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
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
    - $160/4 \times 90/4 \times 128/4 = 40 \times 25 \times 32$ @1080p
  - Used for transparent geometry and volumetrics
Frustum Space

- Items indexed by bits
  - Lights
  - Reflection Probes
  - Density Volumes
  - Decals [0]
Mesh Rendering
Geometry overview
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 makes a huge visual impact at moderate adaptive performance hit. Here exaggerated for visual presentation.
Shadow Map Cache
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
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
  - Downsampling 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
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 \( \max(\text{intensity0...}) \)
  Perform a single trace in key light direction
Particle Lighting
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
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
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

<table>
<thead>
<tr>
<th>RGB SH1 512 CS Job</th>
<th>Time (ms) @PS4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lighting</td>
<td>0.1-0.7</td>
</tr>
<tr>
<td>Scattering</td>
<td>0.1</td>
</tr>
</tbody>
</table>
Multi-Res Rendering
Initially developed for VFX rendering

- Dense VFX results in heavy overdraw – need to optimize
- VFX team wanted to keep sorting ‘as is’
  - Classic Low Res rendering requires merge pass injected during rendering
  - Changes / Complicates sorting order
- MSAA based Multi Resolution Rendering pipeline
  - Allows to keep rendering ‘as is’
  - Individual Effects / Materials can be tagged for ‘Low Res Rendering’
  - Console only for now (pending IHV support for MSAA extensions)
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.
Notice geometric edges marked with multiple 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
Alpha Buffer
Notice geometric edges marked with multiple 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
  - 0.3ms – 0.4ms up-sample / resolve / reconstruct pass
    - Variance comes from amount of micro tiles that need all subsamples
  - Mileage may vary depending on GPU MSAA efficiency

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
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
- XYZ blend zones
- Stored as CubeArray of 64 x 128^2 BC6 textures
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

<table>
<thead>
<tr>
<th>Shader</th>
<th>time (ms) @PS4</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT Culling of 64 reflection probes</td>
<td></td>
</tr>
<tr>
<td>in avg. open scene</td>
<td></td>
</tr>
<tr>
<td>32x24x48</td>
<td>0.185</td>
</tr>
</tbody>
</table>
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.
Relightable Reflection Probes: Performance

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

<table>
<thead>
<tr>
<th>Compute Job on 128^2 Cubemap</th>
<th>Time (ms) @ PS4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relighting (depends on # of dynamic lights)</td>
<td>0.1-0.2</td>
</tr>
<tr>
<td>Filtering all MIPs</td>
<td>0.31</td>
</tr>
<tr>
<td>BC6 compression all MIPs</td>
<td>0.18</td>
</tr>
</tbody>
</table>
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 normalization.
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 (<0.1 roughness)
- Derive specular from lightgrid data (evalSH(reflectionDir))
- Blend with reflection probe in transition threshold
- ~0.5ms average savings

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
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
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.
Very cheap in comparison with fully traced methods

<table>
<thead>
<tr>
<th>Shader</th>
<th>Time (ms) @PS4 @ 1080p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scene mip generation</td>
<td>0.25</td>
</tr>
<tr>
<td>Full screen SS Reflection surface</td>
<td>+0.3</td>
</tr>
<tr>
<td>Full screen SS Refraction surface</td>
<td>+0.5</td>
</tr>
</tbody>
</table>
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)
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
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

<table>
<thead>
<tr>
<th>CS Job</th>
<th>Fixed R/W BW</th>
<th>Execution time</th>
<th>Cost</th>
<th>VGPR</th>
<th>Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple Passes</td>
<td>30,920 KB</td>
<td>~2.0 ms</td>
<td>100%</td>
<td>MAX( 48 )</td>
<td>Mixed</td>
</tr>
<tr>
<td>Merged</td>
<td>7,730 KB</td>
<td>1.2 ms</td>
<td>60%</td>
<td>48</td>
<td>4x4x4</td>
</tr>
</tbody>
</table>
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
# Core Texture Slots

