**Using UAV Overlap Extension in Intel’s D3D Extension Framework**

Contents

[**Introduction** 1](#_Toc53129456)

[**Using the Intel D3D Extensions Framework (Pre-Alpha)** 2](#_Toc53129457)

[**Load the extension library into your process** 2](#_Toc53129458)

[**(Optional) Get the list of supported versions of the Extensions Framework** 2](#_Toc53129459)

[**Create an ExtensionContext object** 2](#_Toc53129460)

[**Destroy the ExtensionContext object** 3](#_Toc53129461)

[**Unload the Extensions Library** 3](#_Toc53129462)

[**Using UAV Overlap** 3](#_Toc53129463)

[**Begin UAV Overlap** 3](#_Toc53129464)

[**End UAV Overlap** 3](#_Toc53129465)

[Code Sample 4](#_Toc53129466)

[**Performance Result** 5](#_Toc53129467)

[**Conclusion** 6](#_Toc53129468)

**Introduction**

UAV Overlap is an extension in the Intel driver that exposes in DirectX 11 the ability for multiple dispatches or draws that access the same Unordered Access View (UAV) to run concurrently. In DirectX 11, these dispatches or draws are serialized to avoid read-after-write (RAW) or write-after-write (WAW) hazards, as there is no synchronization mechanism available. This can negatively affect performance. There are cases, such as GPU emitted particles, where multiple dispatches or draws use different regions of a UAV memory and the algorithm ensures that there are no read/write dependencies on these regions. Avoiding this serialization can increase performance. This makes UAV access similar to how DirectX 12 handles UAV access.

In this paper, we provide instruction and code snippets to first enable the Intel D3D Extensions Framework for your application by loading the extension library, getting a list of supported features, creating an extension context and how to cleanup by destroying the extension context and unloading the library. Next, once you have the extension context, we’ll show the commands on how to let the driver you know which segment of code you intend for UAV overlap by providing the proper begin and end commands, along with sample code to demonstrate usage. Finally, we will provide performance numbers for the sample.

**Using the Intel D3D Extensions Framework (Pre-Alpha)**

To use the extensions supported by Intel's D3D Extensions Framework, you must include igdext.h in your project. You must also set your linker to include the static library igdext64.lib with your project to provide entry points.

**Load the extension library into your process**

HRESULT INTC\_LoadExtensionsLibrary(bool useCurrentProcessDir = false);

Loads the Extension Framework .dll into the currently active process. This step must be done first, as it fills in the rest of the entry points found in igdext.h.

**(Optional) Get the list of supported versions of the Extensions Framework**

It is recommended to check feature support for UAV overlap on your platform before you attempt to use UAV overlap. Below is the function used to query support on your platform.

HRESULT INTC\_D3D11\_GetSupportedVersions(

ID3D11Device\* pDevice,

INTCExtensionVersion\* pSupportedExtVersions,

uint32\_t\* pSupportedExtVersionsCount);

The first call to INTC\_D3D11\_GetSupportedVersions fills in the number of supported versions in pSupportedExtVersionCount. Based on this, the application is responsible for allocating memory for pSupportedExtVersions. Once allocated, another call is done to INTC\_D3D11\_GetSupportedVersions to fill in pSupportedExtVersions. You are now ready to search and determine if your hardware supports the required hardware feature level and API version to enable UAV Overlap.

INTCExtensionVersion is a struct with 3 integer components: { HWFeatureLevel, APIVersion, Revision }.

* HWFeatureLevel - Nearly every extension requires some specific hardware support, so it is important to check the HWFeatureLevel to determine if your GPU has the required feature level for the extension(s) you want to use..
* APIVersion - Software version of the Extensions Framework itself. It gets incremented over time as new extensions are added to the Framework.
* Revision - Reserved for bugfixes.

Production extensions are guaranteed to be backwards-compatible, but experimental extensions make no such guarantee.

For UAV Overlap a minimum of API Version 1 and feature version 1 are needed.

Once you have determined hardware and API support is or is not available, remember to deallocate the memory from pSupportedExtVersions.

There is a matching entrypoint for D3D12 extensions.

**Create an Extension Context**

To start using UAV overlap, you need to declare an object of type INTCExtensionContext\* and pass its address to INTC\_D3D11\_CreateDeviceExtensionContext. Below is the signature of this function. The memory for this object is allocated by the Extension Framework.

