Intel® XeSS Super Resolution (XeSS-SR) Developer Guide 2.0

Intel® XeSS Super Resolution (XeSS-SR) delivers innovative, framerate-boosting technology, which is supported by Intel® Arc™ Graphics and other GPU vendors. By upscaling the image with AI deep learning, it offers higher framerates at no cost to image quality.

Table of Contents

Introduction

Intel® XeSS Super Resolution (XeSS-SR) is an AI-based temporal super sampling and anti-aliasing technology, implemented as a sequence of compute shader passes, which is executed before the post-processing stage (as described in the section entitled `'TAA and XeSS-SR'`). It generally accepts rendered low resolution aliased jittered color as input, as well as motion vectors and depth from the game, and produces upsampled anti-aliased color buffer at target resolution.

Update from XeSS SDK 1.2 and older versions

XeSS SDK 1.3 introduced new Ultra-Performance (3.0x upscaling), Ultra Quality Plus (1.3x), Native Anti-Aliasing (1.0x upscaling) presets, and **increases resolution scaling for existing quality presets**:

Preset	Resolution scaling in previous XeSS-SR versions	Resolution scaling in XeSS-SR 1.3+
Native Anti-Aliasing	N/A	1.0x (Native resolution)
Ultra Quality Plus	N/A	1.3x
Ultra Quality	1.3x	1.5x
Quality	1.5x	1.7x
Balanced	1.7x	2.0x
Performance	2.0x	2.3x
Ultra Performance	N/A	3.0x
Off	Turns Intel XeSS-SR off	Turns Intel XeSS-SR off

When updating to XeSS-SR 1.3+ from previous XeSS-SR versions, it is critical to verify that XeSS-SR integration in the game is satisfying the following:

- Resolution scaling ratios, additional mip bias and jitter sequence length should not be hardcoded per quality preset in the game code. Corresponding function calls and formulas should be used in the runtime
- xessGetOptimalInputResolution function should be used to query input resolution
- Additional texture mip-bias should be calculated using the formula: $\log_2(\frac{Input\ Width}{Target\ Width})$. For example, XeSS 1.3 Performance quality preset should result in additional mip bias of -1.202, and XeSS 1.3 Ultra Performance quality preset should result in additional mip bias of -1.585.
- Jitter sequence must be at least $8 * \left(\frac{Target\ Width}{Input\ Width}\right)^2$. For example, XeSS 1.3 Ultra Performance quality preset should have jitter sequence length of at least 72.

Please refer to the corresponding sections of this document for more details

XeSS-SR Components

XeSS-SR is accessible through the XeSS-SR SDK, which provides an API for integration into a game engine, and includes the following components:

- An HLSL-based cross-vendor implementation that runs on any GPU supporting SM 6.4. Hardware acceleration for DP4a or equivalent is recommended.
- An Intel implementation optimized to run on Intel® Arc™ Graphics, and Intel® Iris® Xe Graphics.

- An implementation dispatcher, which loads either the XeSS-SR runtime shipped with the game, the version provided with the Intel® Graphics driver, or the cross-vendor implementation.

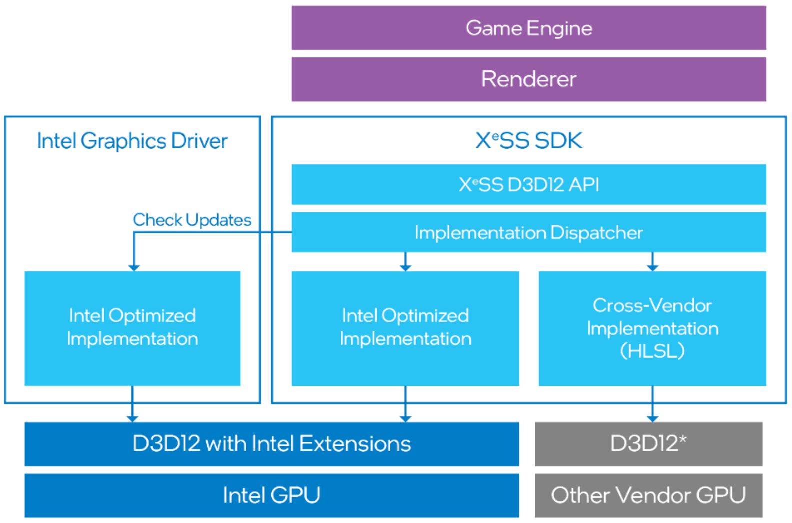
Game Engine
Renderer
Intel Graphics Driver
XeSS SDK
XeSS D3D12 API
Check Updates
Implementation Dispatcher
Intel Optimized Implementation
Intel Optimized Implementation
Cross-Vendor Implementation (HLSL)
D3D12 with Intel Extensions
D3D12*
Intel GPU
Other Vendor GPU

Figure 1. XeSS-SR SDK D3D12 components for both Intel-specific, and cross-vendor solutions.

Compatibility

All future Intel® Graphics driver releases provide compatibility with previous XeSS-SR versions.

During initialization XeSS-SR will check driver compatibility and choose which version to use – version distributed with an application or version bundled with the driver.

Requirements

- Windows 10/11 x64 - 10.0.19043/22000 or later
- DirectX12
  - Intel® Iris® Xe GPU or later
  - Other vendor GPU supporting SM 6.4 and with hardware acceleration for DP4a
- DirectX11
  - Intel® Arc™ Graphics or later
- Vulkan 1.1
  - Intel® Iris® Xe GPU or later
  - use SDK functions to query for required instance and device extensions and device features
  - Other vendor GPU supporting shaderIntegerDotProduct, shaderStorageReadWithoutFormat and mutableDescriptorType features

TAA and XeSS-SR

XeSS-SR is a temporally amortized super-sampling/up-sampling technique that replaces the Temporal Anti-Aliasing (TAA) stage in the game renderer, achieving significantly better image quality than current state-of-the-art techniques in games.

Figure 2 shows a renderer with TAA. The renderer jitters the camera in every frame to sample different coordinates in screen space. The TAA stage accumulates these samples temporally to produce a super-sampled image. The previously accumulated frame (history) is warped using renderer-generated motion vectors to align it with the current frame before accumulation. Unfortunately, the warped sample history can be mismatched, with respect to the current pixel, due to frame-to-frame changes in visibility and shading or errors in the motion vector. This typically results in ghosting artifacts. TAA implementations use heuristics such as neighborhood clamping to detect mismatches and reject the history. However, these heuristics often fail, and produce a noticeable amount of ghosting, over-blurring, or flickering.

