Developer Guide

Game Development

Intel® Xe Super Sampling (XeSS) API Developer Guide v1.1

Use this guide to understand how to optimize image quality and performance without impacting frame rates.

Contents

Intel® Xe Super Sampling (XeSS) delivers innovative, framerate-boosting technology, which is supported by Intel® Arc™ graphics cards and other GPU vendors. It upscales with AI deep learning, so offers higher framerates at no cost to image quality. The XeSS API is for any game developer who wants to optimize image quality and performance.

This developer guide supplements the *XeSS API Reference Guide*.

Introduction

Xe Super Sampling is implemented as a sequence of Microsoft* Direct3D* 12 (DX3D) compute shader passes, executed before the rendering engine's post-processing stage (as described in the section entitled 'TAA and XeSS'). The rendering engine initializes XeSS by passing a Direct3D* 12 (D3D12) device, which is being used for the main rendering, and a pointer to a descriptor heap, where XeSS creates all its internal resource descriptors. XeSS allocates GPU resources for one of two categories:

- Persistent allocations, such as network weights, and other constant data.
- Temporary allocations, such as network activations.

The game engine can control the location where XeSS makes its temporary allocations by passing XESS_INIT_FLAG_EXTERNAL_DESCRIPTOR_HEAP initialization flag to xessD3D12Init call and a pointer to an external resource heap in the xessD3D12Execute call. To ensure optimal game performance with XeSS when game engine provides external resource heap, this heap should have HIGH memory residency priority. Persistent allocations are always owned by the XeSS library.

XeSS Components

XeSS is accessible through the XeSS SDK, which provides a D3D12-based API for integration into a game engine, and includes the following D3D12 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 runtime shipped

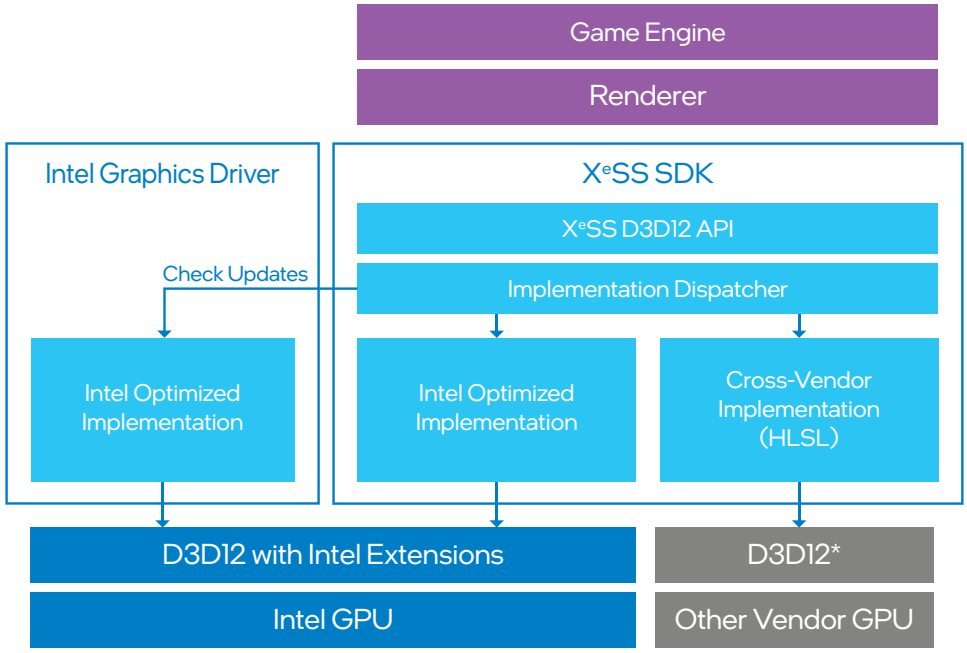

Figure 1. XeSS SDK components for both Intel-specific, and cross-vendor solutions.

with the game, the version provided with the Intel graphics drivers, or the cross-vendor implementation.

