Rasterizer Order Views 101: a Primer

Introduction – What are Rasterizer Order Views?

One of the new features of DirectX* 12 is Rasterizer Ordered Views, which allow read/write access to resources, such as buffers, textures, and texture arrays, without multisampling from multiple threads and without generating memory conflicts through the use of atomic functions. This feature means resources created with Unordered Access Views (UAV) can mark in the pixel shader code certain resources to follow strict ordering rules similar to those used to ensure the correct pixel blending during draw operations. Raster ordered views (ROVs) allow the creation of a whole range of new algorithms, such as Order Independent Transparency (OIT), Adaptive Volumetric Shadow Maps (AVSM), and custom blending operations, that are not possible in the fixed function blending pipeline.

How did Rasterizer Order Views come about?

Certain graphics problems like order independent transparency are vital for realistic looking smoke, foliage, hair, and water but don't fit into the traditional rendering pipeline. The flexibility and power of programmable shaders would seem to offer a solution to these kinds of problems, except that, even with atomics, it is not possible to read and modify data in a deterministic manner inside a shader using UAVs, leading to potential visual artifacts. ROVs are important because they can help solve this problem by synchronizing shader execution in order of triangle submission.

Order Independent Transparency is vital for realistic looking of smoke.

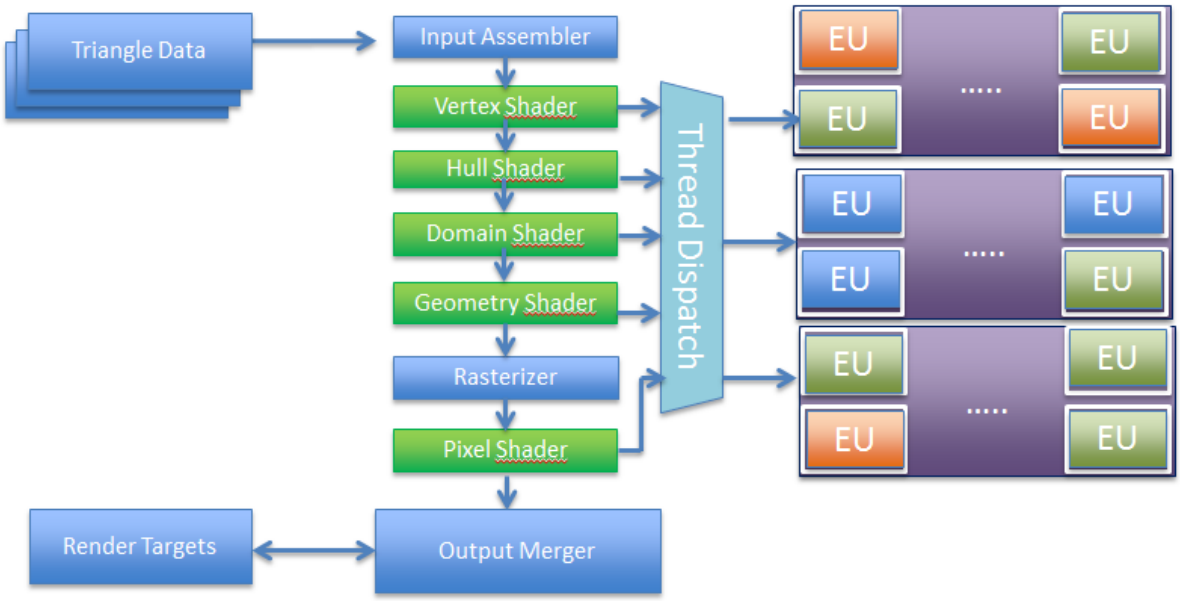
So, why is Intel talking about rasterizer order views in DirectX 12? Well, two years ago Intel introduced similar functionality with the release of PixelSync as part of 4th generation Intel® Core™ processors.

