OIT to Volumetric Shadow Mapping, 101 Uses for Raster Ordered Views using DirectX 12

Leigh Davies
March 05, 2015

Agenda

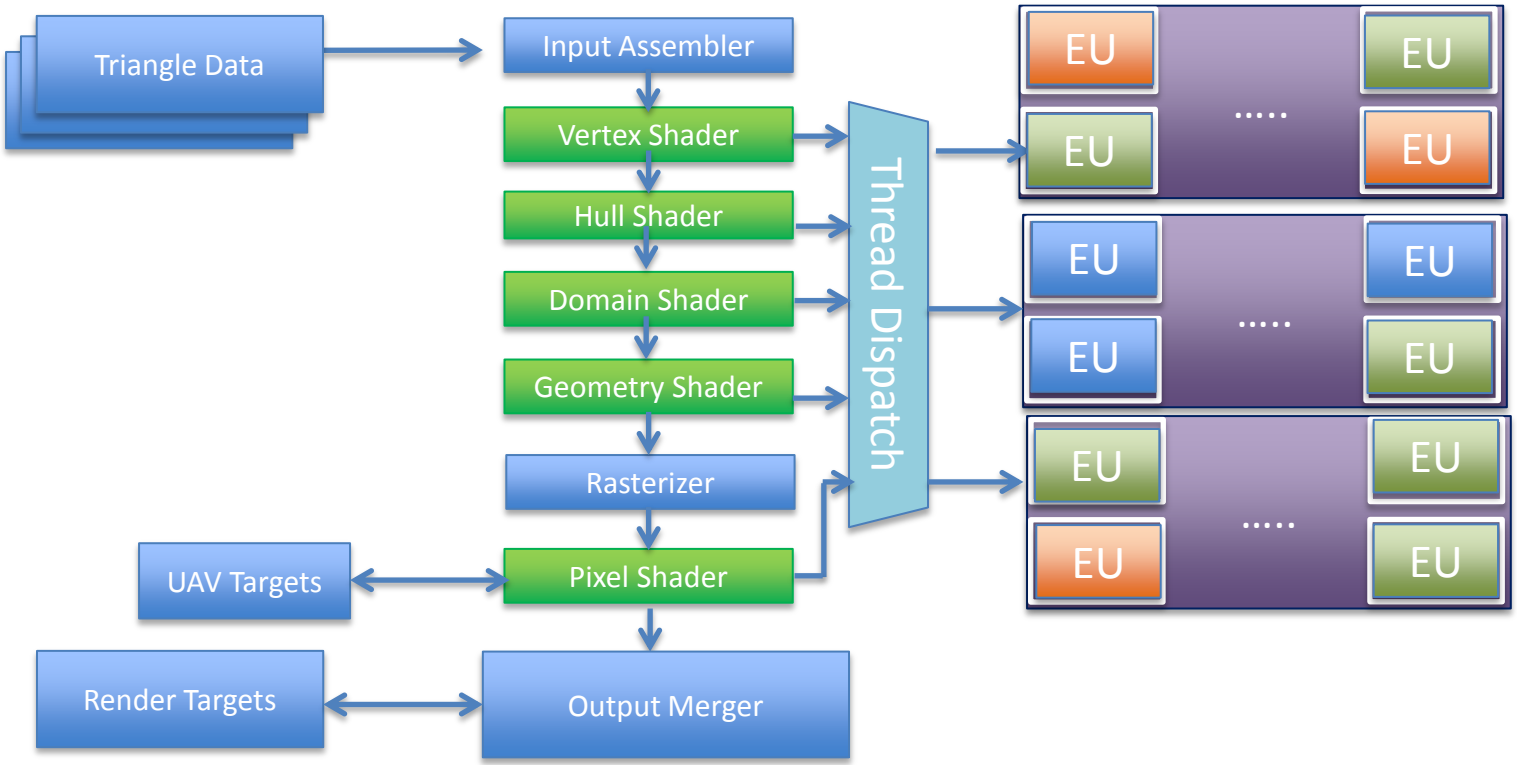
- Introduction
- Raster Ordered Views
- Applications + R&D Topics
- Performance Tips & Tricks
- Summary
- Q&A

Legal

Why Raster Ordered Views?

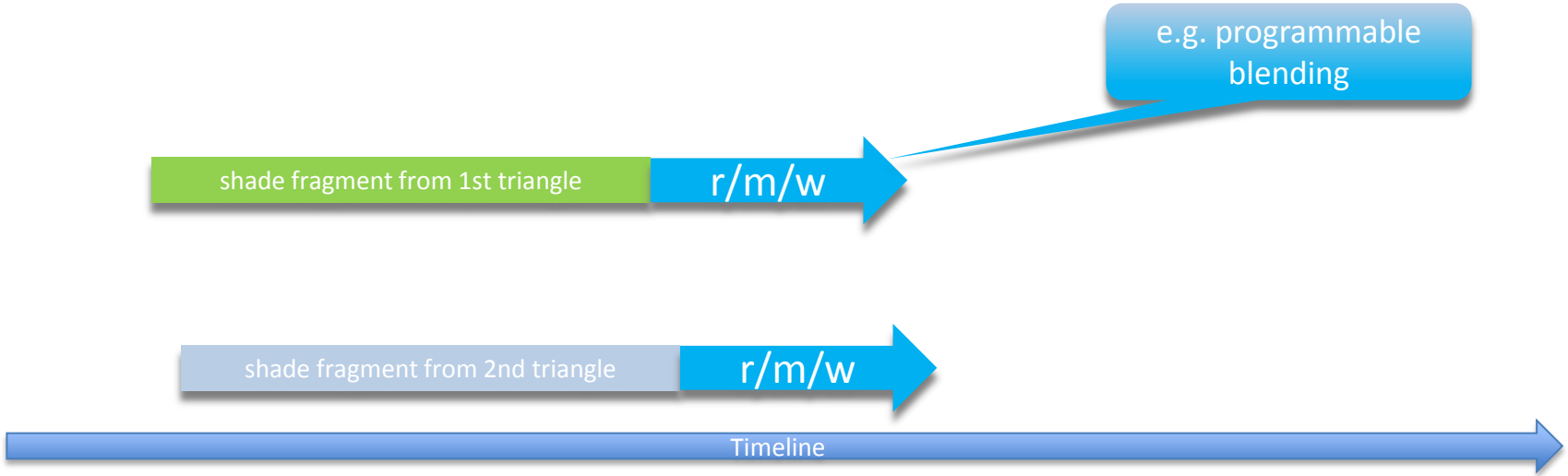
- The DX API specifies “in order” processing rules for the raster pipeline
- If two triangles sent to the GPU touch the same XY screen location, the GPU hardware guarantees that triangle “A” will blend its color result before “B” blends it.
- Hardware in the ROP is responsible for enforcing this ordering requirement.
- Pipeline back-end* still not programmable, color, z & stencil operations from a fixed menu..
- DX11 added Unordered Access Views, leverage power of shaders