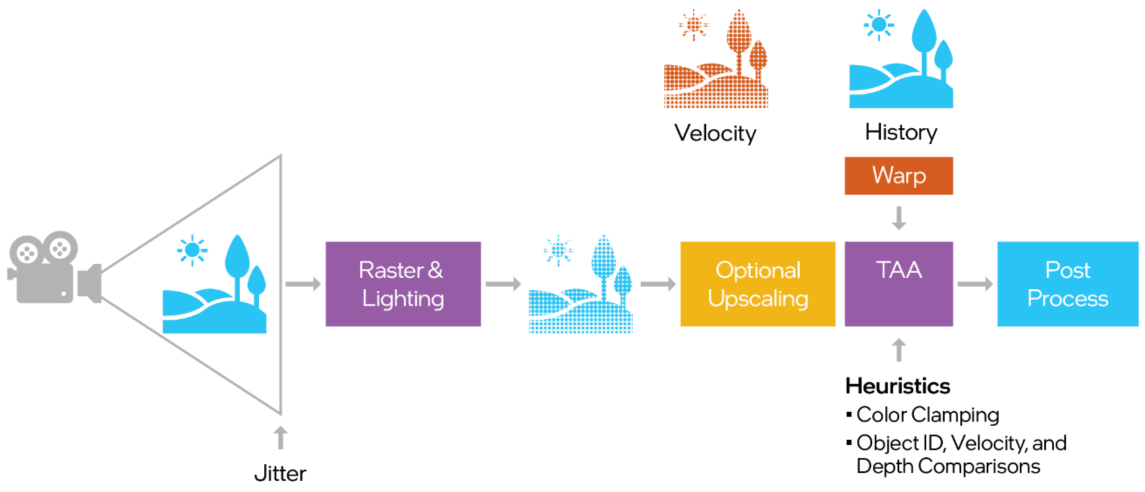

Figure 2. Flow chart of a typical rendering pipeline with TAA.

XeSS-SR replaces the TAA stage with a neural-network-based approach, as shown below, with the same set of inputs and outputs as TAA. Please refer to this report for an overview of TAA techniques.

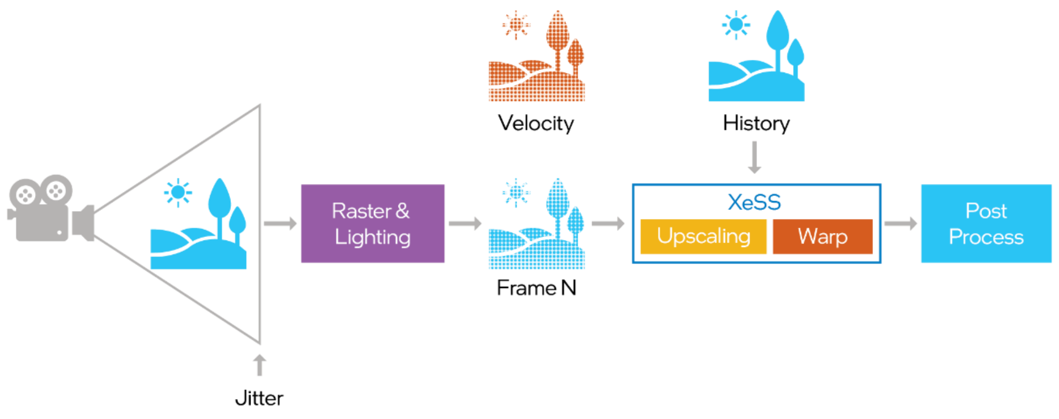

Figure 3. XeSS-SR inclusion into the rendering pipeline.

Game Setting Recommendations

When integrating XeSS-SR into your game, make sure you follow these guidelines for your titles so that your users have a consistent experience when modifying XeSS-SR options.

Naming conventions and branding guidance

Please see "XeSS 2 Naming Structure and Examples.pdf" for approved naming conventions, branding guidance and settings menu examples.

The official font for XeSS-SR-related communication is IntelOneText-Regular.

Label	Intel XeSS-SR
Short Description	Intel® XeSS Super Resolution (XeSS-SR) technology uses AI to deliver more performance with exceptional image quality. XeSS is optimized for Intel® Arc™ GPUs with AI acceleration hardware.
Minimum Description	Intel® XeSS Super Resolution (XeSS-SR) technology uses AI to deliver more performance with exceptional image quality.

Game Graphics Settings Menu/Game Installer/Launcher Settings

Game-title graphics settings should clearly display the XeSS-SR option name and allow the user to choose the quality/performance level option settings, as follows.

Preset	Description	Recommended Resolution
Native Anti-Aliasing	AI-based Anti-Aliasing for maximum visual quality	1080p and above
Ultra Quality Plus	Highest quality visual upscale	1080p and above
Ultra Quality	Higher quality visual upscale	1080p and above
Quality	High quality visual upscale	1080p and above
Balanced	Best balance between performance and visual quality	1080p and above
Performance	High performance improvement	1440p and above
Ultra Performance	Highest performance improvement	1440p and above
Off	Turns Intel XeSS-SR off	NA

Note: To enable XeSS-SR, your title needs to disable other upscaling technologies, such as NVIDIA DLSS* and AMD FSR*, and temporal anti-aliasing (TAA) technologies, to reduce the possibility of any incompatibility issues.

Graphics Preset Default Recommendations

The XeSS-SR preset selected by default in the game's menu should be based on the target resolution that the user has set. The entries below are the recommended default settings.

Default XeSS -SR recommendations	Description	Recommended Setting
Resolution specific	Your game adjusts the XeSS-SR default preset based on the output resolution	1080p and lower set to 'Balanced' 1440p and higher set to 'Performance'
General	Your game selects one XeSS-SR preset as default.	Intel XeSS-SR ON set to 'Performance'

Note: All Intel XeSS Super Resolution settings should be exposed to the user through a selection menu, if supported, to encourage customization.