Versioning

XeSS uses major.minor.patch version format, and Numeric 90+ scheme, for development stage builds. The XeSS version is specified by the 64-bit function [xess_version_t] structure, 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 XeSS version is baked into the XeSS 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 sampless.

Compatibility

All future Intel graphics driver releases provide compatibility with previous and future XeSS versions.

Specifically, on Intel platforms the loader will operate according to the following rules:

- The loader will check the compatibility of the XeSS version installed with the game, and the installed driver on the system.
- If compatible, the loader will use the game title installed version of XeSS.
- If not compatible, and the driver is newer, the loader will ignore the game title version of XeSS, and use the version distributed with the driver.
- If not compatible and the driver is older, the loader will return a failure code, and XeSS will not initialize.

Naming Conventions

The XeSS API uses the following naming conventions:

- All functions must be prefixed with xess
- All functions must use camel case xessObjectAction convention
- All macros must use all caps XESS_NAME convention
- All structures, enumerations, and other types must follow xess_name_t snake case convention
- All structure members and function parameters must use camel case convention
- All enumerator values must use all caps XESS_ENUM_ETOR_NAME convention
- All handle types must end with handle_t
- All parameter structures must end with params_t
- All property structures must end with properties_t

- All flag enumerations must end with flags_t

TAA and XeSS

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

The figure below 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.

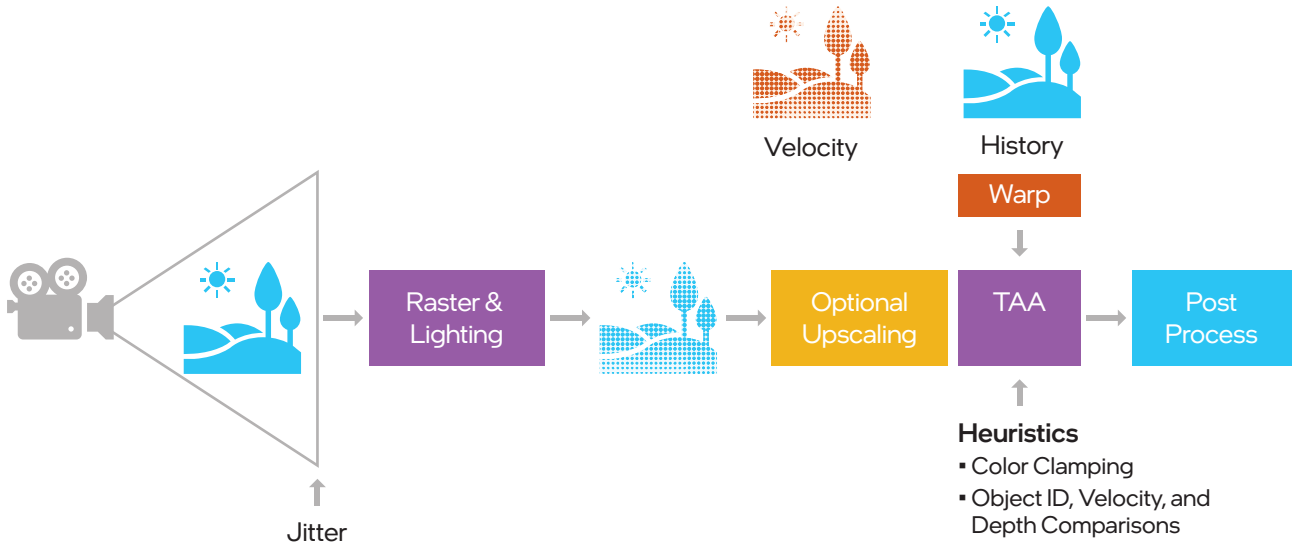

Figure 2. Flow chart of a typical rendering pipeline with TAA.