DX Pipeline



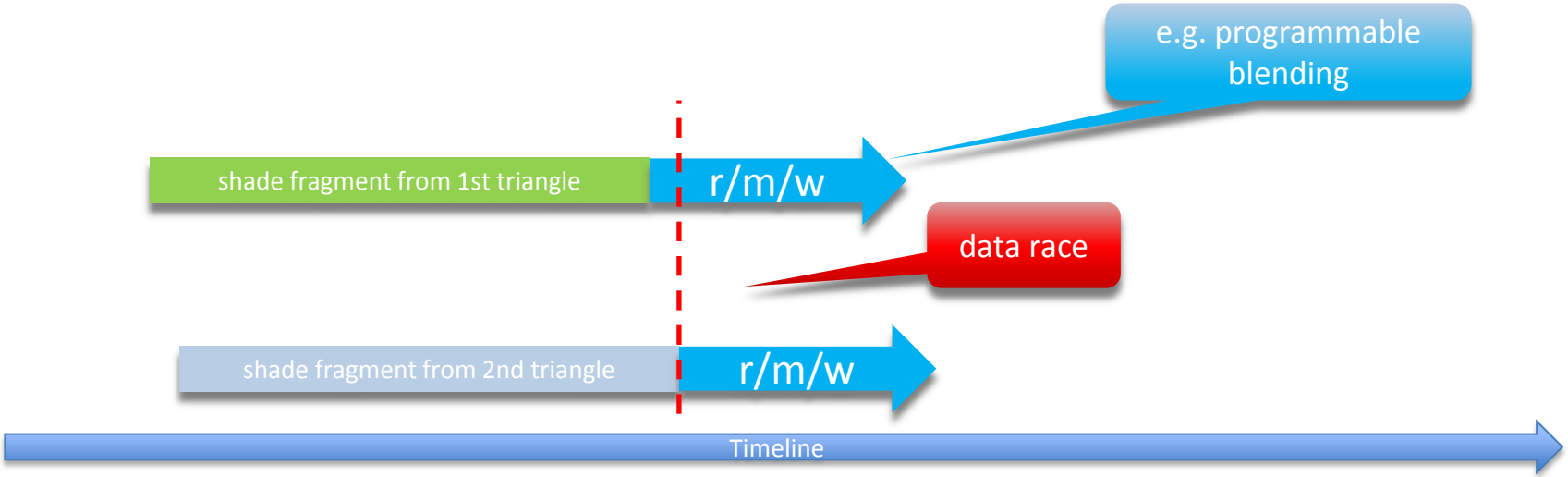
UAV Limitations

- UAV's enable arbitrary R/W memory ops from a pixel shader but no ordering of data input…



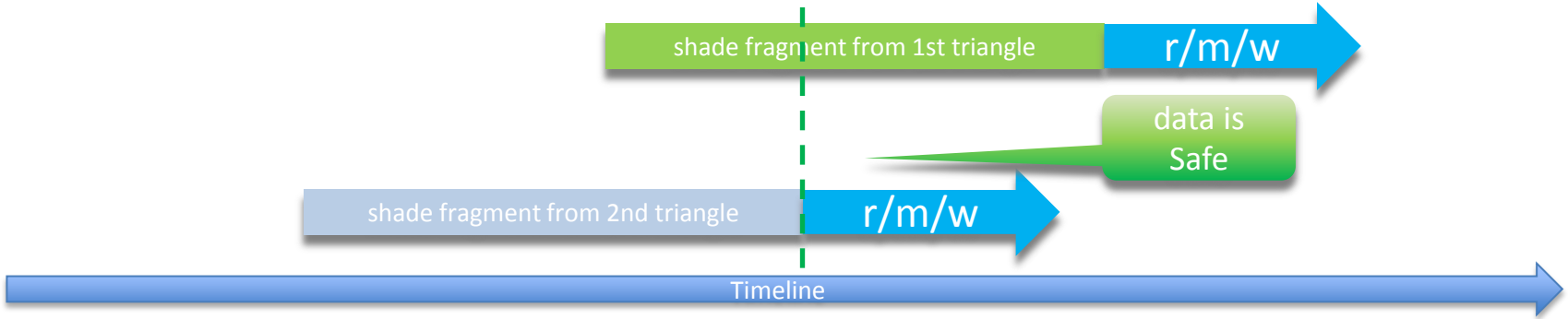
UAV Limitations

- UAV's enable arbitrary R/W memory ops from a pixel shader but no ordering of data input…
- Fragments mapping to same pixel can cause data races



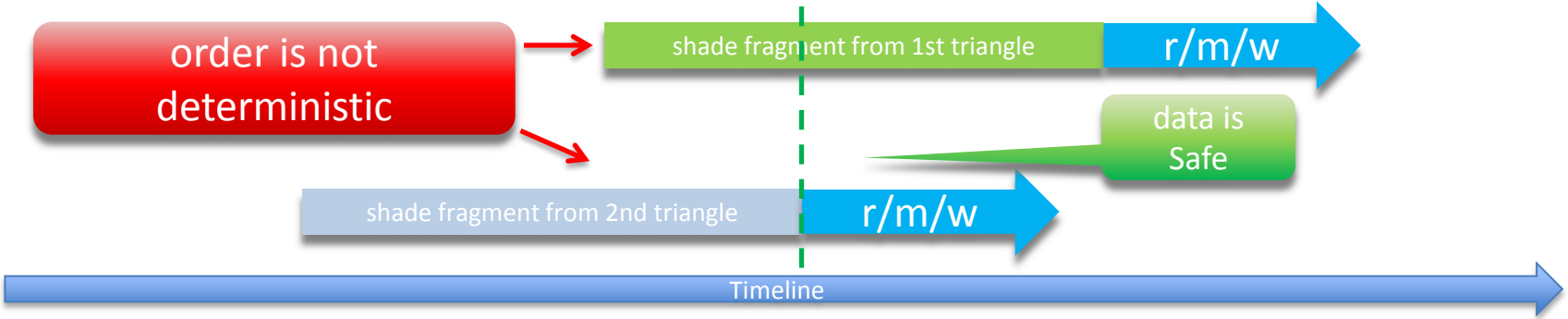
UAV Limitations

- UAV's enable arbitrary R/W memory ops from a pixel shader but no ordering of data input…
- Fragments mapping to same pixel can cause data races