HRESULT INTC\_D3D11\_CreateDeviceExtensionContext(

ID3D11Device\* pDevice,

INTCExtensionContext\*\* ppExtensionContext,

INTCExtensionInfo\* pExtensionInfo,

INTCExtensionAppInfo\* pExtensionAppInfo);

The INTCExtensionInfo struct contains primarily output members, but the one required input is an INTCExtensionVersion RequestedExtensionVersion, indicating the version of the Extensions Framework you want to initialize.

The INTCExtensionAppInfo struct contains completely optional input members.

If this function returns successfully, the INTCExtensionContext object will have been created, and it will be used as an input parameter to every other extension entrypoint.

Note: There is a matching entrypoint for D3D12 extensions.

**Destroy the Extension Context**

The Extensions Framework is responsible for allocating and deallocating the memory for the INTCExtensionContext object. The below function is required to properly destroy the context and free its memory. This needs to be done on application exit or as part of your selected memory cleanup mechanism.

HRESULT INTC\_DestroyDeviceExtensionContext(INTCExtensionContext\*\* ppExtensionContext);

**Unload the Extensions Library**

Upon application exit, or if initialization fails, the Extensions Framework needs to be unloaded.

void INTC\_UnloadExtensionsLibrary();

**Using UAV Overlap**

In this section, we describe begin and end mechanisms to enable a region of code to use UAV Overlap.

**Begin UAV Overlap**

HRESULT INTC\_D3D11\_BeginUAVOverlap(INTCExtensionContext\* pExtensionContext);

Marks the beginning point for disabling GPU synchronization between consecutive draws and dispatches that share UAV resources.

**End UAV Overlap**

HRESULT INTC\_D3D11\_EndUAVOverlap(INTCExtensionContext\* pExtensionContext);

Marks the end point for disabling GPU synchronization between consecutive draws and dispatches that share UAV resources.

