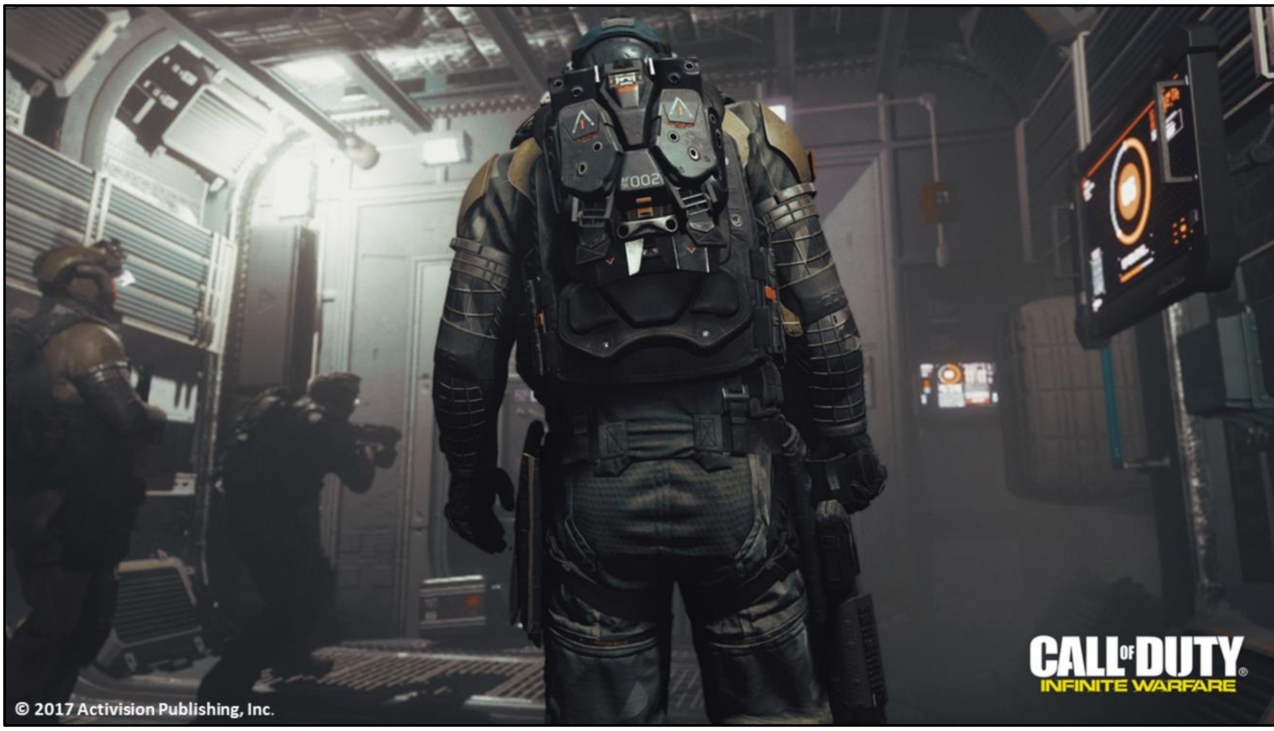
Digital Dragons

Rendering of

CALL OF DUTY
INFINITE WARFARE

Michal Drobot
Principal Rendering Engineer
infinity ward ACTIVISION

This talk will present high level overview of some core rendering components of COD : Infinite Warfare



COD : IW is the latest installment of Call of Duty franchise.
From rendering perspective it was a huge challenge.

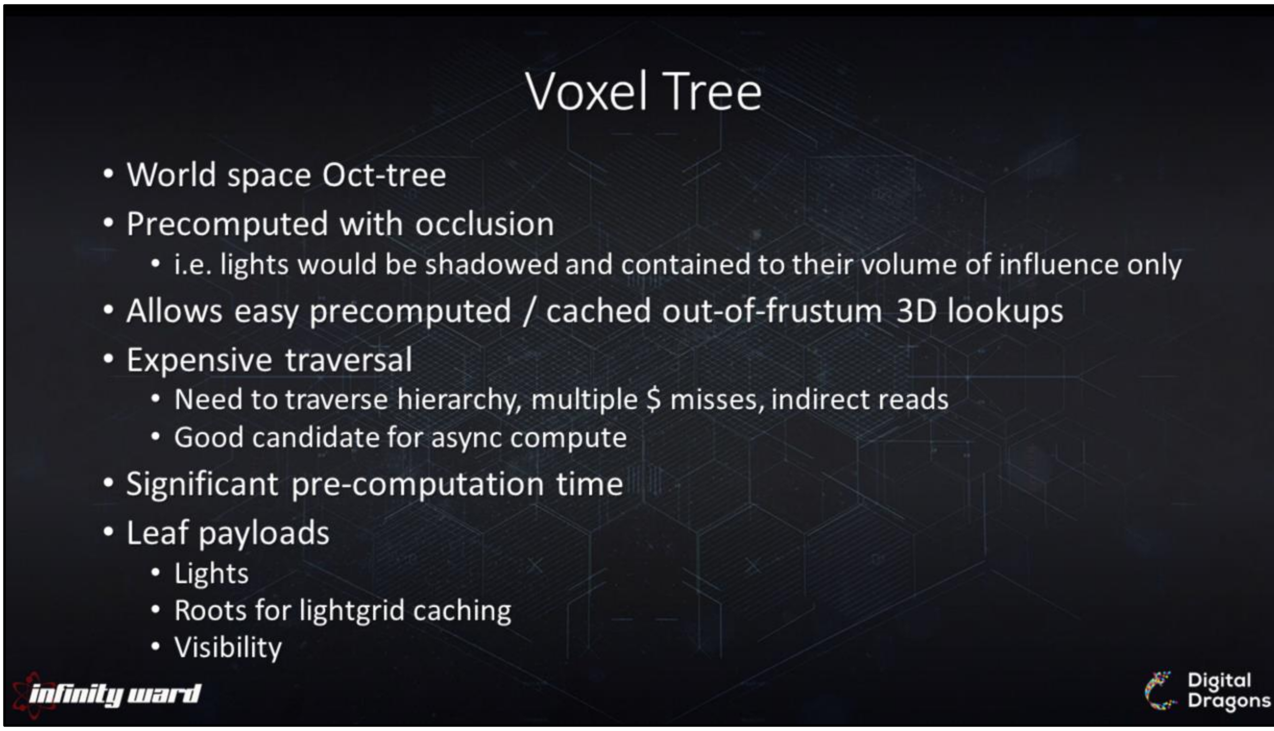
Presentation Outline

1. Forward+ Data Structures
2. Mesh Rendering
3. Shadow Map Cache
4. Particle Lighting
5. Multi-Res Renderer
6. Reflections & Refractions
7. Volumetric Renderer
8. Texture Packer

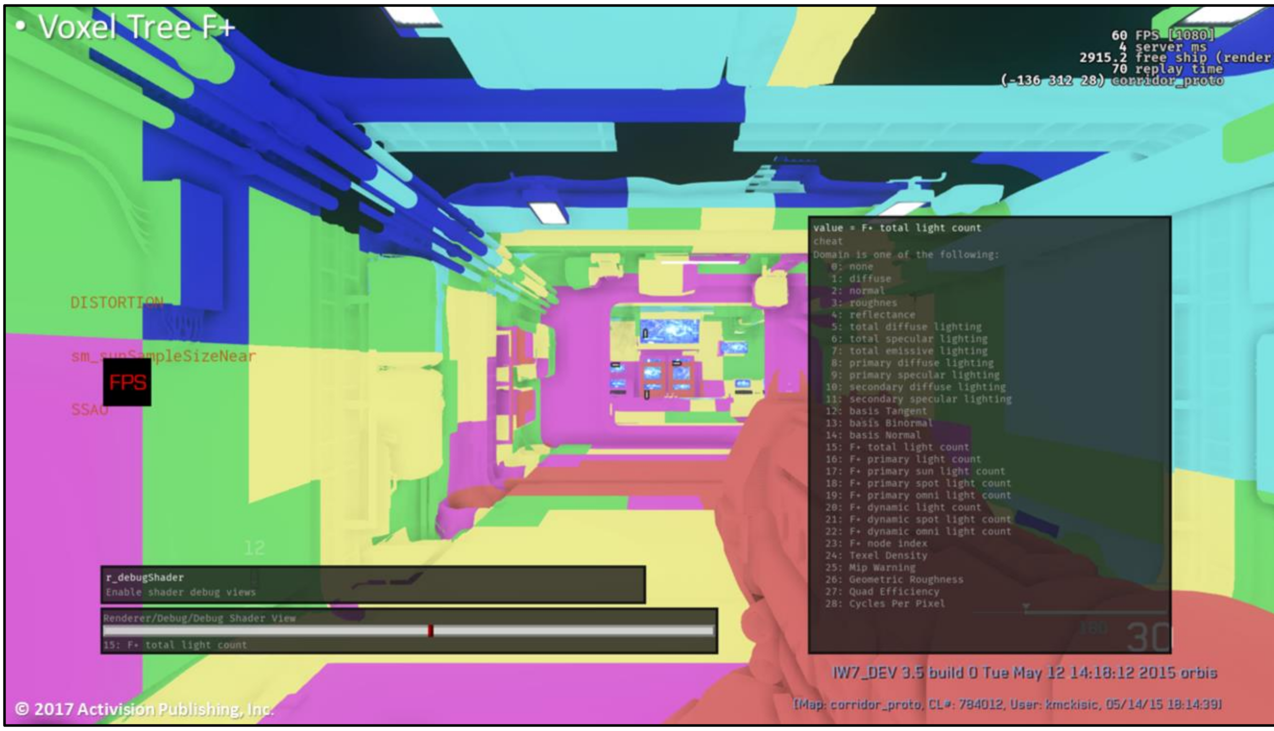
infinity ward

Digital Dragons

Forward+ Data Structures



Used for out of frustum 3D lookups for : lights in dynamic reflection probes, lights for dynamic lightmapped particles, tetrahedron Global Illumination lightgrid



Colors represent amount of lights hitting each pixel. Each voxel stores preculled lights. It is shown here to demonstrate how world space voxels are visualized on surfaces (not used for actual scene lighting).

Frustum Space

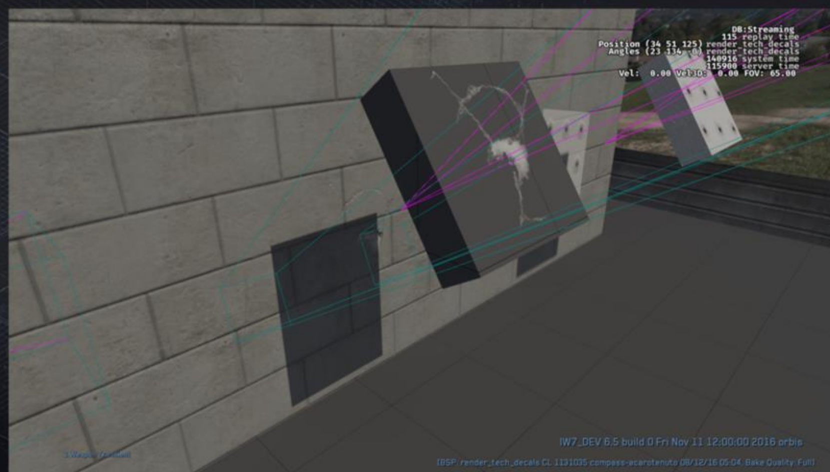
- Tiled based bitmask
  - 8x8 pixel size
  - Used for opaque geometry
- Cluster based bitmask
  - Size to match 4x4x4 kernels from volumetrics
    - 160/4 * 90/4 * 128/4 = 40 x 25 x 32 @1080p
  - Used for transparent geometry and volumetrics

infinity ward

Digital Dragons

Frustum Space

- Items indexed by bits
  - Lights
  - Reflection Probes
  - Density Volumes
  - Decals [0]



infinity ward

Digital Dragons


Mesh Rendering
infinity ward
Digital Dragons


Geometry overview
© 2017 Activision Publishing, Inc.

Smodels / Xmodels

- Static Models / Dynamic Models
- Similar to standard game engine meshes
- Used for props, characters, vehicles, weapons etc.

BSP

- Radiant brush based geometry
- Blocking out levels
- Terrain
- Static Structural parts of environments
- Multiple brushes with individual materials get merged together into optimized sub-meshes and sub-shaders.
- Allows unique detailing of the world at high performance
- Support Tessellation & Displacement mapping

Wireframe of base BSP

Good candidate for physics / AI raycasts. Simple geo, easy to iterate on.

Base BSP with enabled Adaptive Tessellation and Displacement Mapping.
Adaptively tessellate based on displacement deltas, distance to camera, patch angle to camera.
Each generated sub-patch goes through GPU frustum, occlusion and backface culling.


T&D OFF
© 2017 Activision Publishing, Inc.

T&D makes a huge visual impact at moderate adaptive performance hit.
Here exaggerated for visual presentation.


Shadow Map Cache
infinity ward
Digital
Dragons

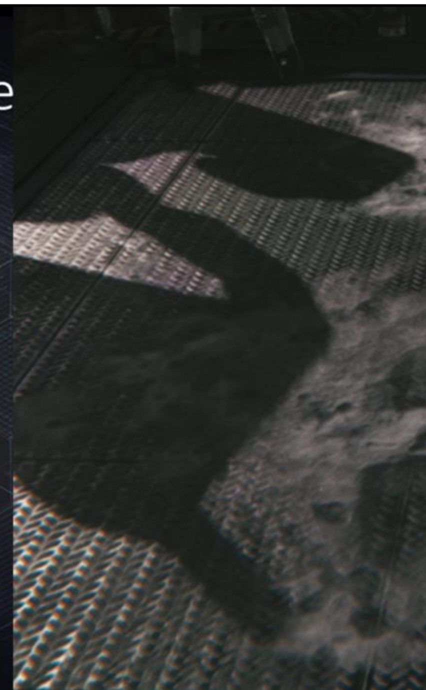
ESM Shadow Map Cache : Motivation

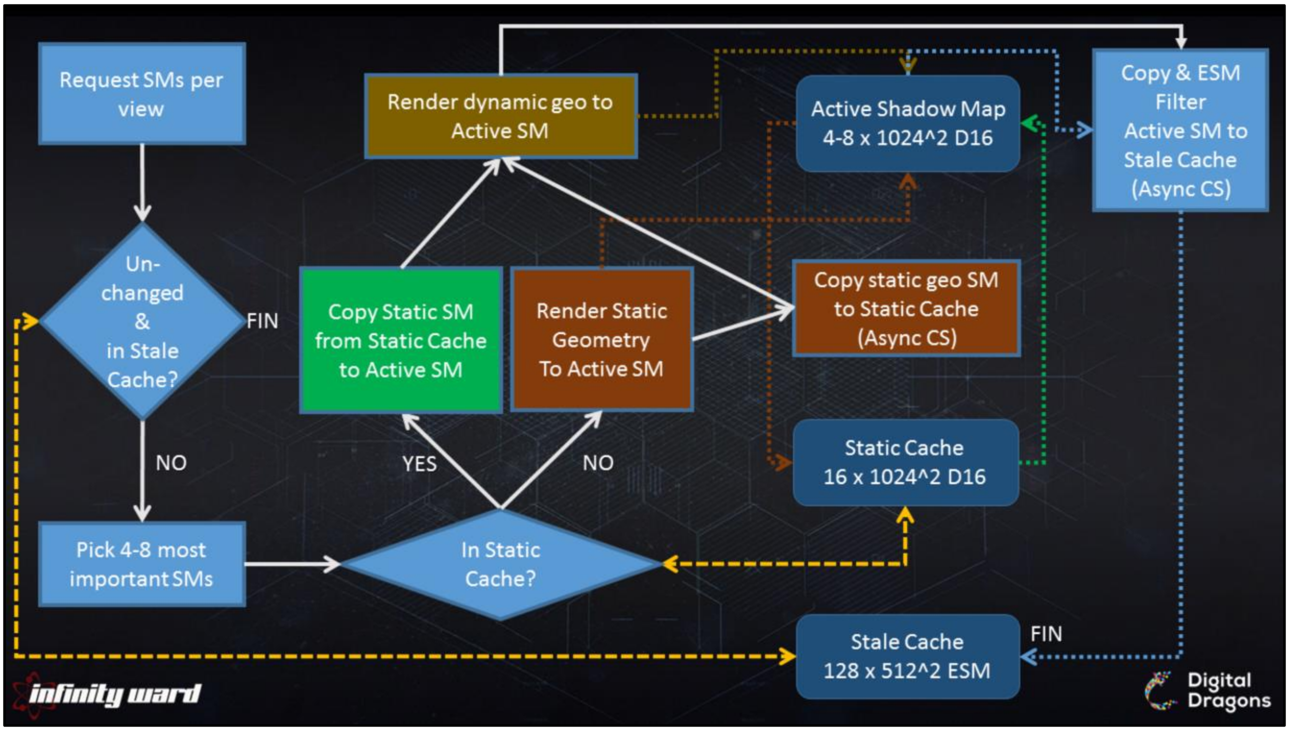
- Tessellation geometry expensive in shadow map rendering
- Majority of lights are stationary
- Many lights
  - < 256 in view frustum
- Many shadows
  - < 128 in view frustum

infinity ward

ESM Shadow Map Cache

- PCF too expensive in F+ (VGPR pressure)
- Emphasis on **static high quality shadowed lights** and **caching**
- Exponential Shadow Maps
  - 512^2 16bit UNORM
  - Downsampled from 1024^2 shadow map
  - 3x3 Gaussian filtered
    - Artistic controls for filtering
  - Pre-filter once and cached





Caching algorithm:
Request SMs per view
  > get all shadow maps visible in view that passed culling tests
Check Stale Cache for SMs that DO NOT need update
  > Is the light resident in Stale Cache?
    > Did the light move in last frame?
    > Did anything move within light frustum in last frame?
    > Was an update forced?
Pick 4-8 most important SMs
  > Sort by priority
    > Artist driven priority (player flashlight etc.)
    > Distance, projected size, intensity
For each picked light
Check Static Cache for SM – static cache hold actual D16 shadow maps that contain only static geometry.
  > Light is cached in Static Cache
    > Copy Static SM from Static Cache to Active
  > Light is NOT cached in Static Cache
    > Render static geometry to Active Cache
    > Copy static geo Shadow Map to Static Cache
Render dynamic geo to Active SM – Active Cache has 4 – 8 D16 shadows maps.
Technically we only need 1, however we overlap multiple async compute jobs from