Deployment

To use XeSS-SR in a project:

- Add "inc" folder to the include path
- Include xess.h and one of the API specific headers: xess_d3d12.h, xess_d3d11.h or xess_vk.h
- For D3D12 and Vulkan: Link with lib/libxess.lib
- For D3D11: Link with lib/libxess_dx11.lib

The following file must be placed next to the executable or in the DLL search path:

- For D3D12 and Vulkan: libxess.dll
- For D3D11: libxess_dx11.dll
- Microsoft Visual C++ Redistributable version 14.40.33810 or later. The following libraries are required:
    - msvcp140.dll
    - vcruntime140.dll
    - vcruntime140_1.dll

Note the deployment uses a separate library for the D3D11 backend, different from library used with the D3D12 and Vulkan backends. Care must be taken to link and call functions from the proper library. Especially common functions from xess.h header, like xessDestroyContext, are exposed by both libraries, but it is not supported to use function from libxess.dll library for XeSS-SR contexts that use D3D11. Similarly, it is not supported to call functions from libxess_dx11.dll library for XeSS-SR contexts that use D3D12 or Vulkan.

Programming Guide

Thread safety

XeSS-SR is not thread safe. The client application must ensure that calls to XeSS-SR functions are performed safely. In general, all calls to the XeSS-SR API must be done from the same thread where XeSS-SR was initialized.

Inputs and Outputs

XeSS-SR requires a minimum set of inputs every frame:

- Jitter
- Input color
- Dilated high-res motion vectors

In place of the high-res motion vectors, the renderer can provide the motion vectors at the input resolution—along with the depth values:

- Undilated low-res motion vectors
- Depth

In the latter case, motion vectors will be dilated and upsampled inside XeSS-SR.

XeSS-SR produces a single output into a texture provided by the application.

Jitter

As a temporal super-sampling technique, XeSS-SR requires a sub-pixel jitter offset *(J_x,J_y)* to be applied to the projection matrix in every frame. This process produces a new subpixel sample location every frame and guarantees temporal convergence even on static scenes. Jitter offset values should be in the range *[-0.5,0.5]*. This jitter can be applied by adding a shear transform to the camera projection matrix:

```
ProjectionMatrix[2][0] += Jx * 2.0f / InputWidth
ProjectionMatrix[2][1] -= Jy * 2.0f / InputHeight
```

The jitter applied to the camera results in a displacement of the sample points in the frame, as shown below, where the target image is scaled 2x in width and height. Note that effective jitter is negated with respect to *(Jx, Jy)*, because the projection matrix is applied to geometry, and it corresponds to a negative camera jitter.

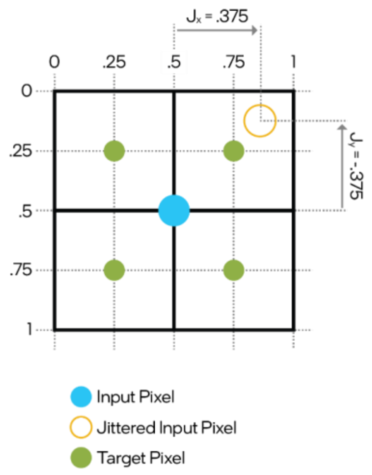

Figure 4. Jitter displacement of sample points.

Jitter Sequence

A quasi-random sampling sequence with a good spatial distribution of characteristics is required to get the best quality from the XeSS-SR algorithm (Halton sequence would be a fair choice). The scaling factor should be considered when using such a sequence to modify the length of a repeated pattern. For example: if the game is using Halton sequence of a length eight in native rendering, it must become $8 * \left(\frac{Target\ Width}{Input\ Width}\right)^2$ if used with XeSS-SR upscaling to ensure a good distribution of samples in the area covered by a single low-resolution pixel. Sometimes, increasing the length even more leads to an additional quality improvement, so we encourage experimentation with the sequence length. Avoid sampling techniques that bias the jitter sample distribution regarding the input pixel, however.

Color

XeSS-SR accepts both low and high dynamic range (LDR and HDR) input colors in any linear color format, for example:

- R16G16B16A16_FLOAT
- R11G11B10_FLOAT
- R8G8B8A8_UNORM

Only UNORM integer formats can be used as color input, other integer formats are not supported.

The application should inform XeSS-SR if LDR input color is provided by setting the input flag XESS_INIT_FLAG_LDR_INPUT_COLOR.

The input colors are expected to be in the scRGB color space, which is scene-referred—i.e., the color values represent luminance levels. A value of (1.0,1.0,1.0) encodes D65 white at 80 nits and represents the maximum luminance for SDR displays. The color values can exceed (1.0,1.0,1.0) for HDR content.

It is recommended to provide HDR input color to XeSS-SR, since it can internally apply tonemapping tuned for XeSS-SR AI Models optimizing visual quality. If LDR input color is provided, XeSS-SR quality may decrease.

If HDR input color values have not been adjusted for the exposure, or if they are scaled differently from the sRGB space, a separate scale value can be provided in the following ways:

- If no exposure value is available, XeSS-SR can calculate it when the XESS_INIT_FLAG_ENABLE_AUTOEXPOSURE flag is used at initialization. Note that the use of this flag will cause a measurable performance impact.
- Input exposure value can be provided by passing value inside xess_*_execute_params_t structure or by providing an exposure scale texture.

If the application provides LDR input color to XeSS-SR, it is recommended to set the exposure value to 1 and avoid using auto-exposure.

If the input color from the application is pre-exposed, it is recommended to provide pre-exposure in one of following ways:

- Provide exposure value divided by pre-exposure.
- Provide inverse of pre-exposure value as input to xessSetExposureMultiplier.

Exposure multiplier will be applied to all exposure values including one generated by built-in auto-exposure functionality.

Exposure values are applied to input color in following way:

```
if (useAutoexposure)
{
 scale = XeSSCalculatedExposure(…)
}
else if (useExposureScaleTexture)
{
 scale = exposureScaleTexture.Load(int3(0, 0, 0)).x
}
else
{
 scale = inputScale
}

inputColor *= scale * exposure_multiplier
```

If a scale value is applied to the input, as shown above, the inverse of this scale is applied to the output color.