UAV Limitations

- UAV's enable arbitrary R/W memory ops from a pixel shader but no ordering of data input…
- Fragments mapping to same pixel can cause data races
- Fragments can be shaded out-of-order, can't support order-dependent algorithms



Out-of-Order Artefacts

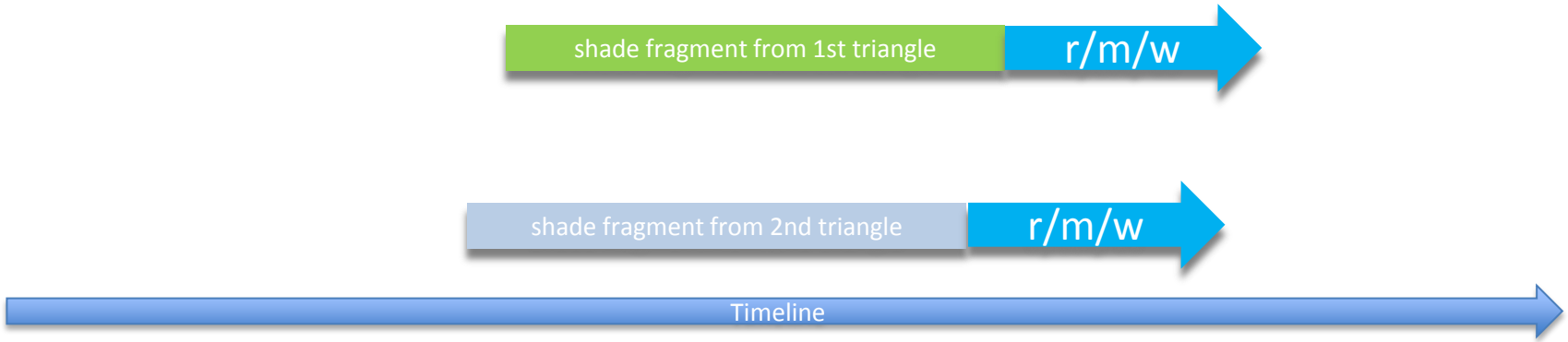
- The more parallel the GPU the greater race conditions

Programmable Back-End

- We need the hardware to detect dependencies among fragments writing to the same X,Y screen coordinate and..
  - Avoid data races
  - Guarantee primitive order for R/M/W operations

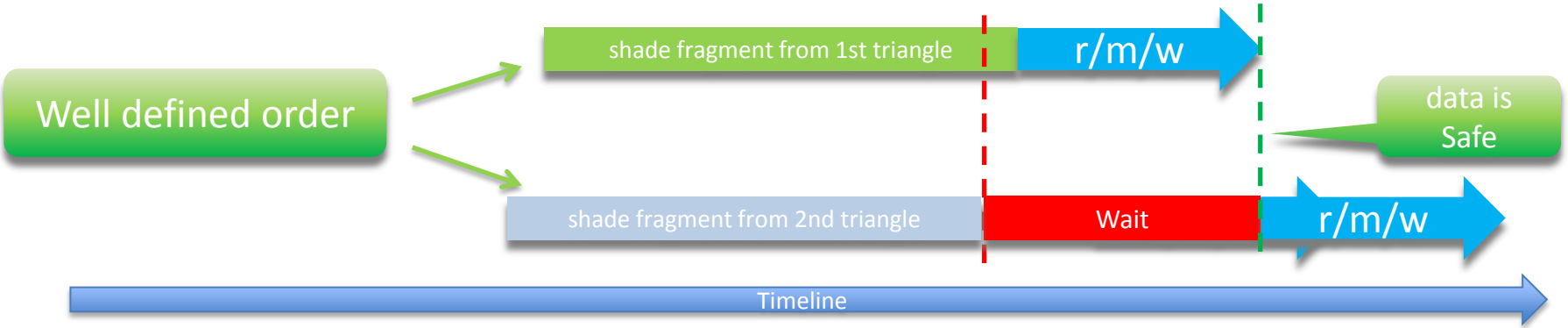
Programmable Back-End

- That's easy what happens if fragment 2 runs first?



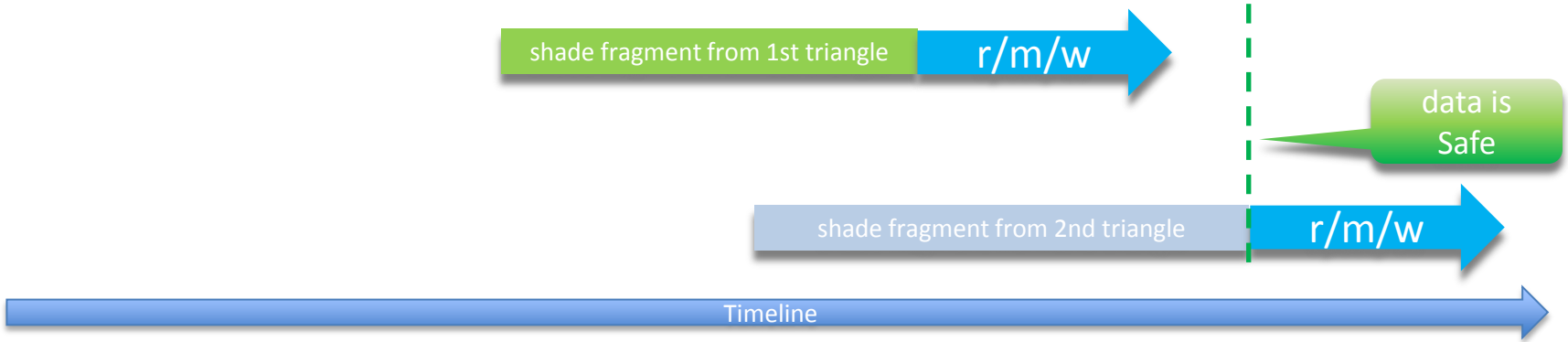
Programmable Back-End

- That's easy what happens if fragment 2 runs first?
- We just get a longer wait☺
- This isn't "just" a critical section



Programmable Back-End

- Little to no performance impact in most cases
  - Fragment 1 completes before Fragment 2 hits barrier
  - Fragments don't touch same X,Y screen Coordinate



Haven’t we seen this before?