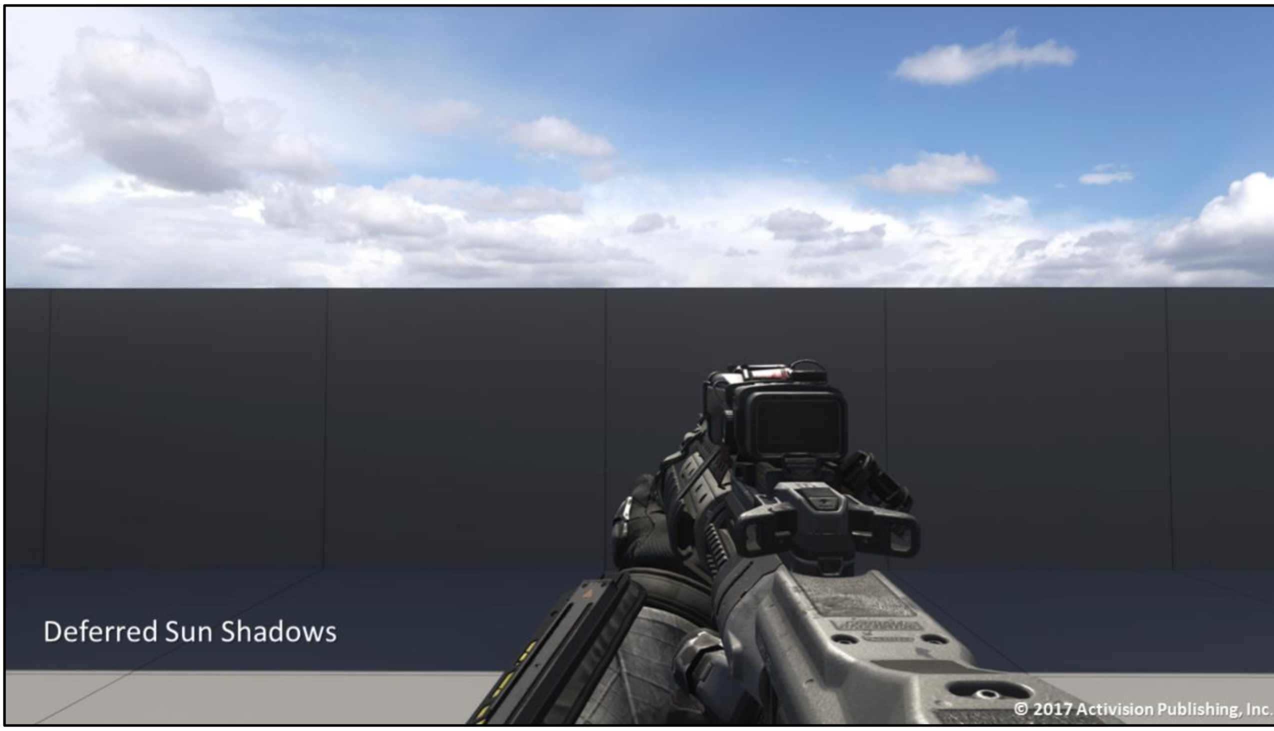
shadow cache system, over rendering of actual shadow maps (i.e. copies, ESM filtering, downsampling, shadow map clears). So CS jobs for shadow map 0 would be overlapped with rendering of shadow map 1.
Copy and ESM process Active SM to Stale Cache

ESM Shadow Map Cache : Performance

- Cache copies and ESM jobs use Async Compute
- Overlapped with ‘next’ shadow map generation work
- Average real cost : < 0.1ms per shadow map exclusive of rendering
- Low sampling cost in Forward+
    - ALU Fully amortized
    - No register (VGPR) impact

infinity ward

Digital Dragons



Need high quality shadows
- Cinematic characters
- View model

Need multiple high resolution object space shadow maps
- Too much pressure on standard Shadow Map Cache
  - High number of active slots needed

Screen Space Shadow
- Do a depth buffer raytrace in direction of the light source

Deferred pass for Sun Only
- Optimized for view model (depth bounds / stencil test)
- Works well if run on whole scene

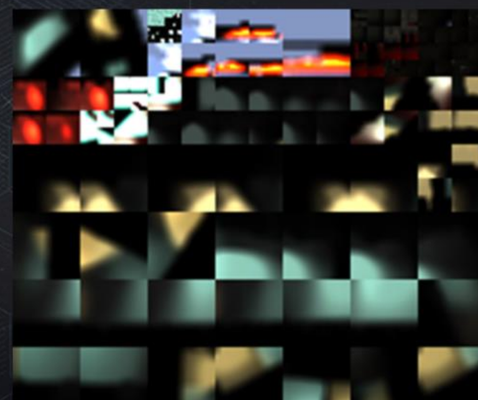
Integrated into F+

- Store strongest light source per-pixel
  - Set by artists as key light or derived from runtime computation as max(intensity0…)
- Perform a single trace in key light direction


Particle Lighting
infinity ward
Digital Dragons

Particle Lighting with lightmaps

- Each quad automatically allocates 1x1 - 32x32 lightmap tile
  - Resolution depends on projected screen space size of quad
- Per each texel
  - Store position for each sampling point
  - CS samples lightgrid for ambient contribution
  - CS primary lighting
    - World Space Voxel Tree
    - Sampling points can be out of frustum bounds
    - Transform and store as RGB SH1

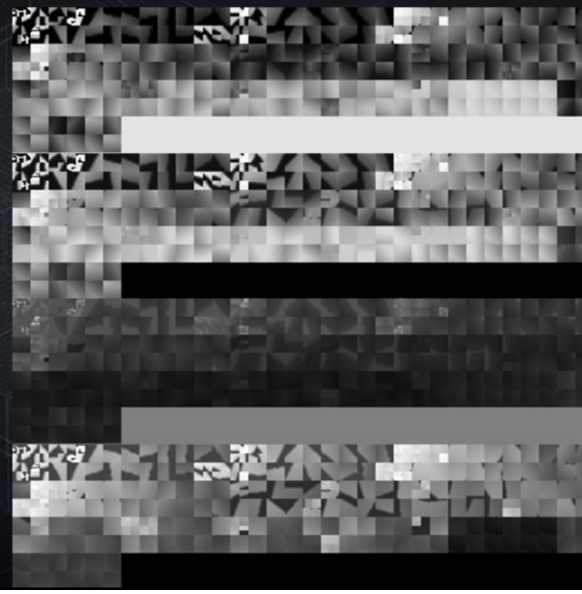


infinity ward

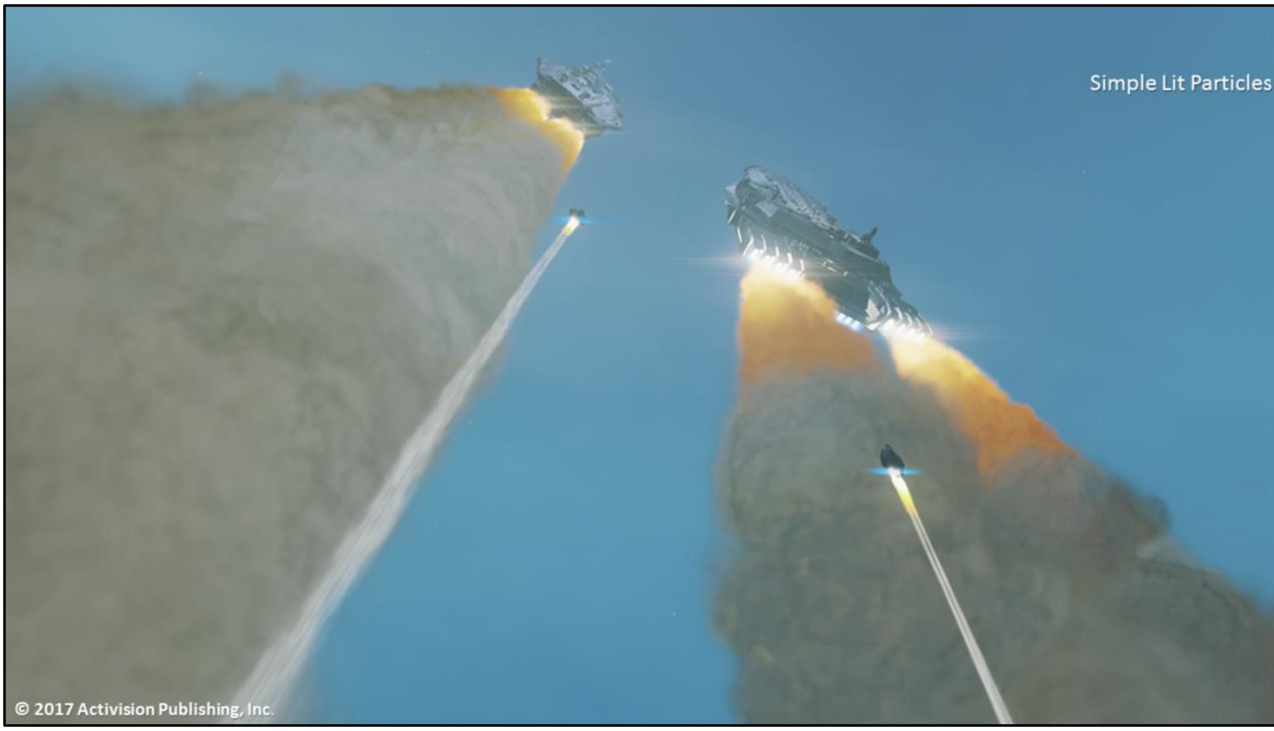
Digital Dragons

Deferred Lightmap

- 512^2 RGB FP11_11_10
  - Omni-directional lighting
  - Simple particles
- 3 x 512^2 RGBA FP16_16_16_16
  - Directional lighting
  - Stored as RGB x SH1
    (4 coefficients stored in RGBA)
  - Complex particles with normal maps
- Normal mapped particles support specular reflections through F+
- Support for
  - VFX impact marks
  - Decal meshes



infinity ward



Simple Lit particles with omni-directional lighting.
Complex lighting scenario. Strong direction sun lights with color tones opposed to bright skydome lighting.
Results in washed out, mixed color flat rendering.

SH deferred lightmap, used with normal mapped particles, correctly separates lighting direction and colors adding a great sense of depth.
This is further improved by our Extinction Shadow Maps used for Sun only.

Blocky magnification artifacts can occur due to relative size differences between particles on screen and lightmap texel size resulting in undersampling.
Lack of light multi-scattering (lightmap stores only primary scattering).
Both issues can be improved by lightmap scattering pass.

Lightmap Lighting Scattering

- Per each tile
  - CS scattering pass
    - Blur to simulate scattering
    - Inverse tonemapping for anti-aliasing
- CS packed and sorted by tile sizes for highest occupancy

In addition (or instead of scattering) implement expensive cubic filtering during particle rendering

- Made per-particle rendering ~10% slower
- Did not ship

Particle Lighting : Performance

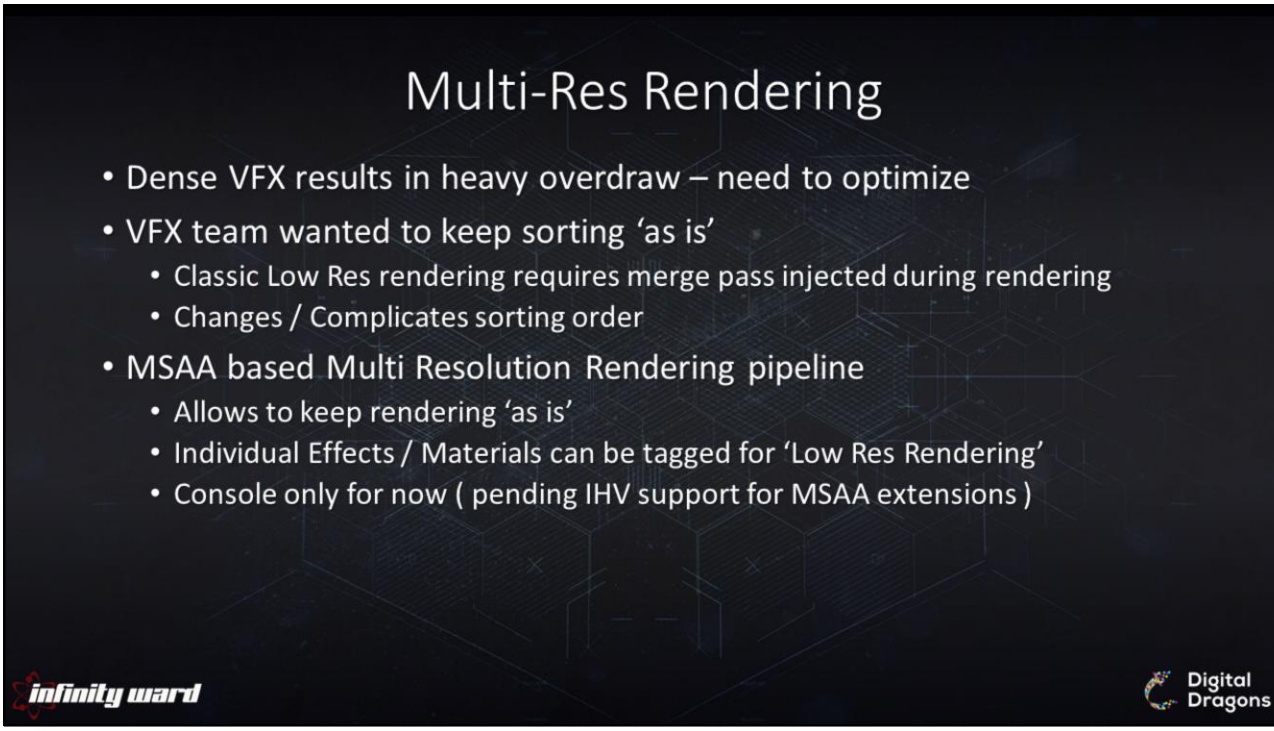
- All processing utilizes Async Compute
  - In most cases amortized over opaque geometry pass

RGB SH1 512 CS Job	Time (ms) @PS4
Lighting	0.1-0.7
Scattering	0.1

infinity ward

Digital Dragons

Multi-Res Rendering

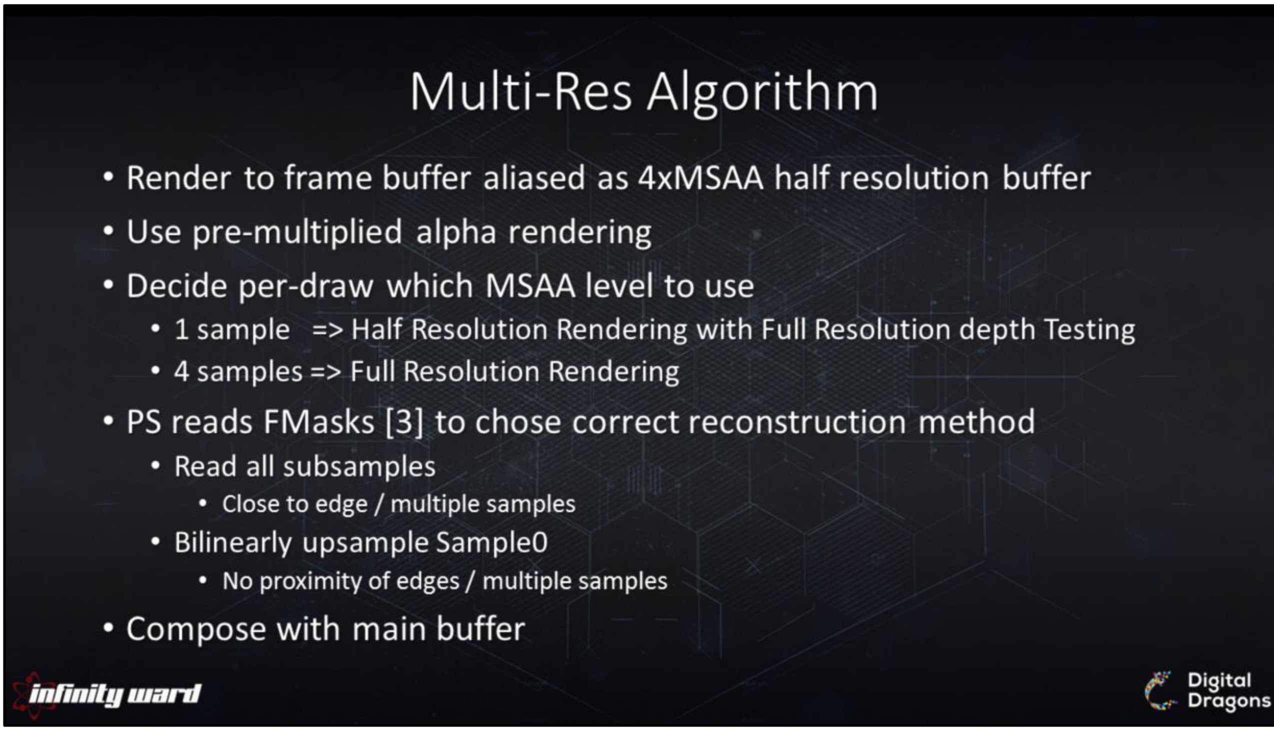


Initially developed for VFX rendering

Full Res : 5ms VFX
© 2017 Activision Publishing, Inc.

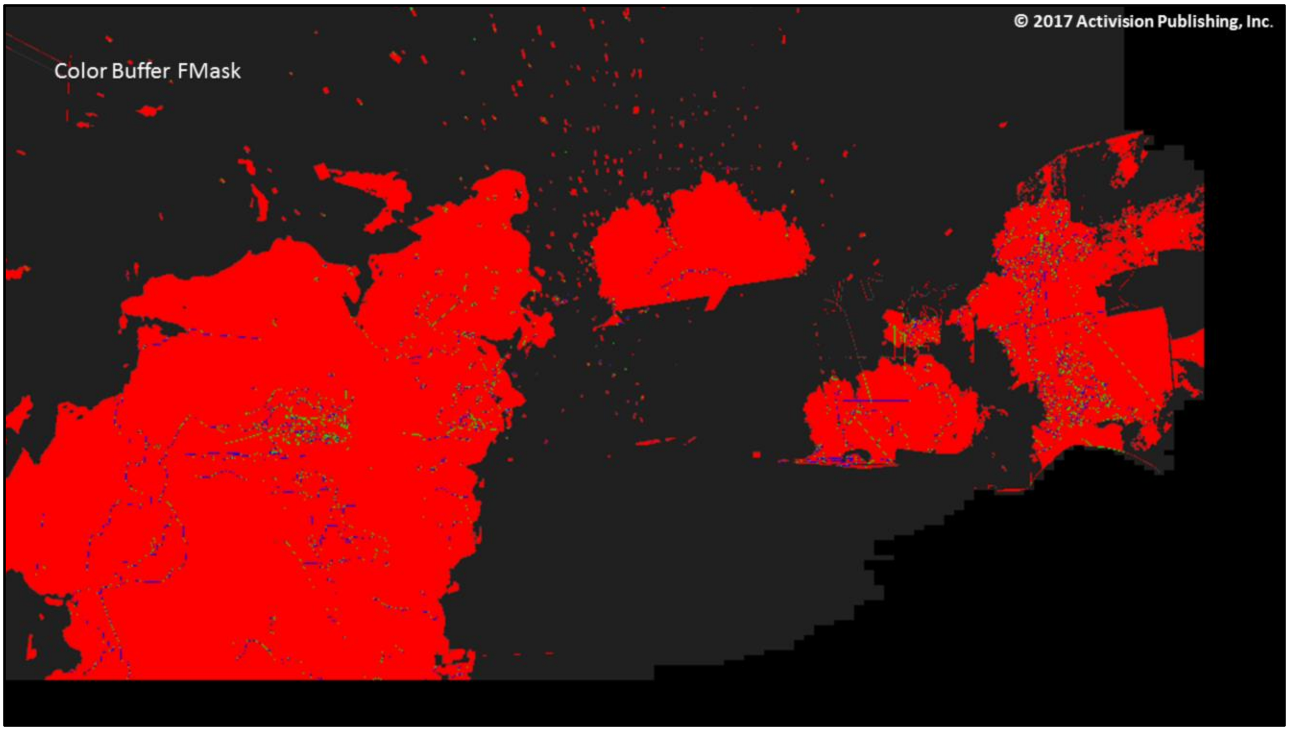

Multi Res : 2.8 ms VFX
© 2017 Activision Publishing, Inc.