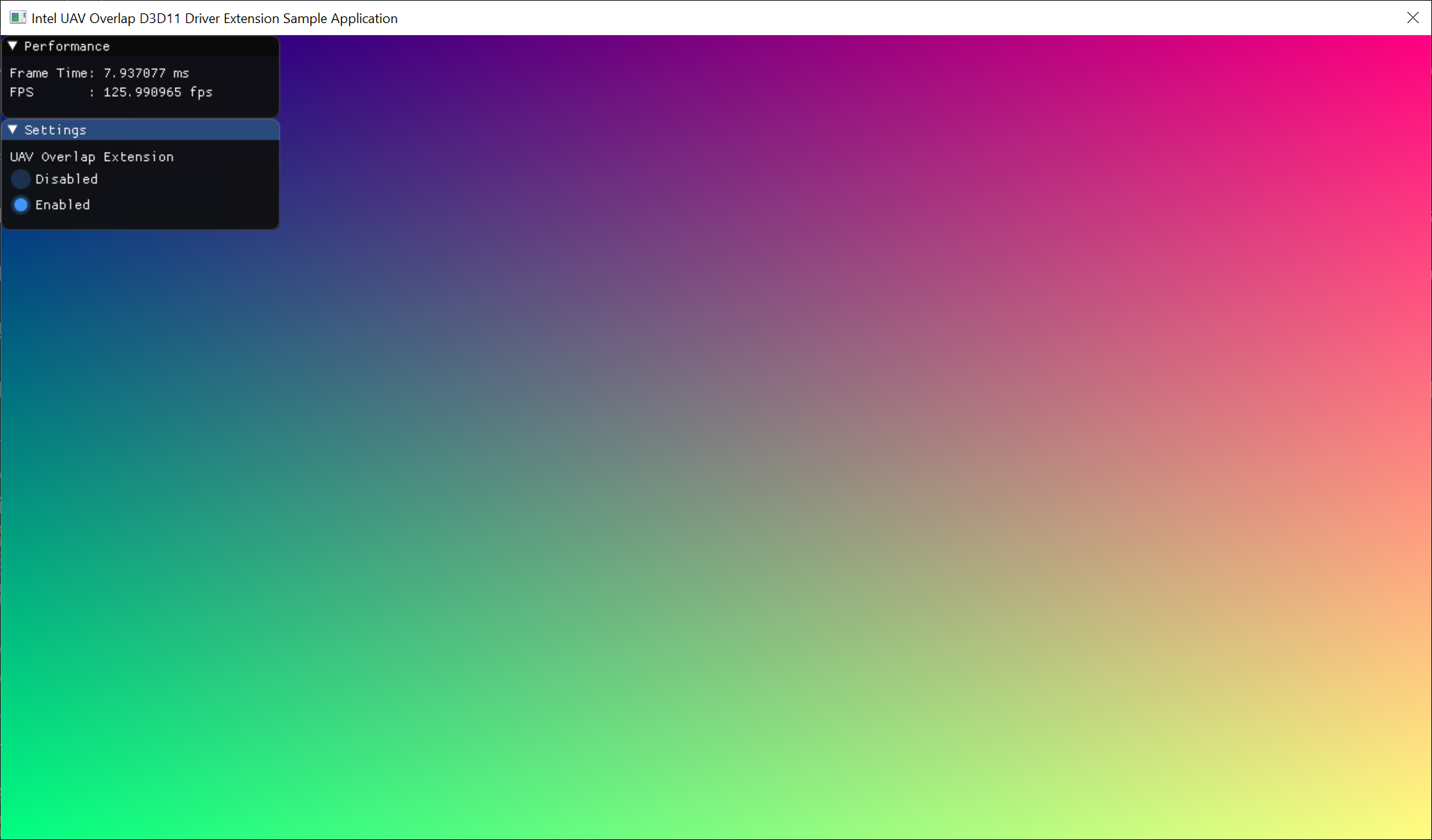
### Handling HRESULT

When working with any extension function that returns an HRESULT, on successful call the function will return a code of S\_OK. Any other result needs to be handled as behavior of UAV overlap is no longer guaranteed. Depending on the application, this can be integrated into exception handling framework, logging, etc. If application abortion will happen due to this, it is recommended to attempt to destroy the extension context and unload the library, as seen in the above sections.

### Code Sample

In this code sample, we simulate an algorithm that an application developer might write where a set of dispatches would be guaranteed to access unique portions of the same UAV.

To make the performance cost of serializing these dispatches abundantly clear, this sample runs 3600 dispatches back-to-back, each of which writes to a unique 16x16 block of pixels in a 2D texture, which is then rendered to the screen as a 1280x720 render target.



The key piece of code to learn from is this section, where we wrap our dispatches between calls to INTC\_D3D11\_BeginUAVOverlap and INTC\_D3D11\_EndUAVOverlap:

// Disable UAV syncs until a call to INTC\_D3D11\_EndUAVOverlap() is encountered

INTC\_D3D11\_BeginUAVOverlap(mINTCExtensionContext);

...

for (uint32\_t x = 0; x < numDispatchesX; x++)

{

for (uint32\_t y = 0; y < numDispatchesY; y++)

{

// Update the sample constant buffer with the current groupID (x,y)

...

mImmediateContext->Dispatch(1, 1, 1);

}

}

// Re-enable UAV syncs

INTC\_D3D11\_EndUAVOverlap(mINTCExtensionContext);

...

We use enough 1x1x1 Dispatches to run 1 thread per pixel in our Texture2D that we bind as a UAV. Each dispatch kicks off one 16x16x1 thread group, and receives some unique constant data, effectively representing a GroupID over the image.

In the compute shader we reconstruct a DipatchThreadID using that constant data and the GroupThreadID, and write to the UAV at that location. Thus, we guarantee that each write to the UAV is going to touch a unique pixel, and overlapping writes should be perfectly safe.

[numthreads(16, 16, 1)]

void CS(uint3 mGroupThreadID : SV\_GroupThreadID)

{

// Compute screen coordinates for the current thread

uint xcoord = dispatchX \* 16 + mGroupThreadID.x;

uint ycoord = dispatchY \* 16 + mGroupThreadID.y;

uint2 coord = uint2(xcoord, ycoord);

// Write out a color to the bound UAV at this thread's screen coordinate

gOutput[coord] = float4((float)xcoord / windowWidth, (float)ycoord / windowHeight, 0.5, 1.0);

}

**Performance Result**

Early driver results on this sample show that with UAV Overlap on, the average frame time is 4.6 ms and without UAV overlap, the average frame time is 16.6 ms. This results in a speedup of 3.6x. An applications resulting speedup is dependent on the work being done, or may not show any benefit at all.

Figure 1: Frame times with UAV overlap off vs UAV overlap on

**Conclusion**

The UAV Overlap Extension is another tool in a graphics programmers toolbox to provide DirectX 12 like UAV access to UAV resources without the implicit serialization of subsequent commands that access the same UAV buffer. This whitepaper showed how to setup and use UAV Overlap and provided sample code on what typical startup, usage, and shutdown should look like. Performance on the simple sample showed a 3.6x increase in performance, but this is dependent on workload.