XeSS-SR maintains an internal history state to perform temporal accumulation of incoming samples. That means the history should be dropped if the scene or view suddenly changes. This is achieved by setting the resetHistory flag in xess_xxx_execute_params_t.

The output must be in the same color space as the input. It can be any linear color format similar to the input.

Motion Vectors

Motion vectors specify the screen-space motion in pixels from the current frame to the previous frame. If the application uses normalized device coordinates (NDC) for motion vectors it should pass an additional flag (XESS_INIT_FLAG_USE_NDC_VELOCITY) during XeSS-SR context initialization. XeSS-SR accepts motion vectors in the format R16G16_FLOAT, where the R channel encodes the motion in x, and the G in y. The motion vectors do not include motion induced by the camera jitter. Motion vectors can be low-res (default), or high-res (XESS_INIT_FLAG_HIGH_RES_MV). Low-res motion vectors are represented by a 2D texture at the input resolution, whereas high-res motion vectors are represented by a 2D texture at the target resolution.

In the case of high-res motion vectors, the velocity component resulting from camera animations is computed at the target resolution in a deferred pass, using the camera transformation and depth values. However, the velocity component related to particles and object animations is typically computed at the input resolution and stored in the G-Buffer. This velocity component is upsampled and combined with the camera velocity to produce the texture for high-res motion vectors. XeSS-SR also expects the high-res motion vectors to be dilated. For example, the motion vectors represent the motion of the foremost surface in a small neighborhood of input pixels (such as 3 * 3). High-res motion vectors can be computed in a separate pass by the user.

Low-res motion vectors are not dilated, and directly represent the velocity sampled at each jittered pixel position. XeSS-SR internally up-samples motion vectors to the target grid and uses the depth texture to dilate them. The figure 5 shows the same motion specified with low-res and high-res motion vectors.

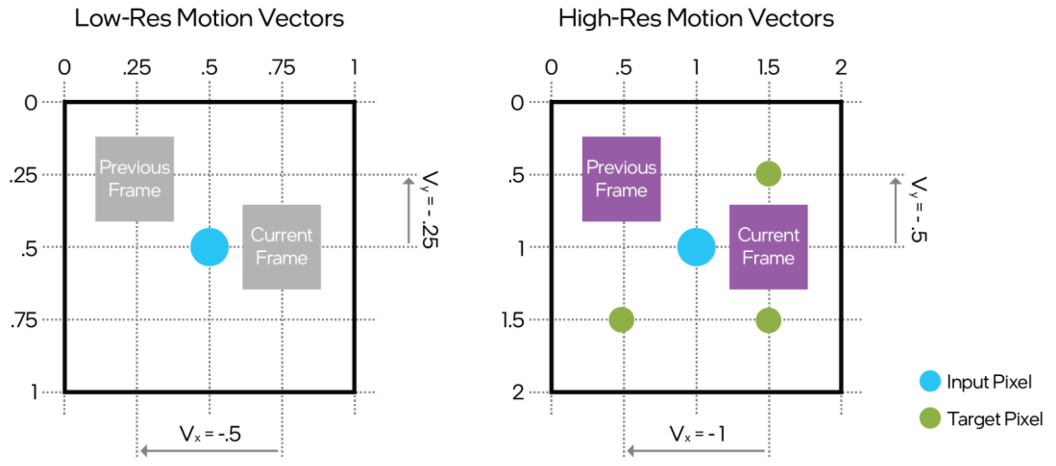

Figure 5. Convention for specifying low-res and high-res motion vectors to XeSS-SR.

Some game engines only render objects into the gbuffer, and quickly compute the camera velocity in the TAA shader. In such cases, an additional pass is required before XeSS-SR execution to merge object and camera velocities and generate a flattened velocity buffer. In such scenarios, high-res motion vectors might be a better choice, as the flattening pass can be executed at the target resolution.

Depth

If XeSS-SR is used with low-res motion vectors, it also requires a depth texture for velocity dilation. Any depth format, such as D32_FLOAT or D24_UNORM, is supported. By default, XeSS-SR assumes that smaller depth values are closer to the camera. However, several game engines use inverted depth, and this can be enabled by setting XESS_INIT_FLAG_INVERTED_DEPTH.

Responsive Pixel Mask

Although XeSS-SR is a generalized technique that should handle a wide range of rendering scenarios, there may be rare cases where objects without valid motion vectors may produce ghosting artifacts, for example particles. In such cases, a responsive pixel mask texture can be provided, masking these objects. Responsive mask allows explicit control over how much history affects masked pixels.

The mask should contain float values in the range [0.0, 1.0]. Mask pixel value 1.0 indicates that accumulated history should be fully ignored for the corresponding color pixel, favoring the current frame. Mask pixel value 0.0 indicates no modification to XeSS-SR AI Model's decision for the corresponding pixel, equivalent to the case when no responsive pixel mask is provided. By default, the responsive mask values are clamped at 0.8 internally by XeSS-SR when applying the values: it's a reasonable value to reduce unexpected aliasing and various flicker, that is common for objects masked with highest "responsiveness" values close to 1.0. Default clamping value should only be overridden by the application after validating that higher values provide better result. Clamping value can be changed with the xessSetMaxResponsiveMaskValue function.

The application can provide a responsive pixel mask as a texture in any format where the R channel can be read as a float value. Responsive pixel masks should be in input resolution and shouldn't contain any jitter.

The application should only use the responsive pixel mask feature when it clearly improves visual quality.

Output

Requirements for the output texture depend on the color input format. The output must be in the same format and color space as the input. It must be big enough to contain the output resolution sized (scaled up) image.

If the output texture is bigger than output resolution, the output image will be placed at the top-left corner of the output texture. The application can provide offsets in xess*Execute to change the output image location.

XeSS doesn't preserve the alpha channel of the output texture - it will be filled with 1.0.

Resource States

XeSS-SR expects all input textures to be in the following states:

- D3D12
  - D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE for input textures.
  - D3D12_RESOURCE_STATE_UNORDERED_ACCESS for the output texture.
- Vulkan
  - VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL for input textures
  - VK_IMAGE_LAYOUT_GENERAL for the output texture

In Vulkan, XeSS-SR accesses resources in VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT stage, using VK_ACCESS_SHADER_READ_BIT access for input textures and VK_ACCESS_SHADER_READ_BIT | VK_ACCESS_SHADER_WRITE_BIT for the output texture.