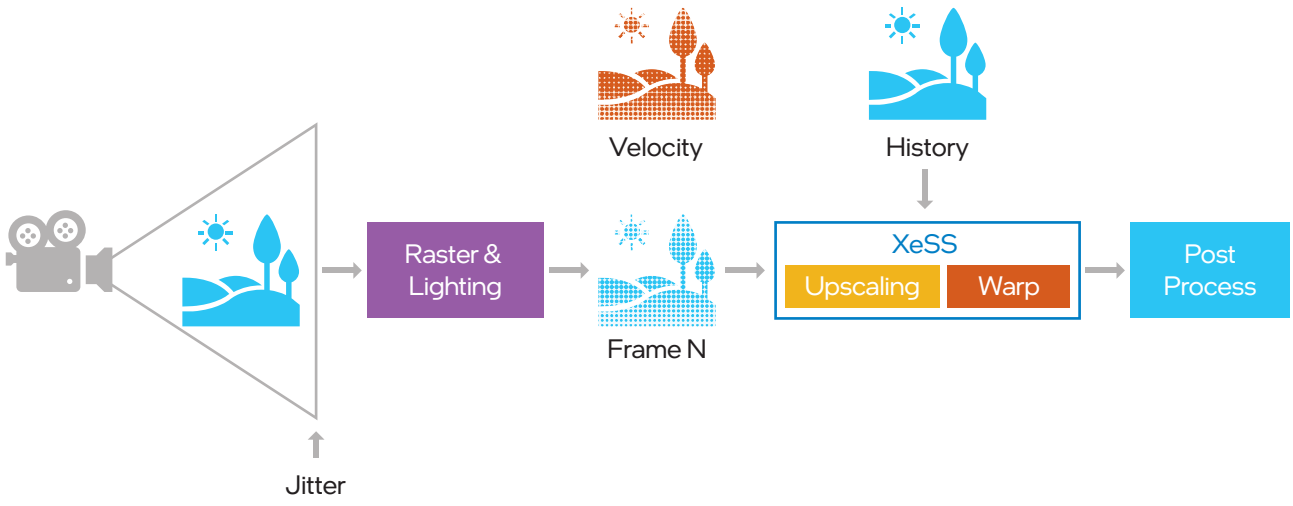

Figure 3. XeSS inclusion into the rendering pipeline.

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.

XeSS 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.

Notes:

- To enable XeSS, your title needs to disable other upscaling technologies, such as DLSS and FSR, and temporal anti-aliasing (TAA) technologies, to reduce the possibility of any incompatibility issues.
- All Intel Xe Super Sampling settings should be exposed to a user through a selection menu, if supported, to encourage customization.

XeSS Game Setting Recommendations

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

Table 1: Naming Conventions

Label	Intel® XeSS
Short Description	Intel Xe Super Sampling (XeSS) technology uses machine learning to deliver higher performance with exceptional image quality. Hardware accelerated XeSS is optimized for Xe-HPG microarchitecture-based GPUs.
Minimum Description	Intel Xe Super Sampling (XeSS) technology uses machine learning to deliver higher performance with exceptional image quality.

XeSS options.

There are also guidelines for the font, official naming, and descriptions of the

Table 2: Game Graphics Settings Menu

Preset	Description	Recommended Resolution
Ultra Quality	Delivers the highest quality visual upscale	1080p and above
Quality	Delivers high quality visual upscale	1080p and above
Balanced	Delivers optimal performance and image quality	1080p and above
Performance	Improves overall gaming performance	1440p and above
Off	Turns Intel XeSS off	NA

XeSS functionality below.

Naming Conventions for Intel Xe Super Sampling Branding

Table 3: Graphics Presets

Default XeSS Recommendation	Description	Recommended Setting
Resolution Specific	Your game adjusts the XeSS 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 preset as default.	Intel XeSS ON set to 'Performance'

You should use the approved naming conventions for XeSS in your settings menus and descriptions. The official font for XeSS-related communication is IntelOneText-Regular. Please use the official superscripted e in XeSS, unless the font system does not support superscript, in which case XeSS is acceptable. For the smaller e in XeSS , you can reduce the font size for just that character to keep the proportions.

