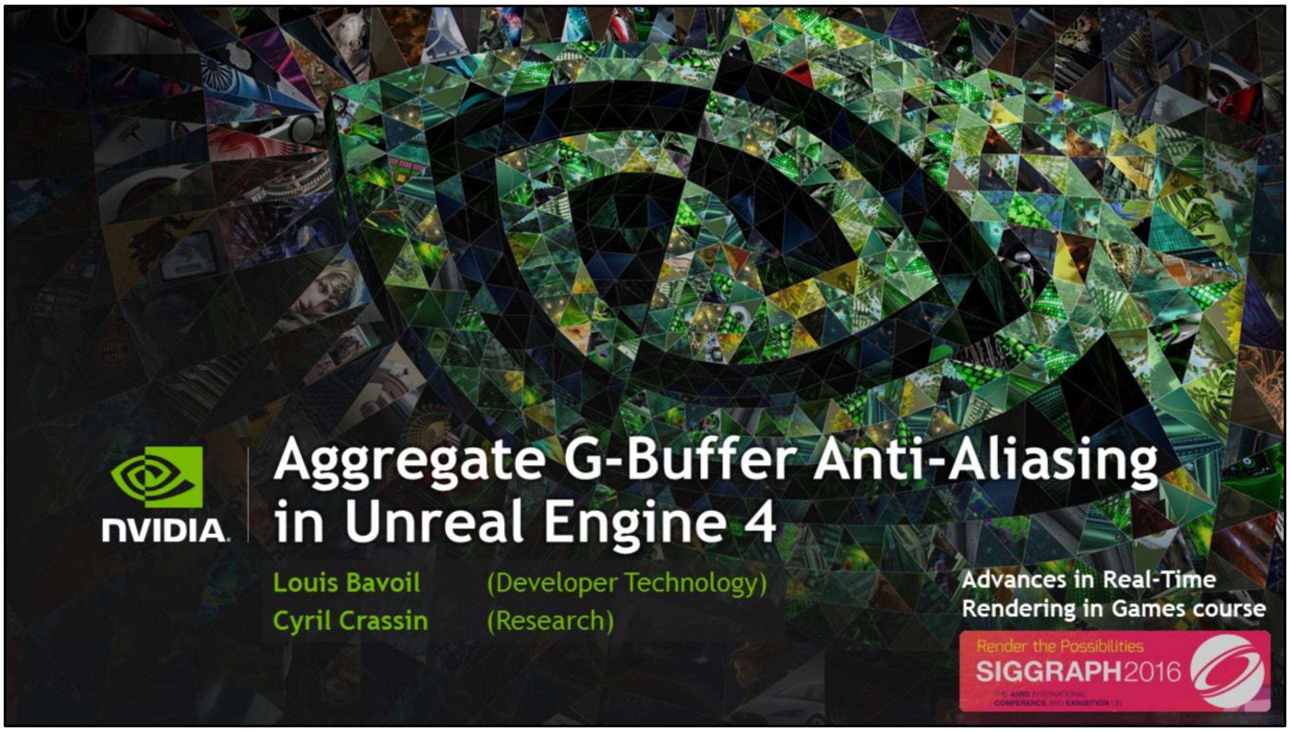


Aggregate G-Buffer Anti-Aliasing in Unreal Engine 4

Louis Bavoil (Developer Technology)
Cyril Crassin (Research)

Advances in Real-Time Rendering in Games course

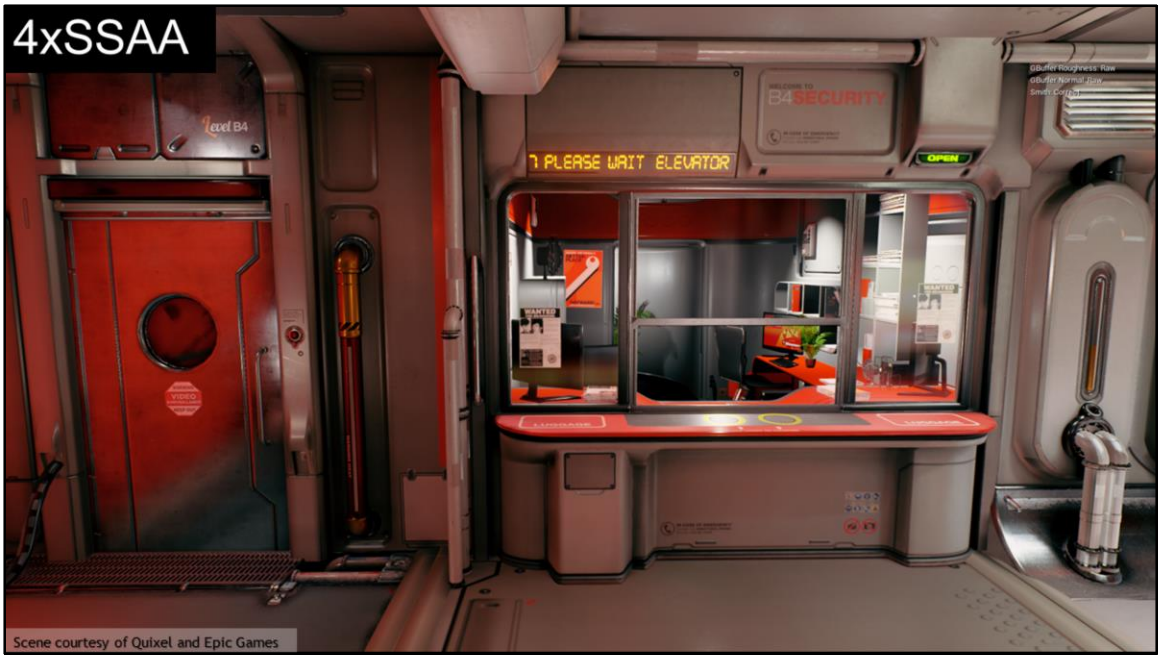
Render the Possibilities
SIGGRAPH2016

4x TAA = 4x SSAA with TAA

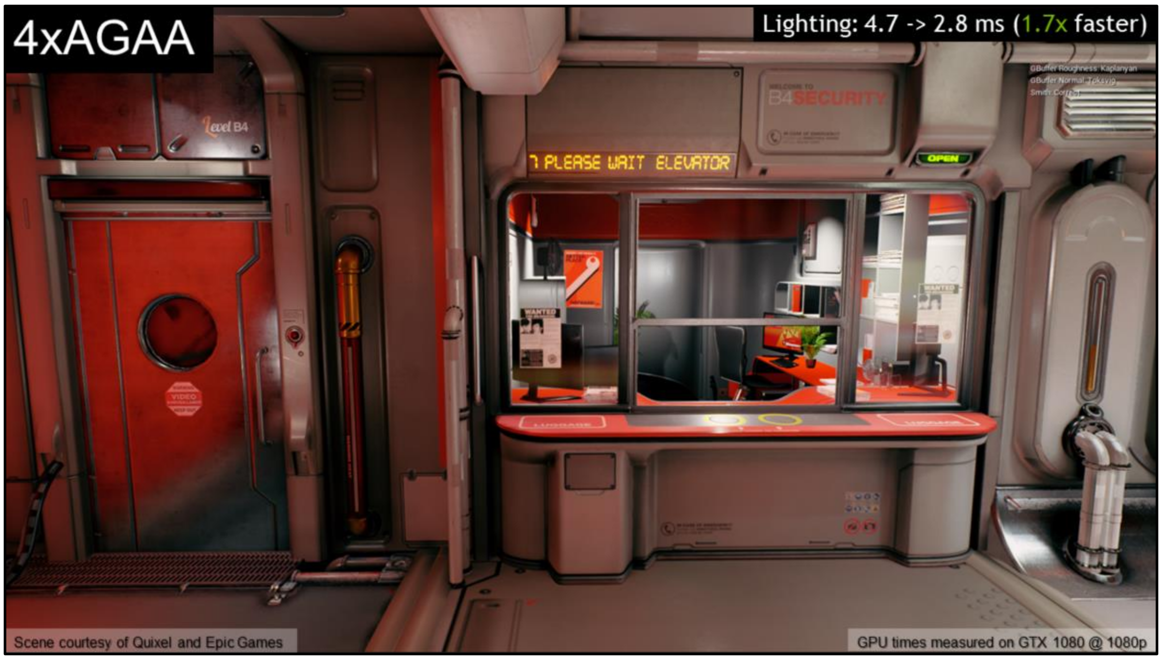
Level B4
7 PLEASE WAIT ELEVATOR
WELCOME TO
B4SECURITY
OPEN
WANTED
WANTED
GBuffer Roughness: Raw
GBuffer Normal: Raw
Smith: Correct
Scene courtesy of Quixel and Epic Games

Currently in UE4, the best out-of-the-box experience you can get for anti-aliasing is by using “Temporal Anti-Aliasing”, applied on a super-sampled color buffer, typically rendered using 4x supersampling.

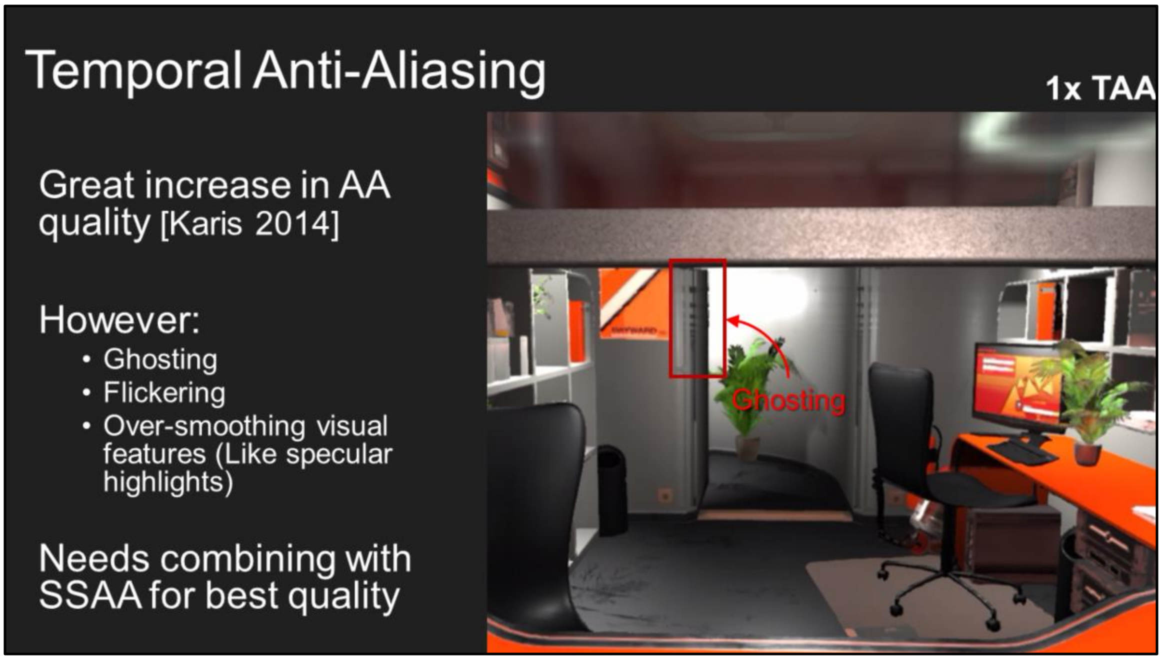
It’s the same for other major game engines. Supersampling plus TAA is an option which is exposed and already shipping in major AAA games from Frostbite, some UE4 titles and probably others.



This is the result we get doing only 4x SSAA, without TAA…


4xAGAA
Lighting: 4.7 -> 2.8 ms (1.7x faster)
GBuffer Roughness: Kaplanyan
GBuffer Normal: Tokuyio
Smith: Correlated
7 PLEASE WAIT ELEVATOR
Scene courtesy of Quixel and Epic Games
GPU times measured on GTX 1080 @ 1080p

And this is also 4x SSAA, but running the lighting pass 1.7x faster using Aggregate G-Buffer Anti-Aliasing.