In D3D12 and Vulkan, XeSS-SR does not perform any memory synchronization on input and output resources. It is application's responsibility to provide resources in specified layout.

Resource Formats

XeSS-SR expects all input textures to be typed.

D3D12 and D3D11 only

XeSS-SR can accept some typeless formats for input textures. In case of typeless formats they will be interpreted internally according to following table:

Input typeless format	Internal typed interpretation
DXGI_FORMAT_R32G32B32A32_TYPELESS	DXGI_FORMAT_R32G32B32A32_FLOAT
DXGI_FORMAT_R32G32B32_TYPELESS	DXGI_FORMAT_R32G32B32_FLOAT
DXGI_FORMAT_R16G16B16A16_TYPELESS	DXGI_FORMAT_R16G16B16A16_FLOAT
DXGI_FORMAT_R32G32_TYPELESS	DXGI_FORMAT_R32G32_FLOAT
DXGI_FORMAT_R32G8X24_TYPELESS	DXGI_FORMAT_R32_FLOAT_X8X24_TYPELESS
DXGI_FORMAT_R10G10B10A2_TYPELESS	DXGI_FORMAT_R10G10B10A2_UNORM
DXGI_FORMAT_R8G8B8A8_TYPELESS	DXGI_FORMAT_R8G8B8A8_UNORM
DXGI_FORMAT_R16G16_TYPELESS	DXGI_FORMAT_R16G16_FLOAT
DXGI_FORMAT_R32_TYPELESS	DXGI_FORMAT_R32_FLOAT
DXGI_FORMAT_R24G8_TYPELESS	DXGI_FORMAT_R24_UNORM_X8_TYPELESS
DXGI_FORMAT_R8G8_TYPELESS	DXGI_FORMAT_R8G8_UNORM
DXGI_FORMAT_R16_TYPELESS	DXGI_FORMAT_R16_FLOAT
DXGI_FORMAT_R8_TYPELESS	DXGI_FORMAT_R8_UNORM
DXGI_FORMAT_B8G8R8A8_TYPELESS	DXGI_FORMAT_B8G8R8A8_UNORM
DXGI_FORMAT_B8G8R8X8_TYPELESS	DXGI_FORMAT_B8G8R8X8_UNORM
DXGI_FORMAT_D16_UNORM	DXGI_FORMAT_R16_UNORM
DXGI_FORMAT_D32_FLOAT	DXGI_FORMAT_R32_FLOAT
DXGI_FORMAT_D24_UNORM_S8_UINT	DXGI_FORMAT_R24_UNORM_X8_TYPELESS
DXGI_FORMAT_D32_FLOAT_S8X24_UINT	DXGI_FORMAT_R32_FLOAT_X8X24_TYPELESS

Mip Bias

To preserve texture details at the target resolution, XeSS-SR requires an additional mip bias. It is recommended to use following formula:

$$\log_2\left(\frac{Input\ Width}{Target\ Width}\right)$$

For example, a mip bias of -1 should be applied for 2.0x resolution scaling that corresponds to the Balanced quality preset. In certain cases, increasing the mip bias even more leads to an additional visual quality improvement; however this comes with a potential performance overhead, due to increased memory bandwidth requirements, and potentially lower temporal stability resulting in flickering and moire. It’s encouraged to experiment with different texture LOD biases to find the optimal value, but it’s recommended to start with a value calculated based on input and output resolutions as suggested above.

In dynamic resolution scenario, the mip bias is recommended to be updated every time the XeSS-SR input resolution changes. If this is not possible, consider keeping a set of estimated mip biases.

Initialization

D3D12 specific

First create an XeSS-SR context, as shown below. On Intel GPUs, this step loads the latest Intel-optimized implementation of XeSS-SR. The returned context handle can then be used for initialization and execution.

```
xess_context_handle_t context;
xessD3D12CreateContext(pD3D12Device, &context)
```

Before initializing XeSS-SR, the application can request a pipeline pre-build process to avoid costly kernel compilation and pipeline creation during initialization.

```
xessD3D12BuildPipelines(context, NULL, false, initFlags);
```

The application can check pipeline build status by calling xessGetPipelineBuildStatus.

The xessD3D12Init function is then called to initialize XeSS-SR. During initialization, XeSS-SR can create staging buffers and copy queues to upload weights. These will be destroyed at the end of initialization. The XeSS-SR storage and layer specializations are determined by the target resolution. Therefore, the target width and height must be set during initialization.

```
xess_d3d12_init_params_t initParams;
initParams.outputWidth = 3840;
initParams.outputHeight = 2160;
initParams.initFlags = XESS_INIT_FLAG_HIGH_RES_MV;
initParams.pTempStorageHeap = NULL;

xessD3D12Init(&context, &initParams);
```

XeSS-SR uses three types of storage:

- **Persistent Output-Independent Storage:** persistent storage such as network weights are internally allocated and uploaded by XeSS-SR during initialization.
- **Persistent Output-Dependent Storage:** persistent storage such as internal history texture.
- **Temporary Storage:** temporary storage such as network activations, only has valid data during the execution of XeSS-SR.

Persistent storage is always owned by XeSS-SR. The application can optionally provide resource heaps for temporary storage via the pTempTextureHeap and pTempBufferHeap fields of the xess_d3d12_init_params_t structure. Allocated resource heaps must use the default heap type (D3D12_HEAP_TYPE_DEFAULT). And to ensure optimal performance this heap should have HIGH memory residency priority. These heaps can be reused after XeSS-SR execution. Required sizes for these heaps can be obtained by calling xessGetProperties.

```
// Get required heap sizes
xess2d_t output_res {width, height};
xess_properties_t props;
xessGetProperties(context, &output_res, &props);

// Create buffer heap
ComPtr<ID3D12Heap> pBufferHeap;
CD3DX12_HEAP_DESC bufferHeapDesc(props.tempBufferHeapSize,
 D3D12_HEAP_TYPE_DEFAULT);
d3dDevice->CreateHeap(&bufferHeapDesc , IID_PPV_ARGS(&pBufferHeap));

// Create texture heap
ComPtr<ID3D12Heap> pTextureHeap;
CD3DX12_HEAP_DESC textureHeapDesc(props.tempTextureHeapSize,
 D3D12_HEAP_TYPE_DEFAULT);
d3dDevice->CreateHeap(&textureHeapDesc, IID_PPV_ARGS(&pTextureHeap));

// Pass heaps to XeSS
initParams.pTempBufferHeap = pBufferHeap.Get();
initParams.bufferHeapOffset = 0;
initParams.pTempTextureHeap= pTextureHeap.Get();
initParams.textureHeapOffset = 0;

xessD3D12Init(&context, &initParams);
```