Low Res : 2.4 ms VFX
© 2017 Activision Publishing, Inc.

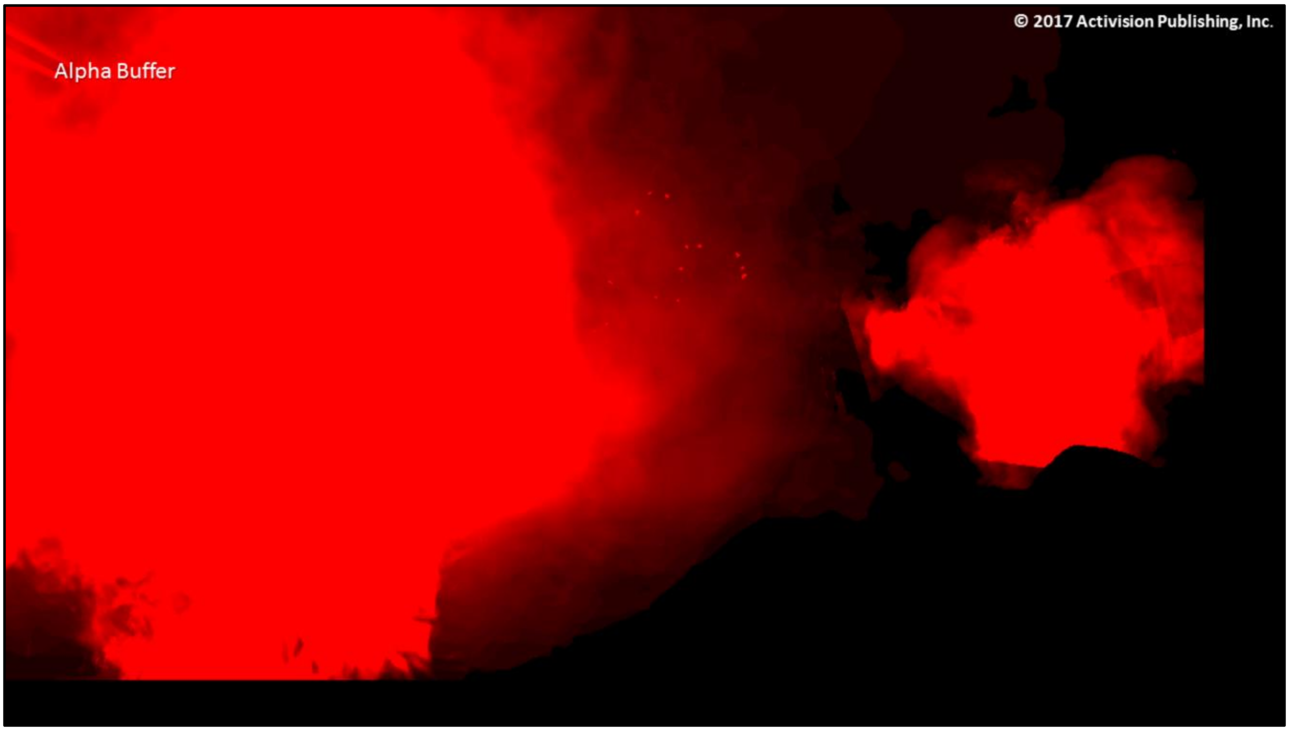


GCN RT formats prevent direct aliasing. Therefore we need to actually re-write and re-swizzle the depth buffer manually in CS.
This step is amortized with other depth related processing.

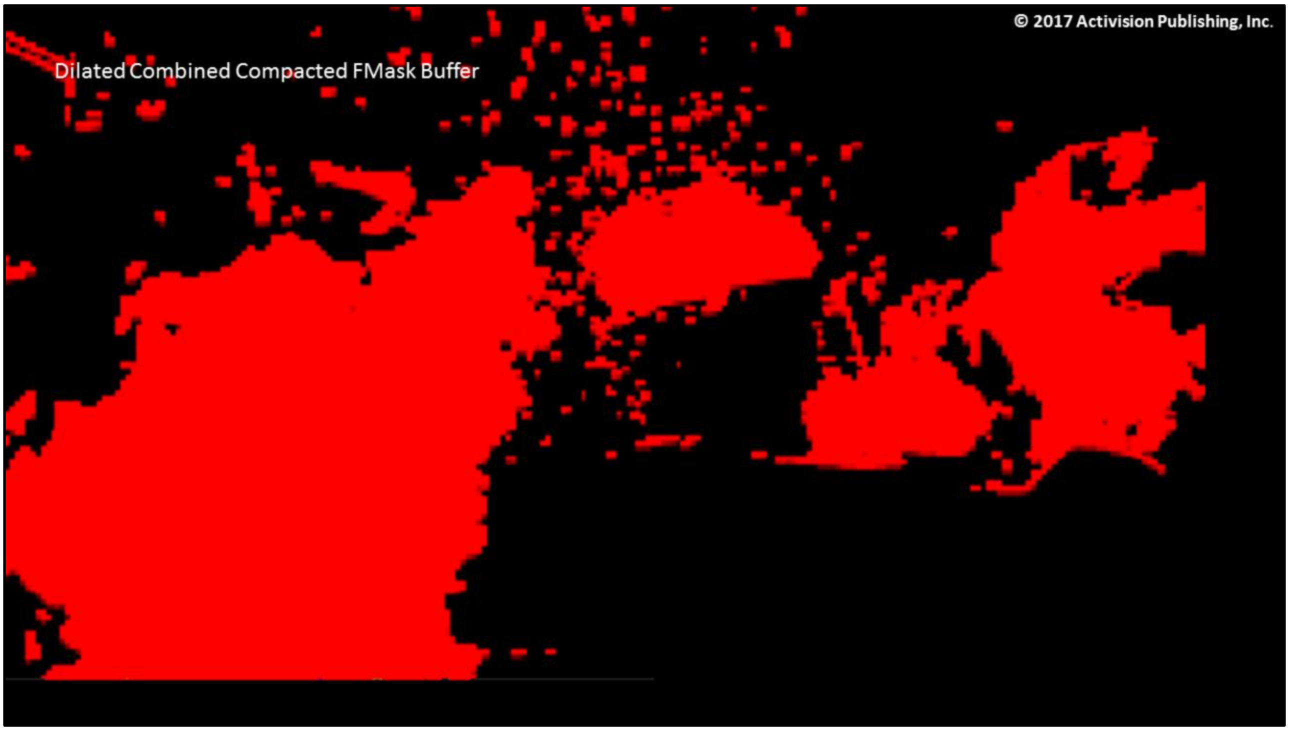

Color Buffer
© 2017 Activision Publishing, Inc.



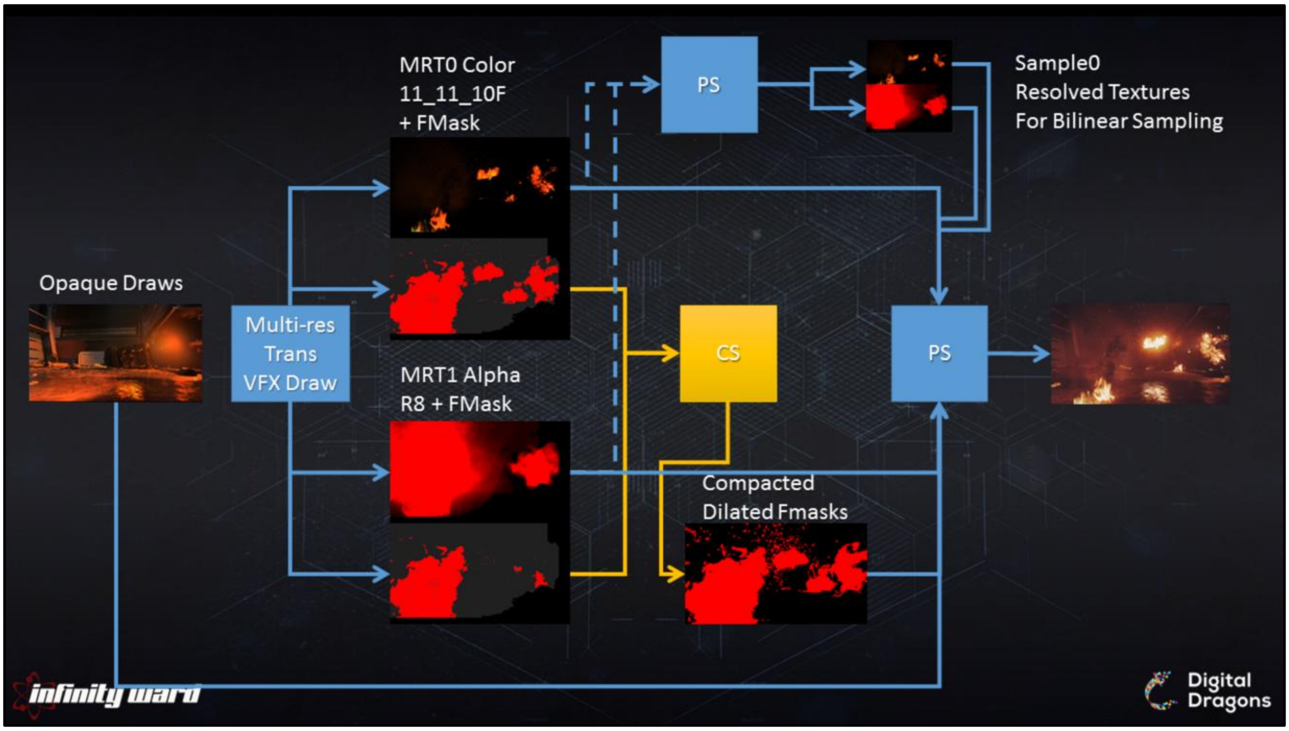
Notice geometric edges marked with multipe samples
Gray -> 1 sample / Cmask touched for blending
Blue / Green /-> 2 / 3 samples passing rasterizer due to depth intersections
Red -> 4 samples due to full resolution rendering or depth intersection hitting all subsamples


© 2017 Activision Publishing, Inc.
Alpha Buffer

Notice geometric edges marked with multipe samples
Also difference between Color Fmask, due to possible different blend mode.

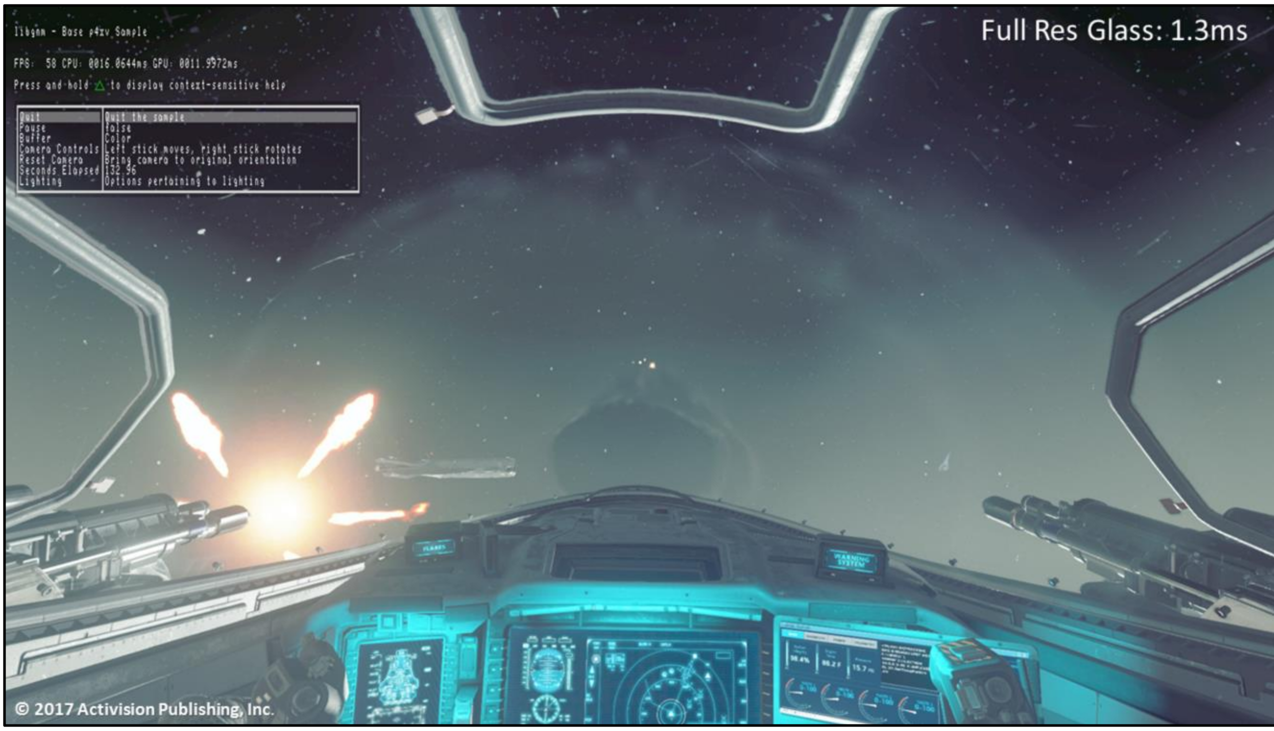


Compacted FMASK
Fmask Color > 0 || Fmask Alpha > 0
Packed into 16 bit buffer – 16 bool values per pixel



FMask / CMask – can be different between color and alpha

- Depends on blend mode setup and HW setup
  - Alpha blend
  - Add
  - Fast Blending mode (HW specific)



Full resolution transparencies can be very expensive i.e. player helmet, visors or vehicle windshields.

Renderer allows mixed resolution of transparencies and regular meshes

- Windshield for vehicles
- Glass

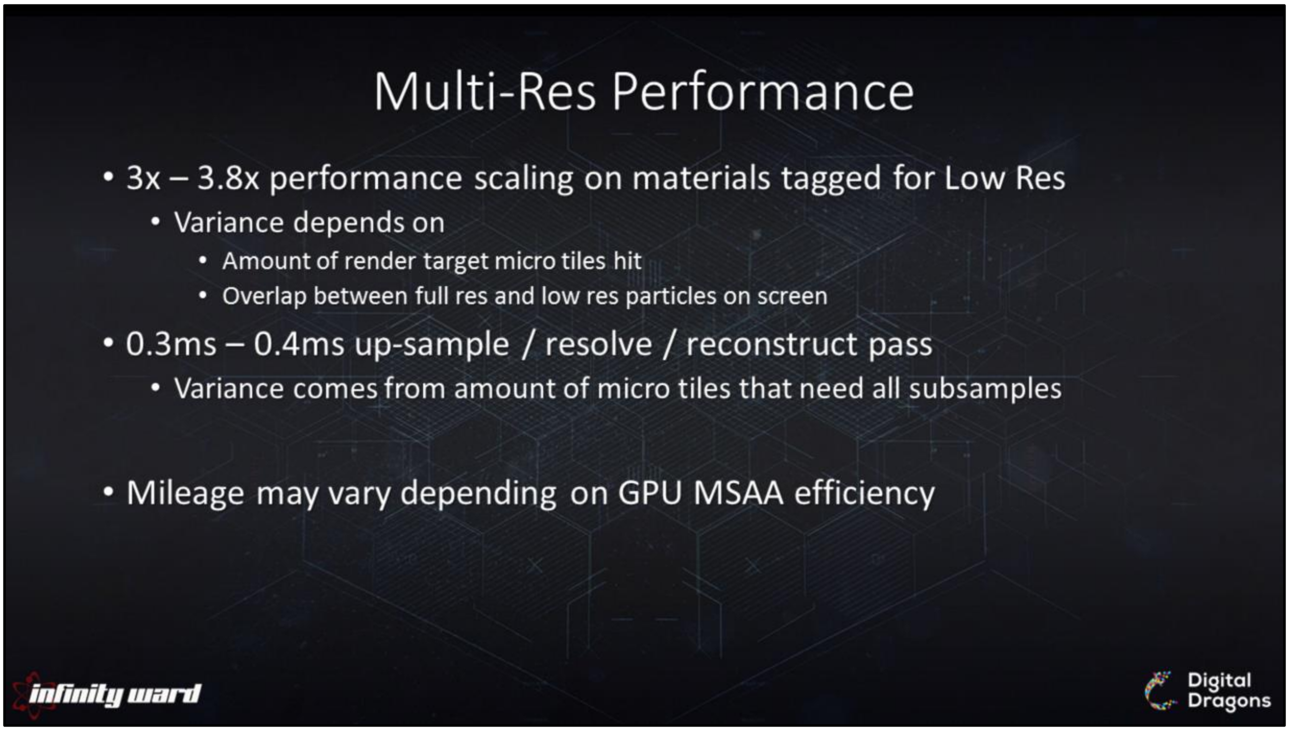
Significant performance improvement (1.3ms -> 0.4ms).
Quality degradation – mostly visible on high frequency details such as scratches on glass.