Temporal Anti-Aliasing

Great increase in AA quality [Karis 2014]

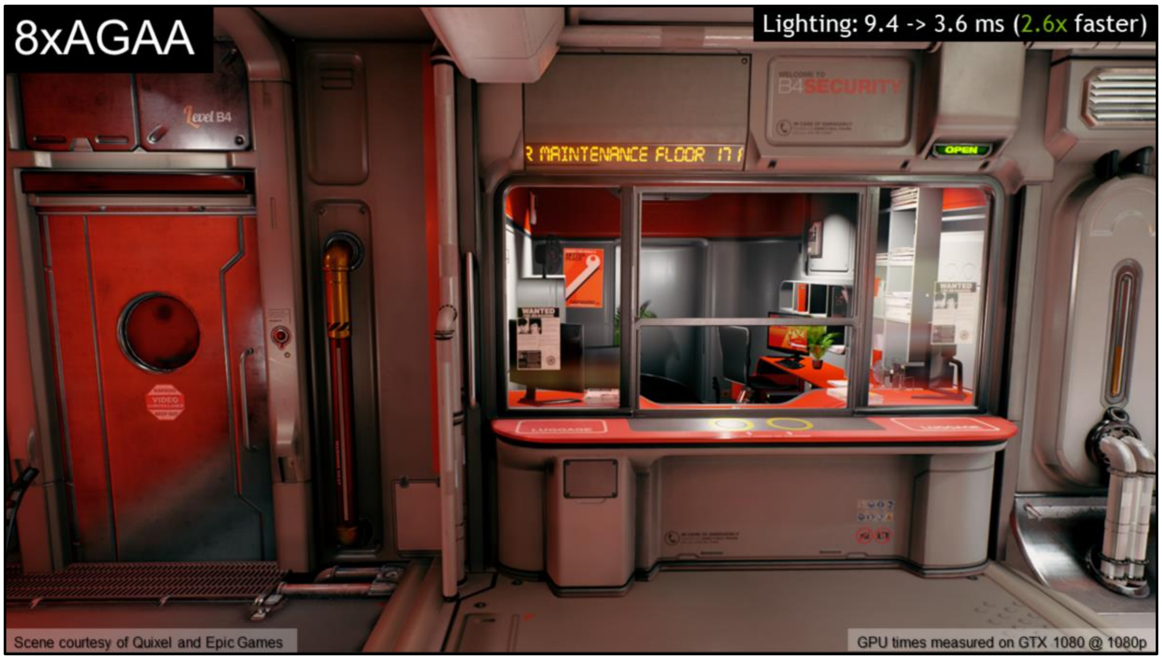
However:
- Ghosting
- Flickering
- Over-smoothing visual features (Like specular highlights)

Needs combining with SSAA for best quality

1x TAA
Ghosting

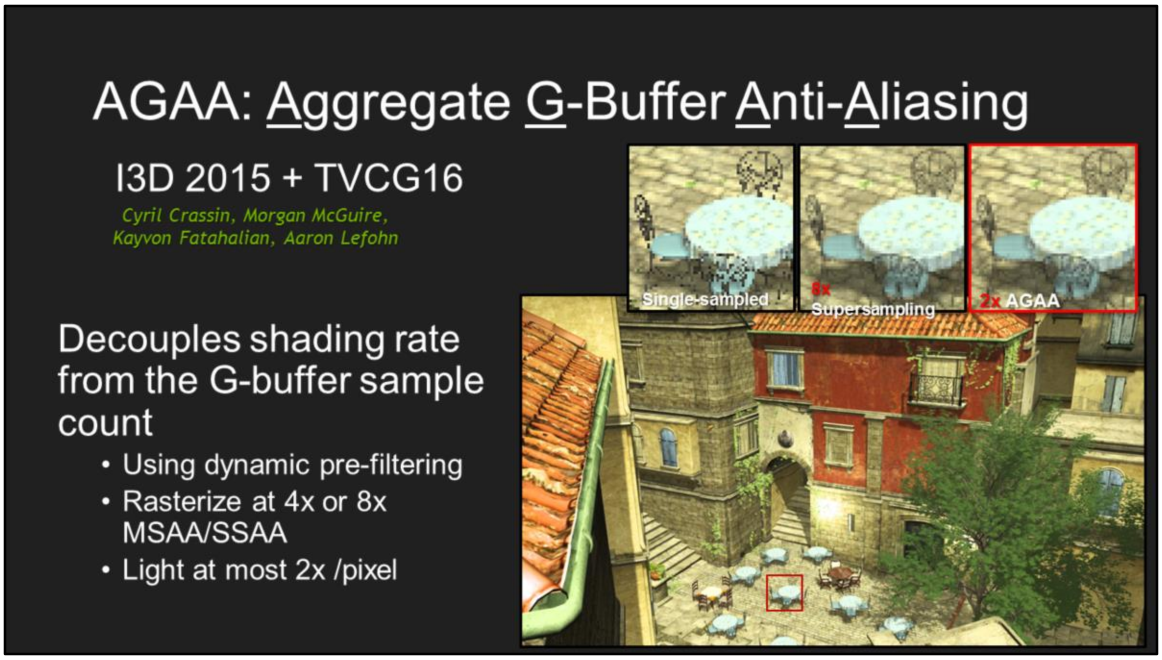
TAA is a massive improvement in anti-aliasing quality in games. However, there is some issues when used without super sampling:
There can be some ghosting, some flickering or some over-blurring of visual features.

In general, it greatly benefits from providing more screen-space and temporal coherence. That’s why it is usually combined with supersampling for best image quality.
The problem is that super-sampled rendering is quite costly….


8xAGAA
Lighting: 9.4 -> 3.6 ms (2.6x faster)
Scene courtesy of Quixel and Epic Games
GPU times measured on GTX 1080 @ 1080p
R MAINTENANCE FLOOR
B4SECURITY
OPEN
WANTED

Given that AGAA makes supersampling less expensive, you can imagine pushing to higher sampling rates, maybe high enough that you can turn off TAA. In this example we do 8x AGAA without TAA. Running lighting 2.6x faster than 8x supersampling.

This gives us image quality close to 8x SSAA, at a similar cost to 4x SSAA.



AGAA: Aggregate G-Buffer Anti-Aliasing

I3D 2015 + TVCG16

Cyril Crassin, Morgan McGuire, Kayvon Fatahalian, Aaron Lefohn

Decouples shading rate from the G-buffer sample count

- Using dynamic pre-filtering
- Rasterize at 4x or 8x MSAA/SSAA
- Light at most 2x /pixel

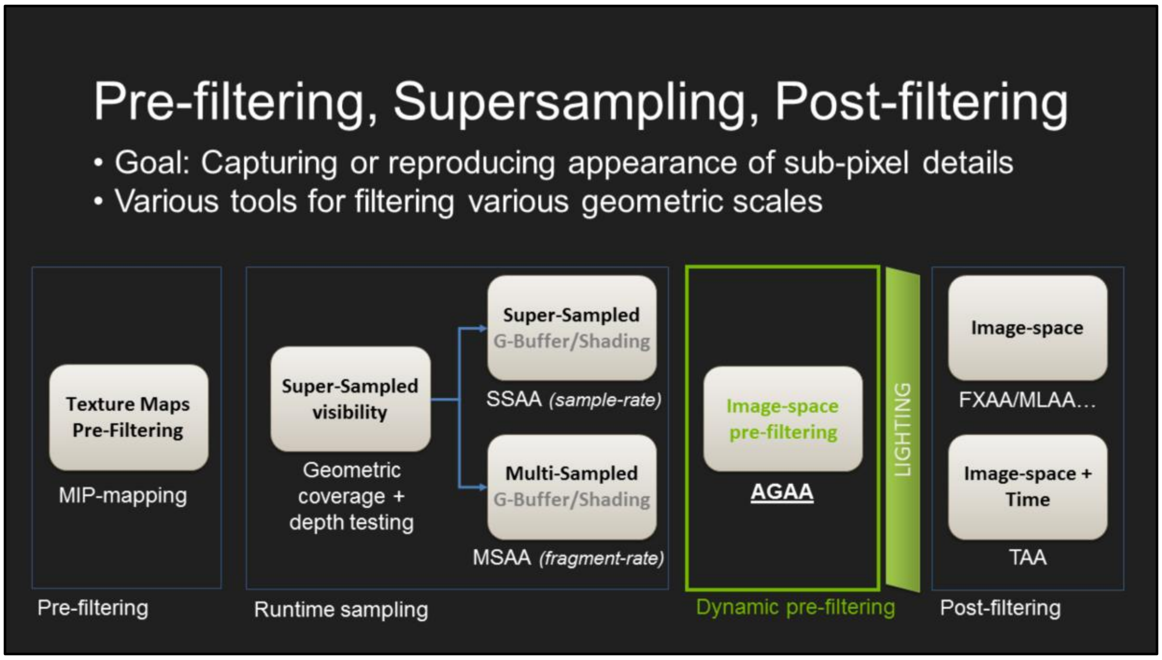
Single-sampled
8x Supersampling
2x AGAA

So last year at I3D we introduced AGAA. And there is now an extended version with more details we published at TVCG.

The high-level idea of this technique is to decouple the shading rate, from the geometry and materials sampling rates, in the context of deferred shading pipelines. For that, we use some form of dynamic pre-filtering in screen-space.

This allows us to greatly reduce the cost of lighting.

What we want to do in this talk, it to give you an overview of the current state of our exploration around taking these research ideas into a game engine.
We will talk about some of the algorithmic changes, and the challenges we faced.



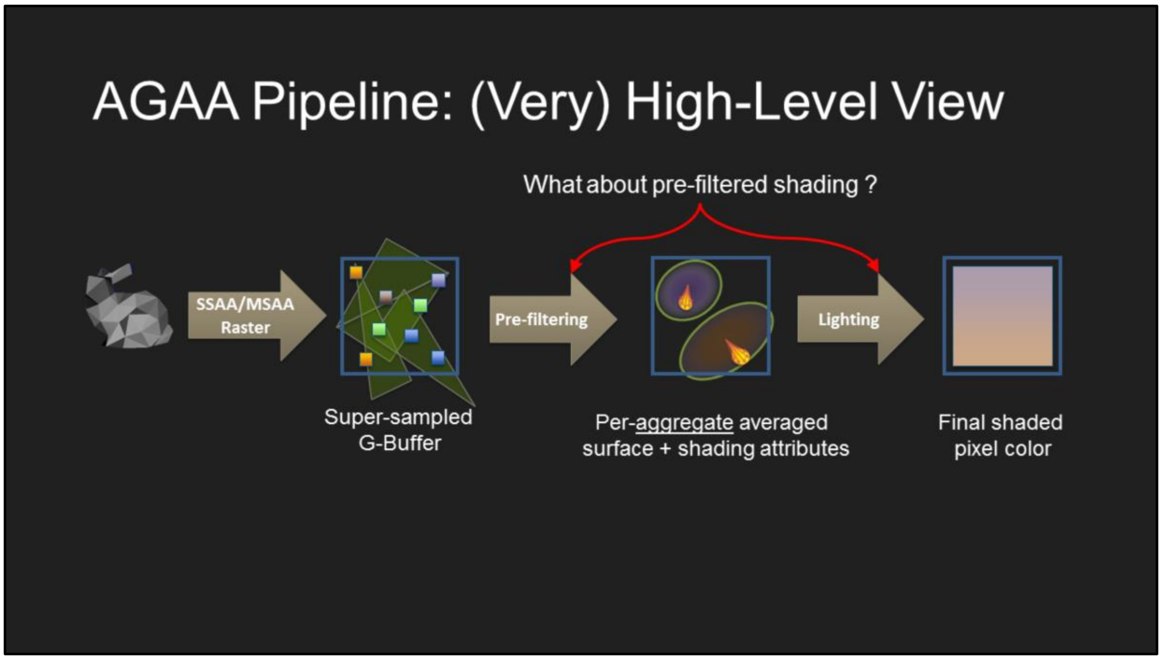
Pre-filtering, Supersampling, Post-filtering

- Goal: Capturing or reproducing appearance of sub-pixel details
- Various tools for filtering various geometric scales

Let’s take a step back and look at the most common options that we have for anti-aliased rendering in real-time applications:

- Most simple: Supersampled rasterization (SSAA), and its optimization Multi-sampled rasterization (MSAA). MSAA allows capturing actual triangle details, with a maximum of 8 samples per-pixel on current GPUs.
- Even when using supersampling, the number of samples that can be evaluated for each pixel is necessarily bounded, and higher-frequency details will not be captured and filtered properly. That’s why pre-filtering tools are also needed. Real-time renderers typically represent micro details in texture-maps instead of actual geometry, and those maps are pre-filtered using MIP-mapping.
- Finally, post-filtering techniques operate post-lighting, on the final shaded color values of the pixels/samples, and reconstruct or hallucinate sub-pixel details by taking advantage of spatial coherence with neighboring pixels (for FXAA/MLAA…), or also temporal coherence by re-projecting samples from previous-frames.

All those tools are generally combined in order to obtain the best possible quality within a given time budget.
AGAA proposes a new kind of pre-filtering in the context of deferred shading pipelines, which is performed in image-space and computed on the fly from dynamically sampled shading attributed.