Johan Andersson, Technical Director at DICE for Battlefield 4, when asked about what he wanted to see in the next generation of GPUs from all hardware vendors, even mentioned PixelSync: "We have a pretty long list…but one very concrete thing we'd like to see, and actually Intel has already done this on their hardware, they call it PixelSync, which is their method of synchronizing the graphics pipeline in a very efficient way on a per-pixel basis. You can do a lot of cool techniques with it, such as order independent transparency for hair rendering or for foliage rendering. And they can do programmable blending where you want to have full control over the blending instead of using the fixed-function units in the GPU. There's a lot of cool components that can be enabled by such a programmability primitive there…". ROVs now bring a standard way of accessing the PixelSync functionality that Johan liked across a wide variety of hardware from different vendors.

DirectX Pipeline and the limitations of UAVs

As mentioned in the introduction, rasterizer order views are important because they allow you to read and modify data in a deterministic manner inside a shader using UAVs. So why isn't this possible without ROVs? To understand that, you need to understand how data passes through the various stages in the graphics pipeline.

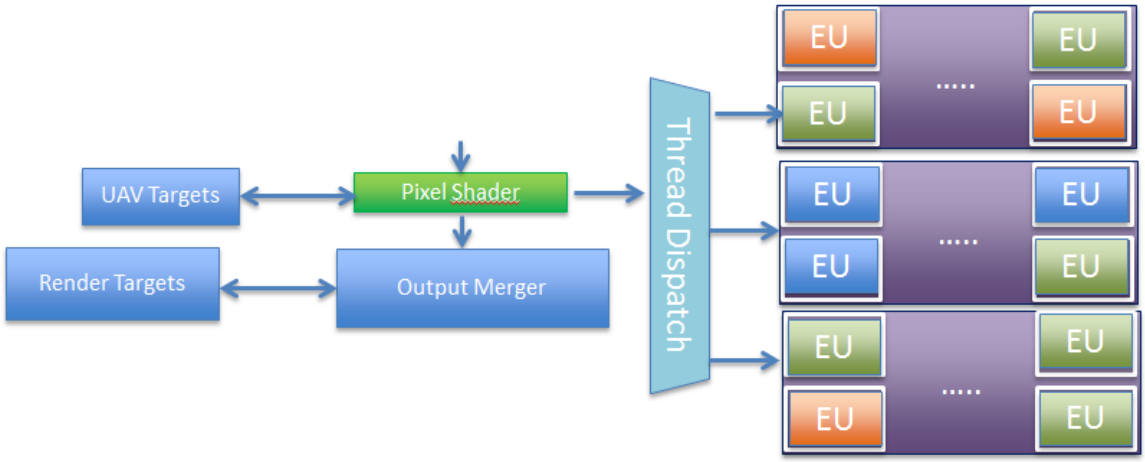
DirectX rendering follows a strict set of rule that ensure triangles are always rendered in the order they are submitted: if two triangles are overlapping on the screen, the hardware guarantees that Triangle 1 will have its color result blended to the screen before Triangle 2 is processed and blended.