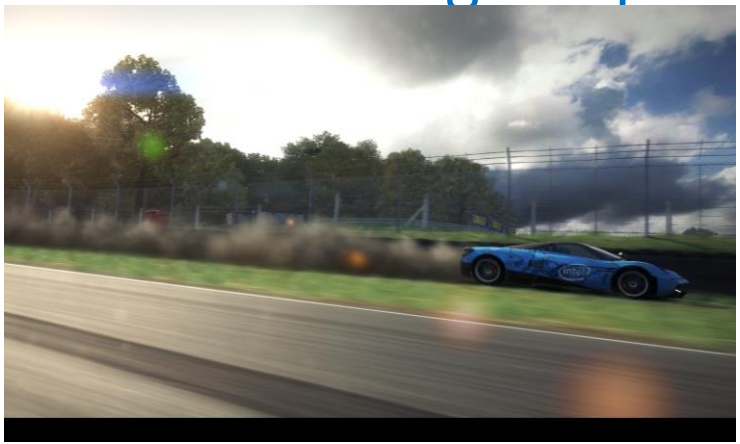
“We have a pretty long list of all the stuff we typically go through and then talk to them with, but ***one very concrete thing we’d like to see, and actually Intel has already done this on their hardware, they call it PixelSync, which is their method of synchronizing the graphics pipeline in a very 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 and* ***I would like to see AMD and NVidia implement something similar to that as well.*** *It's also very power-efficient and efficient overall on Intel's hardware, so I guess the challenge for NVidia and AMD would be if they were able to do that efficiently because they have a lot of the different architectures there.* ***So that's one thing.”***

Johan Andersson, DICE (2013)

Source: http://www.tomshardware.com/reviews/johan-andersson-battlefield-4-interview,3688-2.html

Raster Ordered Views == Pixel Sync

- Almost perfect match for Intel Pixel Synchronization extension introduced on 4th Gen Core processors in 2013.
- Lots of existing samples and even games using it!



Rasterizer ordered views API

- HLSL only construct, modifies access behavior to UAV's
- New (HLSL) objects, only available to the pixel shader:
  - RasterizerOrderedBuffer
  - RasterizerOrderedByteAddressBuffer
  - RasterizerOrderedStructuredBuffer
  - RasterizerOrderedTexture1D
  - RasterizerOrderedTexture1DArray
  - RasterizerOrderedTexture2D
  - RasterizerOrderedTexture2DArray
  - RasterizerOrderedTexture3D
- Used in the same manner as other UAV objects

ROV's vs Pixel Sync

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
}
```

UAV Refresher

- Writing to a UAV disables both Early-Z and Hi-Z
- Declaring [**earlydepthstencil**] in front of a pixel shader guarantees the depth test happens early even if the shader writes to a UAV
- [**earlydepthstencil**] means Depth is updated even if you discard in the pixel shader, unless completely disabled

Applications

Programmable Blending Applications

- New blending operators, non-linear color spaces, exotic encodings, etc.
  - e.g. RGBE, LogLuv, etc.

Intel RGBE blending sample

HDR in to custom R10G10B10A2 in GRID Autosport

Blending on a RGBE color buffer

```
RasterOrderedTexture2D<t> gRGBEBuffer;

void PS_RGBE_Blend (...)
{
        float3 rgb = ...
        float alpha = ...

        uint rgbe = gRGBEBuffer[xy];
        float3 dstRGB = RGBE_to_RGB(rgbe);
        dstRGB = alpha * rgb + (1 – alpha) * dstRGB;
        gRGBEBuffer[xy] = RGB_to_RGBE(dstRGB);
}
```

Ordered UAV structure

Compute fragment color & alpha

Wait on access

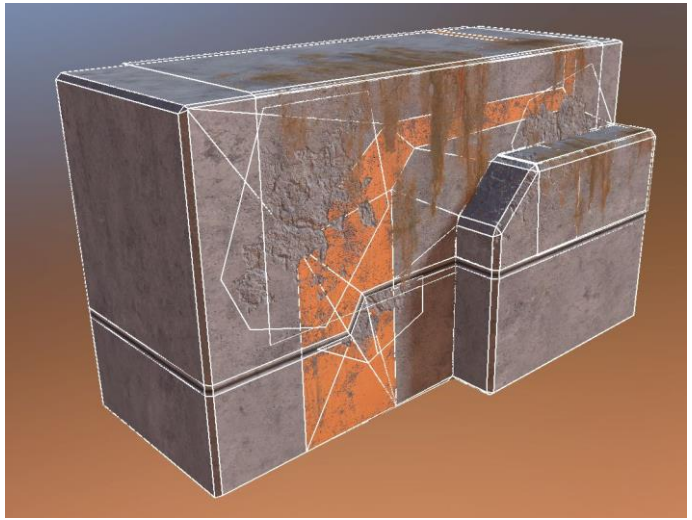
Read RGBE buffer & convert to RGB

Alpha-blending in RGB space

Conversion to RGBE & buffer write

Blending for deferred shaders

- To apply a bullet hole or an axe mark…
- simply
  - Render your G-Buffer
  - Take a normal map of a decal
  - Blend it with the G-Buffer
  - Result will be a correctly mapped bullet hole



K-Buffer

- Generalization of the Z-Buffer*
- Render N-layers of the image in a single pass
- Under DX11 requires Per pixel linked lists and append buffers
  - Unbounded memory requirement
  - Memory bandwidth heavy
- Countless applications:
  - Depth-peeling
  - Constructive solid geometry
  - Depth-of-field & motion blur
  - Volume rendering
  - <insert your idea here ☺>

K-Buffer: Single-Pass Depth Peeling

Compute fragment color, z, etc..

Read N fragments from K-buffer

Write N fragments to K-buffer

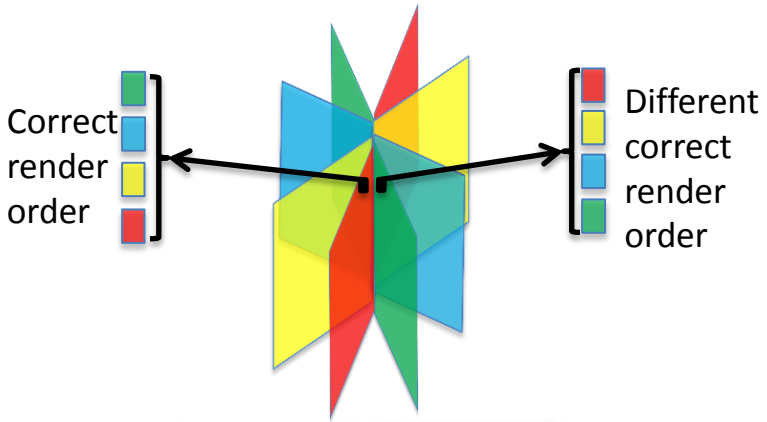
```
RasterizerOrderedBuffer gBuffer;
void PSMain(...)
{
        Fragment frag = {...};