AGAA Pipeline: (Very) High-Level View

What about pre-filtered shading ?
SSAA/MSAA Raster
Super-sampled G-Buffer
Pre-filtering
Per-aggregate averaged surface + shading attributes
Lighting
Final shaded pixel color

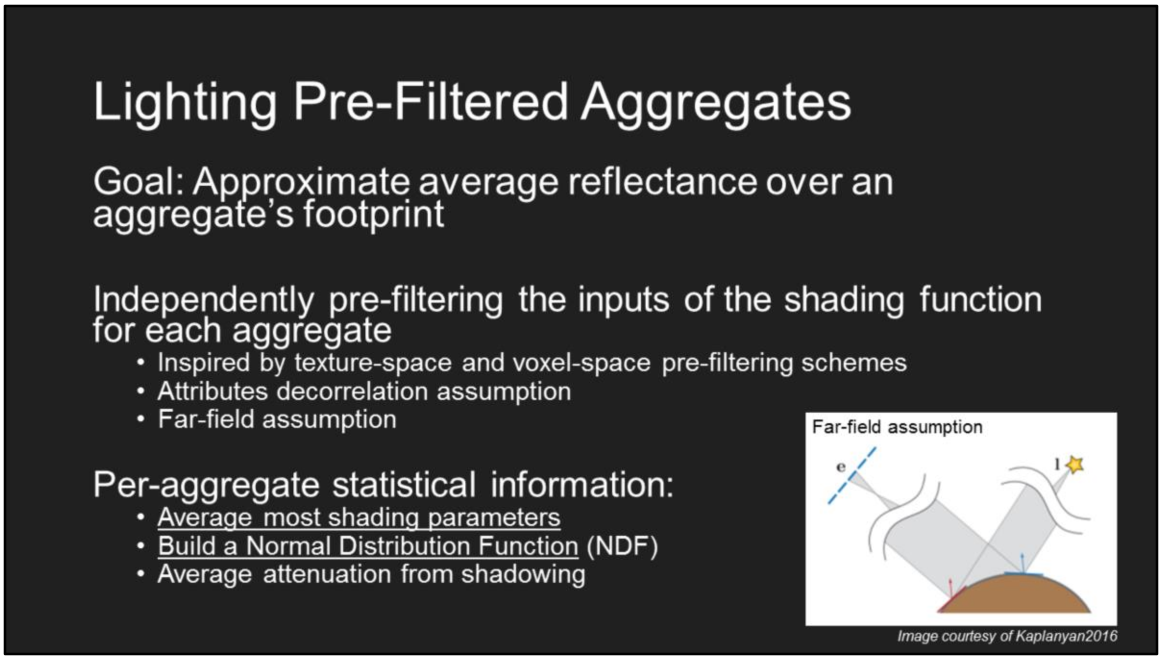
Now, let' see how AGAA works in UE4:

- # The starting point of AGAA is a supersampled G-buffer, which uses the exact same layout as the standard UE4 G-buffer.

- # Then, instead of shading each sub-sample individually, as we would normally do, we combine those sub-samples together inside what we call the "aggregates".
  The aggregates contain pre-filtered shading attributes averaged from the sample values.
  From our experience, using two aggregates is generally enough with up to 8x MSAA.

- # Finally, after generating the aggregates, we will light each of them once, regardless of what rate we sampled the g-buffer at.

# Before digging a little bit more inside the details of this pipeline, let's take a look at the pre-filtering.



Lighting Pre-Filtered Aggregates

Goal: Approximate average reflectance over an aggregate's footprint

Independently pre-filtering the inputs of the shading function for each aggregate

- Inspired by texture-space and voxel-space pre-filtering schemes
- Attributes decorrelation assumption
- Far-field assumption

Per-aggregate statistical information:

- Average most shading parameters
- Build a Normal Distribution Function (NDF)
- Average attenuation from shadowing

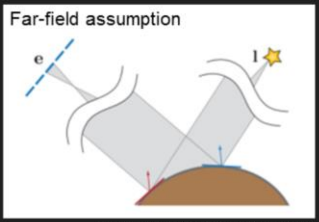


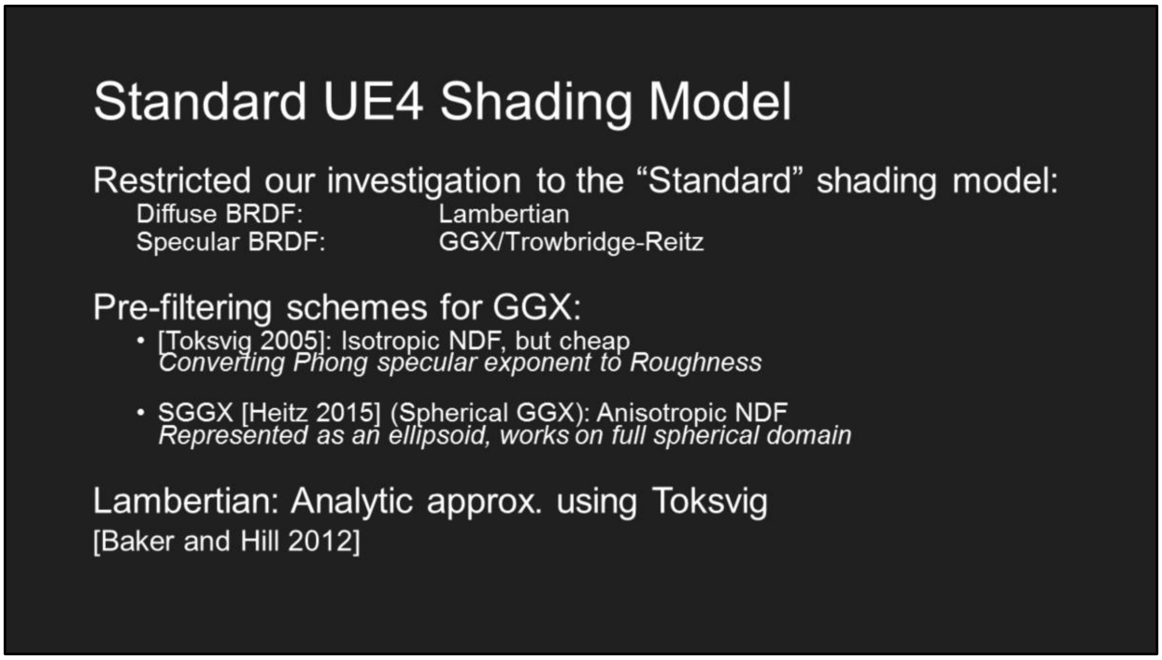
Image courtesy of Kaplanyan2016

The goal of the pre-filtering scheme is to allow approximating the average reflectance over the footprint of an aggregate, by linearly and independently pre-filtering all the inputs of the shading function.

For that, we take inspiration from texture-space and voxel-space pre-filtering schemes for analytic microfacet BRDFs.

In practice, we average most of the shading parameters, and we build a normal distribution function used to filter the microfacet BRDF model.

Among other details, it's interesting to note here that we also pre-filter shadowing from UE's attenuation buffer.


Standard UE4 Shading Model

Restricted our investigation to the “Standard” shading model:

Diffuse BRDF:	Lambertian
Specular BRDF:	GGX/Trowbridge-Reitz

Pre-filtering schemes for GGX:

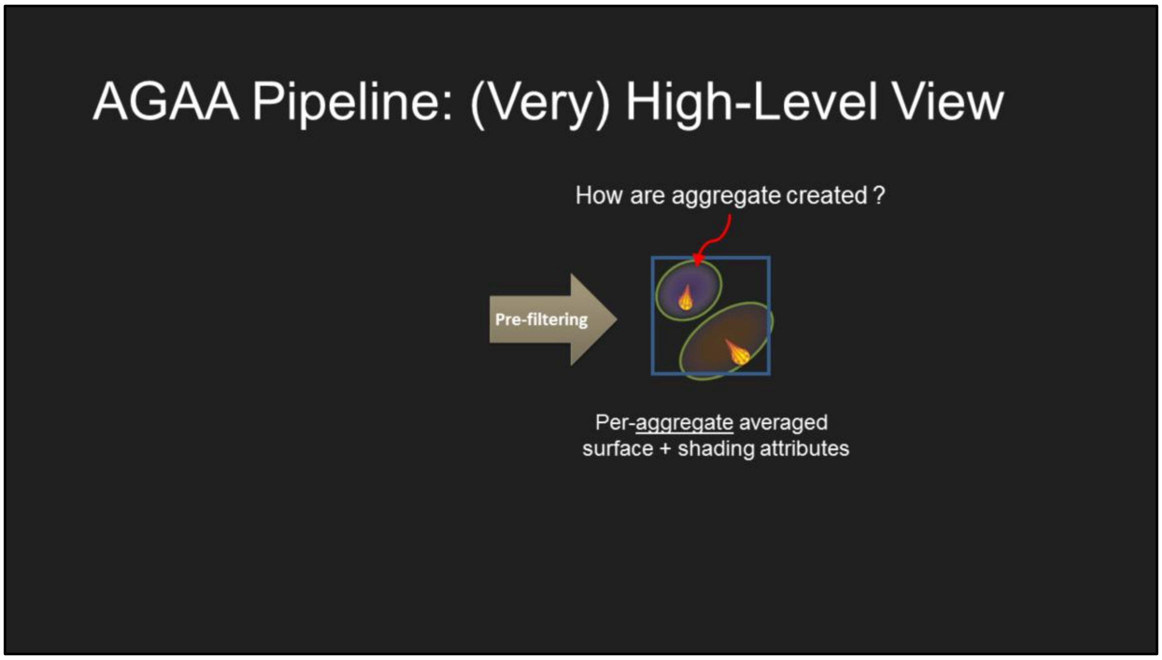
- [Toksvig 2005]: Isotropic NDF, but cheap
  Converting Phong specular exponent to Roughness
- SGGX [Heitz 2015] (Spherical GGX): Anisotropic NDF
  Represented as an ellipsoid, works on full spherical domain

Lambertian: Analytic approx. using Toksvig
[Baker and Hill 2012]

So far, we have focused only on pre-filtering the “standard” shading model in UE4. It s based on a simple Lambertian diffuse BRDF, and a GGX specular BRDF.

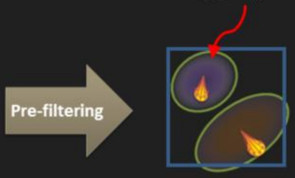
The specular GGX BRDF is the most important one to filter. For that, we experimented both with Toksvig, which only encodes an isotropic variance.
SGGX generalizes the GGX definition and allows encoding microfacets spread spatially, and encodes anisotropic variance.

We have also experimented with the Lambertian analytic approximation from *[Baker and Hill 2012]* and found that it made subtle differences on certain diffuse objects, and almost made no visual difference on the SciFiHallway scene. All our results in this presentation were generated without it.

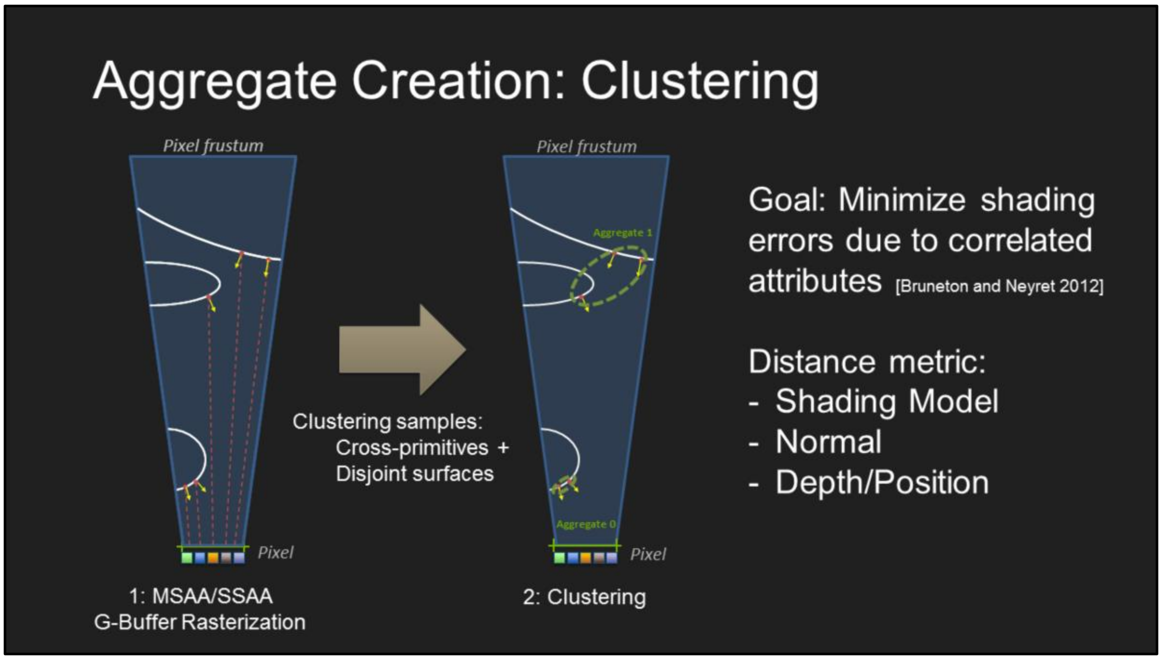


AGAA Pipeline: (Very) High-Level View

How are aggregate created ?