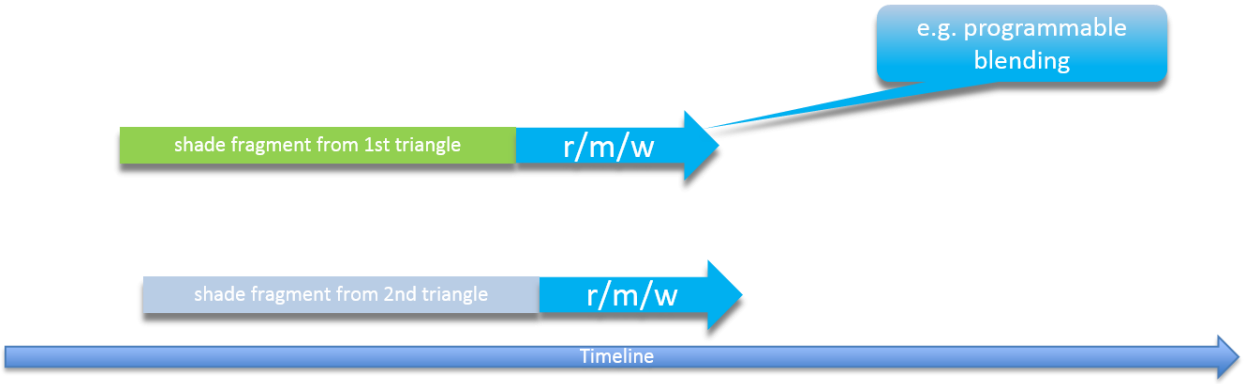
When triangles are submitted, they run through the input assembler then through the vertex shader, hulls shader, or geometry shader depending on which parts are being enabled, before reaching the rasterizer and then to the pixel shader. These shaders are run on programmable units called EUs (execution units) on Intel® hardware, with the system designed to run many shaders in parallel across different EUs. Hardware on the backend, called the raster operations pipeline (ROP), is responsible for enforcing this ordering requirement ensuring pixels from Triangle 1 are rendered before Triangle 2.

However it's not programmable; you can only do a fixed menu of operations on the color, z, and stencil buffers. Another major limitation of the pipeline shown above is the fact the input data sources have to be different than the output render targets—a shader can't modify its own incoming data. DirectX 11 introduced a way around this particular limitation with the introduction of UAVs.

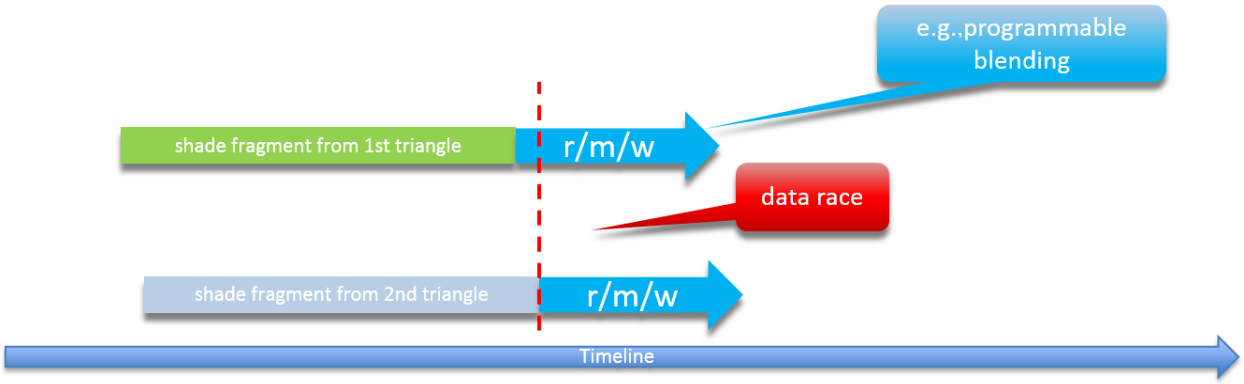
UAVs are resources (which include buffers, textures, and texture arrays) that are directly connected to one of the shaders and are processed therefore before the output merger. They're processed before the pipeline part that specifies the in-order behavior between individual triangles.



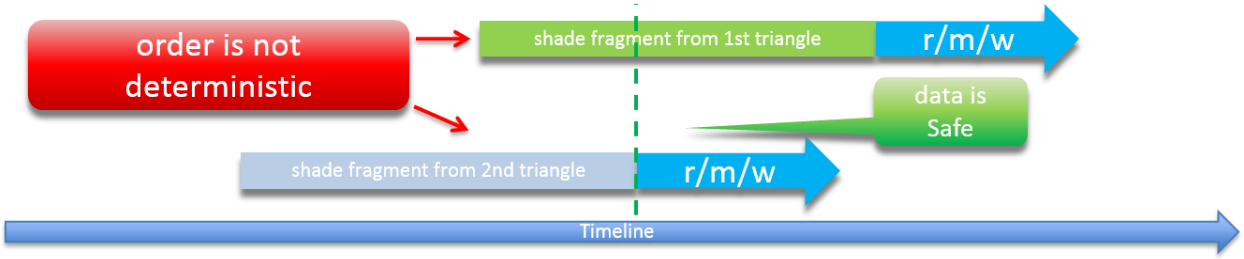
So what are the UAV limitations stemming from operating before the output merge? Imagine two triangles entering the pipeline. The first pixel shader from Triangle 1 does a read/modify/write (r/m/w) operation on data in a UAV using the screen location as an index, then the second comes along and tries to do a r/m/w. If the triangles overlap, they potentially could access the exact same location.