        Fragment fragArray[N] = gBuffer[xy];
        for (int i = 0; i < N; i++) {
                if (frag.Z < fragArray[i].Z) {
                        Fragment temp = frag;
                        frag = fragArray[i];
                        fragArray[i] = temp;
                        }
                }
        gBuffer[xy] = fragArray;
}
```

Enable pixel synchronization

Bubble sort (1 pass)

Order-Independent Transparency

- Correct compositing, rendering foliage & fences with zero aliasing etc..
- DX11-style order-independent transparency has significant drawbacks
  - Requires unbounded memory (per-pixel lists)
  - Not so great performance due to global atomics, fragments sorting, etc.



Order-Independent Transparency

- Raster Ordered View enable a new approach
  - Single geometry pass and fixed memory requirements
  - Stable and predictable performance
  - Scalable: easily trade-off image quality for performance/memory

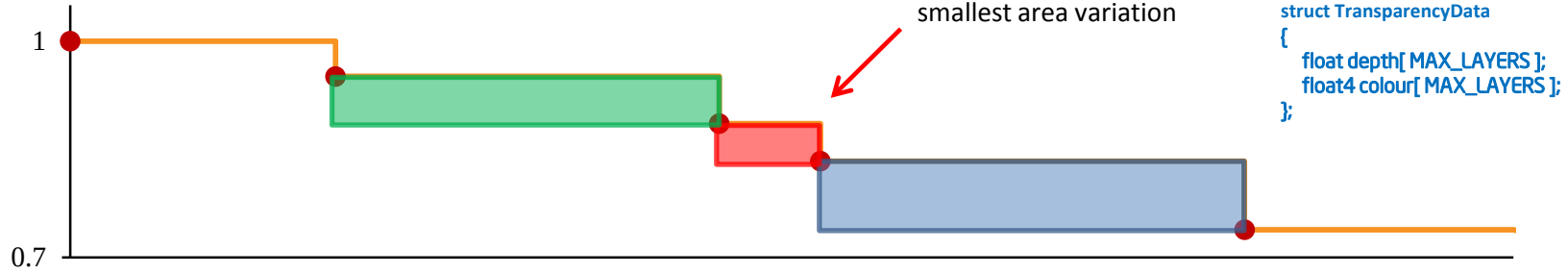
GRID Autosport 2014
OIT on 15watt 4th Gen Core™ Processor

Multi-Layer Alpha Blending

- Step 1: Improve alpha-blending
  - Use depth to decide whether to composite incoming fragment over or under
  - Much better than vanilla alpha-blending but in some cases not quite correct
- Step 3: Use more layers to trade-off image quality for perf/memory
  - Sort N Layers, blend furthest fragments
- Step 3:Or use an Adaptive Transparency Function

Adaptive Visibility Function

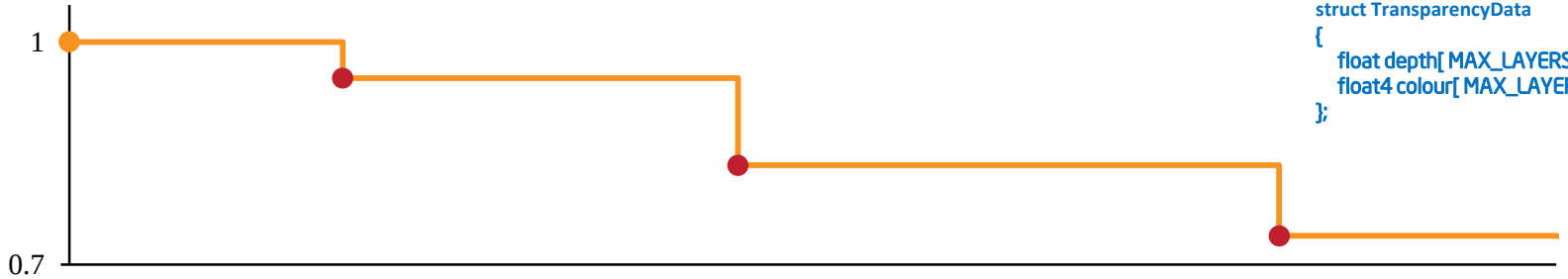
- Store Visibility Function as a sorted fixed-size array of nodes, in UAV surface
- Each ***red*** node corresponds to a pair of values for depth and transmittance: (d,t)
- To compress visibility we remove the node that generates the smallest area variation




```
struct TransparencyData
{
    float depth[ MAX_LAYERS ];
    float4 colour[ MAX_LAYERS ];
};
```

Adaptive Visibility Function

- Store Visibility Function as a sorted fixed-size array of nodes, in UAV surface
- Each ***red*** node corresponds to a pair of values for depth and transmittance: (d,t)
- To compress visibility we remove the node that generates the smallest area variation