Per-aggregate averaged
surface + shading attributes



Aggregate Creation: Clustering

Clustering samples:
Cross-primitives +
Disjoint surfaces

1: MSAA/SSAA
G-Buffer Rasterization

2: Clustering

Goal: Minimize shading errors due to correlated attributes [Bruneton and Neyret 2012]

Distance metric:
- Shading Model
- Normal
- Depth/Position

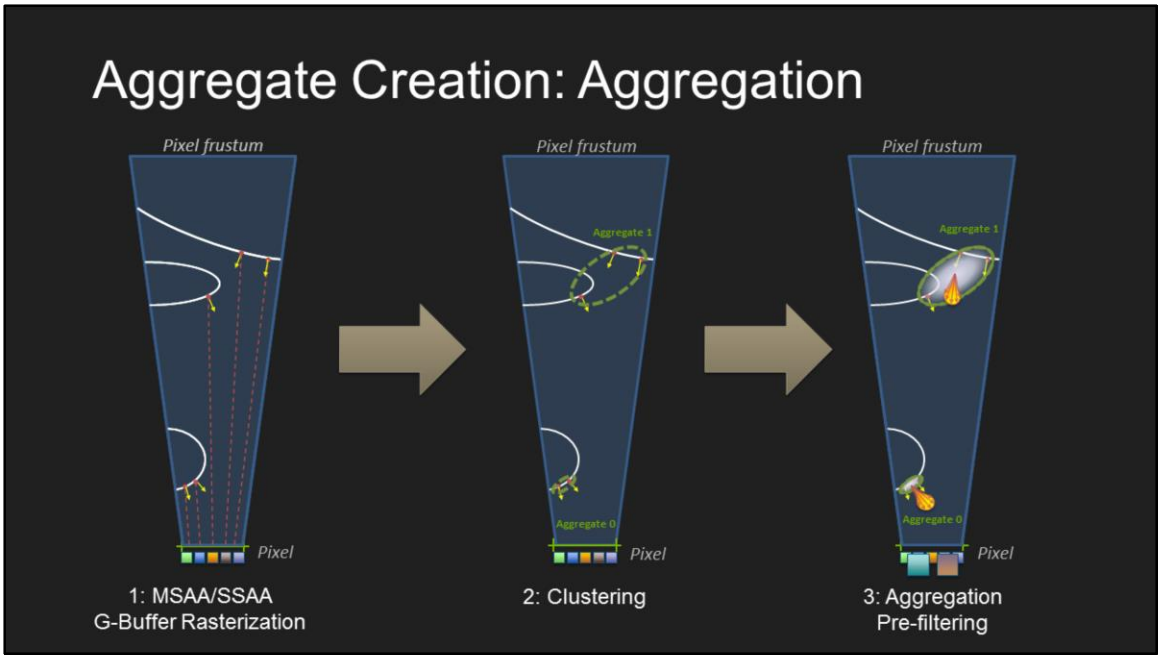
This diagram gives an example of some visible geometry within the frustum of a given pixel.

# Once we have generated the full supersampled g-buffer, the first step is to associate each sub-sample to one of the two aggregates.

# We use a clustering pass for that.

In contrast to texture-space pre-filtering schemes, aggregates can contain samples from totally different surfaces that can be spread spatially.
What we want to do here with the clustering is basically to optimize the quality of the pre-filtering, and minimize the shading errors.

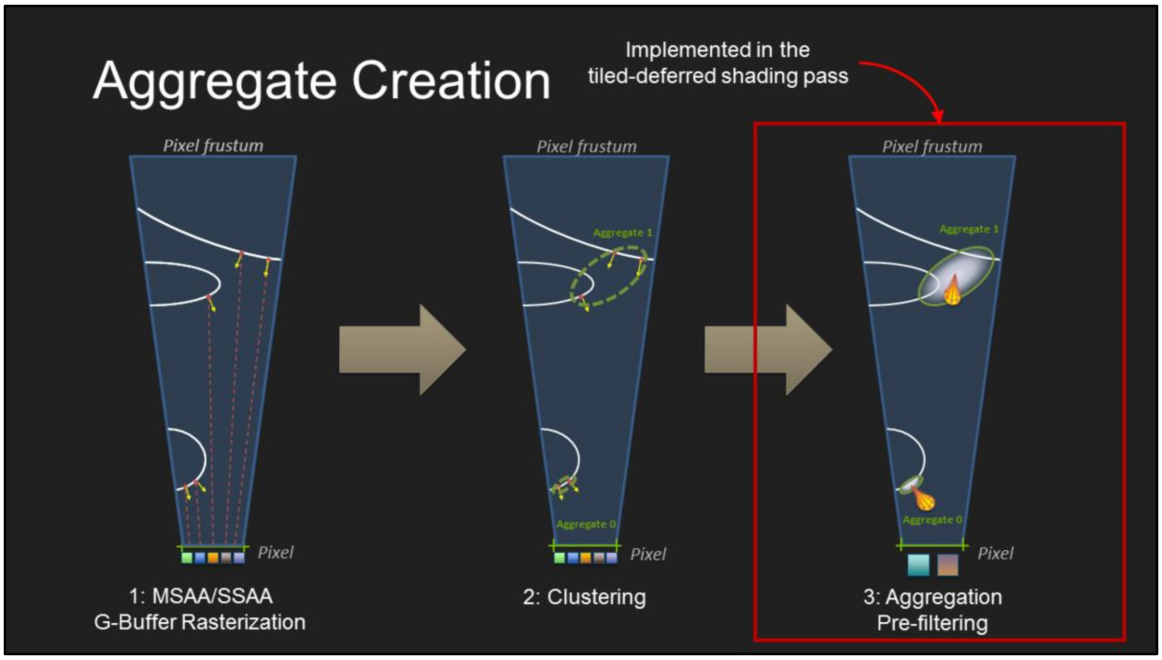
The first important thing is to separate samples with different shading models, given that pre-filtered shading schemes assume a unique shading model per-aggregate.
Then, we built a simple distance metric which maximizes spatial locality while avoiding aggregating together very divergent normal directions.



Aggregate Creation: Aggregation

# Once we have this sample to aggregate mapping, the remaining step is to actually performs the filtering of the shading attributes per-aggregate,
# using the data from the super-sampled G-buffer.

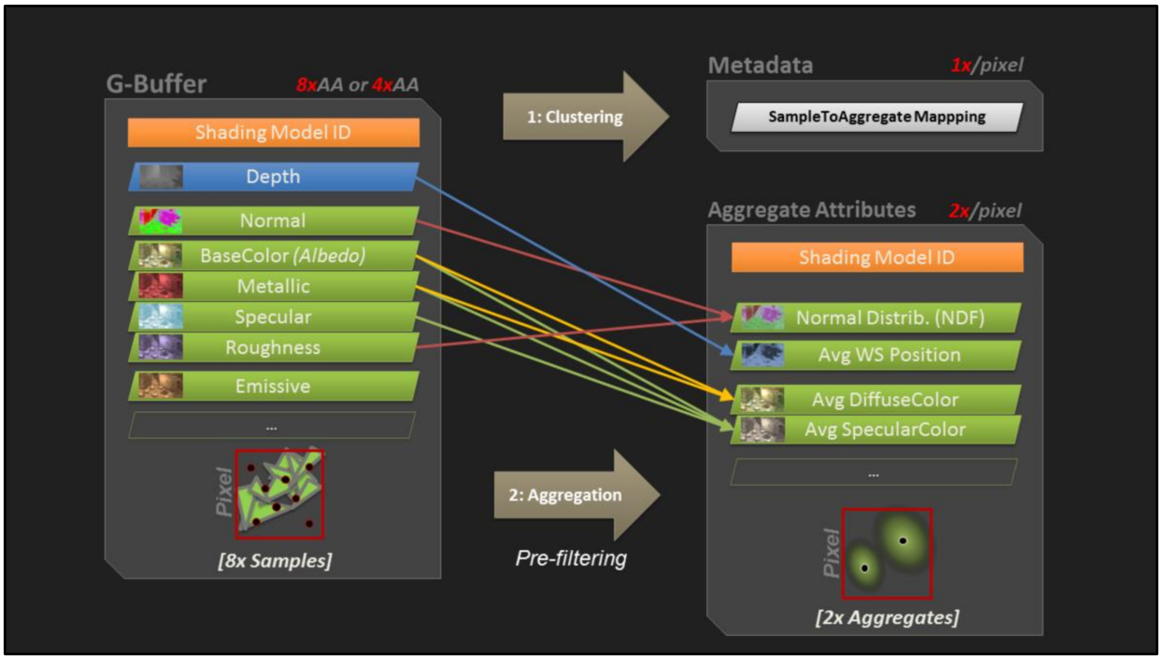
To recap, we average most of the shading attributes, and we transform the per-sample normal and roughness information into the Normal Distribution representation associated to our BRDF model, which allows us to linearly combine them, and approximate their variance over an aggregate's footprint.



Aggregate Creation

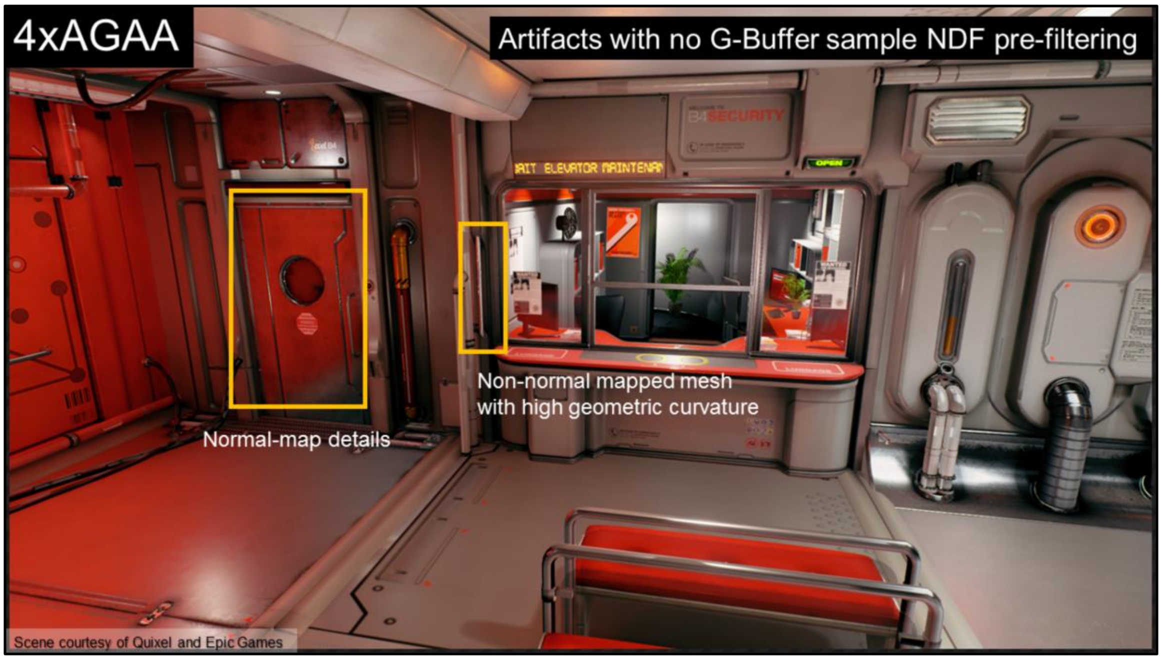
In UE4, there are multiple possible render paths to perform shading. One of them is Tiled Deferred shading.
We have configured the engine to use the Tiled Deferred path whenever possible and implemented the AGAA aggregation and lighting in the Tiled Deferred Shading compute shader.

For getting best performance, we have implemented the G-Buffer aggregation step at the beginning of the shading phase of this compute shader, which lets us re-use the aggregated attributes for all processed lights for the current tile.



The clustering pass fetches all of the ShadingModel ID, depths and normals for each pixel and computes the mapping from sample ID to aggregate ID. Because there are only 2 aggregates per pixel, the per-sample aggregate IDs can be stored with one bit per sample. We store these bits in a R16_UINT render target.
We also store the per-aggregate ShadingModel IDs in the same render target.