This would set up a data race condition and a situation where some data may get dropped from either Triangle 1 or 2. Triangle 1 could have read some data from the UAV and be in the middle of processing before writing its value back. Triangle 2, because of the race condition, could potentially read the same starting data, do its own set of calculations, and write another result back to the UAV surface wiping over the result from Triangle 1. The effect would the same as if Triangle 1 was never run.



Even if the data is safe, there still are potential issues. Although Triangle 1 starts processing first, there's no guarantee that the shader that accesses the UAV referred to by Triangle 1 will run first. For example, if Triangle 1 happened to access new data from outside the local cache or took a more complex dynamic path through its shaders, Triangle 2 could actually run the code that accesses the UAV first.

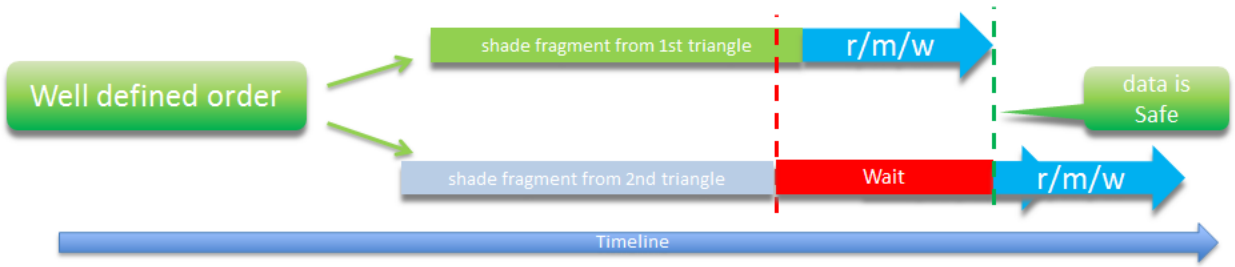


Even if the r/m/w operation doesn't cause a race condition, the nondeterministic order of execution can cause significant issues for certain algorithms. Here's a video that shows an example of what happens if there isn't pixel sync and you use UAVs to implement order independent transparency using a lossy compression algorithm. You can actually see flickering occur because the pixels are getting operated on in the wrong order and the written values are changing between frames even though the scene isn't changing. It's very obvious to the viewer, and frequently the more powerful the hardware (and the more threads executed in parallel), the more flicker will be seen.

<Video>

We need the hardware to detect dependencies among fragments writing to the same x/y screen coordinate and enforce the same ordering rules normally used by the ROP, but at the point of access to the UAV. This would avoid data races and guarantee primitive ordering for r/m/w operations.

So, what happens when you have the ability to enforce order between triangles? If Triangle 1 is in middle of its r/m/w operation and Triangle 2 hits the same point in the shader, rather than starting the algorithm it will block and wait until the first triangle is finished to avoid any race conditions. Even if Triangle 2 gets there first, it will wait until Triangle 1 is run and finished before starting Triangle 2, which creates a determined order and also avoids race conditions.



When the fragments are not overlapping and don't write the same x/y coordinate, there is no performance impact. Even if two fragments are in flight and reference the same x/y coordinate, the performance cost is minimal if the hardware executes them in the original submitted order.