Multi-Res Issues

- Can result in ‘point sampled visuals’
  - Low Res pixels are stored as single color sample
  - Blending occurs in HW during rendering
  - Blender duplicates lower samples if blending with higher ones is required

- Per-Pixel shading can introduce aliasing if executed only for sample0
  - Z-Feather
- Use temporal dithering to mitigate issues
  - Helps with color bit-depth issues

Fail case for a pixel:
Render Low res draw – writes sample0 (fire effect)
FMask set to 1 sample – can be still bilinearly upsampled
Render Blend High res draw – duplicates src sample0, blends per sample (glass in front of fire effect)
FMask set to >1 sample – can not bilinearly upsample



3x – 3.8x performance scaling on Materials tagged for Low Res

- Variance depends on
  - Amount of render target micro tiles hit
  - Overlap between full res and low res particles on screen
    - Fast Blend / MSAA bandwidth benefits are lost as soon as MRT micro tile gets tagged for decompress == Full Res Rendering occurs
- Always less pixel work

0.3ms – 0.4ms Upsample / Resolve / Reconstruct pass

- Variance comes from amount of micro tiles that need all subsamples

All performance numbers are based off AMD GCN GPUs performance

Multi-Frequency Rendering: R&D

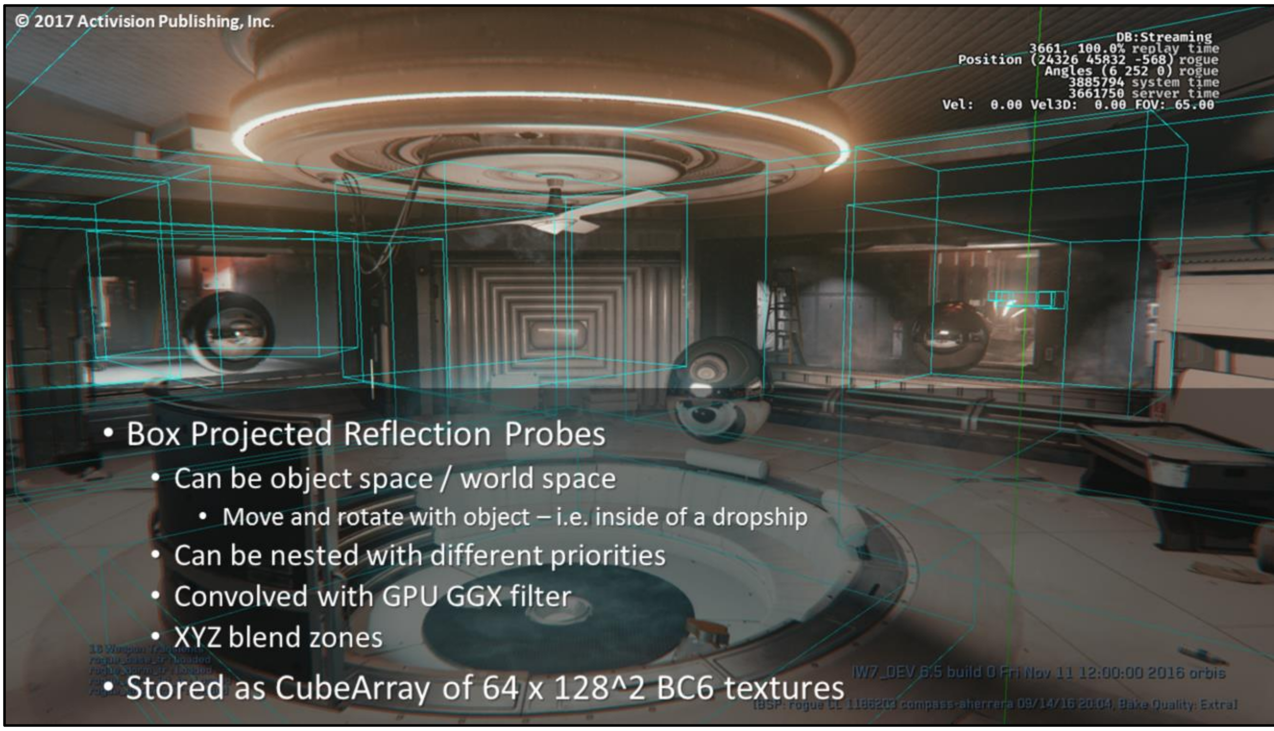
- Experiment with 8xMSAA
- Allow 1, 2, 4, 8 samples
  - Change sample patterns in conjunction with temporal supersampling
- Temporal Stochastic MSAA based OIT
- Render opaque scene using MFR
  - Pick objects of interest at high resolution i.e. character
  - Randomly change sample counts on less important objects

infinity ward

Digital Dragons

Reflections & Refractions

Reflection probes are a first class citizen.
Static and dynamic, applied in uniform way to all geometry through F+



Box Projected Reflection Probes

Can be object space / world space

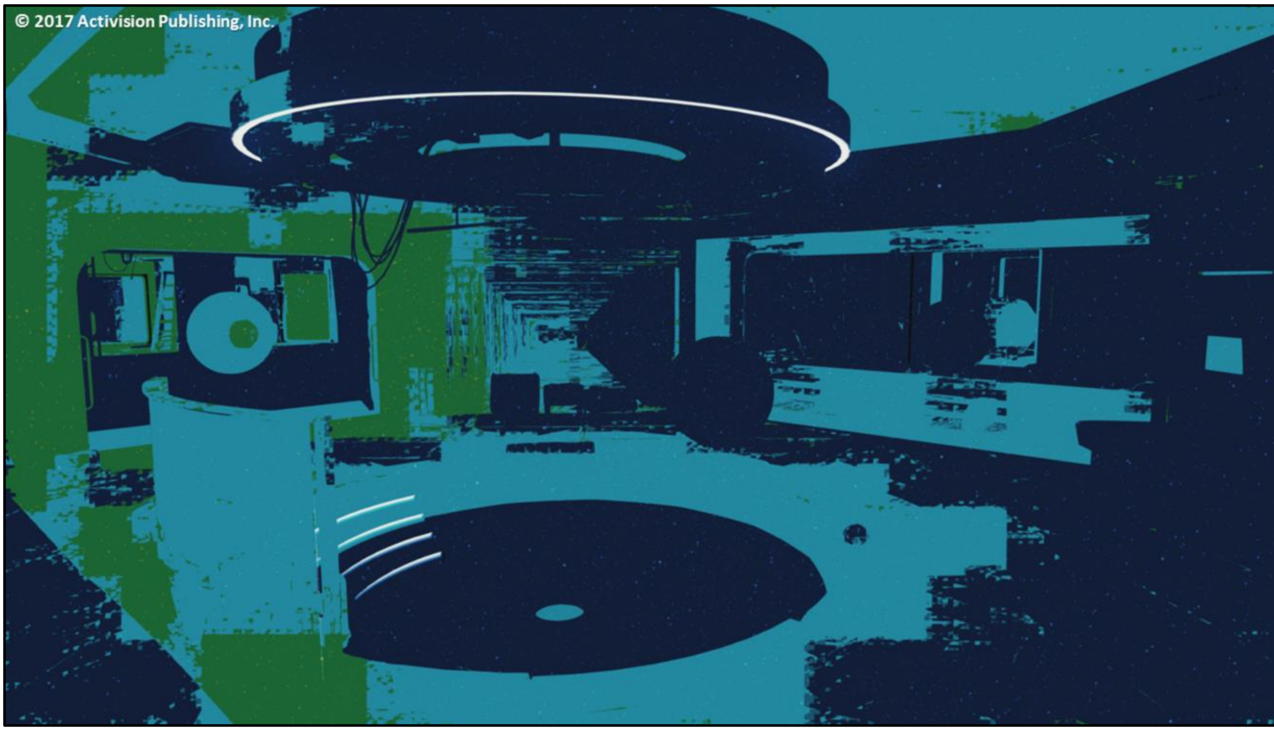
Move and rotate with object – i.e. inside of a dropship

Can be nested with different priorities

Convolved with GPU GGX filter

Allows blending of arbitrary amount of probes per pixel
Support XYZ Blend regions defined per reflection probe volume

Screen shows overlapping reflection probes and their weights



Screen shows effective post-cull regions of cube map overlaps

Reflection Probes

- CS GPU Culling: Separating Axis Theorem
  - Up to 64 cubemaps in view
  - 32x24x48 x 64bits
  - Per-pixel : additional culling steps inside PS
- Cost fully amortized on Async Compute pipeline

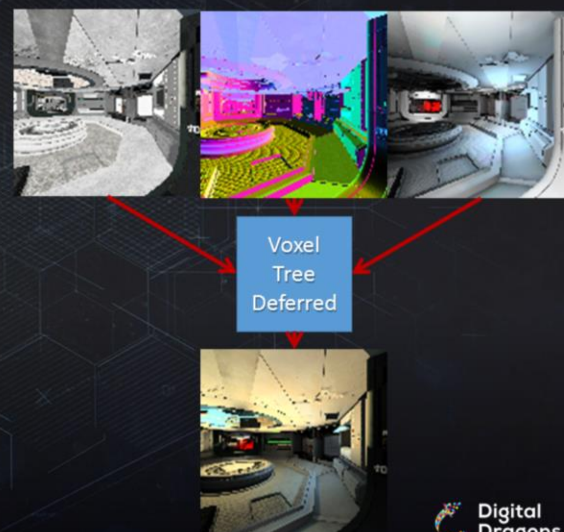
Shader	time (ms) @PS4
SAT Culling of 64 reflection probes in avg. open scene 32x24x48	0.185

infinity ward

Digital Dragons

Relightable Reflection Probes

- During bake - generate packed cube map g-buffers
  - Combined albedo + specular
  - Depth
  - Normal
  - Emissive / Base ambient lighting

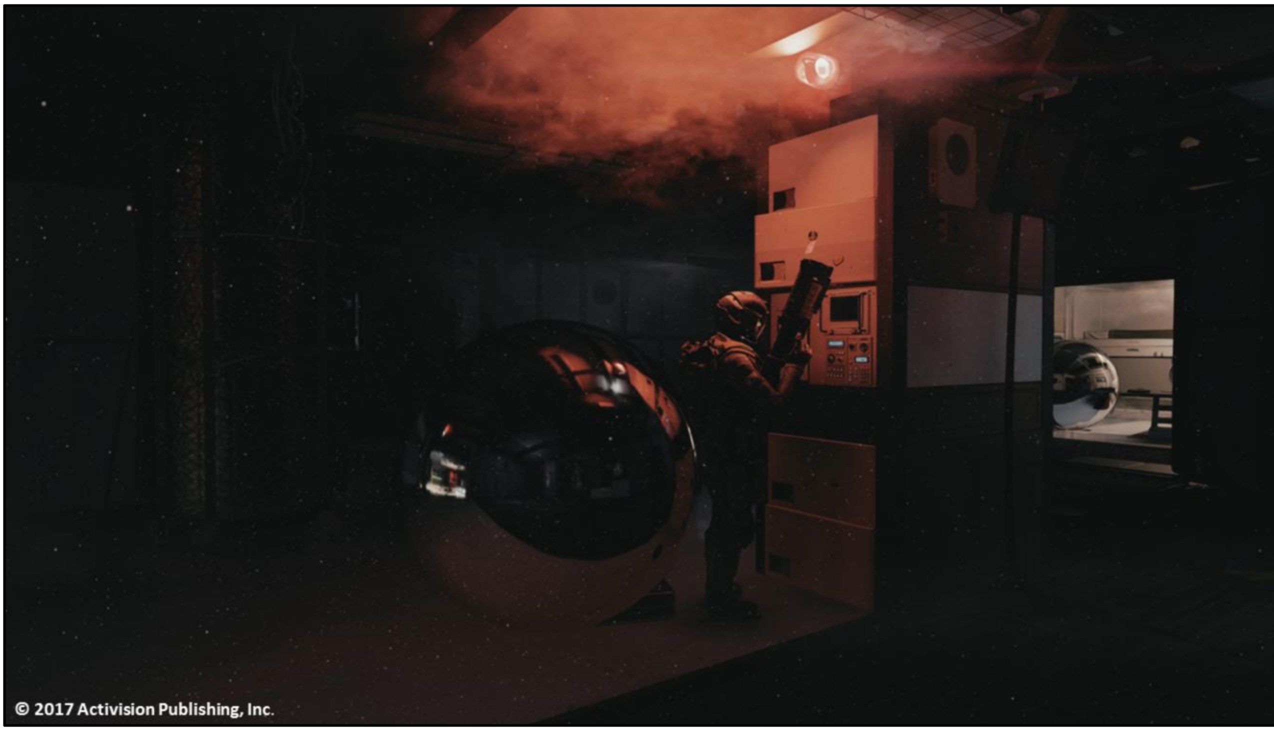


Relightable Reflection Probes

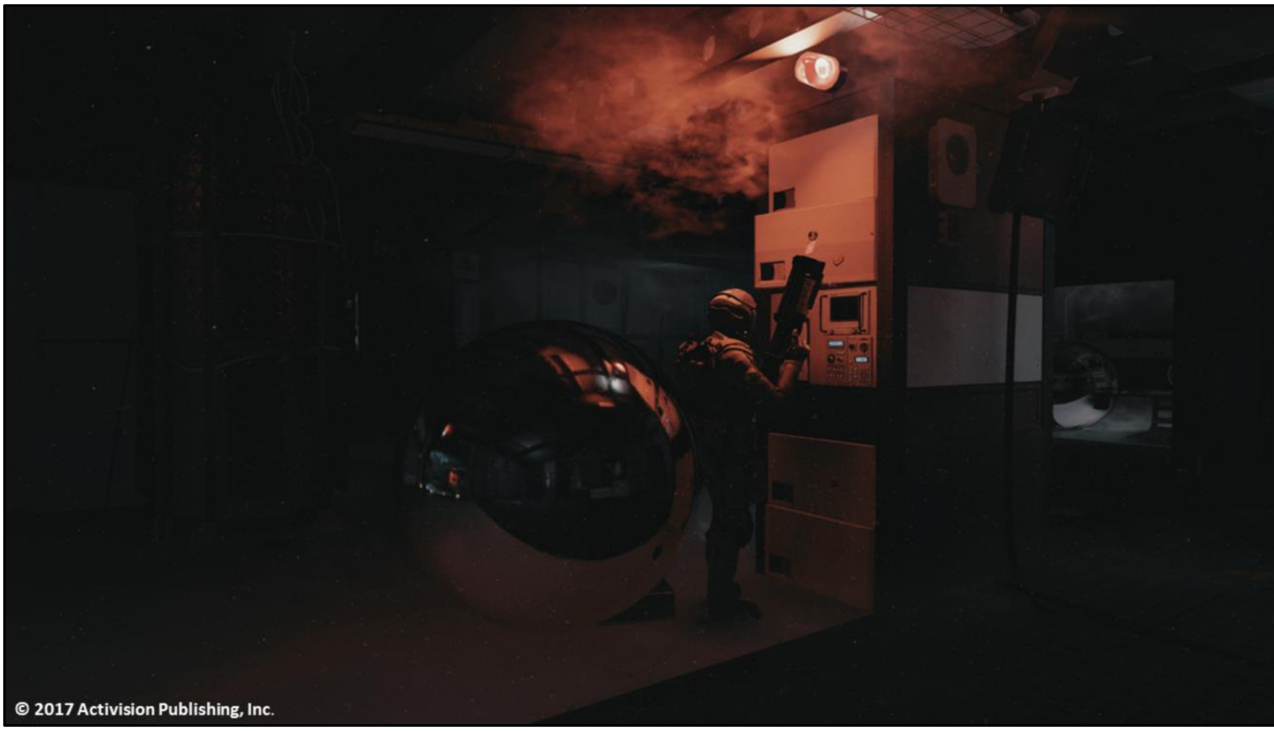
- Each frame
  - Pick N reflection probes that need relighting
  - CS World Space Voxel Tree Lighting on each cube g-buffer
  - CS Filter reflection probes [4]
    - Calculates SH2 ambient light data
  - CS compress copy to BC6h array of reflection cubemaps
- Used as normal reflection probe

One of our maps required dynamic permutations of arbitrary amount of dynamic lights, including full blackout situation.

You can see how reflection probes react to sequential light changes to adjacent rooms.



Notice the reflection of gun mounted light on the ceiling and in reflection probe.



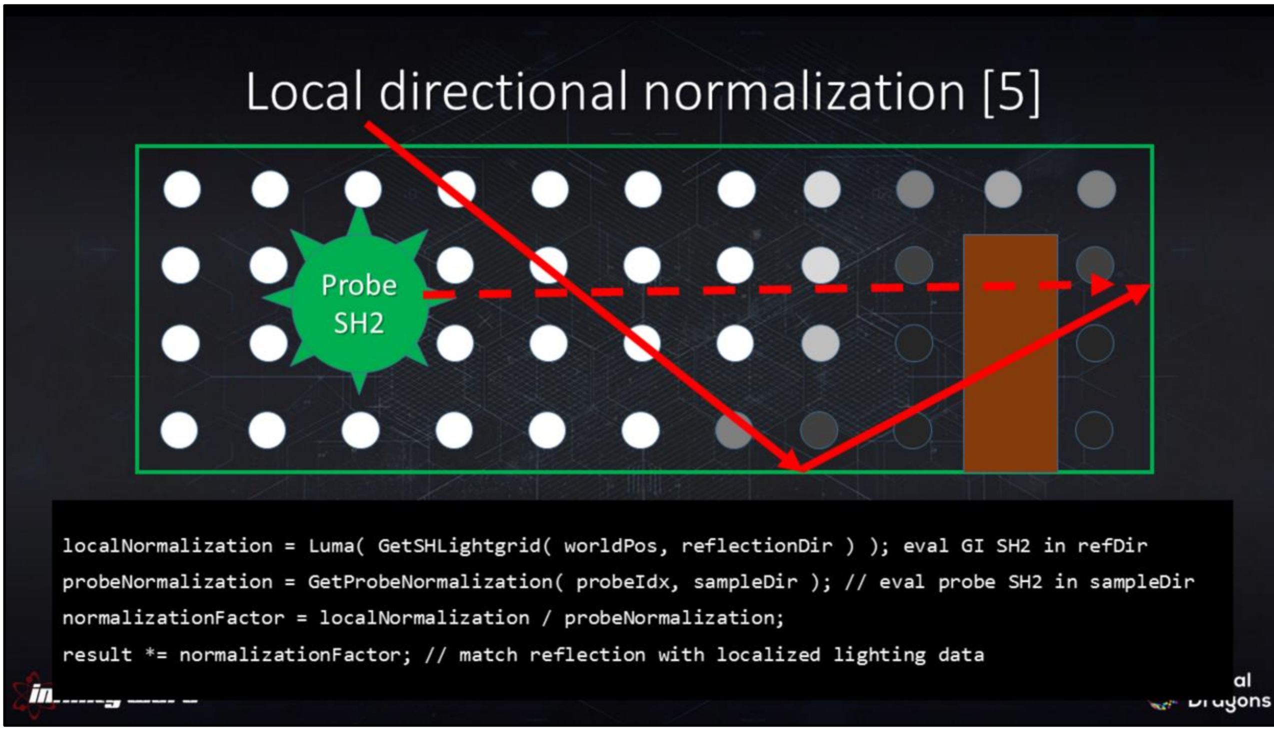
When the character moves, you can see the reflection updated in real time.


© 2017 Activision Publishing, Inc.

Relightable Reflection Probes : Performance

- Renderer updates 1 probe per frame
- All processing utilizes Async Compute

Compute Job on 128^2 Cubemap	Time (ms) @PS4
Relighting (depends on # of dynamic lights)	0.1-0.2
Filtering all MIPs	0.31
BC6 compression all MIPs	0.18



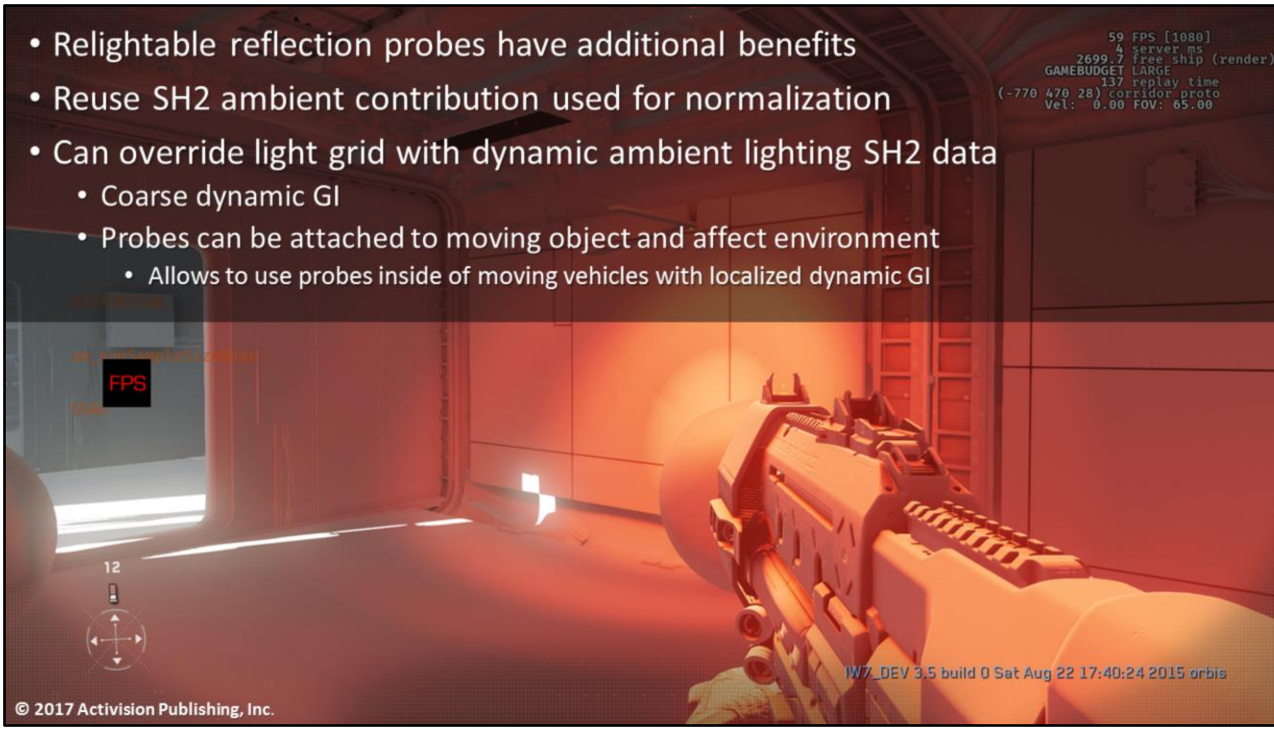
Local directional normalization by lightgrid SH

- Each probe at generation time stores its own SH luma
- Relightable probes calculate SH luma during filtering process
- Lightgrid luma SH value is evaluated, during shading, in direction of specular reflection from data already sampled for GI.
- Sampled Reflection probe data is scaled to match evaluated lightgrid value.