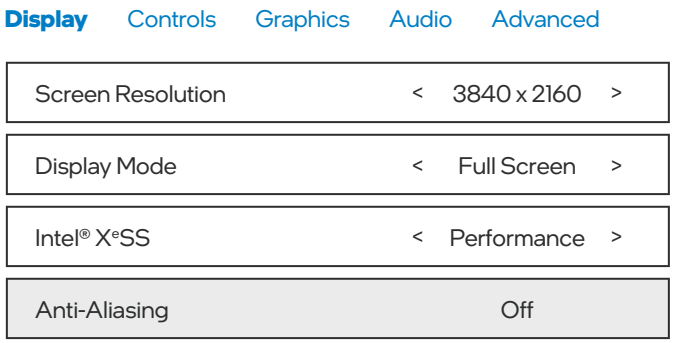

Figure 4. Example of game UI with XeSS settings.

Notes:

- **Screen resolution:** XeSS supports resolutions of 1080p and above
- **Display mode:** XeSS supports full screen, borderless windowed mode, and windowed mode
- **XeSS:** Off, Performance, Balanced, Quality, Ultra Quality
- **Anti-Aliasing:** AA mode should be returned to the previous setting when Intel XeSS is disabled

Game Graphics Settings Menu /Game Installer/ Launcher Settings

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

Graphics Preset Default Recommendations

The XeSS 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.

Intel Xe Super Sampling UI Example

Programming Guidance

Inputs and Outputs

XeSS 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 up sampled inside XeSS.

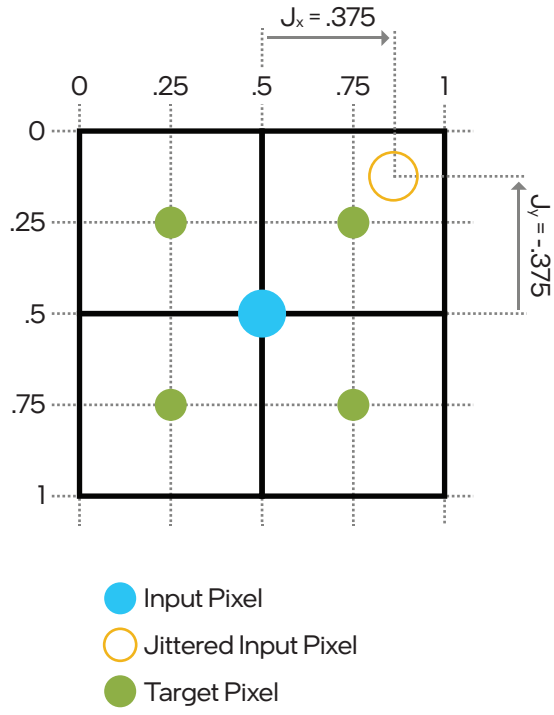

Figure 5. Jitter displacement of sample points.

Jitter

As a temporal super-sampling technique, XeSS requires a sub-pixel jitter offset *(J_x,J_y)* to be applied to the projection matrix every frame. This process essentially 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.M[2][0] += Jx * 2.0f /
InputWidth

ProjectionMatrix.M[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 *(J_x, J_y)*, because the projection matrix is applied to geometry, and it corresponds to a negative camera jitter.

Jitter Sequence

A quasi-random sampling sequence with a good spatial distribution of characteristics is required to get the best quality of XeSS 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 * scale^2* if used with XeSS 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 with regard to the input pixel, however.

Color

XeSS accepts both SDR and HDR input colors in any linear color format, for example: DXGI_FORMAT_R16G16B16A16_FLOAT, DXGI_FORMAT_R11G11B10_FLOAT, DXGI_FORMAT_R8G8B8A8_UNORM, etc. 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.