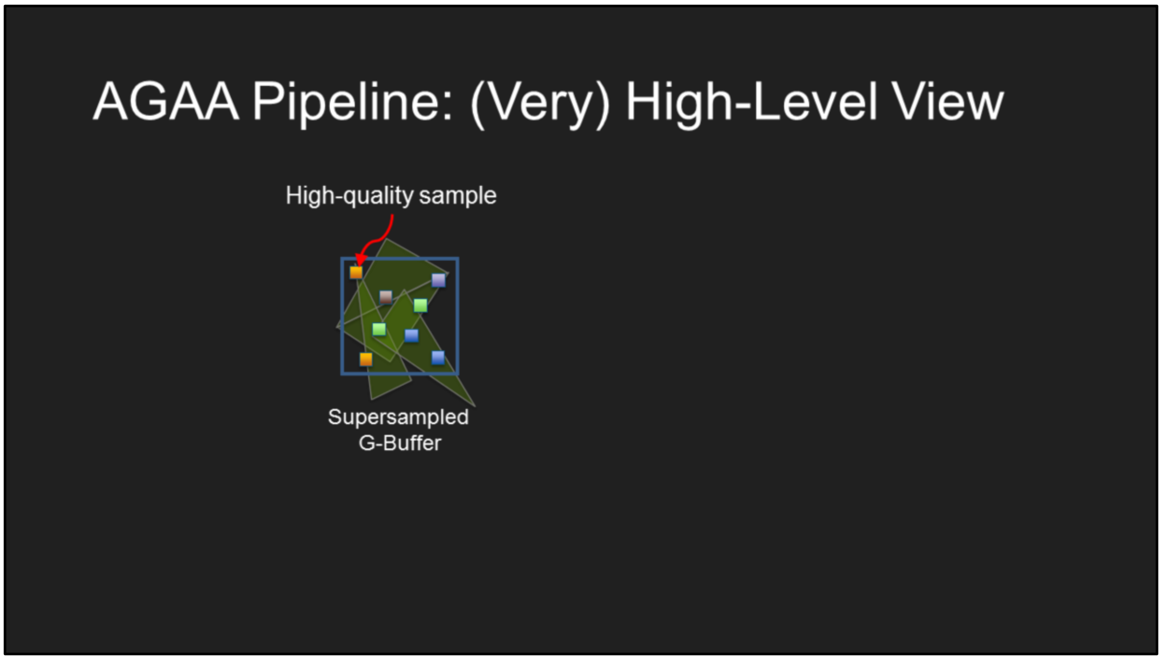
Next for each GBuffer sample, we reconstruct the shading parameters (in the format used for aggregate shading) based on the GBuffer data.
And we average these shading parameters across all samples within each aggregate.
There is one exception though: for the world-space positions, we have found that it is much faster to first calculate the average view-space depth per aggregate, and then reconstruct a world-space position by assuming that the screen-space XY coordinate for this position is at the pixel center.
This approximation does not introduce any visible artifacts in our tests.


4xAGAA
Artifacts with no G-Buffer sample NDF pre-filtering
Normal-map details
Non-normal mapped mesh with high geometric curvature
Scene courtesy of Quixel and Epic Games

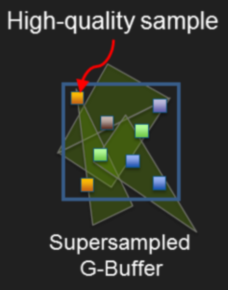
After doing all that, we were still seeing aliasing & flickering on specular highlights on certain objects:

- The door on this screenshot is a flat surface, and there was aliasing on the bumps from the normal map
- The vertical chrome pipe also had aliasing artifacts

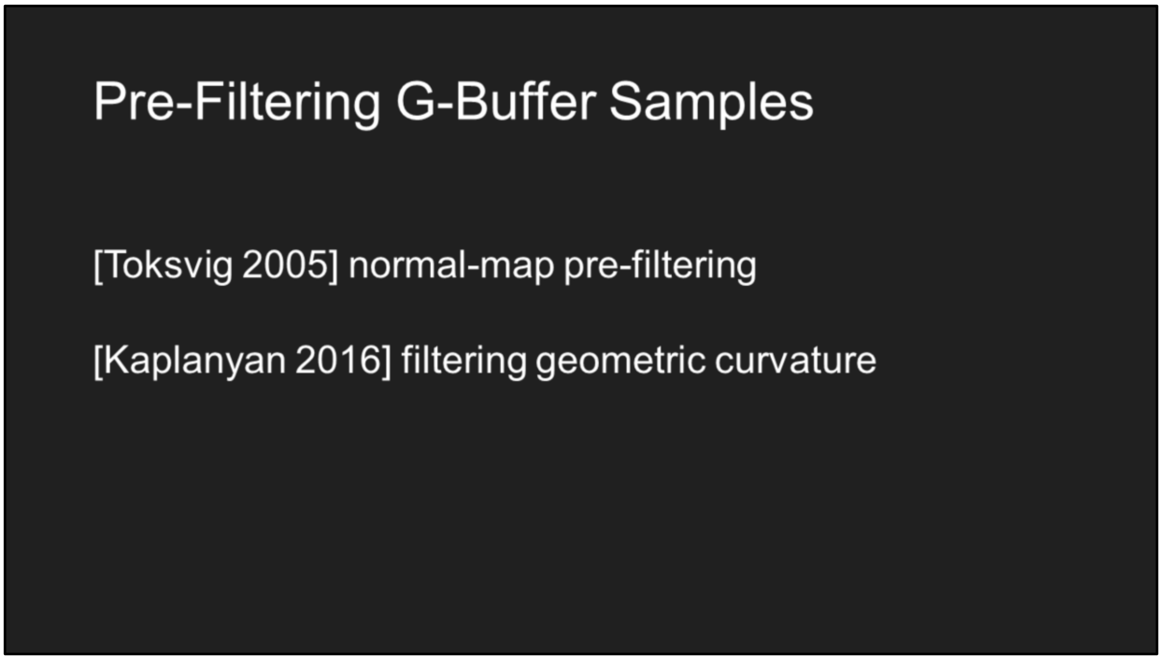
At this point, we realized that we were just using a super-sampled GBuffer but not doing any NDF pre-filtering at the GBuffer sample level.



AGAA Pipeline: (Very) High-Level View



# In addition to pre-filtering the shading attributes at the scale of the aggregates, AGAA also needs sample values to be filtered as best as possible



Pre-Filtering G-Buffer Samples

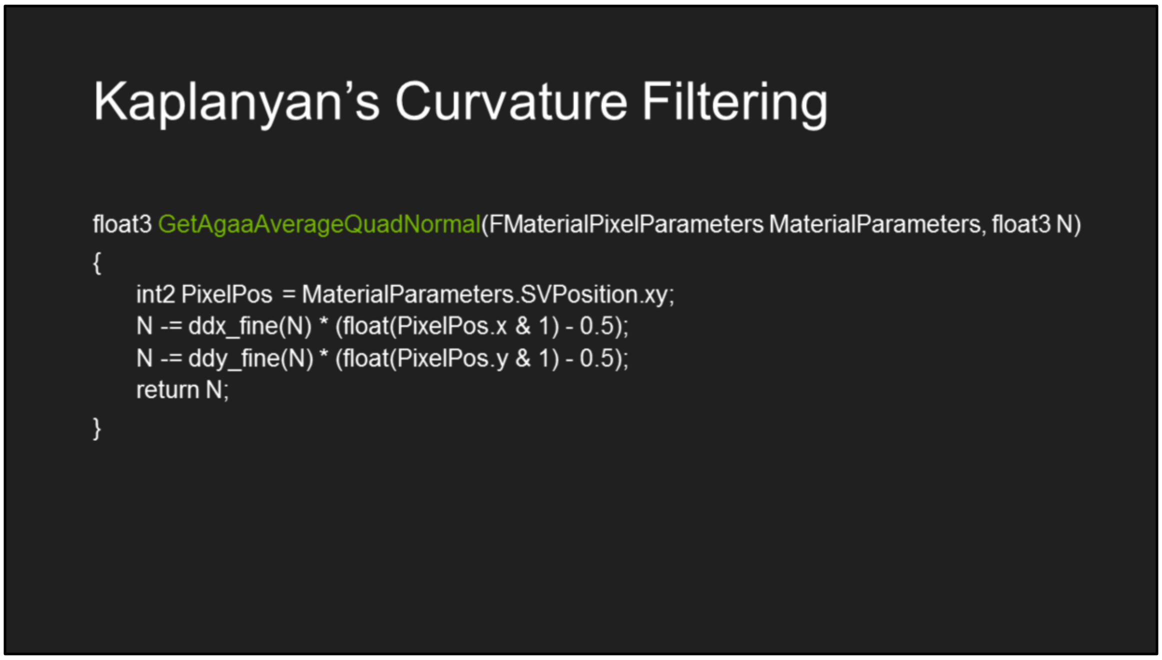
[Toksvig 2005] normal-map pre-filtering

[Kaplanyan 2016] filtering geometric curvature

When fetching the normal of a given GBuffer sample, we want to be able to not only fetch a normal vector, but also reconstruct some information on its variance.

We use Toksvig normal-map pre-filtering, assuming that the material fetches a single normal map and plugs it directly into the GBuffer.
We will see later that material graphs can be arbitrarily complex.

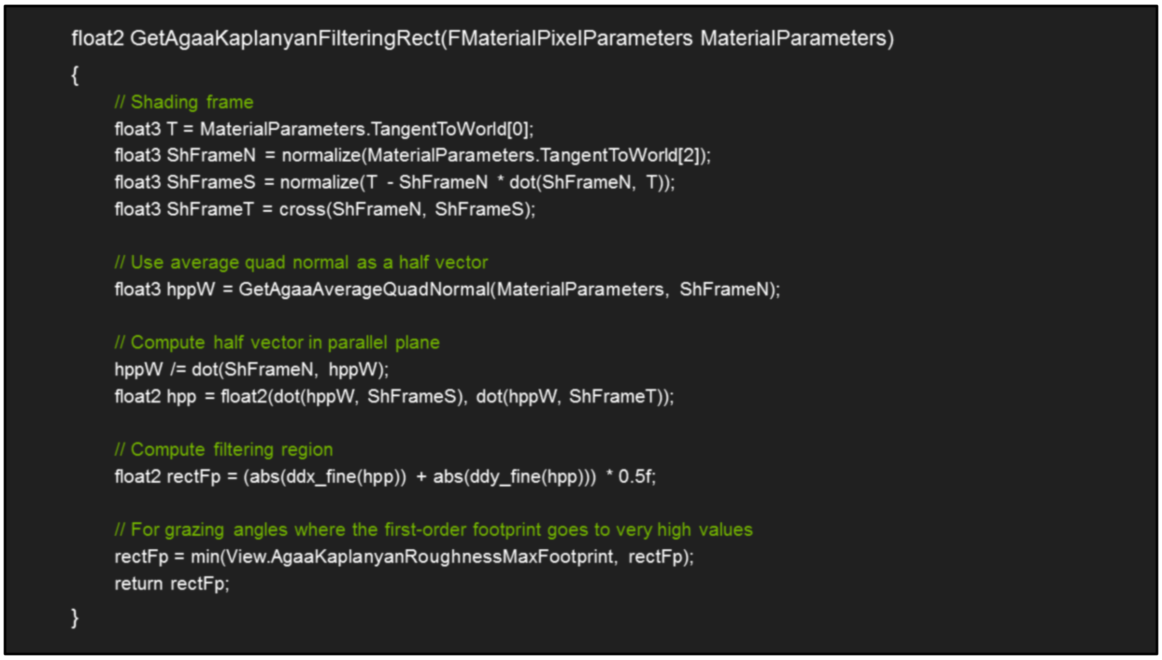
In addition, we also filter the geometric curvature, coming from the actual triangle mesh, using Kaplanyan's analytic algorithm which operates during geometry rendering.



Kaplanyan's Curvature Filtering

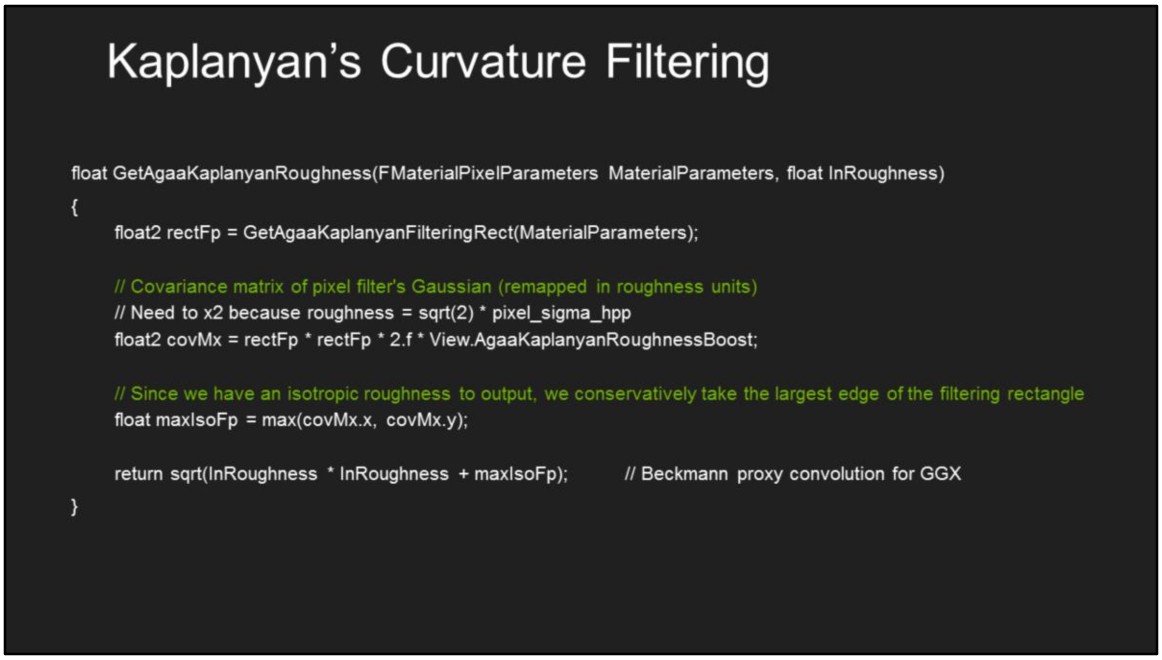
```
float3 GetAgaaAverageQuadNormal(FMaterialPixelParameters MaterialParameters, float3 N)
{
    int2 PixelPos  = MaterialParameters.SVPosition.xy;
    N -= ddx_fine(N) * (float(PixelPos.x & 1) - 0.5);
    N -= ddy_fine(N) * (float(PixelPos.y & 1) - 0.5);
    return N;
}
```