Notice unevenly lit golden foil air ducts. Also consoles on right wall.

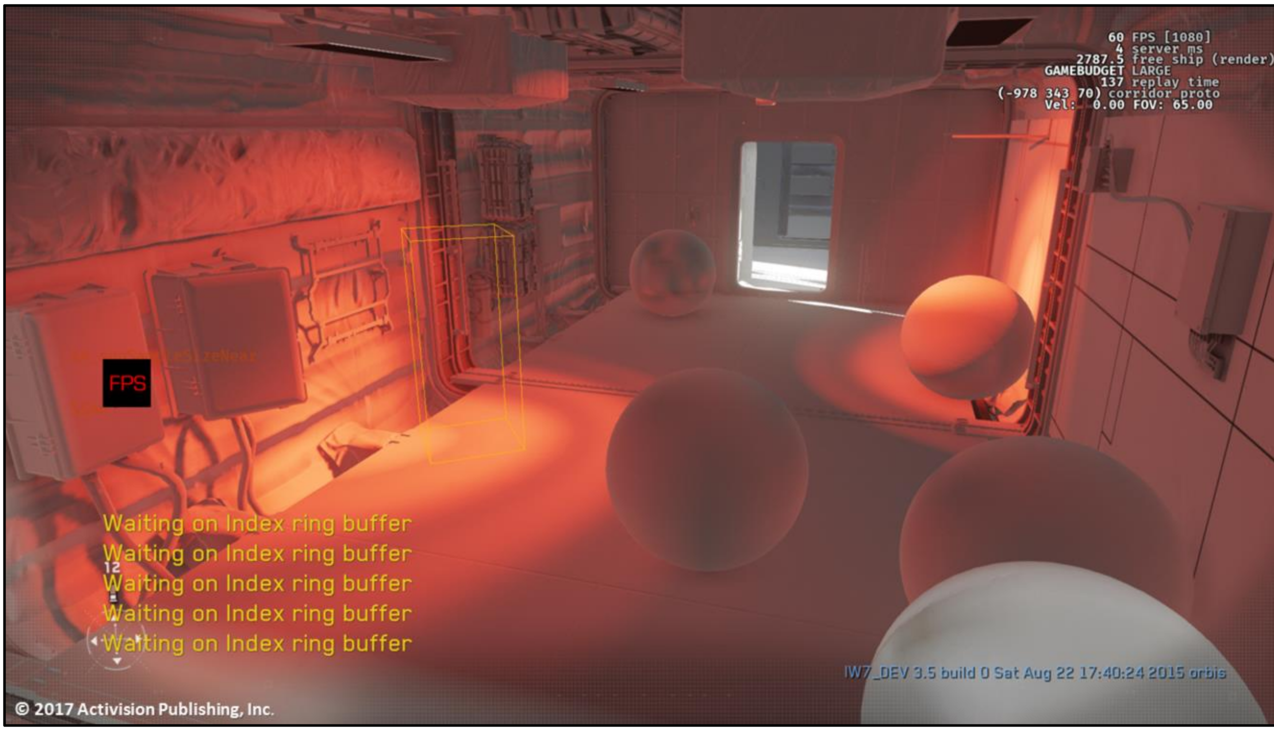
With localized normalization, integration of scene is much improved.

Relightable reflection probes have additional benefits.
We already calculate SH2 ambient contribution for each probe, used for normalziation.

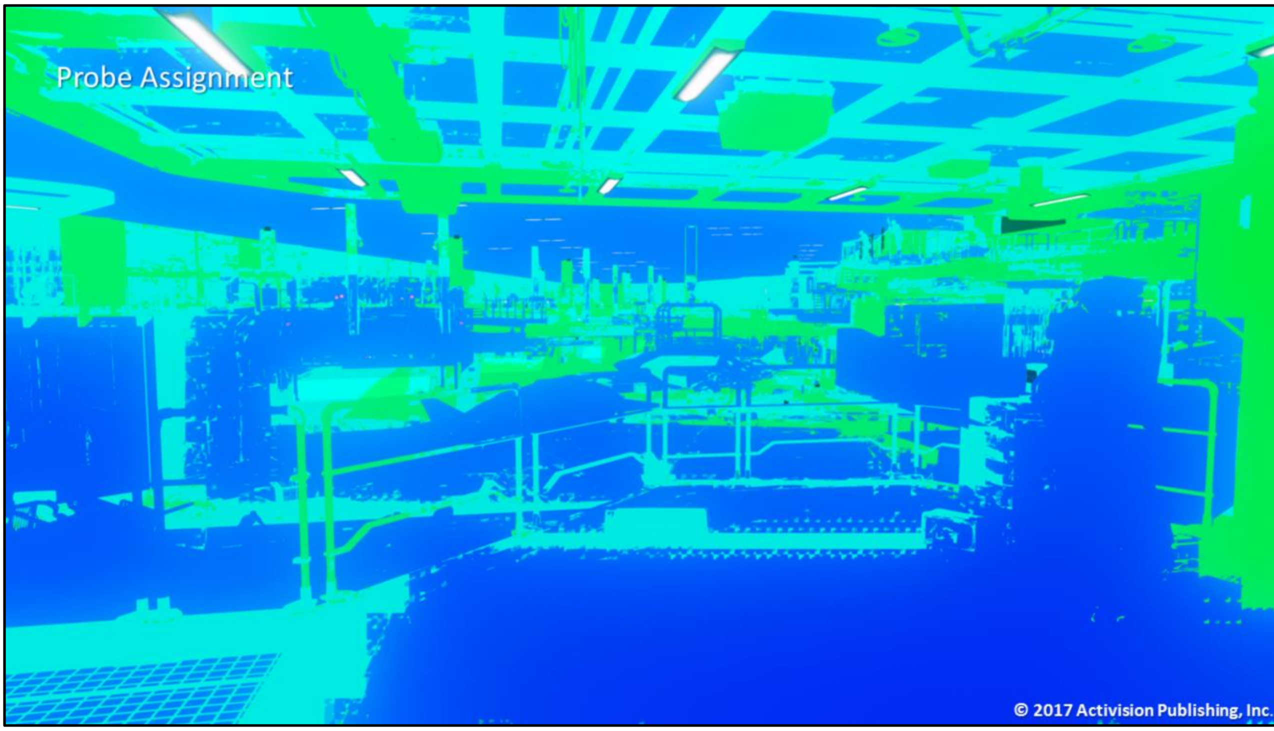


Coarse Dynamic GI based off reflection probes
Add Delta Light SH2 from probes to ambient term (lightmap / lightgrid)

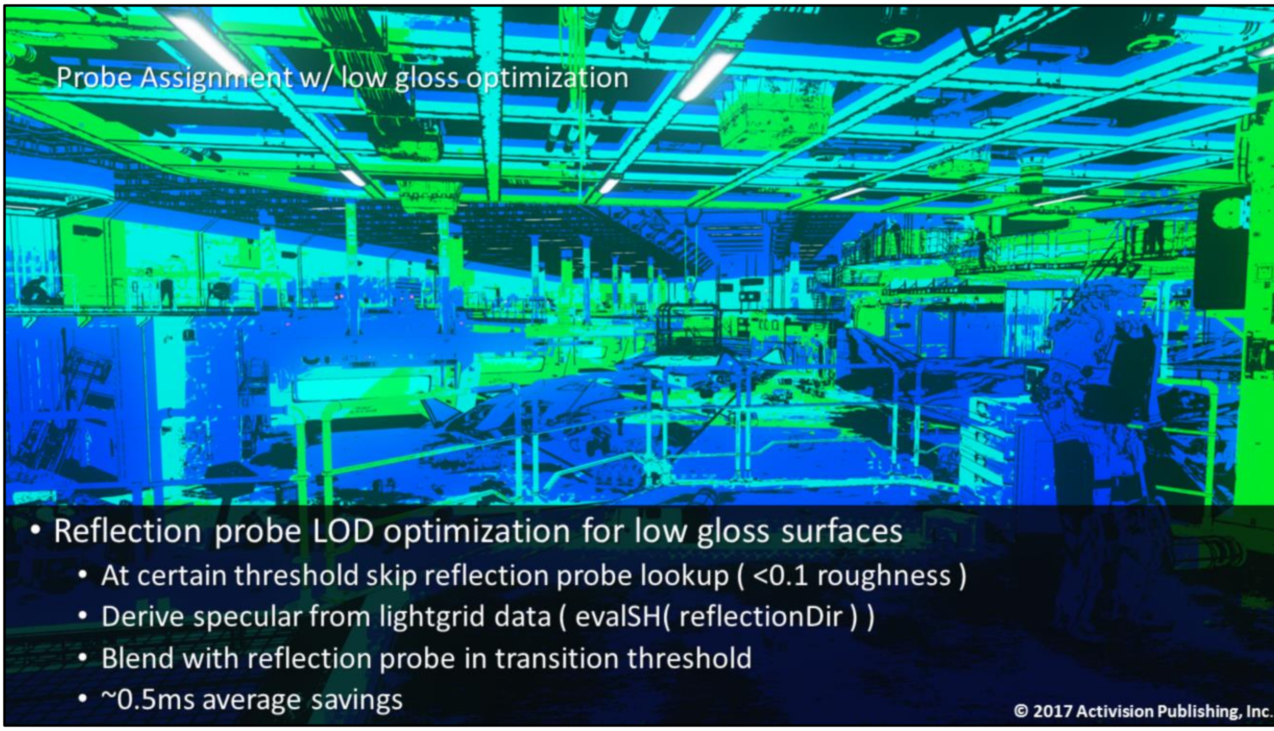
Another view on coarse dynamic GI. Disabled.



Another view on coarse dynamic GI. Enabled.



Every single pixel samples at least one reflection probe



Reflection probe LOD optimization for low gloss surfaces

At certain threshold skip reflection probe lookup

Derive specular from lightgrid data

~0.5ms saving in average scene (filtering low mips of cubemaps is really expensive with cube wrap filtering mode – you want to avoid it at all cost)

Screen represent a scene that has majority of metal materials of varying roughness.

Screen Space Reflections / Refractions

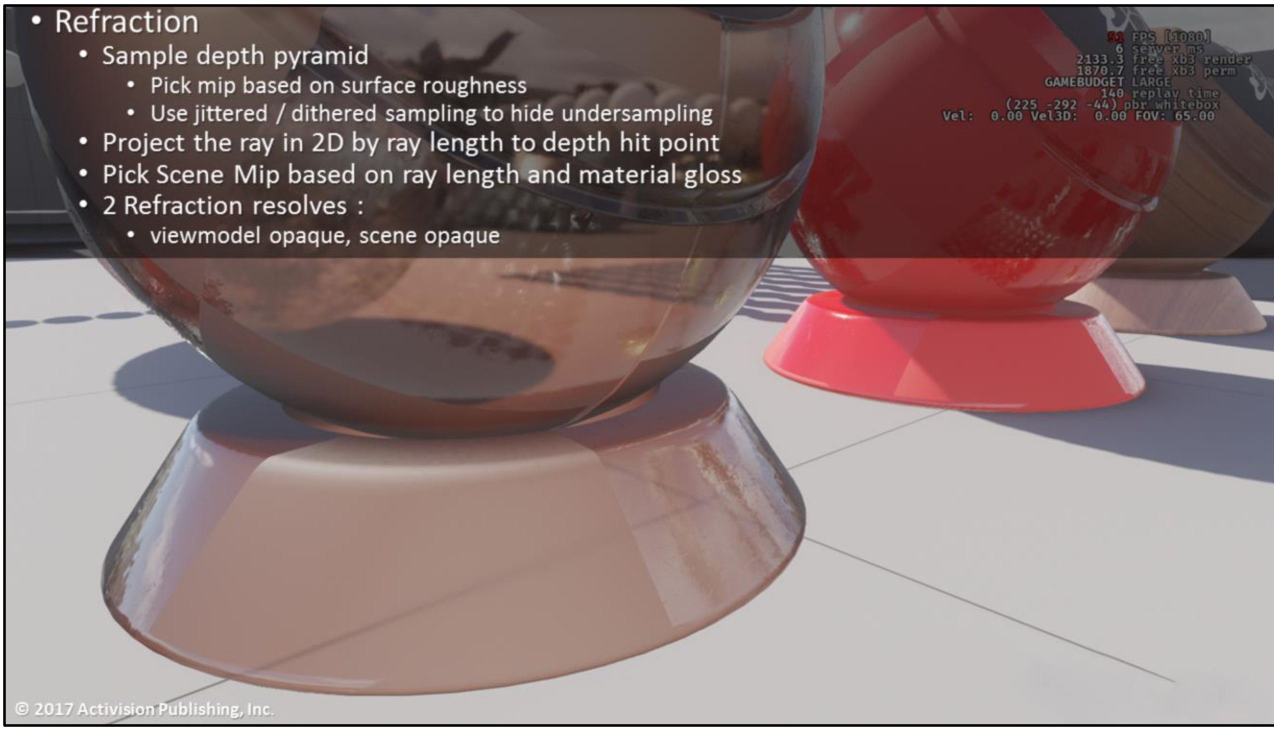
- Generate Scene Mip chain prior to tone map resolve
  - Use BRDF screen space filter to match mips to gloss BRDF similar to cubemaps
  - Reproject previous frame mip chain
- Reflections
  - Reuse intersection from Box Projection Reflection Probes
  - No additional tracing needed
  - Pick mip based on material gloss and ray length
  - Reflection as good as your Box Projection match

infinity ward

Digital Dragons

Mix of all presented techniques working together.
Box projected reflection probes
Relightable reflection probes
Box projected screen space reflections

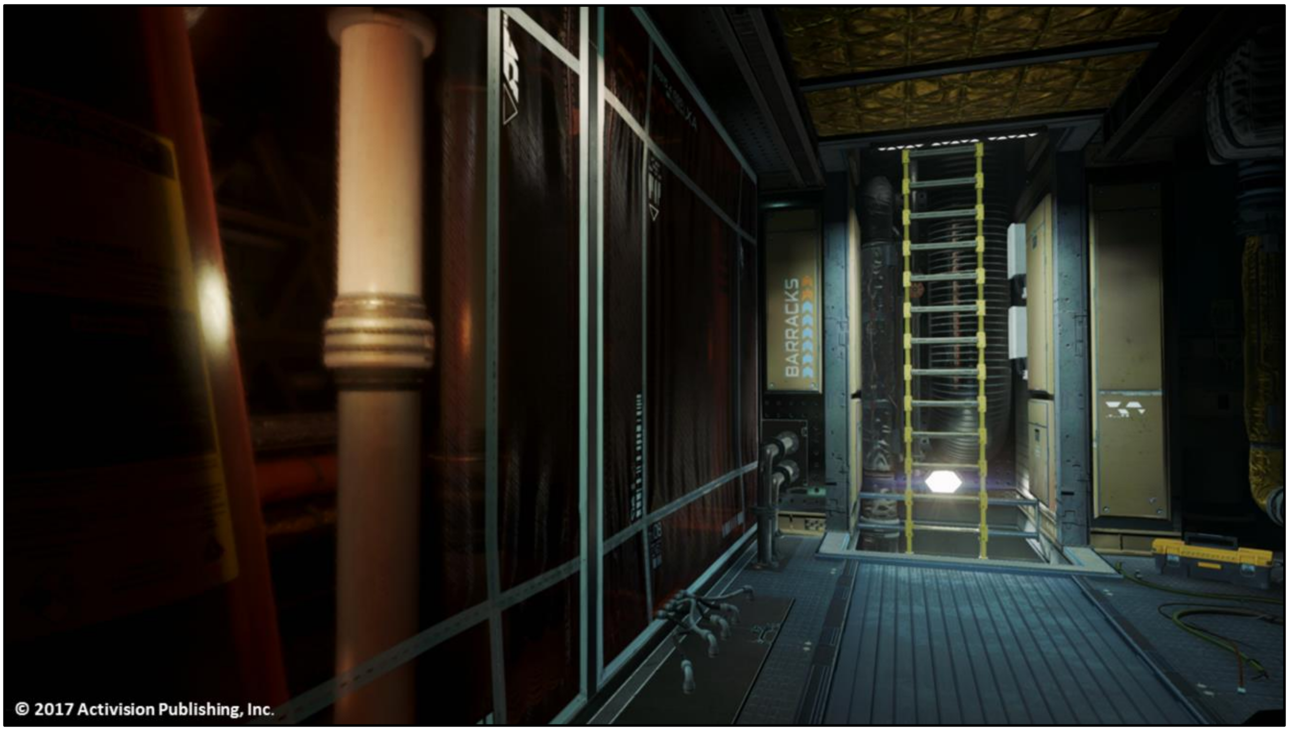
Mix of all presented techniques working together.
Box projected reflection probes
Relightable reflection probes
Box projected screen space reflections



- Refraction
  - Sample depth pyramid
    - Pick mip based on surface roughness
    - Use jittered / dithered sampling to hide undersampling
  - Project the ray in 2D by ray length to depth hit point
  - Pick Scene Mip based on ray length and material gloss
  - 2 Refraction resolves :
    - viewmodel opaque, scene opaque

FPS [1080]
6 server ms
2133.3 free xb3 render
1870.7 free xb3 perm
GAMEBUDGET LARGE
140 replay time
(225 -292 -44) pbr whitebox
Vel: 0.00 Vel3D: 0.00 FOV: 65.00

© 2017 Activision Publishing, Inc.