For D3D12 the application can optionally provide an external descriptor heap, where XeSS-SR creates all its internal resource descriptors. The amount of required descriptors can be obtained by calling xessGetProperties. The application has to specify the XESS_INIT_FLAG_EXTERNAL_DESCRIPTOR_HEAP flag on initialization and pass the descriptor heap and an offset during execution. The descriptor heap must be of type D3D12_SRV_UAV_CBV_DESCRIPTOR_HEAP and shader visible.

The application can re-initialize XeSS-SR if there is a change in the target resolution, or any other initialization parameter. However, pending XeSS-SR command lists must be completed before re-initialization. When temporary XeSS-SR storage is allocated, it’s the application’sresponsibility to deallocate, or reallocate the heap. Quality preset changes are cheap, but any other parameters change may lead to longer xessD3D12Init execution times.

D3D11 specific

Currently, on D3D11 the XeSS-SR support is limited to Intel® Arc™ Graphics or later . Only an Intel-optimized implementation of XeSS-SR is available.

Creating a context on not-supported devices, especially non-Intel devices or Intel® Iris® Xe will fail with XESS_RESULT_ERROR_UNSUPPORTED_DEVICE.

To use XeSS-SR on D3D11, a context must be created as shown below.

```
xess_context_handle_t context;
xessD3D11CreateContext(pD3D11Device, &context);
```

The xessD3D11Init function is then called to initialize XeSS-SR. During initialization, XeSS-SR can create staging buffers and copy queues to upload weights. These will be destroyed at the end of initialization. The XeSS-SR storage and layer specializations are determined by the target resolution. Therefore, the target width and height must be set during initialization.

```
xess_d3d11_init_params_t initParams;
initParams.outputWidth = 3840;
initParams.outputHeight = 2160;
initParams.initFlags = XESS_INIT_FLAG_HIGH_RES_MV;
xessD3D11Init (&context, &initParams);
```

You can also re-initialize XeSS-SR if there is a change in the target resolution, or initialization flag. However, pending XeSS-SR executions must be completed before re-initialization. Quality preset changes are cheap, but any other parameters change may lead to longer xessD3D11Init execution times.

Unlike D3D12, in D3D11 the temporary storage for data needed for execution of XeSS-SR is managed by the library and temporary storage heaps cannot be provided by the user during initialization.

The XESS_INIT_FLAG_EXTERNAL_DESCRIPTOR_HEAP flag is not supported by the D3D11 path.

Pipeline build functions, like xess*BuildPipelines or xessGetPipelineBuildStatus are not available in D3D11.

Vulkan specific

To use XeSS-SR, Vulkan 1.1 must be supported. Additionally, XeSS-SR requires Vulkan extensions and features, that should be queried as shown below. For baseline operation XeSS-SR currently requires:

- `shaderStorageReadWithoutFormat` feature.
- `mutableDescriptorType` feature from `VK_EXT_mutable_descriptor_type` extension.
- `shaderIntegerDotProduct` feature.

XeSS-SR may request additional features and extensions for improved performance at runtime, for example `shaderInt8`. Therefore, XeSS-SR exposes a set of functions for querying needed extensions and features.

Before creating a VkInstance object, the application should query XeSS-SR for needed instance extensions and Vulkan API version.

```
uint32_t instance_extension_count;
const char* const* instance_extensions;
uint32_t api_version;
xessVKGetRequiredInstanceExtensions(&instance_extension_count,
 &instance_extensions, &api_version);
```

If the count of returned extensions is not 0, returned extensions must be enabled for the VkInstance object that wwill be passed to xessVKCreateContext. When creating the VkInstance, the VkApplicationInfo.apiVersion field must be equal or greater than the returned value .

It' is guaranteed that returned extensions and API version are supported on given platform. If required extensions are not supported, the function fails with XESS_RESULT_ERROR_UNSUPPORTED_DRIVER.

The memory backing for the returned extension list is owned by XeSS-SR and must not be freed by the application.

After creating the VkInstance object, before creating the VkDevice object, the application should query XeSS-SR for required device extensions and device features.

```
uint32_t device_extension_count;
const char* const* device_extensions;
xessVKGetRequiredDeviceExtensions(instance, physical_device,
 &device_extension_count, &device_extensions);


VkPhysicalDeviceFeatures2 physical_device_features2 = {...};
void* features = &physical_device_features2; // or NULL
xessVKGetRequiredDeviceFeatures(instance, physical_device, &features);
```

If the count of returned extensions is not 0, returned extensions must be enabled for the VkDevice object that will be passed to xessVKCreateContext.

The function returns required features by adding them to the chain of given VkPhysicalDevice...Features structures. The chain should be formed by connecting the structures by pNext field. ThexessVKGetRequiredDeviceFeatures function returns required features by writing field values to existing structures and chaining new structures, if needed.

Alternatively, this function can be called with features equal to NULL, in which case it would return new feature structure chain that the application should merge into it's own structure chain passed to vkCreateDevice. The application must take care of possible duplicated structure types if XeSS-SR and application would want to chain same structure type.

It's an error to pass an VkDeviceCreateInfo structure with non-null pEnabledFeatures field, as this field is const and cannot be modified by XeSS-SR. The VkPhysicalDeviceFeatures2 structure should be chained via pNext field instead of using pEnabledFeatures field.

Returned features must be chained to the VkDeviceCreateInfo structure passed to the vkCreateDevice call when creating the VkDevice object which will be passed to the xessVKCreateContext function.

It's guaranteed that returned extensions and features are supported on given device. If required extensions or features are not supported, the function fails with XESS_RESULT_ERROR_UNSUPPORTED_DRIVER.

The memory backing for the returned extension list and feature structures added to the chain is owned by XeSS-SR and must not be freed by the application.