The first step of the algorithm calculates the average normal for the current 2x2 pixel quad in the Gbuffer Fill pass.
This is done using ddx/ddy instructions on the geometric normal (with no normal maps applied).



```
float2 GetAgaaKaplanyanFilteringRect(FMaterialPixelParameters MaterialParameters)
{
    // Shading frame
    float3 T = MaterialParameters.TangentToWorld[0];
    float3 ShFrameN = normalize(MaterialParameters.TangentToWorld[2]);
    float3 ShFrameS = normalize(T - ShFrameN * dot(ShFrameN, T));
    float3 ShFrameT = cross(ShFrameN, ShFrameS);

    // Use average quad normal as a half vector
    float3 hppW = GetAgaaAverageQuadNormal(MaterialParameters, ShFrameN);

    // Compute half vector in parallel plane
    hppW /= dot(ShFrameN, hppW);
    float2 hpp = float2(dot(hppW, ShFrameS), dot(hppW, ShFrameT));

    // Compute filtering region
    float2 rectFp = (abs(ddx_fine(hpp)) + abs(ddy_fine(hpp))) * 0.5f;

    // For grazing angles where the first-order footprint goes to very high values
    rectFp = min(View.AgaaKaplanyanRoughnessMaxFootprint, rectFp);
    return rectFp;
}
```

This **average pixel-quad normal** is then used together with the **normal of the current fragment** to estimate **curvature variance** information.



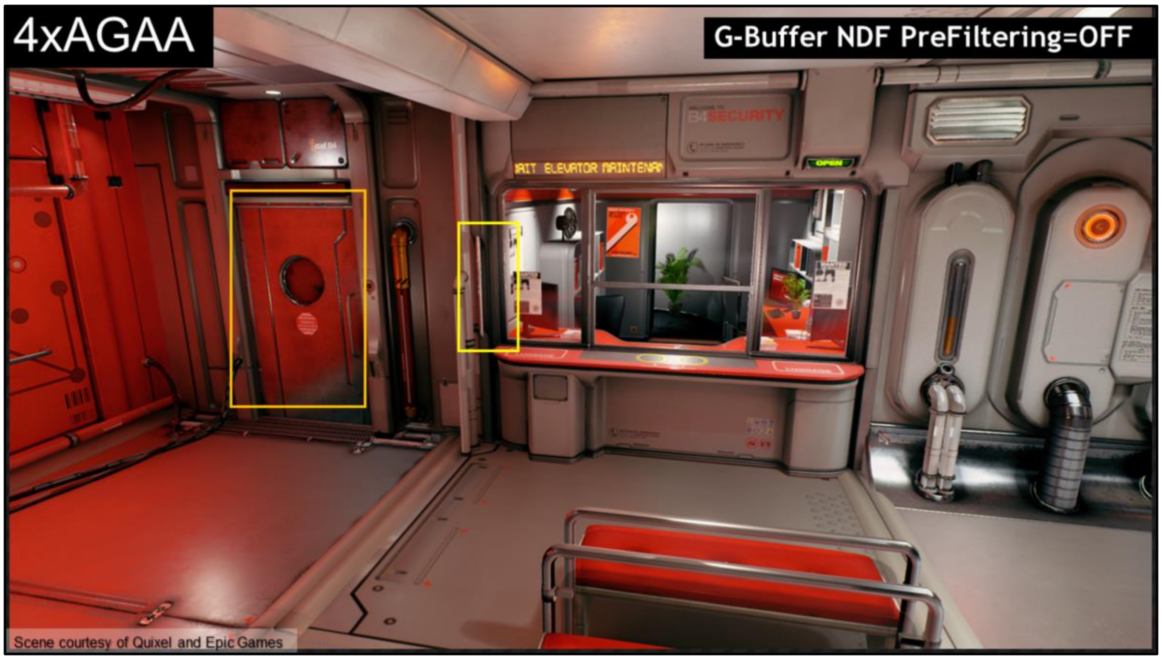
Kaplanyan's Curvature Filtering

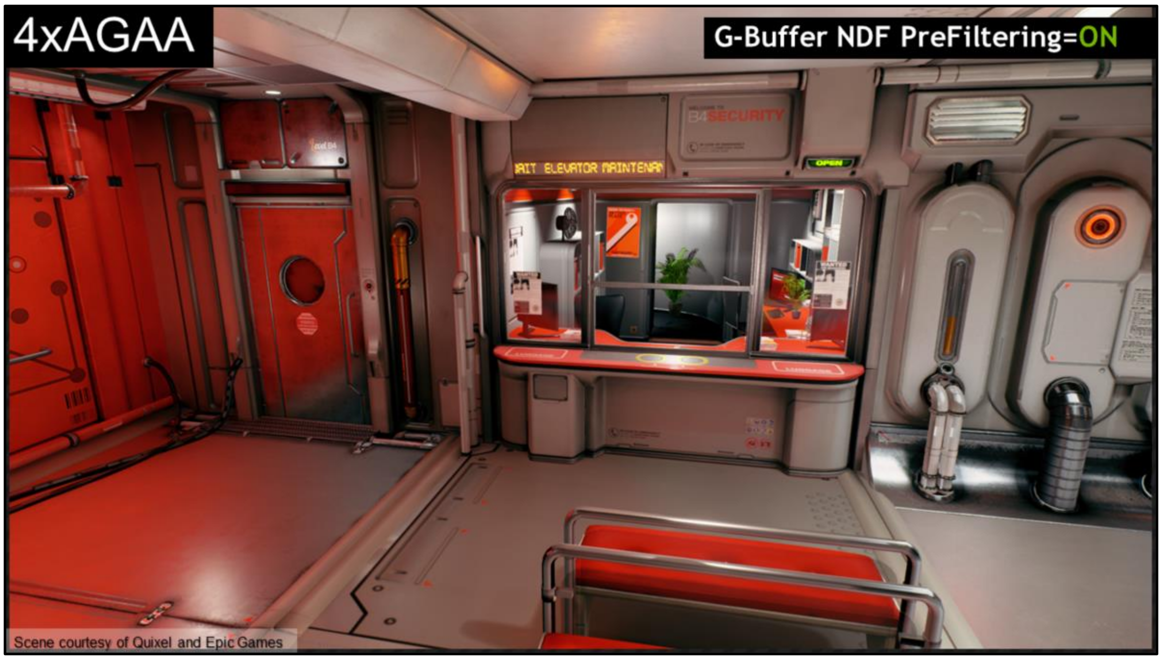
```
float GetAgaaKaplanyanRoughness(FMaterialPixelParameters MaterialParameters, float InRoughness)
{
	float2 rectFp = GetAgaaKaplanyanFilteringRect(MaterialParameters);

	// Covariance matrix of pixel filter's Gaussian (remapped in roughness units)
	// Need to x2 because roughness = sqrt(2) * pixel_sigma_hpp
	float2 covMx = rectFp * rectFp * 2.f * View.AgaaKaplanyanRoughnessBoost;

	// Since we have an isotropic roughness to output, we conservatively take the largest edge of the filtering rectangle
	float maxIsoFp = max(covMx.x, covMx.y);

	return sqrt(InRoughness * InRoughness + maxIsoFp);	// Beckmann proxy convolution for GGX
}
```

Finally, the curvature variance information is converted to an isotropic roughness bias and the original roughness is modified.

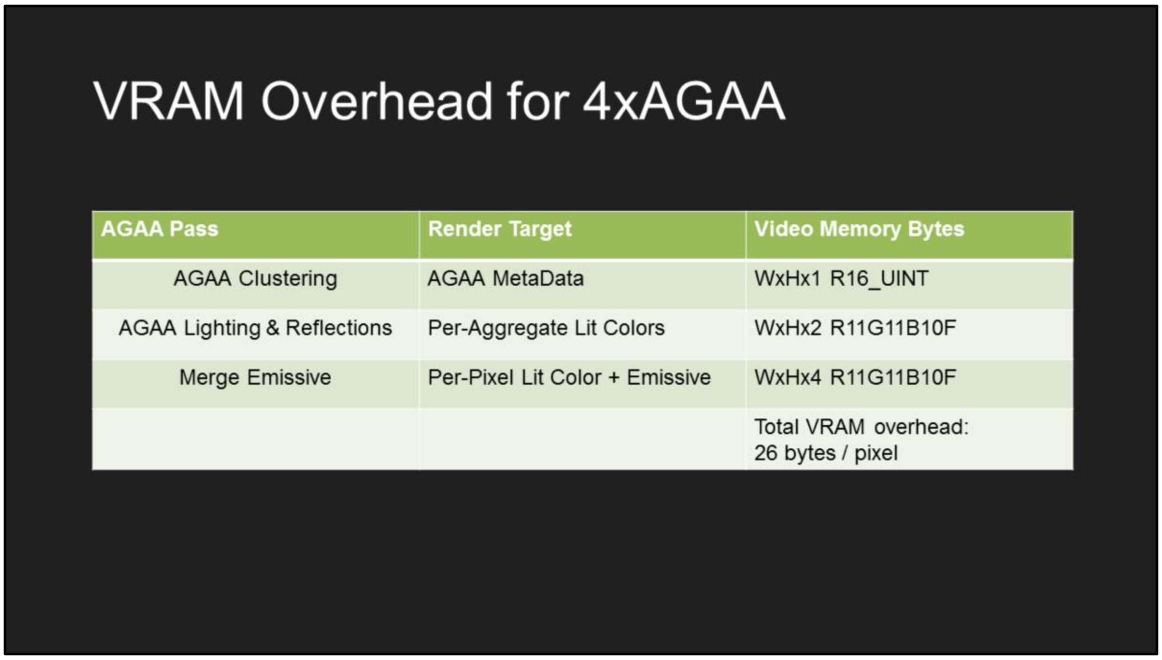
So regions that have a high curvature get an increased roughness, which removes specular shading aliasing.

For the record, we use:
AgaaKaplanyanRoughnessMaxFootprint=0.3
AgaaKaplanyanRoughnessBoost=2.0


4xAGAA
G-Buffer NDF PreFiltering=OFF
AIT ELEVATOR MAINTENAN
SECURITY
OPEN
Scene courtesy of Quixel and Epic Games


4xAGAA
G-Buffer NDF PreFiltering=ON
Scene courtesy of Quixel and Epic Games

By adding Toksvig's and Kaplanyan's pre-filtering methods to the screen-space pre-filtering of AGAA, we were able to get rid of the specular shading aliasing artifacts in this scene.



VRAM Overhead for 4xAGAA

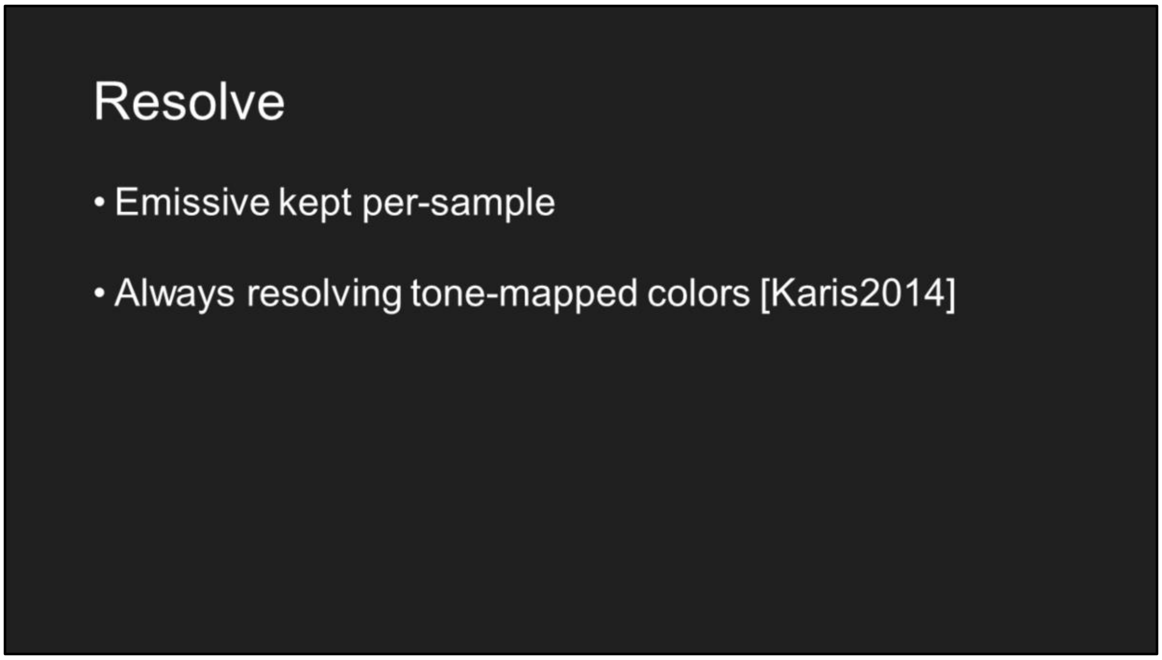
AGAA Pass	Render Target	Video Memory Bytes
AGAA Clustering	AGAA MetaData	WxHx1 R16_UINT
AGAA Lighting & Reflections	Per-Aggregate Lit Colors	WxHx2 R11G11B10F
Merge Emissive	Per-Pixel Lit Color + Emissive	WxHx4 R11G11B10F
		Total VRAM overhead: 26 bytes / pixel

Let’s recap the passes of our AGAA implementation and their impact on video memory usage.