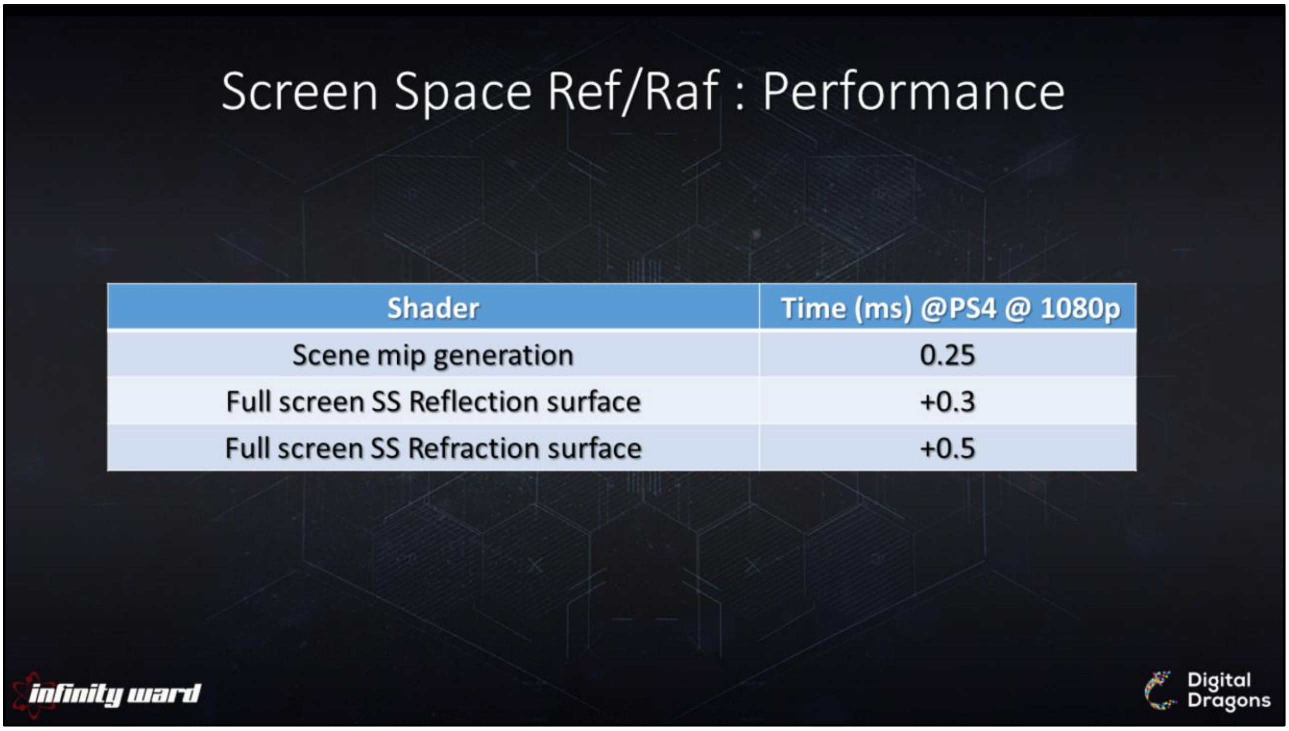


Used for multiple surfaces ranging from frosted glass to plastic curtains

Our weapon artists and users love to see how internal weapon parts operate, seen through semi-translucent pieces using screen space glossy refractions

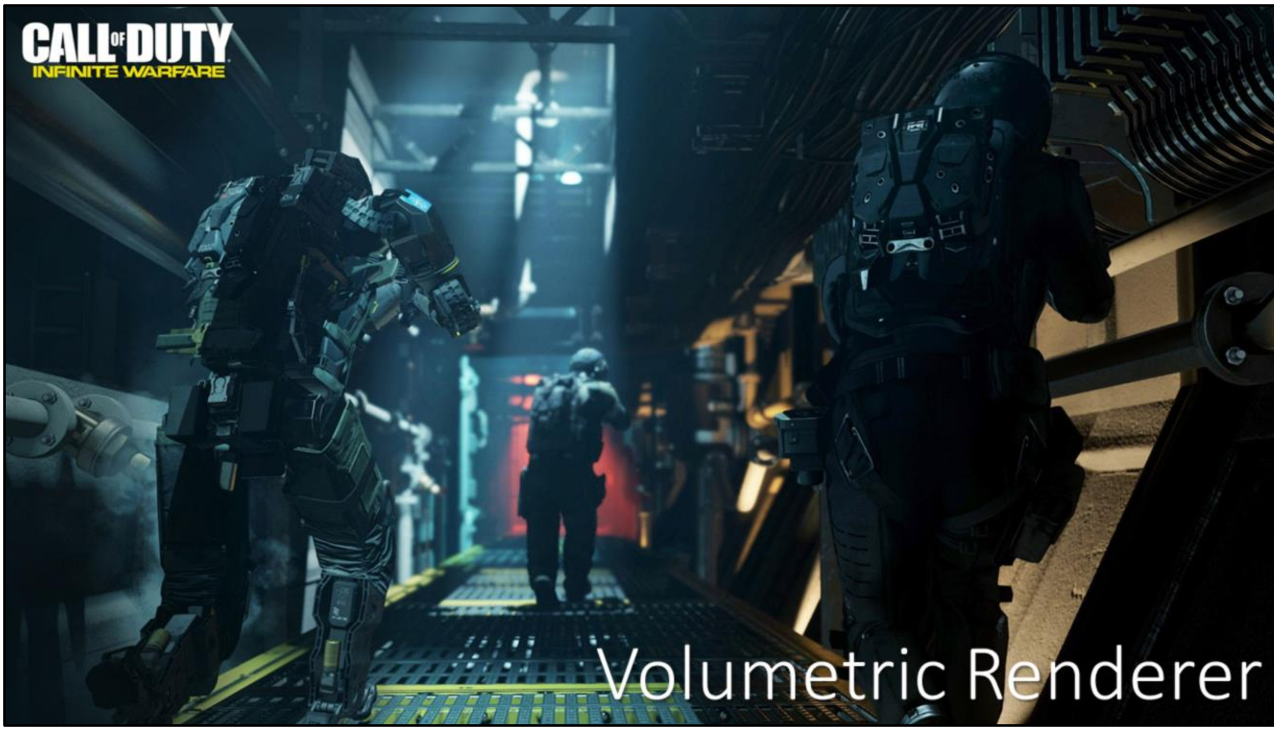
Our weapon artists and users love to see how internal weapon parts operate, seen through semi-translucent pieces using screen space glossy refractions

Our weapon artists and users love to see how internal weapon parts operate, seen through semi-translucent pieces using screen space glossy refractions



Shader	Time (ms) @PS4 @ 1080p
Scene mip generation	0.25
Full screen SS Reflection surface	+0.3
Full screen SS Refraction surface	+0.5

Very cheap in comparison with fully traced methods



Volumetrics were important part of COD:IW look. Could not ship without them, nor use them as quality setting.

Volumetrics use Froxel buffer [6][7]

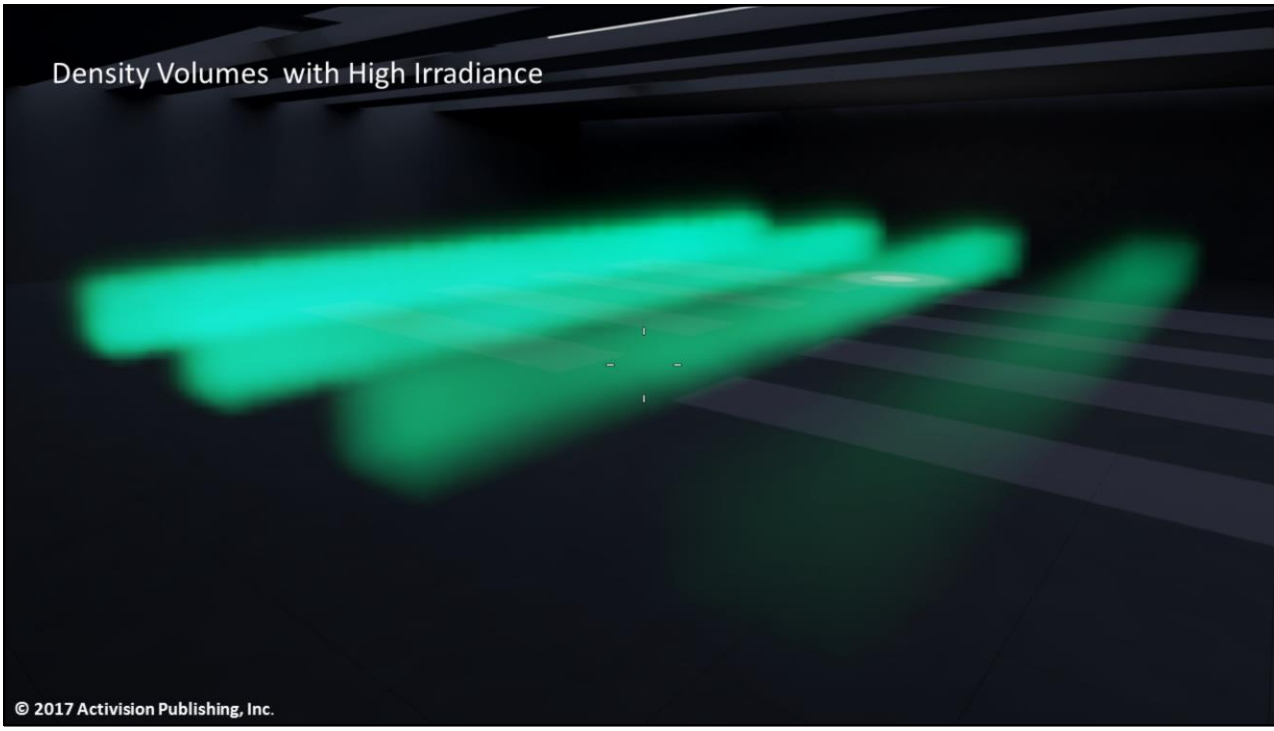
Static lighting and GI resamples the lightgrid.

Supports all light types

- Static / Ambient Light
  - samples light grid
- Dynamic Lights
  - evaluates using unified code path for scene rendering

Temporal Re-projection to stabilize

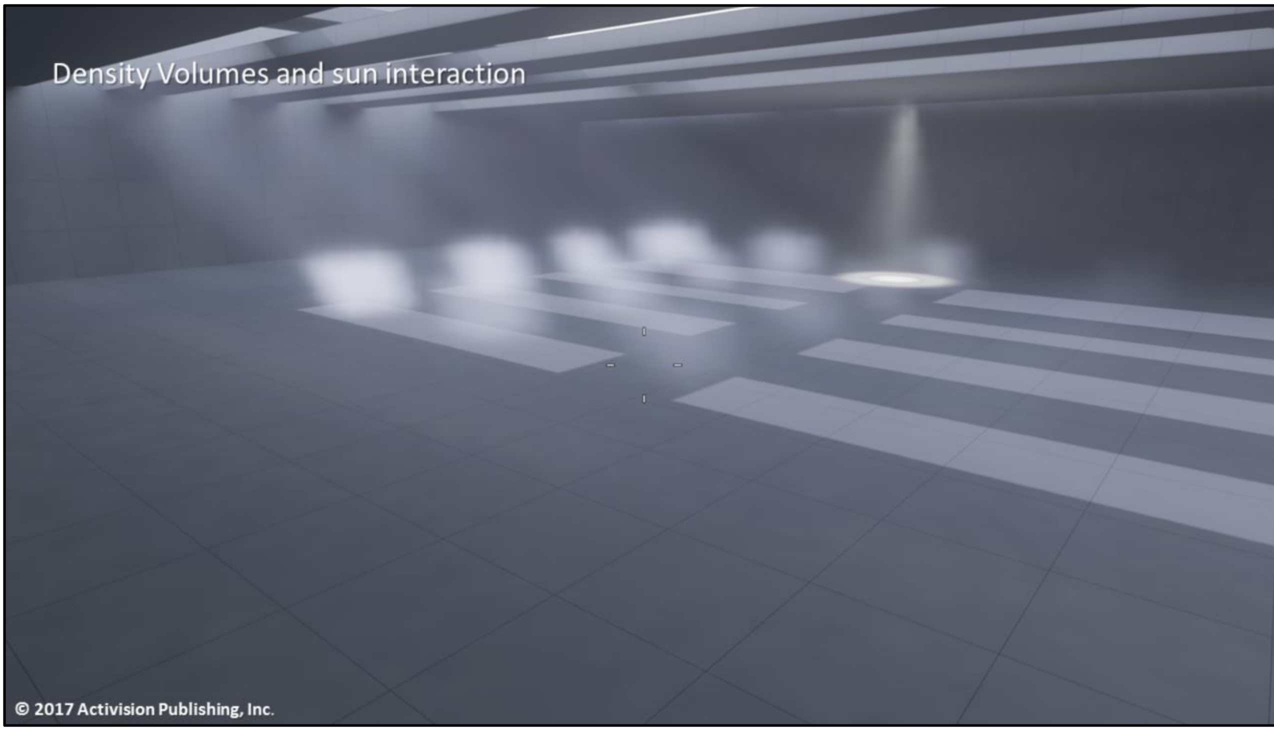
- 2 x memory & bandwidth consumption



Artists can manually place localized density (fog) volumes.
Each density volume

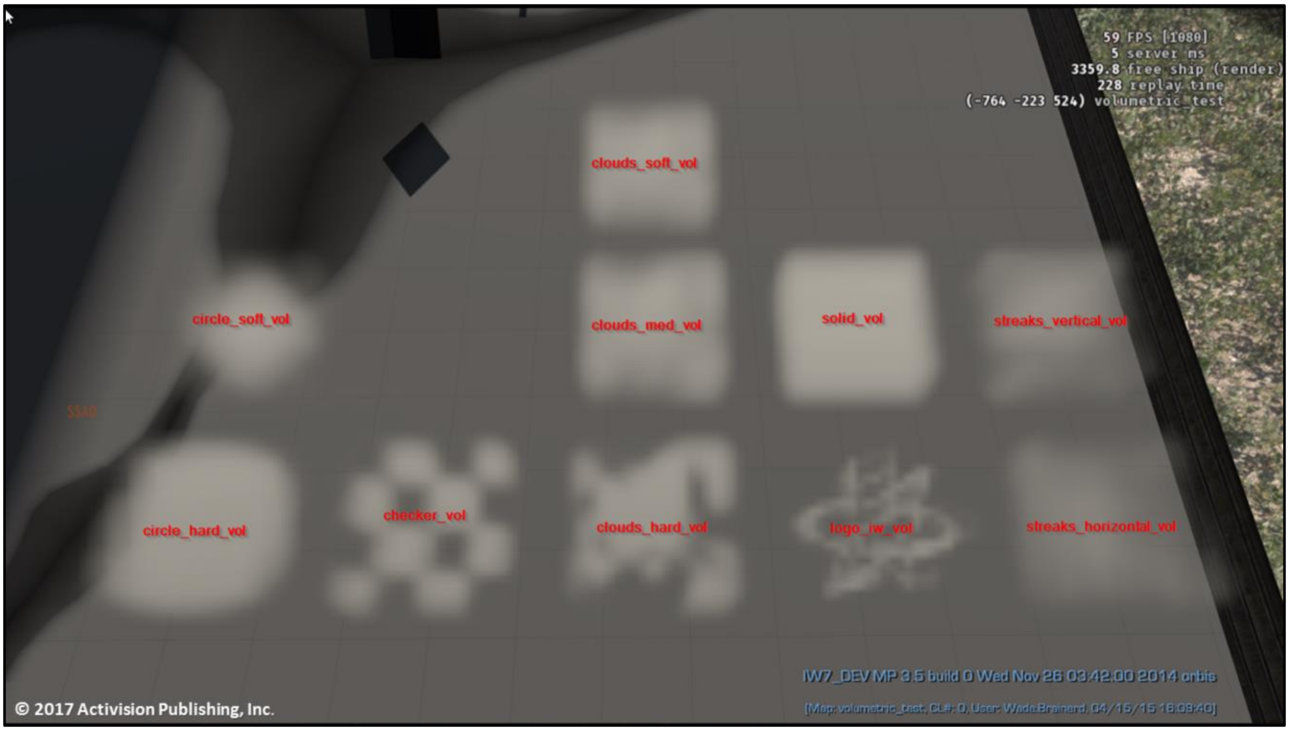
- World Space Bounding Box
- Base density
- Irradiance

Screenshot shows Rendering of high irradiance density volumes



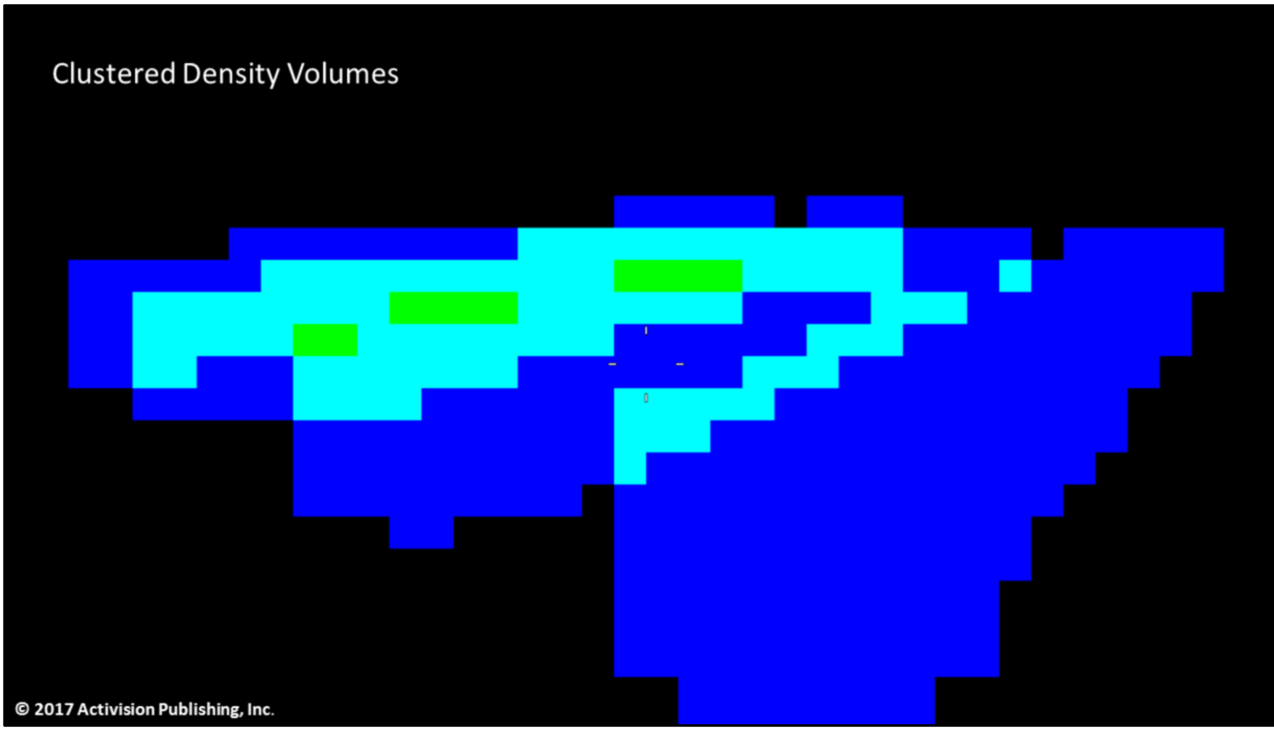
Used to 'localize' fog in map
- Often placed in interiors, without affecting global fog settings

Screenshot shows Sunlit density volumes of varying densities



Density can be masked by up to 4 axis aligned projected textures
Animated UV scrolling (i.e. animated knee height fog)