```
struct TransparencyData
{
    float depth[ MAX_LAYERS ];
    float4 colour[ MAX_LAYERS ];
};
```

- Used for early Order Independent Transparency samples

Combining OIT with sorted Transparency

- Only add transparent pixels into the OIT structure if there is already OIT data in the buffer
  - Store mask of pixels in R32 target to flag pixels that contain OIT data
  - Use the mask to identify pixels where traditional transparency needs to be merged with OIT
  - All other pixels are rendered with normal alpha blending, because there is no overlap

Hair Rendering

Alpha Blending

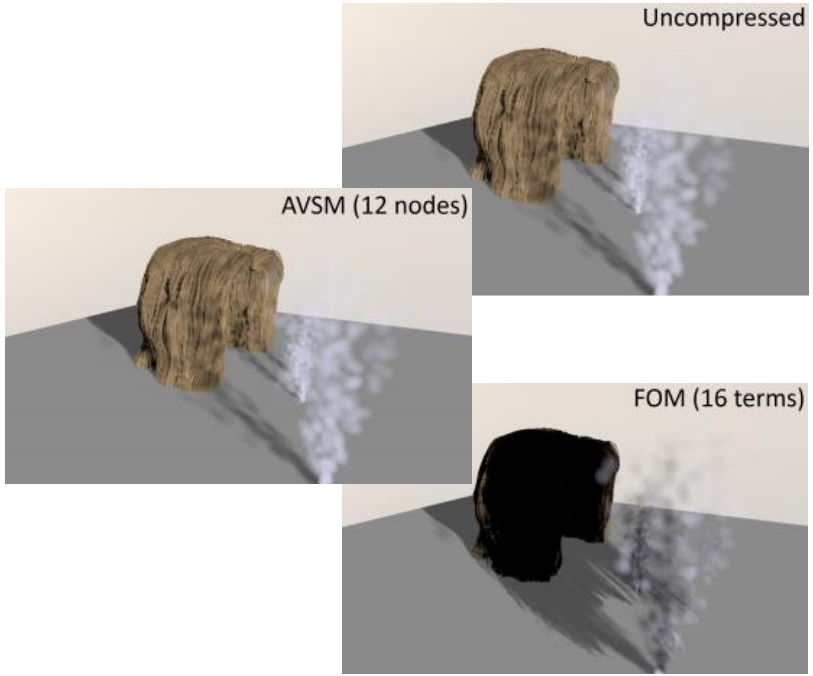
Adaptive Transparency

Adaptive Volumetric Shadow Maps

- Like Deep Shadow Maps but designed for real-time rendering
- Encode per-pixel visibility function from light point-of-view
- Lossy compression of the visibility data
- Raster Ordered Views enables first fixed memory implementation of AVSM

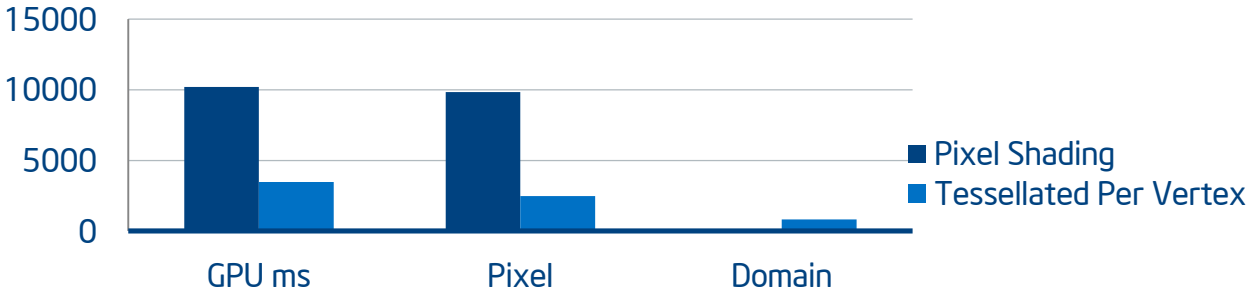
AVSM Quality Comparison

- AVSM with 8 to 12 nodes closely matches ground truth results
- 4 Nodes suitable for many situations, used in GRID 2.
- Fourier Opacity Maps suffer from artefacts generated by high-frequency light blockers like hair and sub-optimal depth bounds
- Hair model courtesy of Cem Yuksel.



Accelerating AVSM

- Overdraw of particles means even small increases in per-pixel costs are noticeable



- Shadow map generation can use simplified particles
  - Less particles, larger area, higher opacity
- Per Pixel lighting of particles took >=10ms…
  - Tessellated vertex lighting actually looked better!
  - Tessellation is 2-3x faster than per-pixel

Volume-Aware Blending

- Billboard sprites representing clouds volumes
- Composited using Alpha blending
- Popping Artefacts as billboard sprites change order

Volume-Aware Blending

- Billboard clouds using volume aware blending
- No popping
- Smooth blending of intersecting volumes

Volume-Aware Blending

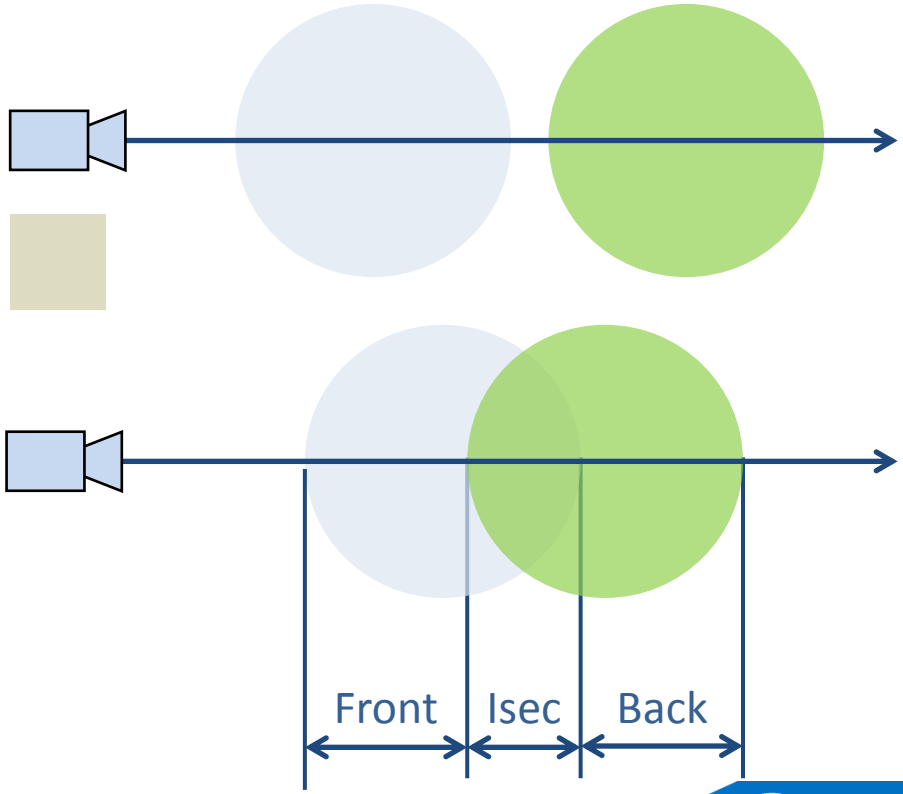
Blending volumetric particles

- If particles do not overlap, blending is trivial
- How can we correctly blend overlapping particles?
- Final color and transparency:

$$T_{Final} = T_{Front} \cdot T_{Isec} \cdot T_{Back}$$

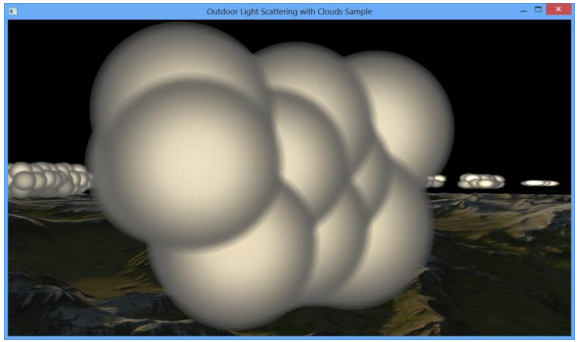
$$C_{Final} = \frac{C_{Front} + C_{Isec} \cdot T_{Front} + C_{Back} \cdot T_{Front} \cdot T_{Isec}}{1 - T_{Final}}$$

- Division by $1 - T_{Final}$ because we do not want alpha pre-multiplied color

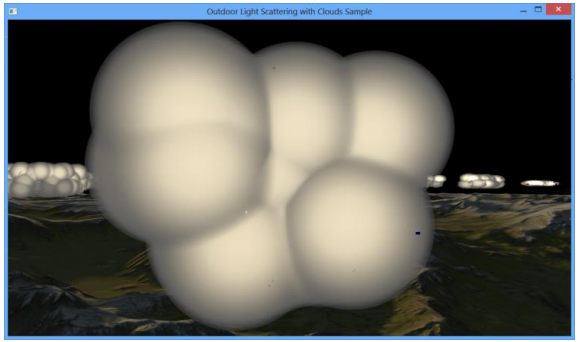


Volume-Aware Blending