During lighting we light the 2 aggregates in the same pass, which produces 2 R11G11B10F colors per pixel.

We then merge the per-sample emissive colors with the per-aggregate non-emissive lit colors, and write the per-sample results in a super-sampled color buffer.

Translucency passes & post processing passes are then performed with super-sampling (could also be MSAA).

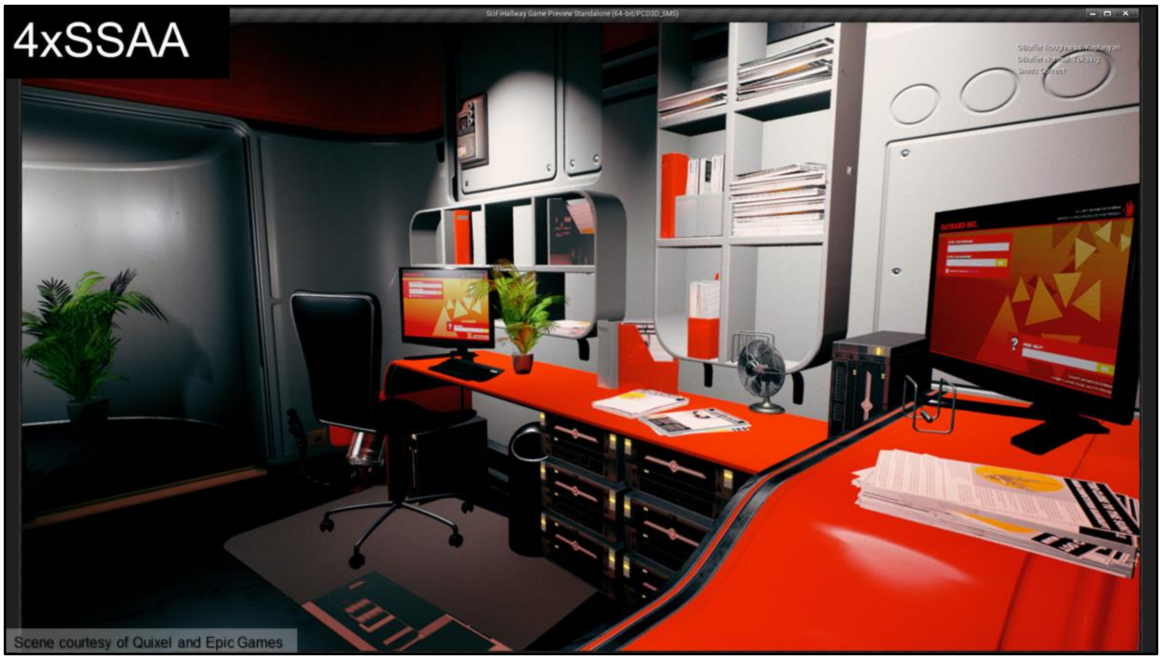

Resolve

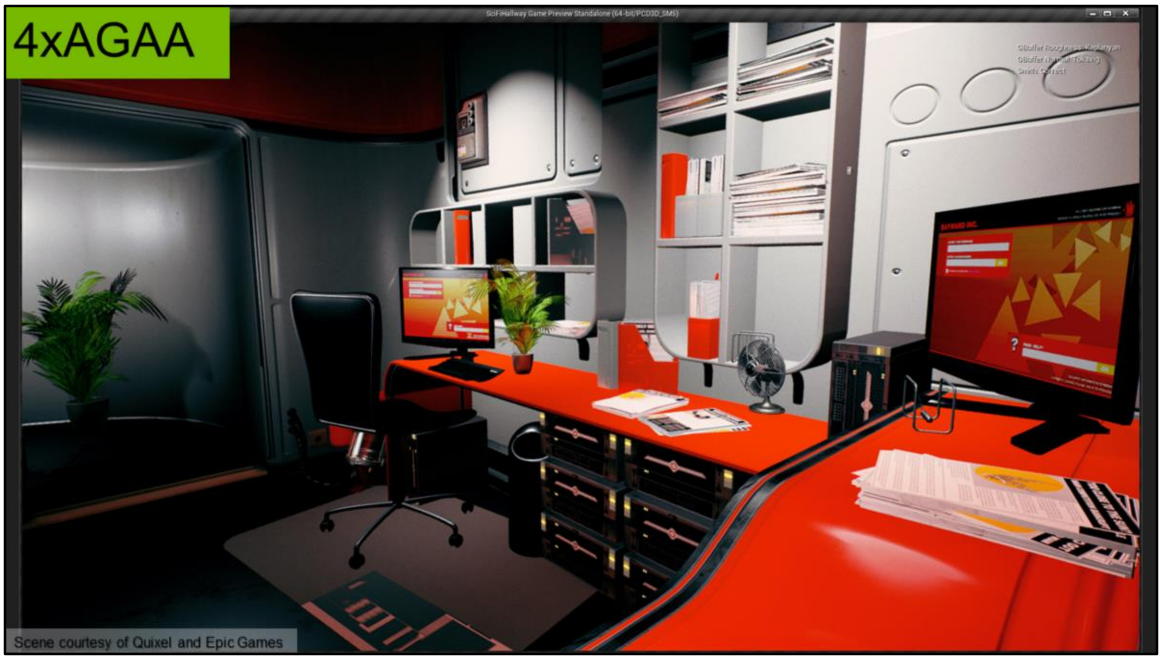
- Emissive kept per-sample
- Always resolving tone-mapped colors [Karis2014]

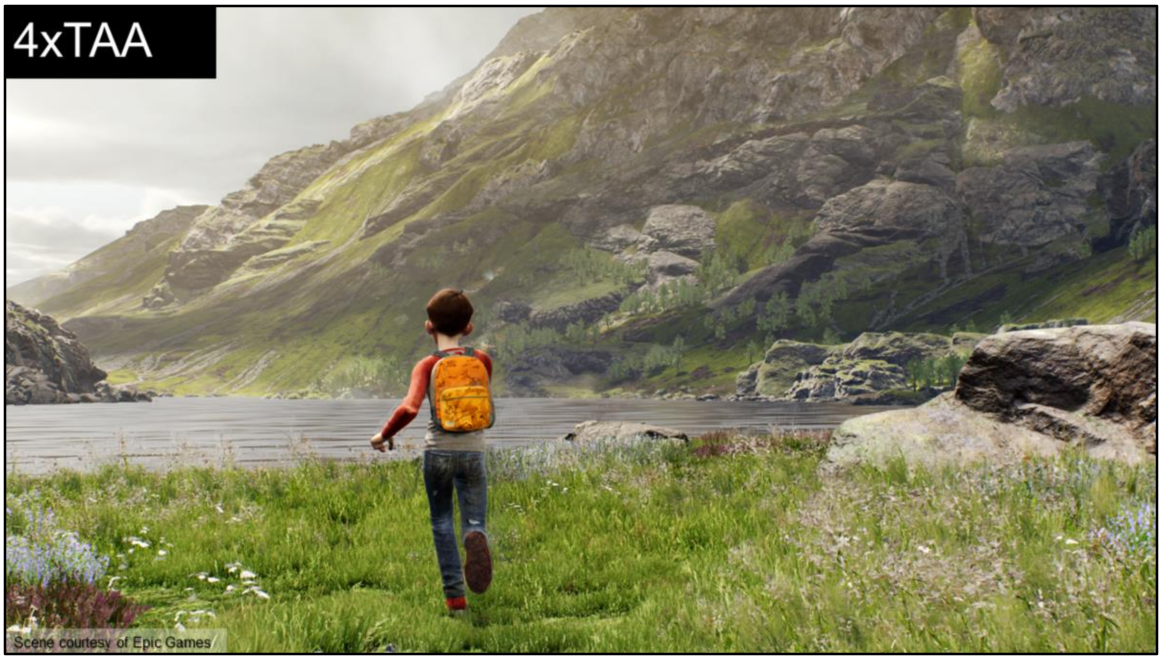
To avoid HDR firefly artifacts, it is important to:
- keep the HDR emissive colors per-sample (do not include the emissive in the per-aggregate lit colors)
- perform the final Resolve (average of the per-sample colors) on tone-mapped colors.

Results

Image quality and performance


4xSSAA
Scene courtesy of Quixel and Epic Games


4xAGAA
Scene courtesy of Quixel and Epic Games

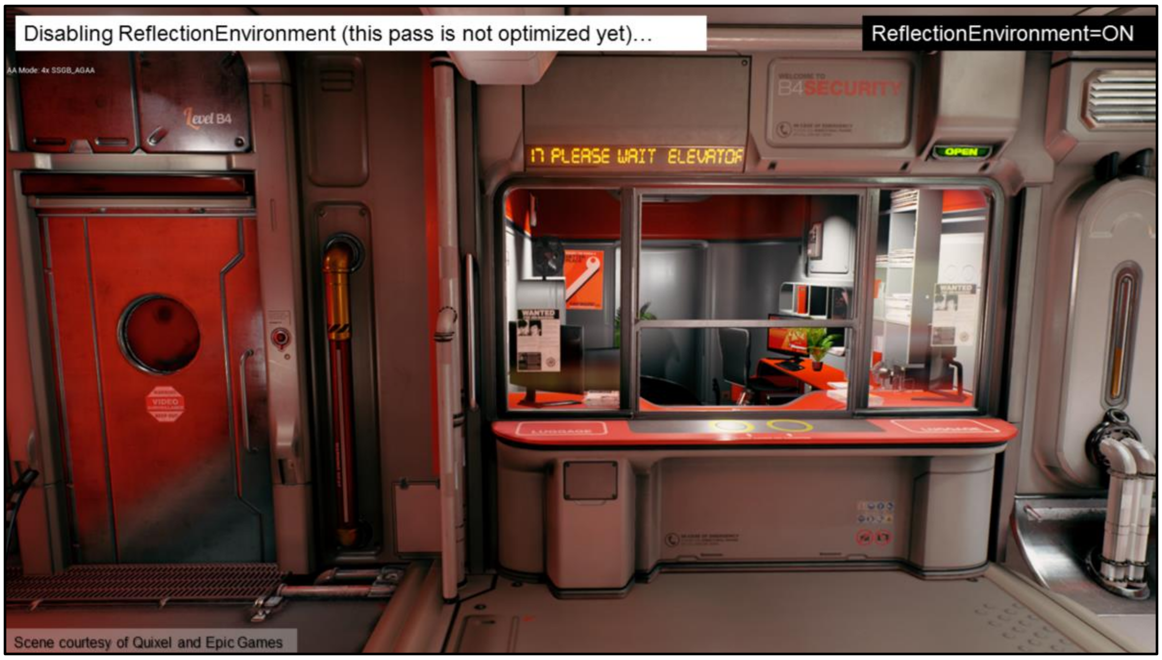

4xTAA
Scene courtesy of Epic Games

4xAGAA
Scene courtesy of Epic Games

4xSSAA
Scene courtesy of Epic Games

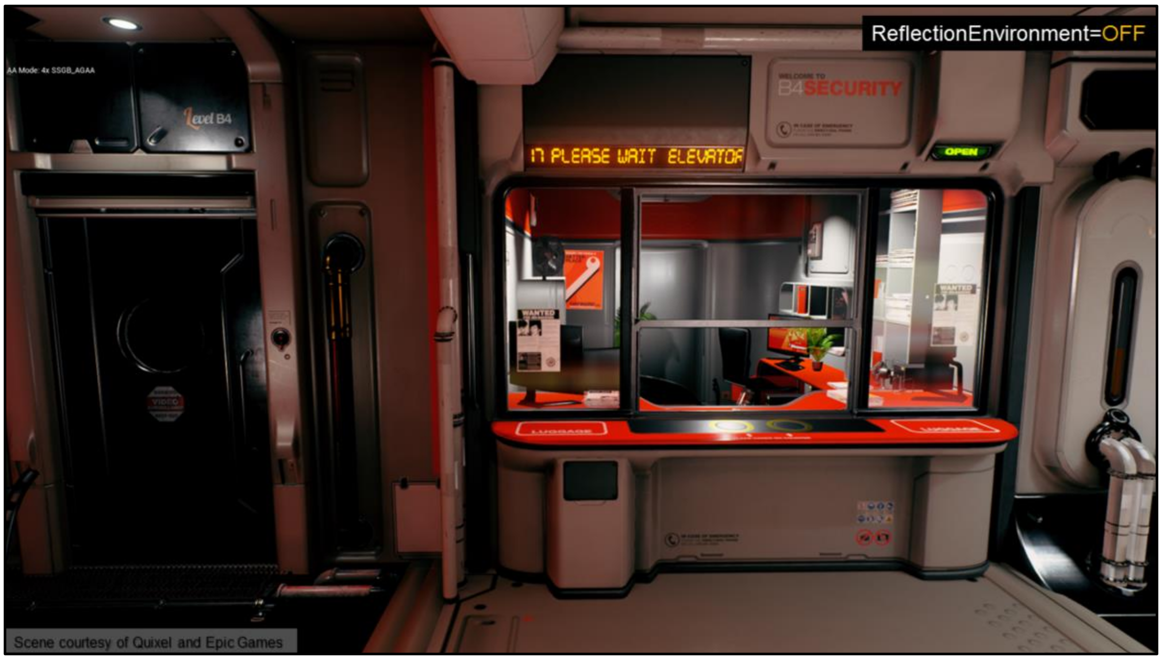
8xSSAA
OPEN
WANTED
Scene courtesy of Quixel and Epic Games