Screenshot shows Multiple textures used to create different density volumes



CS job for clustering

Clusters to match 4x4x4 main CS kernel

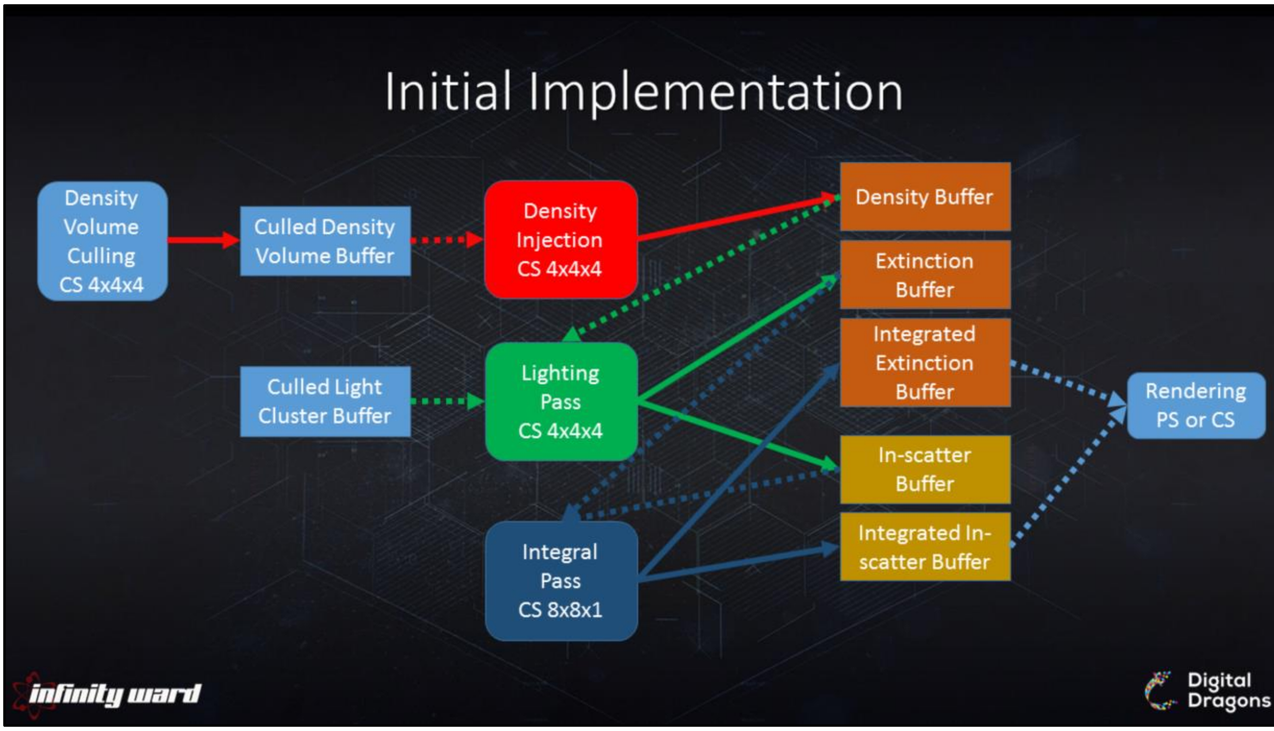
Up to 256 bits indexing density volumes within the frustum

Screenshot shows Clustered view of density volumes

Low Frequency visuals provided by Volumetric Rendering

High Frequency visuals provided by Lit Particles

Blending is seamless due to 3D nature of volumetrics.
Opaque / Transparent just sample 3D texture with integrated in-scatter light and integrated extinction.



Multiple sequential CS jobs

- Density Injection
  - CS optimized for scalarization : execute in 4x4x4 thread kernels
    - match single froxel size from clustered density volume buffer
  - Write (Density Buffer)
- Lighting Pass
  - CS optimized for scalarization : (4x4x4 thread kernel)
    - match single froxel size from clustered light buffer
  - Read (Density Buffer) / Write (In-scatter Buffer / Extinction Buffer)
- Integral Pass
  - Iterate over Z slices (8x8x1 thread kernel)
  - Accumulate scattering and extinction
  - Read (In-scatter Buffer / Extinction Buffer) / Write (Integrated In-scatter / Integrated Extinction Buffer)

Volumetric Renderer : Initial Implementation

- 112x90x128
  - 11_11_10F – (5,150 KB) | in-scatter and integral buffer
  - 16F – (2,580 KB) | density and extinction buffer
- Each pass shown to be BW / Latency limited

CS Job	Constant R/W BW	Execution time	VGPR	Optimal Kernel
Density Injection	5,150 KB	~0.4 ms	36	4x4x4
Lighting Pass	10,310 KB	~1.1 ms	48	4x4x4
Integral Pass	15,460 KB	~0.5 ms	42	8x8x1
Total	30,920 KB	~2.0 ms (ALU only performance ~1.2 ms)	MAX(48)	

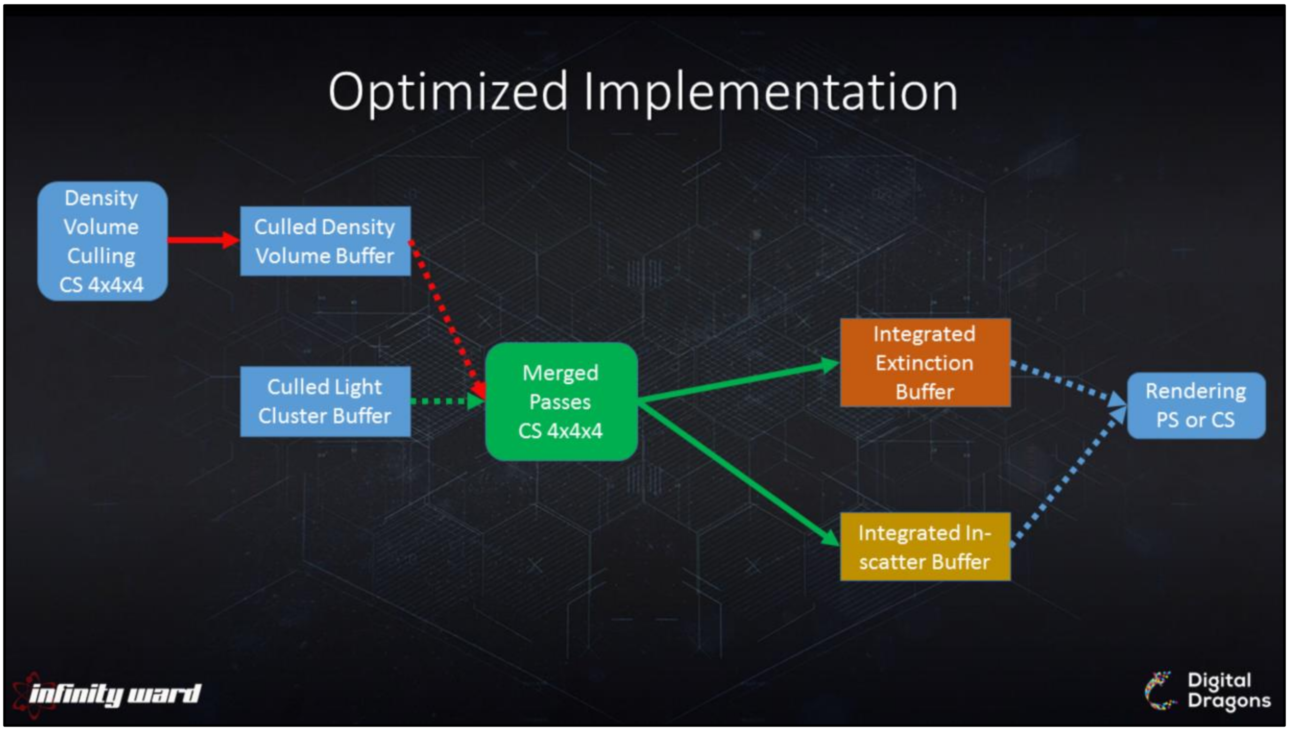
infinity ward

Digital Dragons

Each pass shown to be BW / Latency limited
- 3D texture R/W has high latency
- Pick tile mode to match your CS job R/W pattern (kernel size)
  - TILE_MODE_1D_THIN -> Slice by slice R/W (8x8x1)
  - TILE_MODE_1D_THICK -> 3D block R/W (4x4x4)

B/W bound
- Run wider VGPR
  - Already high occupancy
- Latency bound
  - Compensate with more ALU
- Remove redundant memory transfer

Experiment
- Cap all passes to 48 VGPR
  - Performance unchanged
  - Still latency bound
- Forcing to 64 VPGR (4 occupancy) results in 20% performance loss due to B/W
- We are at optimal occupancy



- Merge all passes
  - Each pass exists as ‘code block’
    - Known VGPR pressure and cycles
  - All input memory read bandwidth optimized
  - Only Write out
    - In-scattering Integral
    - Extinction Integral
  - Need to match kernel size for scalarization
  - Switch all blocks to work with 4x4x4 3D clusters
    - TILE_MODE_1D_THICK
    - In cache, grouped texture Loads and Stores
  - Integral CS required new algorithm to work in 4x4x4 groups
    - Inclusive Prefix Sum
    - Implemented using Lane Swizzles

```
// 8x8x1 Integral ( X Y Z - order )
float accumulate = 0;
for( I = 0; I < sliceCount; i++ )
{
        accumulate += ReadData( xy, i );
}
```

```
// 4x4x4 Integral ( Z X Y - order )
float accumulate = 0;
uint lane = __XB_GetLaneID();
bool laneMask0 = (lane >> 0) & 1;
bool laneMask1 = (lane >> 1) & 1;
for( I = 0; I < sliceCount / 4; i++ )
{
    data = ReadData( xy, i );
    // Inclusive prefix-sum
    sumData = data;
    // Threads: 4th = 4th + 3rd   2nd = 1st + 2nd
    addData0 = QuadSwizzle( sumData, 0, 0, 2, 2 );
    sumData += laneMask0 ? addData0 : 0.0f;
    // Threads: 4th= 4th + 2nd = 3rd + 4th + 1st + 2nd
    addData1 = QuadSwizzle( sumData, 0, 0, 1, 1 );
    sumData += laneMask1 ? addData1 : 0.0f;
    accumulate += sumData ;
    // Broadcast 4th thread to 1st 2nd 3rd
    accumualte = GetLastLane( accumulate );
 }
```

```
// GetLastLane & QuadSwizzle are LaneSwizzle macros
QuadSwizzle( v, n0, n1, n2, n3 ) – for every 1st 2nd 3rd and 4th
lane of register V, swizzle so 1st swizzles to n0, 2nd swizzles
to n1, 3rd swizzles to n2 and 4th swizzles to n3 ( where n0..3
in [0..3] ) lane.
GetLastZLane( v ) – returns last lane in each kernel ( in this
case 4th )
```

#define __LANE_SWIZZLE_MASK(_and, _or, _xor)   ((_and & 0x1F) << 0) | ((_or & 0x1F) << 5) | ((_xor & 0x1F) << 10)
#define __QUAD_SWIZZLE_MASK(_o0, _o1, _o2, _o3) ((_o0 & 0x3) << 0) | ((_o1 & 0x3) << 2) | ((_o2 & 0x3) << 4) | ((_o3 & 0x3) << 6) | (0x1 << 15)

#define LaneSwizzle(_x, _and, _or, _xor)__LaneSwizzle(_x, __LANE_SWIZZLE_MASK(_and, _or, _xor))
#define QuadSwizzle(_x, _o0, _o1, _o2, _o3)__LaneSwizzle(_x, __QUAD_SWIZZLE_MASK(_o0, _o1, _o2, _o3))

Some shader compilers provide QuadSwizzle functionality right away.

Volumetric Renderer : Final Implementation

- Significant optimization allowed this technique to be viable

CS Job	Fixed R/W BW	Execution time	Cost	VGPR	Kernel
Multiple Passes	30,920 KB	~2.0 ms	100%	MAX(48)	Mixed
Merged	7,730 KB	1.2 ms	60%	48	4x4x4

infinity ward

Digital Dragons

View distances were an issue. Our view ranges can dynamically scale between close quarter corridors, open vistas and space battles.
Reconfiguration of slices can happen at runtime and helps with transitions i.e. inside of a building to outside.
Volumetrics can pick a different optimized shader that can sample : sun only, lights only, ambient only, or any permutation. This gave us up to 20% performance boost in certain situations.

Texture Packer

Texture Packer : Motivation

- Disk and runtime memory limits
- Do not want to complicate asset pipeline for artists
- Multiple texture samples unfavorable for
  - Anisotropic Filtering
  - Forward+
- Augmented textures
  - Antialiasing

infinity ward

Digital Dragons

Core Texture Slots

Semantic Slot	Compression type	Bytes per pixel
Diffuse (RGB) + Alpha (A)	BC1 / BC3	0.5 / 1.0
Specular (RGB) + Gloss (A)	BC3	1.0
Normal (XY)	BC5	1.0
Occlusion (A)	BC4	0.5
Reveal (A) <BSP only>	BC4	0.5
Total : 4 – 5 texture samples		4.0 – 4.5

infinity ward

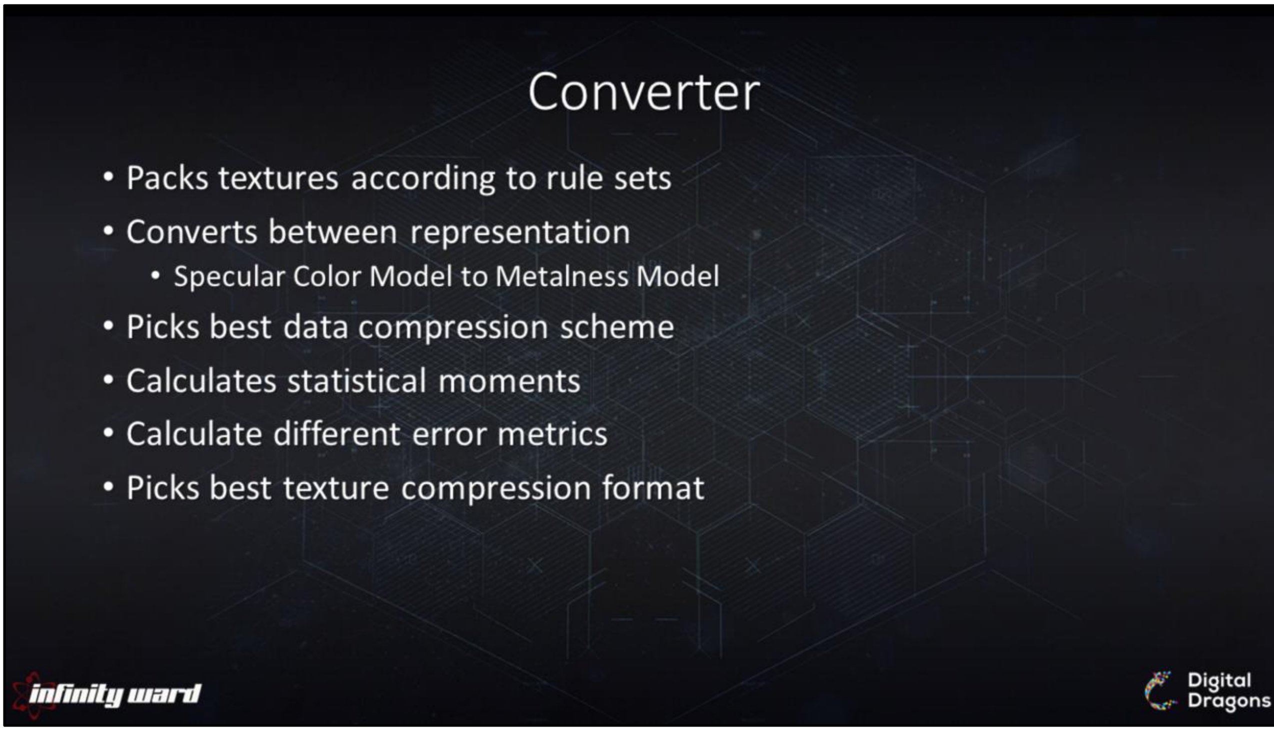
Digital Dragons

Additional Texture Slots

Semantic Slot	Compression type	Bytes per pixel
Thickness	BC4	0.5
Absorption	BC1	0.5
Fluorescence	BC1	0.5
Sheen and Cloth	BC1	0.5
Anisotropy	BC7	1.0
Reveal	BC4	0.5
Thickness	BC4	0.5

infinity ward

Digital Dragons



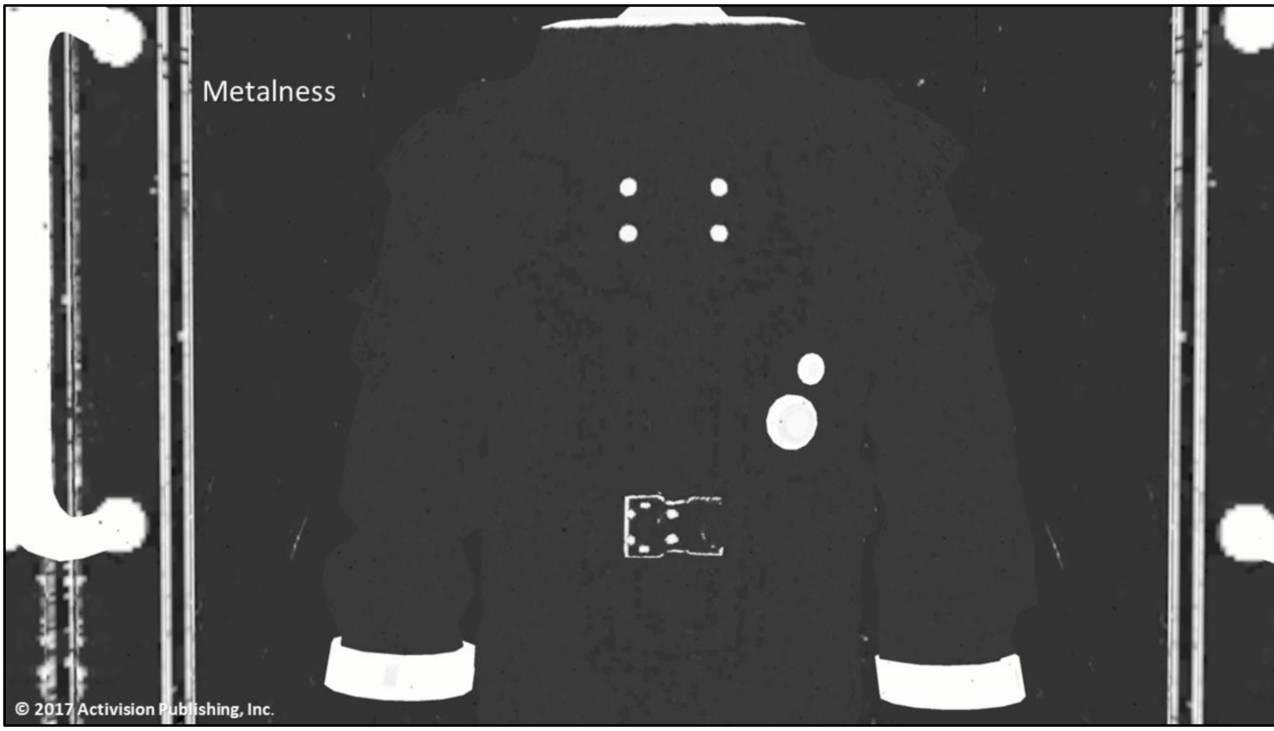
Packs textures according to rule sets
Converts between representation
    Specular Color Model -> Metalness Model
Picks best data compression scheme
    i.e. normal map packing
Calculates statistical moments (1st , 2nd moments over X and Y)
    Augments gloss maps for BRDF anti-aliasing [8]
    Similar to CLEAN / Toksvig mapping [9]
    Supports X/Y gloss maps for anisotropic materials [10]
Calculate different error metrics
    Use error metric to pick best compression schemes matching data
    Pick metric relevant to data – i.e. normal deviation for normal maps instead of color delta
Picks best texture compression format
    BC4 / BC1 / BC7

Specular Model-> Metalness Converter
Fitted ranged curves to convert between Specular and Metalness model

Preserves dielectric / insulator range

No need for separate reflectivity / F0


Fused Diffuse Specular
© 2017 Activision Publishing, Inc.