ROV API

The ROV API is a High Level Shading Language (HLSL) construct that builds on the existing UAV support, so the only code changes to make are within the HLSL shaders themselves. Below is a list of resource types that can be defined within HLSL:

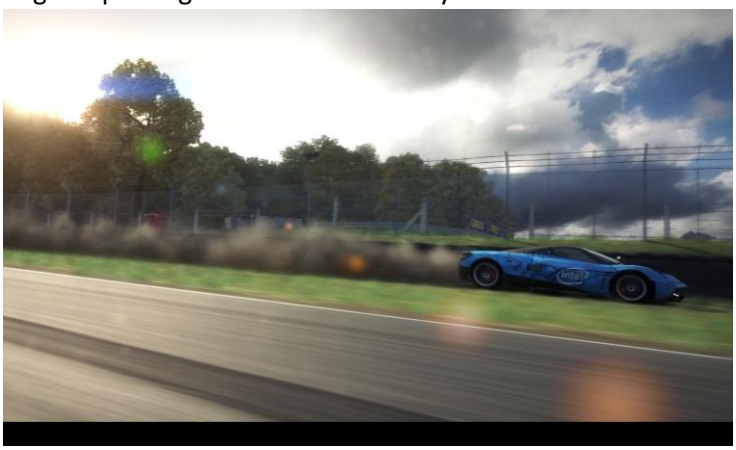
- RasterizerOrderedBuffer
- RasterizerOrderedByteAddressBuffer
- RasterizerOrderedStructuredBuffer
- RasterizerOrderedTexture1D
- RasterizerOrderedTexture1DArray
- RasterizerOrderedTexture2D
- RasterizerOrderedTexture2DArray
- RasterizerOrderedTexture3D

Each of the declarations corresponds to a normal UAV resource, such as a RWBuffer or a RWTexture2D, etc. Outside of the HLSL shader, no code changes calling C++ code are required. Resources are created as normal and a UnorderedAccessView is created and bound using OMSetRenderTargetsAndUnorderedAccessViews.

Porting between PixelSync and ROVs

While there are a couple of subtle differences between writing shader code using Raster Order Views in DirectX 12 and Intel's PixelSync, they basically work the same. An algorithm created in one is easily transferable to the other. That's useful for developers, as for the last three years Intel has been writing code samples for PixelSync. All of these samples are easily transferable to ROVs, enabling support for not just Intel hardware but all hardware vendors. Many of these samples have been used in shipping games. For example, both Grid* 2 from Codemasters and the Total War* series from Sega used OIT for improving foliage. Many of these game development houses continue to use these algorithms in their

engines proving ROVs have the ability to make a real visual difference on current hardware.



The great outdoors in GRID 2 by Codemasters with OIT applied to the foliage and chain link of fencing*

Any PixelSync sample on the Intel web site is easy to transfer over to ROV code. Inside all PixelSync code is an include file titled IntelExtensions.hlsl, a declaration to a UAV resources such as RWTexture2D along with its defined binding slot, and a couple of predefined functions such as IntelExt_Init() and IntelExt_BeginPixelOrderingonUAV. The latter is used to define the actual synchronization point and the UAV surface affected.

The ROV syntax is more simplified. An external include file is not needed, and instead the RGBE buffer gets declared as a raster ordered texture. The shader compiler will automatically insert a synch point at the first instance of the RGBE buffer read. It even knows which UAV slot has been assigned. It's very quick and simple to move from PixelSync to ROV.

Pixel Sync

```
#include "IntelExtensions.hlsl“

RWTexture2D <t> gRGBEBuffer : register( u1 );

void PS_RGBE_Blend (...)
{
    IntelExt_Init();
    // Code that doesn’t reference
        UAV’s
IntelExt_BeginPixelOrderingOnUAV(1);
    // Access UAV
    uint rgbe = gRGBEBuffer[xy];
    // Manipulate UAV
}
```

Raster Ordered View

```
RasterOrderedTexture2D<t> gRGBEBuffer;

void PS_RGBE_Blend (...)
{
    // Code that doesn’t reference
        UAV’s
    // Access UAV
    uint rgbe = gRGBEBuffer[xy];
    // Manipulate UAV
}
```