To use XeSS-SR, a context must be created as shown below.

```
xess_context_handle_t context;
xessVKCreateContext (vulkan_instance, physical_device, device,
  enabled_features2, enabled_extensions_count, enabled_extensions,
  &context)
```

Before initializing XeSS-SR, the application can request a pipeline pre-build process to avoid costly kernel compilation and pipeline creation during initialization.

```
xessVKBuildPipelines(context, pipeline_cache, false, initFlags);
```

The application can check pipeline build status by calling xessGetPipelineBuildStatus.

The xessVKInit function is then called to initialize XeSS-SR. During initialization, XeSS-SR can create staging buffers and copy queues to upload weights. These will be destroyed at the end of initialization. The XeSS-SR storage and layer specializations are determined by the target resolution. Therefore, the target width and height must be set during initialization.

```
xess_d3d12_init_params_t initParams;
initParams.outputWidth = 3840;
initParams.outputHeight = 2160;
initParams.initFlags = XESS_INIT_FLAG_HIGH_RES_MV;
initParams.tempBufferHeap = VK_NULL_HANDLE;
initParams.tempTextureHeap = VK_NULL_HANDLE;
xessVKInit (&context, &initParams);
```

XeSS-SR includes three types of storage:

- **Persistent Output-Independent Storage:** persistent storage such as network weights are internally allocated and uploaded by XeSS-SR during initialization.
- **Persistent Output-Dependent Storage:** persistent storage such as internal history texture.
- **Temporary Storage:** temporary storage such as network activations, only has valid data during the execution of XeSS-SR.

Persistent storage is always owned by XeSS-SR. The application can optionally provide resource heaps for temporary storage via the pTempTextureHeap and pTempBufferHeap fields of the xess_vk_init_params_t structure. Allocated resource heaps should be allocated form device local memory. These heaps can be reused after XeSS-SR execution. Required sizes for these heaps can be obtained by calling xessGetProperties.

The application can also re-initialize XeSS-SR if there is a change in the target resolution, or any other initialization parameter. However, pending XeSS-SR command lists must be completed before re-initialization. When temporary XeSS-SR storage is allocated, it'sthe application's responsibility to de-allocate, or reallocate, the heap. Quality preset changes are cheap, but any other parameters change may lead to longer xessVKInit execution times.

One difference from the D3D12 path is that XESS_INIT_FLAG_EXTERNAL_DESCRIPTOR_HEAP flag is not supported by the Vulkan path.

Logging Callback

The application can obtain additional diagnostic information by registering a logging callback using xessSetLoggingCallback. The requirements for the callback:

- Callback can be called from different threads.
- Callback can be called simultaneously from several threads.
- Message pointer only valid inside function and may be invalid right after return call.
- Message is a null-terminated utf-8 string.

There are four levels of logging:

- XELL_LOGGING_LEVEL_DEBUG,
- XELL_LOGGING_LEVEL_INFO,
- XELL_LOGGING_LEVEL_WARNING,
- XELL_LOGGING_LEVEL_ERROR.

We recommend using error level for production and warning or debug level for development.

Execution

In D3D12 and Vulkan the XeSS-SR execution functions do not involve any GPU workloads, rather they record XeSS-SR commands into the specified command list. The command list is then submitted by the user. That means it'sthe application's responsibility to make sure all input/output resources are alive at the time of the actual GPU execution.

D3D12 specific

By default, XeSS-SR creates an internal descriptor heap, but if you have specified XESS_INIT_FLAG_EXTERNAL_DESCRIPTOR_HEAP at the initialization stage, you can pass the pointer to the external descriptor heap and its offset as execution parameters. This flag is supported only for the D3D12 path.

If the EXTERNAL_DESCRIPTOR_HEAP flag has been specified in xessD3D12Init parameters, you must create descriptors for the input and output buffers in contiguous locations in the same descriptor heap as the internal descriptors. The external descriptor heap is passed via the pDescriptorHeap field of the xess_d3d12_execute_params_t structure. DescriptorHeapOffset should point to the XeSS-SR descriptor table.

D3D11 specific

In D3D11, the XeSS-SR execution function prepares the workload for the GPU, which is then enqueued alongside the user D3D11 commands and eventually executed. Accesses to input/output resources are implicitly synchronized with user D3D11 commands.

Vulkan specific

For Vulkan integrations application must pass additional information together with resource pointers. Structure xess_vk_execute_params uses additional structure to hold resource descriptions:

```
typedef struct _xess_vk_image_view_info
{
    VkImageView imageView;
    VkImage image;
    VkImageSubresourceRange subresourceRange;
    VkFormat format;
    unsigned int width;
    unsigned int height;
} xess_vk_image_view_info;
```

All fields of this structure are important since Vulkan doesn't provide any way to retrieve that data from VkImage/VkImageView.

The subresourceRange and view parameters should point to the image subresource (level and layer) to be used by XeSS-SR. Only 2D view types are supported. For depth images and depth stencil images, subresource and view should point only to the VK_IMAGE_ASPECT_DEPTH_BIT aspect of the image.

Fixed input resolution

Default scenario for using XeSS-SR is using fixed input resolution that depends on the desired quality setting and target resolution. Call xessGetOptimalInputResolution and use pInputResolutionOptimal value to determine input resolution based on quality setting and target resolution. You must provide actual input resolution values as a part of the xess_*_execute_params_t structure in each call to `xess*Execute`.

Please note, that `xessGetInputResolution` function is deprecated starting XeSS 1.2 and kept for compatibility reasons.