<table>
<thead>
<tr>
<th>Semantic Slot</th>
<th>Compression type</th>
<th>Bytes per pixel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diffuse (RGB) + Alpha (A)</td>
<td>BC1 / BC3</td>
<td>0.5 / 1.0</td>
</tr>
<tr>
<td>Specular (RGB) + Gloss (A)</td>
<td>BC3</td>
<td>1.0</td>
</tr>
<tr>
<td>Normal (XY)</td>
<td>BC5</td>
<td>1.0</td>
</tr>
<tr>
<td>Occlusion (A)</td>
<td>BC4</td>
<td>0.5</td>
</tr>
<tr>
<td>Reveal (A) &lt;BSP only&gt;</td>
<td>BC4</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Total: 4 – 5 texture samples 4.0 – 4.5
## Additional Texture Slots

<table>
<thead>
<tr>
<th>Semantic Slot</th>
<th>Compression type</th>
<th>Bytes per pixel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thickness</td>
<td>BC4</td>
<td>0.5</td>
</tr>
<tr>
<td>Absorption</td>
<td>BC1</td>
<td>0.5</td>
</tr>
<tr>
<td>Fluorescence</td>
<td>BC1</td>
<td>0.5</td>
</tr>
<tr>
<td>Sheen and Cloth</td>
<td>BC1</td>
<td>0.5</td>
</tr>
<tr>
<td>Anisotropy</td>
<td>BC7</td>
<td>1.0</td>
</tr>
<tr>
<td>Reveal</td>
<td>BC4</td>
<td>0.5</td>
</tr>
<tr>
<td>Thickness</td>
<td>BC4</td>
<td>0.5</td>
</tr>
</tbody>
</table>
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
Makes assumptions about Insulator Range
  0.0-0.1 insulators => monochromatic specular
  >0.1 dielectrics => color specular
Decompression in shader is only 5ALU
```c
#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_MINFLT );
    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);
}
Quadratic scaling around 0.0
Adds more precision to ‘flat normals’
## Texture Packer

<table>
<thead>
<tr>
<th>Primary set (packed_CS)</th>
<th>Secondary set (packed_NOG)</th>
<th>Tertiary set (packed_ART)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R Fused Diffuse &amp; Specular color</td>
<td>Gloss - fused with Variance *(generated by converter)</td>
<td>Alpha</td>
</tr>
<tr>
<td>G Fused Diffuse &amp; Specular color</td>
<td>Normal X</td>
<td>Reveal</td>
</tr>
<tr>
<td>B Fused Diffuse &amp; Specular color</td>
<td>Occlusion</td>
<td>Thickness</td>
</tr>
<tr>
<td>A Metalness mask <em>(generated by converter)</em></td>
<td>Normal Y</td>
<td></td>
</tr>
</tbody>
</table>
# Packed Texture Sets

<table>
<thead>
<tr>
<th>Final Converted Texture</th>
<th>Compression type</th>
<th>Bytes per pixel</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>BC7</td>
<td>1.0</td>
</tr>
<tr>
<td>NOG</td>
<td>BC7</td>
<td>1.0</td>
</tr>
<tr>
<td>A / R / T &lt;optional&gt;</td>
<td>BC4 / BC1 / BC7</td>
<td>0.0 - 1.0</td>
</tr>
<tr>
<td><strong>Total : 2 – 3 texture samples</strong></td>
<td></td>
<td>2.0 - 3.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Semantic Slot</th>
<th>Compression type</th>
<th>Bytes per pixel</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total : 4 – 5 texture samples</strong></td>
<td>BC1 – BC5</td>
<td>4.0 – 4.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Texture sample savings</th>
<th>Memory savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>50% - 40%</td>
<td>50% - 33%</td>
</tr>
</tbody>
</table>
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
  • Siggraph
    • Indirect Lighting in COD: Infinite Warfare
    • Dynamic Temporal Supersampling and Anti-Aliasing
    • Improved Culling for Tiled and Clustered Rendering
    • Practical Multilayered PBR rendering
  • Microsoft XFest 2017
    • Optimizing the Renderer of Call of Duty: Infinite Warfare

research.activision.com
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
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
research.activision.com

Q&A

michal@infinityward.com
@MichalDrobot
References

[0] “The Devil is in the details : idTech 666”, Tiago Sousa, Siggraph 2016
Bonus Slides
• Directional SH1 occlusion lightmaps
  • High memory cost
    • Need additional 4 channels
    • 2x BC5 (Quality) or 1x BC7 (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.
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] );
}