Makes assumptions about Insulator Range

- 0.0-0.1 insulators => monochromatic specular
- >0.1 dielectrics => color specular

Decompression in shader is only 5ALU

```
#define INSULATOR_SPEC_RANGE 0.1f
void DeriveMetalnessAndFusedAlbedoSpecMap( float3 albedo, float3 specular,
                                           out float3 fusedAlbedoSpec, out float metalness )
{
        float nonmetal   = ( 1.0f / 3.0f ) * ( albedo.r + albedo.g + albedo.b );
        float metal      = ( 1.0f / 3.0f ) * ( specular.r + specular.g + specular.b );

        nonmetal         = saturate( nonmetal + DATA_FORMAT_FP16_MIN_FLT );
        float specE      = saturate( metal - INSULATOR_SPEC_RANGE );
        float specI      = min( metal, INSULATOR_SPEC_RANGE );
        metalness        = specE / (specE + nonmetal );
        fusedAlbedoSpec  = ( saturate( specular - INSULATOR_SPEC_RANGE ) ) + albedo;
        metalness        = specI + ( 1.0f - INSULATOR_SPEC_RANGE ) * metalness;
}
```

Bonus Code Slide

```
void DeriveAlbedoAndSpec( float3 fusedAlbedoSpec, float metalness,
                          out float3 albedo, out float3 specular )
{
        float m         = saturate( metalness - INSULATOR_SPEC_RANGE );
              m         = m * ( 1.0f / ( 1.0f - INSULATOR_SPEC_RANGE ) );
        float r0        = min( metalness, INSULATOR_SPEC_RANGE );
        albedo          = saturate( 1.0f - m ) * fusedAlbedoSpec;
        specular        = r0 + m * ( fusedAlbedoSpec );
}
```

Bonus Code Slide

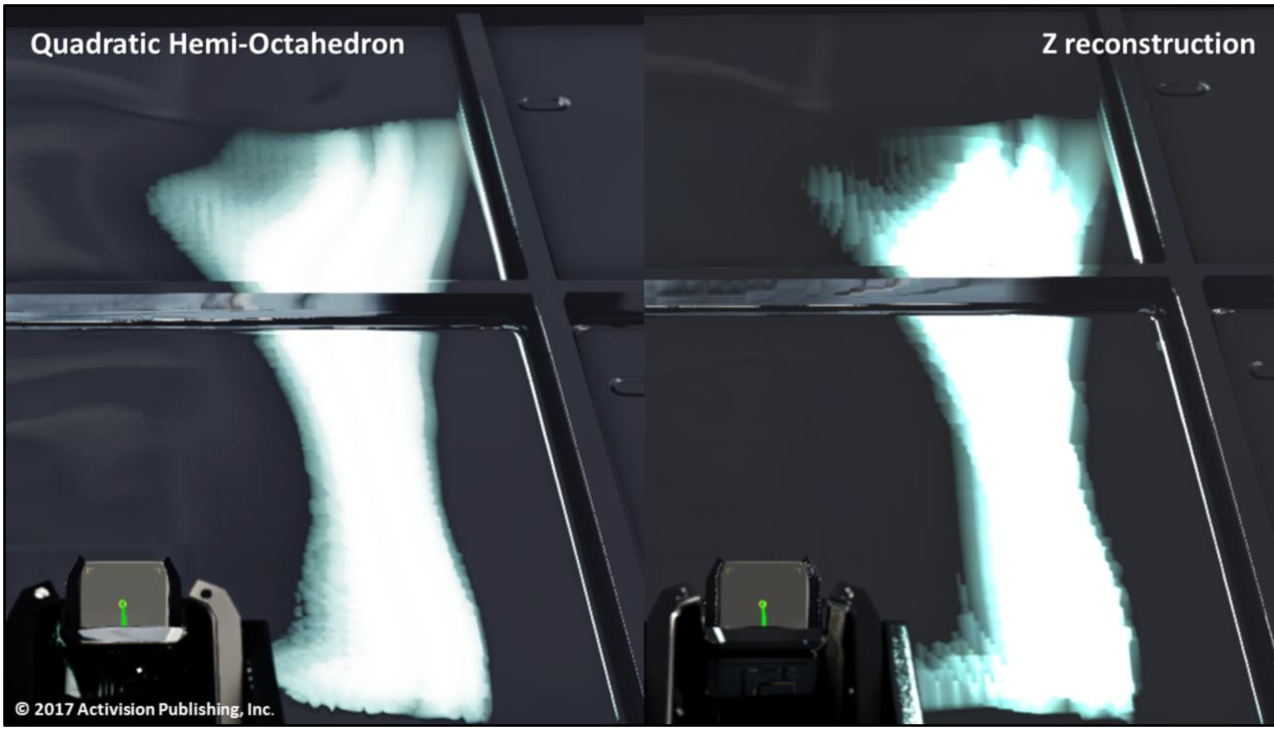
Data Analysis & Nomal Maps

- Hemi-Octahedron Normal Map compression [11]
- Data analysis
  - Use quadratic scaling for “almost flat” normal maps
  - Normal vector scaling values determined by data pipeline
    - Rescaled in shader

```
// x in [-1, 1]
float EncodeSNormQuadraticScaling( float x )
{
	float sqrtX = sqrt( abs( x ) );
	return x > 0.0f ? sqrtX : -sqrtX;
}
```

```
// x in [-1, 1]
float DecodeSNormQuadraticScaling( float x )
{
	float x2 = x * x;
	return x > 0.0f ? x2 : -x2;
}
```

Quadratic scaling around 0.0
Adds more precision to ‘flat normals’


Quadratic Hemi-Octahedron
Z reconstruction
© 2017 Activision Publishing, Inc.

Texture Packer

	Primary set (packed_CS)	Secondary set (packed_NOG)	Tertiary set (packed_ART)
R	Fused Diffuse & Specular color	Gloss - fused with Variance *(generated by converter)	Alpha
G	Fused Diffuse & Specular color	Normal X	Reveal
B	Fused Diffuse & Specular color	Occlusion	Thickness
A	Metalness mask (generated by converter)	Normal Y	

infinity ward

Digital Dragons

Packed Texture Sets

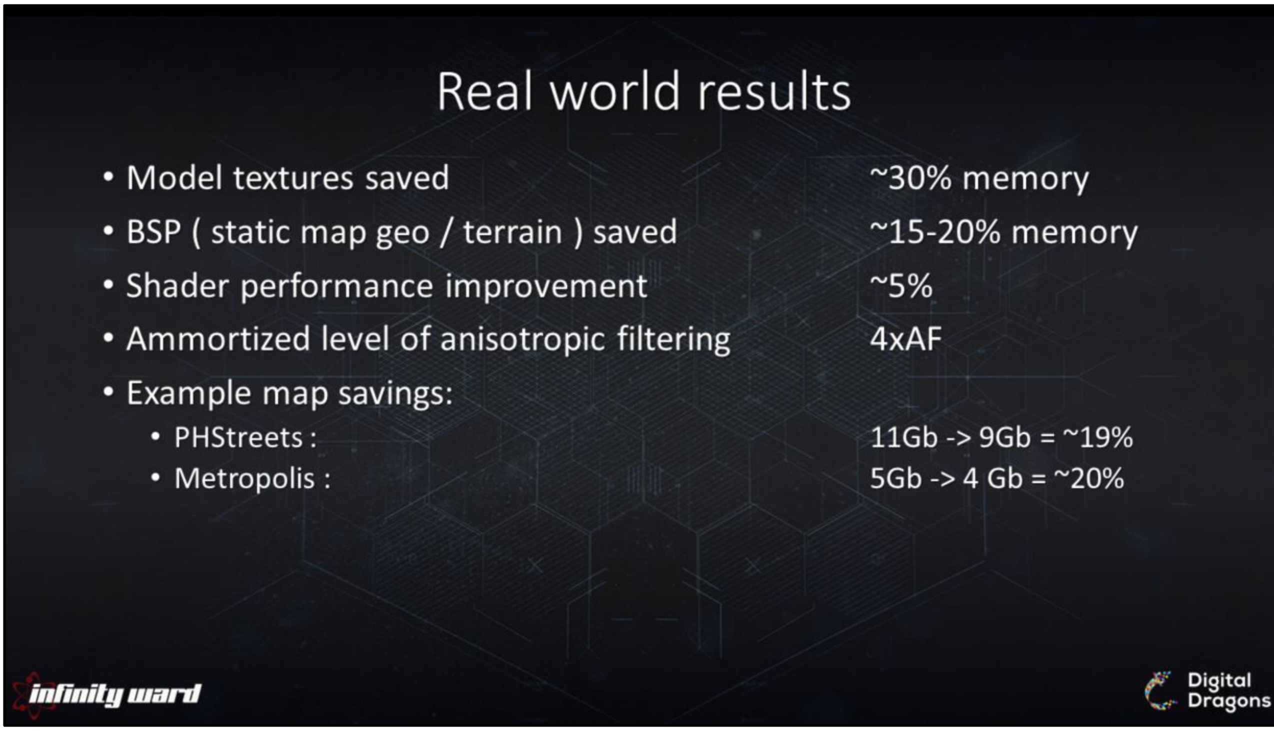
Final Converted Texture	Compression type	Bytes per pixel
CS	BC7	1.0
NOG	BC7	1.0
A / R / T <optional>	BC4 / BC1 / BC7	0.0 - 1.0
Total : 2 – 3 texture samples		2.0 - 3.0

Semantic Slot	Compression type	Bytes per pixel
Total : 4 – 5 texture samples	BC1 – BC5	4.0 – 4.5

Texture sample savings		Memory savings
50% - 40%		50% - 33%

infinity ward

Digital Dragons



Real world results differ a bit from theoretical data.
In certain cases we couldn’t use packing due to mismatched texture resolutions provided by art, thus ambiguity in packing rules.

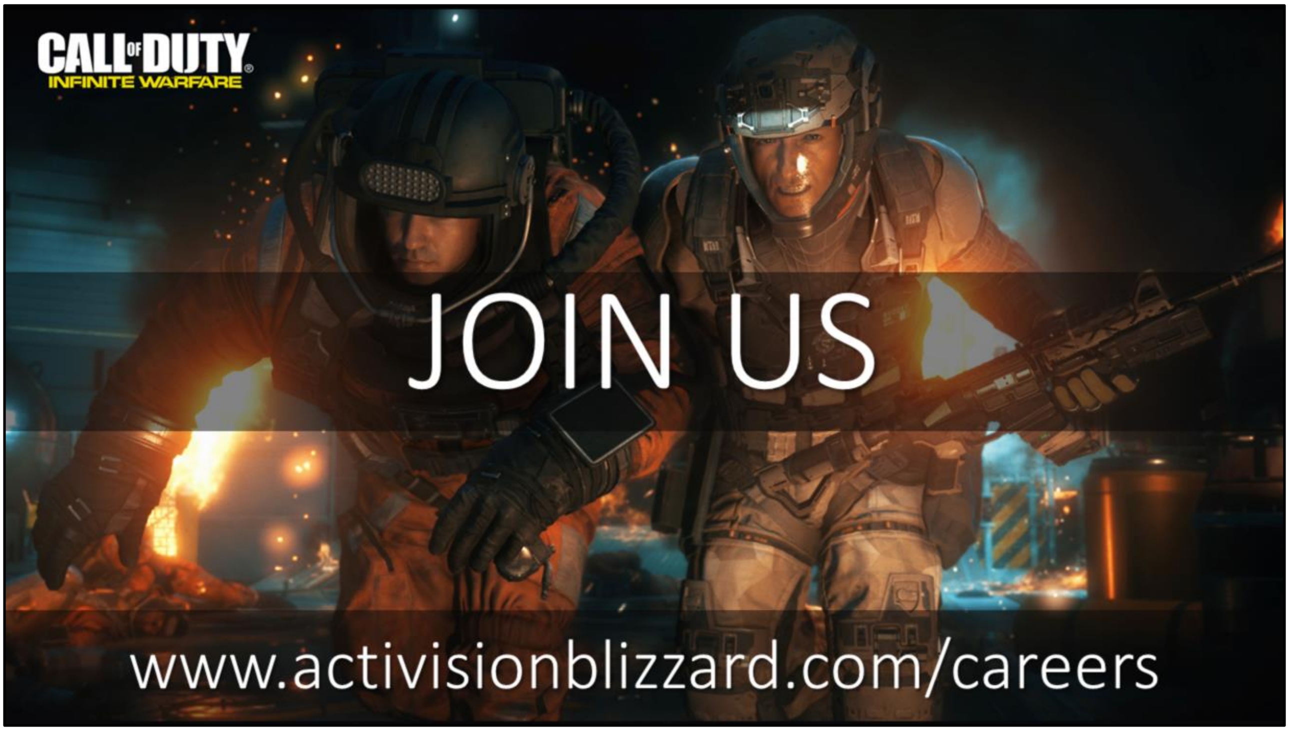
Rendering presentations 2017

- EGSR
  - Ambient Dice — Michal Iwanicki
- Siggraph
  - Indirect Lighting in COD: Infinite Warfare — Michal Iwanicki
  - Dynamic Temporal Supersampling and Anti-Aliasing — Jorge Jimenez
  - Improved Culling for Tiled and Clustered Rendering — Michal Drobot
  - Practical Multilayered PBR rendering — Michal Drobot
- Microsoft XFest 2017
  - Optimizing the Renderer of Call of Duty: Infinite Warfare — Michal Drobot

research.activision.com

infinity ward

Digital Dragons


CALL OF DUTY
INFINITE WARFARE
JOIN US
www.activisionblizzard.com/careers

Special Thanks

- ESM Shadow Map Cache :
  - Felipe Gomez
- Reflections / Refractions
  - Paul Malin
- Lightgrid / GI
  - Michal Iwanicki, Peter Pike Sloan
- Voxel Tree
  - Peter Pon
- Tessellation
  - Paul Edelstein
- Volumetric Renderer
  - Wade Brainerd, Felipe Gomez
- Particle Lighting
  - Charlie Birtwistle
- D+ Renderer
  - Michael Vance
- Texture Packing
  - Ryan Sammartino
- Various
  - Angelo Pesce, Akimitsu Hogge

infinity ward

Digital Dragons

Additional Thanks

- IW Rendering Team
  - Anthony Carotenuto, Rulon Raymond, Peter Pon, Mike Esposito, Vishal Kashyap, Felipe Gomez

- Activision Central Tech
- Infinity Ward
- Sledgehammer Games
- Treyarch
- Raven

infinity ward

Digital Dragons


research.activision.com
Q&A
michal@infinityward.com
@MichalDrobot
CALL OF DUTY
INFINITE WARFARE

References

[0] "The Devil is in the details : idTech 666", Tiago Sousa, Siggraph 2016

[1] "Secrets of CryENGINE 3 Graphics Technology", Tiago Sousa, Siggraph 2011

[2] "Extinction Transmittance Maps", P.Gautron & C.Delandre & J-E Marvie, 2011

[3] "Hybrid Reconstruction Antialiasing", Michal Drobot, GPU PRO 6

[4] "Fast Filtering of Reflection Probes", Josiah Manson and Peter-Pike Sloan, EGSR 2016

[5] "Getting More Physical in Call of Duty: Black Ops II", Dimitar Lazarov, Siggraph 2013

[6] "Physically-based & Unified Volumetric Rendering in Frostbite", Sebastien Hillaire, Siggraph 2015

[7] "Volumetric fog: Unified, compute shader based solution to atmospheric scattering", Bart Wronski, Siggraph 2014

[8] "Lean Mapping", Marc Olano and Dan Baker

[9] "Specular Showdown in the Wild West", Stephen Hill, http://blog.selfshadow.com

[10] "Advanced VR Rendering", Alex Vlachos, GDC 2014

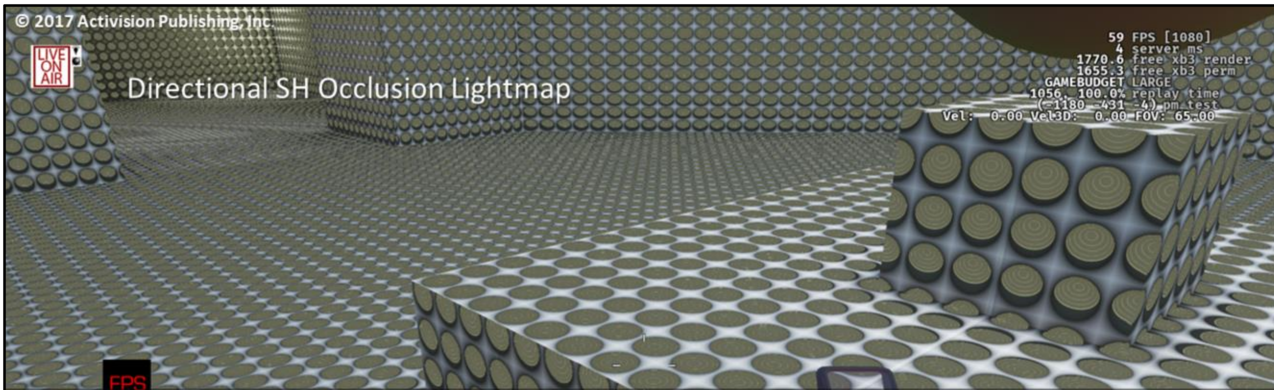
[11] "A Survey of Efficient Representations for Independent Unit Vectors", Cigolle et al., Journal of Computer Graphics Techniques Vol. 3, No. 2, 2014

infinity ward

Digital Dragons

Bonus Slides

Directional SH Occlusion Lightmap



- Directional SH1 occlusion lightmaps
  - High memory cost
    - Need additional 4 channels
    - 2 x BC5 (Quality) or 1xBC7 (Performance & Memory)
  - Each ‘cone’ represents a bent cone calculated out of stored SH1 coefficients
- NOT shipped – but ready for future work

Notice crate on the right lacking reflection shadowing.

Notice significantly improved localized shadowing near room corners.

```
// Pack into hemisphere octahedron
// Assume normalized input on +Z hemisphere. Output [-1, 1].
void EncodeHemiOctaNormal( const float3 v, inout float2 encV )
{
        // Project the hemisphere onto the hemi-octahedron, and then into the xy plane
        float rcp_denom = 1.0f / ( abs( v[0] ) + abs( v[1] ) + v[2] );
        float tx = v[0] * rcp_denom;
        float ty = v[1] * rcp_denom;
        encV[0] = tx + ty;
        encV[1] = tx - ty;
}

void DecodeHemiOctaNormal( const float2 encV, inout float3 v )
{
        //      Rotate and scale the unit square back to the center diamond
        v[0] = ( encV[0] + encV[1] ) * 0.5f;
        v[1] = ( encV[0] - encV[1] ) * 0.5f;
        v[2] =  1.0f - abs( v[0] ) - abs( v[1] );
}
```

Bonus Code Slide