Please note that starting with XeSS 1.3, Quality Preset mapping to resolution scaling has changed. Please refer to the table below for XeSS 1.3:

Preset	Resolution scaling
Native Anti-Aliasing	1.0x (Native resolution)
Ultra Quality Plus	1.3x
Ultra Quality	1.5x
Quality	1.7x
Balanced	2.0x
Performance	2.3x
Ultra Performance	3.0x
Off	Turns Intel XeSS-SR off

XeSS-SR provides a way to restore legacy Quality Preset mapping to resolution scaling. It can be done by using `xessForceLegacyScaleFactors`. Calling this function will result in the following scale factors used in the next call to `xessGetOptimalInputResolution` and `xessD3D12Init`:

Preset	Resolution scaling (Legacy)
Native Anti-Aliasing	1.0x (Native resolution)
Ultra Quality Plus	1.3x
Ultra Quality	1.3x
Quality	1.5x
Balanced	1.7x
Performance	2.0x
Ultra Performance	3.0x
Off	Turns Intel XeSS-SR off

In order to use legacy scale factors, the application should initialize XeSS-SR in the order demonstrated by the following code sample:

```
// 1. Create context
xess*CreateContext(device, &context);
```

```
// 2. Request legacy scale factors
xessForceLegacyScaleFactors(context, true);

// 3. Get optimal input resolution and initialize context
xessGetOptimalInputResolution(context, …);
xess*Init(context, …);
```

Dynamic input resolution

XeSS-SR supports a scenario with dynamic input resolution and fixed target resolution. In this case, it's recommended to ignore pInputResolutionOptimal value and freely change input resolution within the supported range [pInputResolutionMin; pInputResolutionMax], letting the applicationreach optimal visual quality level while keeping fixed FPS. You must provide actual input resolution values as a part of the xess_*_execute_params_t structure in each call to `xess*Execute`. When varying input resolution, it is recommended for the application to keep the rendering resolution aspect ratio as close as possible to the target resolution aspect ratio, to ensure optimal and stable visual quality of XeSS-SR.

Example for D3D12 path, but it would be similar for Vulkan:

```
xess_d3d12_execute_params_t params;
params.jitterOffsetX    = 0.4375f;
params.jitterOffsetY    = 0.3579f;

params.inputWidth = 1920;
params.inputHeight = 1080;

// xess records commands into the command list
xessD3D12Execute(&context, pd3dCommandList, &params);

// Application may record more commands as needed
pD3D12GraphicsCommandList->Close();

// Application submits the command list for GPU execution
pCommandQueue->ExecuteCommandLists(1, &pCommandLists);
```

Please note that starting with XeSS 1.3, Quality Preset mapping to resolution scaling has changed. Please refer to the table below for XeSS 1.3:

Preset	Dynamic resolution scaling range
Native Anti-Aliasing	N/A
Ultra Quality	1.0x - 1.5x
Quality	1.0x - 1.7x
Balanced	1.0x - 2.0x
Performance	1.0x - 2.3x

Ultra Performance	1.0x - 3.0x
Off	Turns Intel XeSS-SR off

Jitter scale

The function xessSetJitterScale applies a scaling factor to the jitter offset. This might be useful if the application stores jitter in units other than pixels. For example: NDC jitter can be converted to a pixel jitter by setting an appropriate scale.

Velocity scale

The function xessSetVelocityScale applies a scaling factor to the velocity. This might be useful if the application stores velocity in units other than pixels. For example, a normalized viewport velocity can be converted to pixel velocity by setting an appropriate scale.

Exposure multiplier

The function xessSetExposureMultiplier applies a multiplier to the exposure value. This multiplier works in all exposure modes:

- Autoexposure.
- Exposure passed as value inside xess_*_execute_params_t structure.
- Exposure texture.

Recommended Practices

Visual Quality

We recommend to run XeSS-SR at the beginning of the post-processing chain before tone-mapping. Execution after tone-mapping is possible in certain scenarios; however, this mode is experimental, and good quality is not guaranteed.

The following considerations should be considered to maximize image quality:

- Use high-, or ultra-high-, quality setting for screen-space ambient occlusion (SSAO) and shadows.
- Turn off any techniques for shading-rate reduction and rendering resolution scaling, such as variable rate shading (VRS), adaptive shading, checkerboard rendering, dither, etc.
- Avoid using quarter-resolution effects before XeSS-SR upscaling.
- Do not rely on XeSS-SR for any kind of denoising; noisy signal significantly hurts reconstruction quality.
- Use fp16 precision for the color buffer in scene linear HDR space.
- Use fp16 precision for the velocity buffer.
- Adjust mip bias to maximize image quality and keep overhead under control.
- Provide an appropriate scene exposure value. Correct exposure is essential to minimize ghosting of moving objects, blurriness, and precise brightness reconstruction.

Driver Verification

For the best performance and quality, install the latest driver. To facilitate this, after initialization with xess*CreateContext, call function xessIsOptimalDriver to verify the driver installed will provide the best possible experience. If XESS_RESULT_WARNING_OLD_DRIVER is returned from this function, an advisory message or notice should be displayed to the user recommending they install a newer driver. XESS_RESULT_WARNING_OLD_DRIVER is not a fatal error, and the user should be allowed to continue.

Debugging Tips

Motion Vectors Debugging

If XeSS-SR is producing an aliased or shaky image, it is worth concentrating on static scene debugging:

- Emulate zero time-delta between frames in the engine to maintain a fully static scene.
- Set 0 motion vector scale to exclude potential issues with motion vectors.
- Significantly increase the length of a repeated jitter pattern.

XeSS-SR should produce high-quality, super sampled images. If this does not happen, there might be problems with jitter sequence or the input textures' contents; otherwise, the problem is in the decoding of motion vectors. Make sure that the motion vector buffer content corresponds to currently set units (NDC or pixels), and that axis directions are correct. Try playing with plus or minus 1 motion vector scale factors to align the coordinate axis appropriately.

Jitter Offset Debugging

If the static scene does not look good, try playing with plus or minus 1 jitter offset scaling to appropriately align the coordinate axis. Make sure jitter does not fall off outside of [-0.5, 0.5] bounds.

Versioning

The library uses a `<major>.<minor>.<patch>` versioning format, and Numeric 90+ scheme, for development stage builds. The version is specified by a 64-bit sized structure (xess_version_t), in which:

- A major version increment indicates a new API, and potentially a break in functionality.
- A minor version increment indicates incremental changes such as optional inputs or flags. This does not change existing functionality.
- A patch version increment may include performance or quality tweaks, or fixes, for known issues. There is no change in the interfaces. Versions beyond 90 are used for development builds to change the interface for the next release.

The version is baked into the XeSS-SR SDK release and can be accessed using the function xessGetVersion. The version is included in the zip file and in the accompanying README, as well as the header of the code samples.

Additional Resources

Survey of temporal anti-aliasing techniques

Intel® Arc™ landing page

Intel® XeSS Plugin for Unreal Engine on GitHub

DirectX download page

Notices