- Blending volumetric particles - Implementation
  - Color, density and min/max extent of the current particle are stored in the UAV buffer
  - Each new particle is tested against the currently stored
    - If new particle is in front of the current, the current is blended into the back buffer and replaced with the new one
    - If new particle overlaps with the current, they are blended and stored
    - Particles need to be sorted



Alpha Blended



Volume Blended

R&D Topics

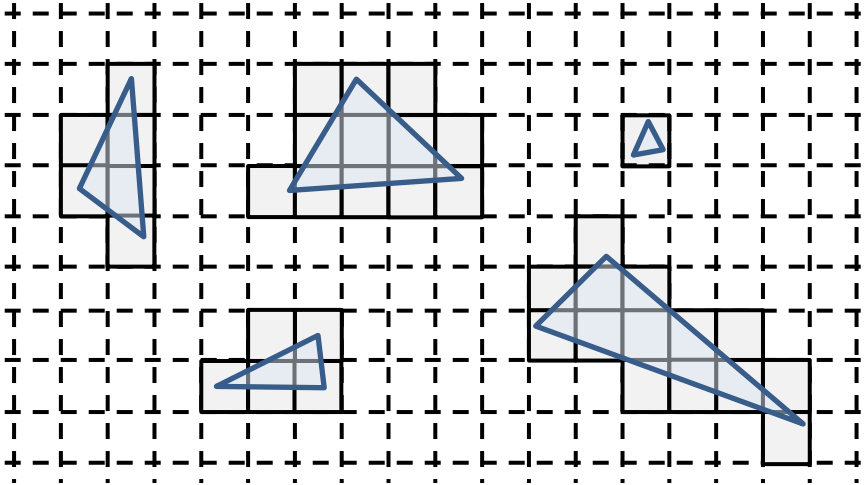
Advanced Anti-Aliasing

- Use Raster Ordered Views to improve or replace multi-sampling anti-aliasing
  - Higher image quality vs. lower memory requirements vs. better performance

- Z^3 anti-aliasing* (1999)
  - Originally developed as HW based high-quality anti-aliasing algorithm
    - Store N fragment per pixel (z, ∂z/∂x, ∂z/∂y, color, coverage)
    - Merge fragments (lossy)

Advanced Anti-Aliasing

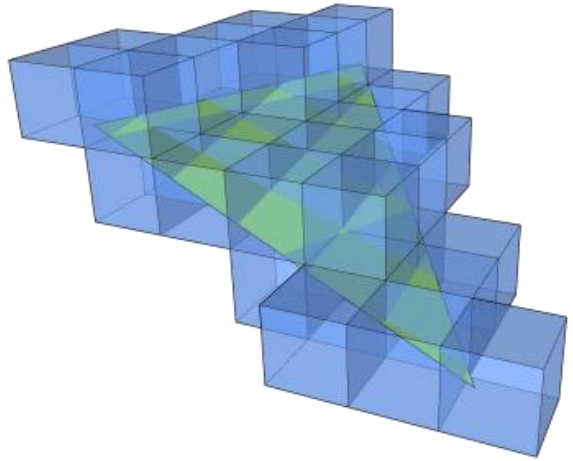
- Analytic methods
  - Render scene using conservative rasterization
  - Build per-pixel spatial subdivision structure using primitive edges
  - Compute fragment weights from fraction of pixel area covered and resolve
  - Think targeted OIT



conservative rasterization renders any pixels that are even partially covered are rasterized

Voxelization

- Build complex per-voxel data structures on the GPU at voxelization time
  - e.g. direction-dependent representations (anisotropic voxels, etc.)
- Voxelization via 2D rasterization projects triangles to XY, YZ or XZ plane

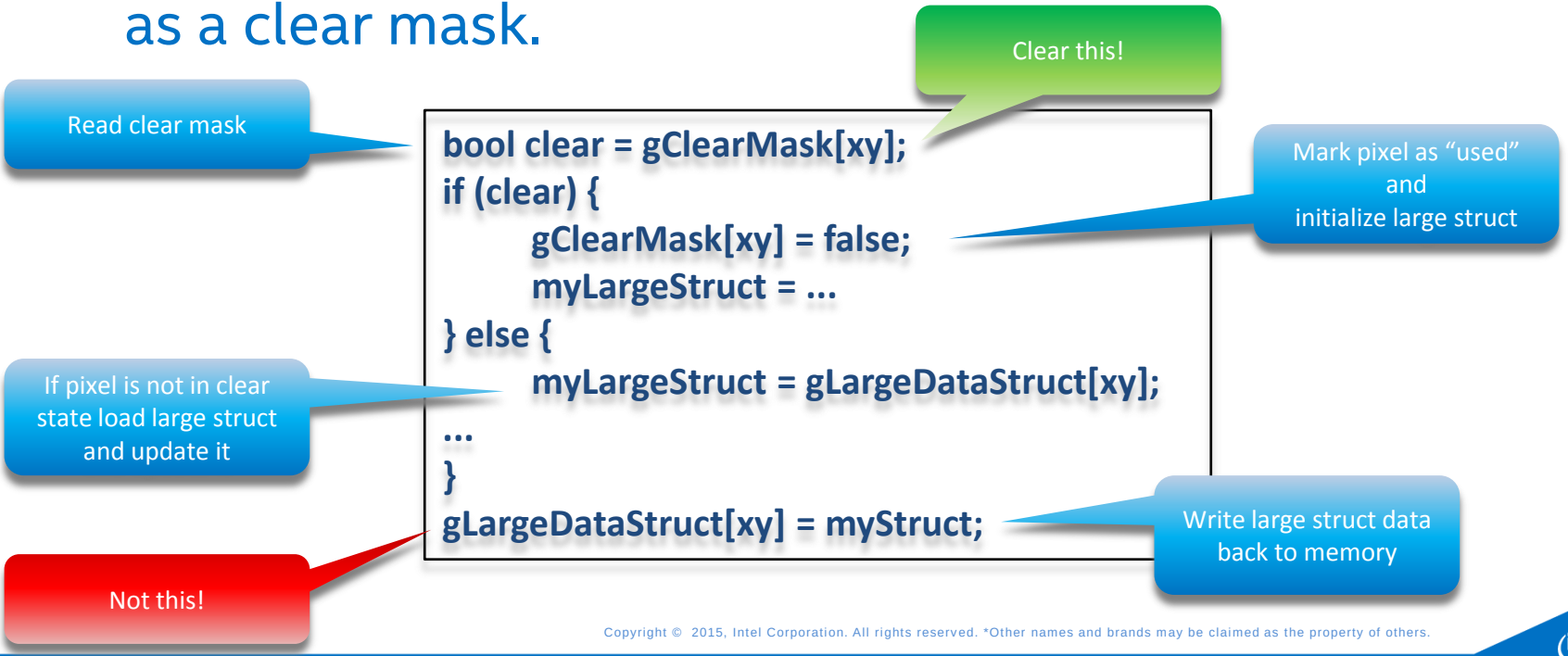


Voxelization

- Use Raster Ordered Views to build 3D data structures at voxelization time, think Depth Peeling
  - Problem: fragment dependencies cannot be tracked over multiple 2D planes in a single render call.
- Easy fix: voxelize onto one 2D plane at time
  - 3 draw calls per mesh, one per 2D plane (i.e. reject triangles that map to other planes)
- Number of generated voxels doesn't change & more flexible than using global atomics
- Can be further optimized by Conservative Rasterization

Performance Tips & Tricks

- Don't clear large buffers. Clear a small buffer and use it as a clear mask.



Balance data size vs Packing

Small(er) data structures can improve performance

Use more instructions to pack/unpack data

Tile Data

- Address 1D structured buffers as tiled to better data exploit locality
- e.g. 1x2 or 2x2 (2D textures), 2x2x2 (voxels), etc..
- Saved 1ms in Grid 2 by Codemasters in OIT.