If the input color values have not been adjusted for the exposure, or if the input color values 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 can calculate it when the XESS_INIT_FLAG_ENABLE_AUTOEXPOSURE flag is used at initialization. Note that use of this flag will cause a measurable performance impact.
- When the XESS_INIT_FLAG_ENABLE_AUTOEXPOSURE flag is not provided, an input exposure scale value can be provided, or an exposure scale texture which can be updated by the GPU.

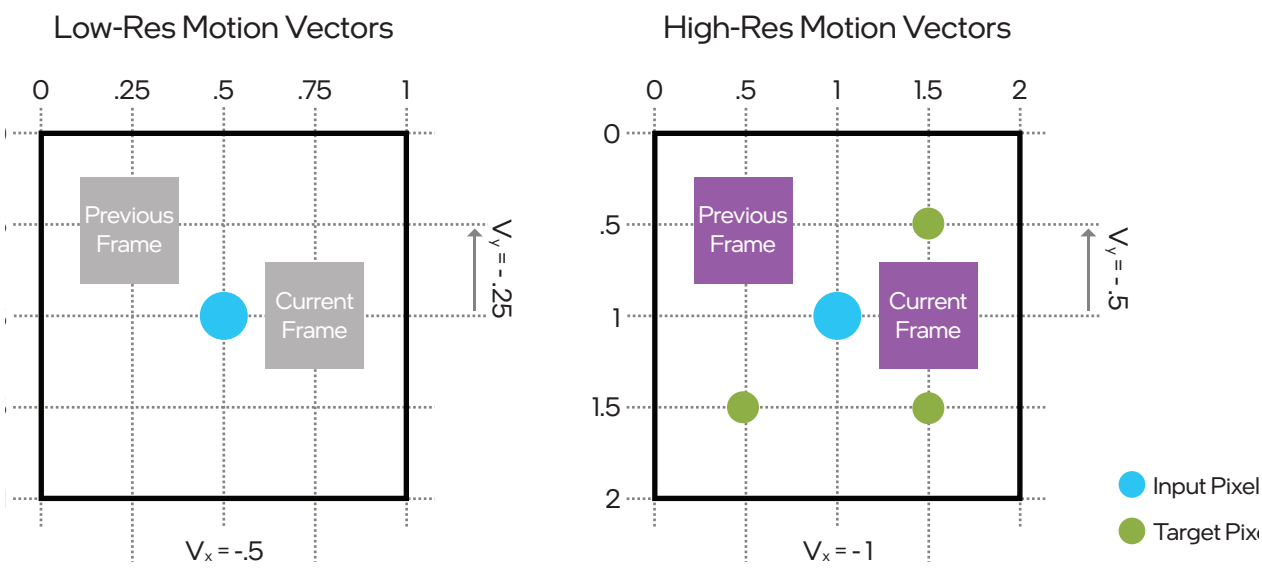

Figure 6. Convention for specifying the low-res and high-res motion vector to XeSS.

These scale values are applied to the input as shown below:

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

    inputColor *= scale
```

The output is in the same color space as the input. It can be any three or four channel linear color format similar to the input. If a scale value is applied to the input, as shown above, the inverse of this scale is applied to the output color. XeSS 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 passing setting historyReset flag in xess_xxx_execute_params_t.

Motion Vectors

Motion vectors specify the screen-space motion in pixels from the previous frame to the current frame. XeSS accepts motion vectors in the format DXGI_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 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 internally up-samples motion vectors to the target grid and uses the depth texture to dilate them. The

Table 4. Resource Formats

Input format	Output format
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

figure below shows the same motion specified with low-res and high-res motion vectors.

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 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 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 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

You could provide a responsive pixel mask with a mask value of 1 to force XeSS to ignore information from previous frames. Although XeSS 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 artifacts, for example particles. In such cases, a responsive pixel mask can be set for these objects. Any texture format can be used for the mask, as long as the mask value is in the R channel.

Resource States

XeSS expects all input textures to be in the state D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE, and the output texture to be in the state D3D12_RESOURCE_STATE_UNORDERED_ACCESS.

Resource Formats

XeSS expects all input textures to be typed. For typeless formats XeSS performs a conversion according to Table 4.

Mip Bias

To preserve texture details at the target resolution, XeSS requires an additional mip bias of (log2(frac{Input Width}{Target Width})). For example, a mip bias of -1 should be applied for 2x resolution scaling. In certain cases, increasing mip bias even more leads to an additional visual quality improvement; this comes with a potential performance overhead, however, due to increased memory bandwidth requirements, and potentially lower temporal stability resulting in flickering and moire. You are, of course, free to experiment with more or less aggressive texture LOD biases to find the right balance.

Initialization

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

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

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

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

The xessD3D12Init function is then called to initialize XeSS. During initialization, XeSS can create staging buffers and copy queues to upload weights. These will be destroyed at the end of initialization. The XeSS 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 includes three types of storage:

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

Allocate temporary storage either internally in a library-managed heap (default), or in a heap provided by the user in the pTempStorageHeap field of the xess_d3d12_init_params_t structure. If you allocate the temporary storage, it can be reused outside of XeSS execution.

```
ComPtr<ID3D12Heap> pHeap;
CD3DX12_HEAP_DESC heapDesc(xessProp.
tempHeapSize,D3D12_HEAP_TYPE_DEFAULT);

d3dDevice->CreateHeap(&heapDesc, IID_PPV_
ARGS(&pHeap));

initParams.tempStorageOffset = 0;
initParams.pTempStorageHeap = pHeap.Get();

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

You can specify the XESS_INIT_FLAG_EXTERNAL_DESCRIPTOR_HEAP initialization flag to use the external descriptor heap later at the execution stage.

You can also re-initialize XeSS if there is a change in the target resolution, or any other initialization parameter. However, pending XeSS command lists must be completed before re-initialization. When temporary XeSS storage is allocated, it is your responsibility to de-allocate, or reallocate, the heap. Quality preset changes are free, but any other parameters change may lead to longer xessD3D12Init execution times.

Execution

The XeSS execution function does not involve any GPU workloads, rather it records XeSS commands into the specified command list. The command list is then enqueued by the user. That means it is your responsibility to make sure all input/output resources are alive at the time of the actual GPU execution.

By default, XeSS 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 in execution parameters.

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 descriptor table.

The resolution of the input image depends on the desired quality setting and target resolution. Call xessGetInputResolution at any point of execution to determine the correct input resolution based on quality setting and target resolution, then provide this input resolution as part of the xess_d3d12_execute_params_t structure., Please note that Ddynamic resolution scaling is not supported by XeSS.

```
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);
```

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.

Debug and Logging Capabilities

Logging Callback

The XeSS SDK provides an API to set logging callback. Use function xessSetLoggingCallback to define a function to be called in the following circumstances:

- 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

Input Dump Functionality

XeSS SDK provides an API to dump SDK inputs, outputs and history state. To dump inputs, the application should call function xessStartDump. Due to internal implementation, SDK can dump fewer frames than provided in frame_count fie field of the _dump_parameters_t structure.

Recommended Practices

Visual Quality

We recommend you run XeSS in the beginning of the post-processing chain, before the 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 upscaling.
- Do not rely on XeSS for any kind of denoising; noisy

Additional Resources

- A Survey of Temporal Antialiasing Techniques (PDF)
- Intel® Arc™ landing page
- Intel® Xe Super Sampling Plugin for Unreal* Engine on GitHub
- DirectX download page

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 xessD3D12CreateContext, 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 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 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 most likely in the decoding of motion vectors. Make sure that motion vectors buffer contents correspond to currently set units (NDC or pixels), and axis directions are correct. Try playing with plus or minus 1 motion vectors scale factors to align 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.