ROV use cases

The ability to have deterministic order access to r/w buffers within a shader opens up a lot of interesting solutions to current graphic problems. One of the most obvious use cases is for programmable blending to replace the fixed function hardware, which opens the possibility for using custom data types as a

render target using a 32-bit render surface to store RGBE data, allowing for greater precision as is shown in this Intel code sample. A simple extension to programmable blending is g-buffer blending, where surfaces containing nonlinear data such as a surface normal can be correctly combined, which is a real benefit for deferred renderers.

A more complex use case is to create a k-buffer that is a generalization of the traditional z-buffer-based framebuffer. Instead of restricting the framebuffers to a single value, the k-buffer uses memory as a r/m/w pool of k entries whose use is programmatically defined by k-buffer operations. Using ROVs to generate a k-buffer allows single pass implementations for depth-peeling, order independent transparency, and depth-of-field and motion blur effects.

In DirectX 11, r/m/w operations had undefined behavior. These algorithms were frequently designed around per-pixel linked lists, which had unbounded memory requirements. In many cases the unbounded memory requirement can be completely removed, as various forms of lossy compression can be done while the data is inserted to keep the data set within a fixed size. An implementation of order independent transparency used in GRID 2 and GRID Autosport did exactly this—the first k pixels were stored in the k buffer and once its limit was reached any additional transparent pixels were merged with the current data using a routine to minimize the variation in the result from the ground truth.

In addition to the algorithms already mentioned and used in games, ROVs might prove useful in several R&D topics, such as using ROVs for custom anti-aliasing solutions especially when combined with another DirectX 12 feature in conservative rasterization and voxelisation.

One problem voxelisation routines often have is the insertion of the data into the 3D voxel grid. This is normally accomplished using atomic operations. ROVs allow much more complex data structures to be modified safe from race conditions with other triangles updating the mesh. Using ROVs for voxelisation does require the geometry to be rasterized into the three planes as separate calls, rather than a single draw submission with the geometry shader choosing the plane to project into. Fragment dependencies cannot be tracked over multiple 2D planes in a single render call, so it offers an interesting tradeoff for the more complex data that can be managed within the structure.

SUMMARY

Rasterizer order views are a new set of tools to help developers control the 3D pipeline. It's very simple to use and offers new solutions for many long standing problems like order independent transparency, depth peeling, and volume rendering and blending, all while improving game performance. As a starting point for experimenting with ROVs, refer to articles on related topics like PixelSync samples or even the OpenGL* extensions that duplicate ROV behavior. With two generations of Intel hardware supporting ROVs and broad support from the rest of the industry, ROVs offer a great addition to the transitional pipeline, one you can use now. Good luck with your game coding!

About the Author

Leigh Davies is a senior application engineer at Intel with over 15 years of programming experience in the PC gaming industry, originally working with several developers in the UK and then with Intel. He is currently a member of the European Visual Computing Software Enabling Team, providing technical support to game developers. Over the last few years Leigh has worked on a wide variety of enabling areas from graphics (optimization, Order Independent Transparency, and Adaptive Volumetric Shadow Mapping) to multi-core, enabling platform optimizations like touch and sensors controls. Over the last 2 years Leigh has worked on Windows* (DirectX 11 and 12) and Android* (GLES 3.1).

REFERENCES

- https://msdn.microsoft.com/en-us/library/windows/desktop/dn914601(v=vs.85).aspx
- https://software.intel.com/en-us/gamedev/code-samples
- http://advances.realtimerendering.com/s2013/2013-07-23-SIGGRAPH-PixelSync.pdf
- https://software.intel.com/en-us/blogs/2013/03/27/programmable-blend-with-pixel-shader-ordering
- http://www.gdcvault.com/play/1020221/Rendering-in-Codemasters-GRID2-and
- https://software.intel.com/sites/default/files/salvi_avsm_egsr2010.pdf