```
uint AOITAddrGen(uint2 addr2D, uint surfaceWidth)
{
#ifdef AOIT_TILED_ADDRESSING

        surfaceWidth         = surfaceWidth >> 1U;
        uint2 tileAddr2D  = addr2D >> 1U;
        uint  tileAddr1D  =
                (tileAddr2D[0] + surfaceWidth * tileAddr2D[1]) << 2U;
        uint2 pixelAddr2D = addr2D & 0x1U;
        uint  pixelAddr1D = (pixelAddr2D[1] << 1U) + pixelAddr2D[0];

        return tileAddr1D | pixelAddr1D;
#else
        return addr2D[0] + surfaceWidth * addr2D[1];
#endif
}
```

Linear address

Synch Late

- Prefer inserting the synchronized UAV read in the second half of the shader
- Increase likelihood of concurrently shading fragments that map to the same pixel


```
RasterOrderedTexture2D<t> gRGBEBuffer;

void PS_RGBE_Blend (...)
{
    uint rgbe = gRGBEBuffer[xy];

    float3 rgb = ... Complex stuff....
    float alpha = ... Potentially can run
    concurrent with other shaders

    uint rgbe = gRGBEBuffer[xy];

}
```

Avoid Synch Altogether!

- Ok not 100% possible but clever organisation of the triangles in can limit sync points
  - Randomize instancing data to avoid consecutive overlapping models.
- Small render targets aren't always your friend, can increase hits to the same pixel.

Summary

- Raster Ordered Views are a new tool that injects new life in the 3D pipeline
- Very simple to use, high performance
- Build complex 3D data structures in bounded memory
- Solve many long standing problems
  - OIT
  - Depth Peeling
  - Volume Rendering and Blending

Questions?

References

- https://software.intel.com/sites/default/files/salvi_avsm_egsr2010.pdf
- http://advances.realtimerendering.com/s2013/2013-07-23-SIGGRAPH-PixelSync.pdf
- https://software.intel.com/en-us/gamedev/code-samples
- http://www.gdcvault.com/play/1020221/Rendering-in-Codemasters-GRID2-and
- https://software.intel.com/en-us/blogs/2014/03/31/cloud-rendering-sample