Then move to Async queue for best results
- **A Prebuilt FidelityFX API DLL integration is the only supported method with FSR 3.1**
- Use the optimal method for UI composition (Callback) – avoid HUDLess mode
- Follow recommendations for UI requirements
- Localization strings can be found on GPUOpen at https://gpuopen.com/fidelityfx-naming-guidelines/