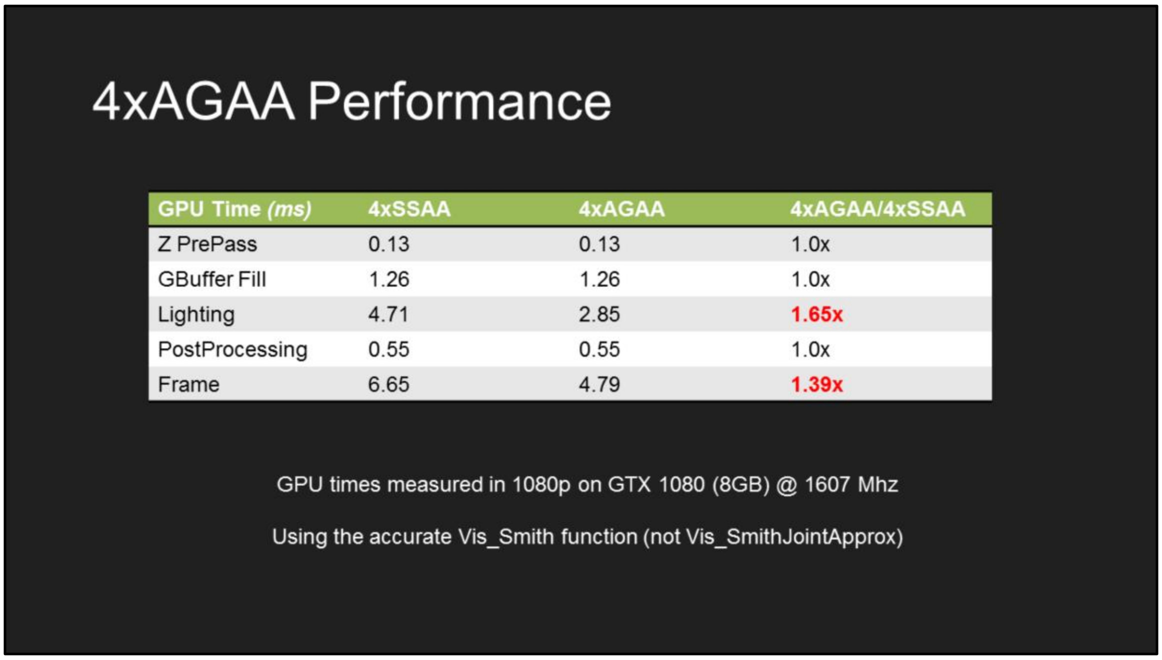
8xAGAA
Scene courtesy of Quixel and Epic Games
OPEN
WANTED



Disabling ReflectionEnvironment (this pass is not optimized yet)…

ReflectionEnvironment=ON


ReflectionEnvironment=OFF



4xAGAA Performance

GPU Time *(ms)*	4xSSAA	4xAGAA	4xAGAA/4xSSAA
Z PrePass	0.13	0.13	1.0x
GBuffer Fill	1.26	1.26	1.0x
Lighting	4.71	2.85	**1.65x**
PostProcessing	0.55	0.55	1.0x
Frame	6.65	4.79	**1.39x**

GPU times measured in 1080p on GTX 1080 (8GB) @ 1607 Mhz

Using the accurate Vis_Smith function (not Vis_SmithJointApprox)

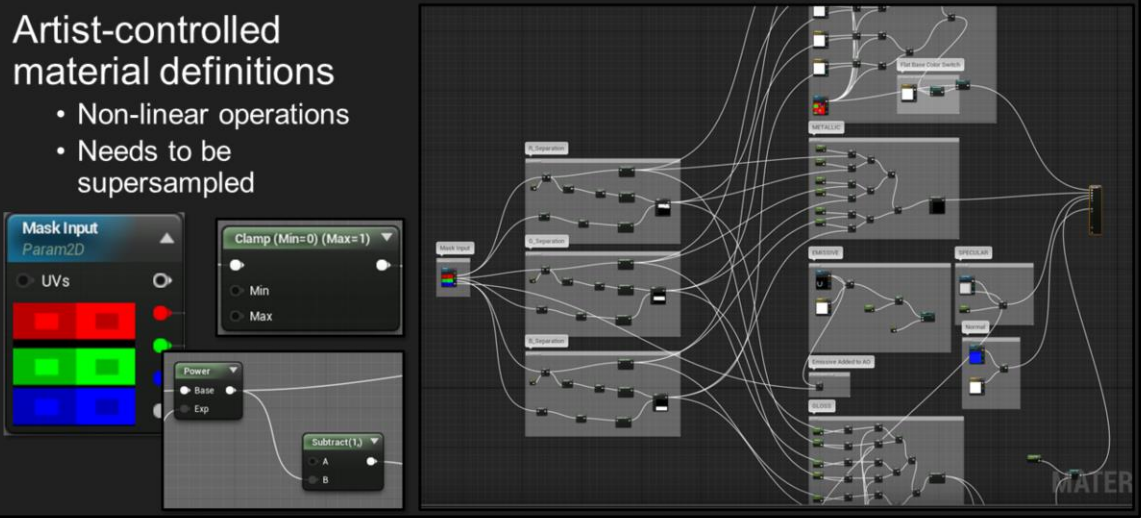
MSAA

The cost of better performance

Why not MSAA ? Complex material graphs

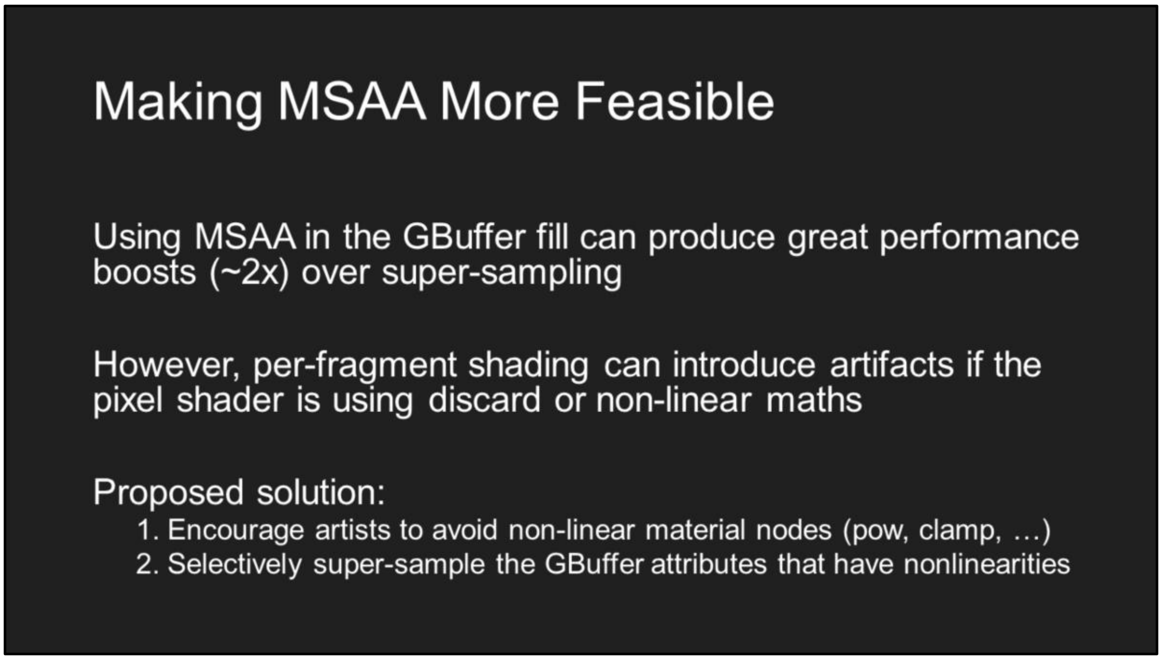
Artist-controlled material definitions

- Non-linear operations
- Needs to be supersampled



The reason is that the shading function needs to be super-sampled.
Even though UE4 mostly relies on texture maps for storing material's input parameters, which can be pre-filtered,
artists control the material definitions using potentially complex material graphs like this one.

Those graphs can contain lots of non-linear operations, such as discard calls, power functions, clamps, masks, conditions, and various combinations of those, which breaks texture-space pre-filtering.


Making MSAA More Feasible

Using MSAA in the GBuffer fill can produce great performance boosts (~2x) over super-sampling

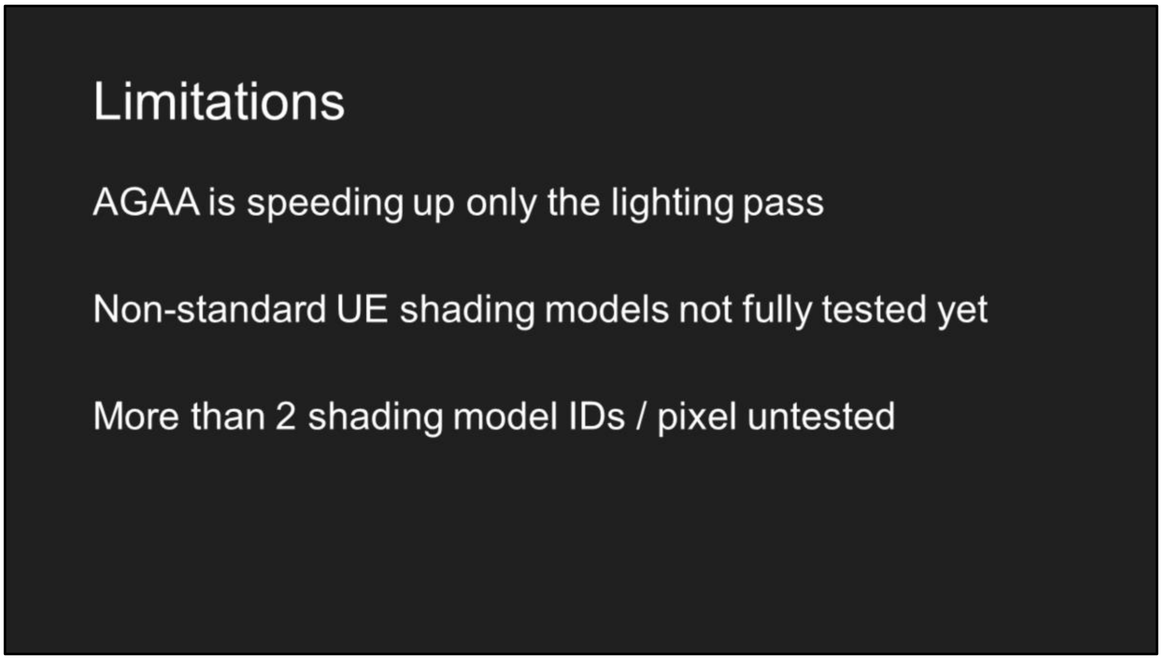
However, per-fragment shading can introduce artifacts if the pixel shader is using discard or non-linear maths

Proposed solution:

1. Encourage artists to avoid non-linear material nodes (pow, clamp, …)
2. Selectively super-sample the GBuffer attributes that have nonlinearities

A solution would be to selectively super-sampler certain GBuffer attributes automatically based on a static analysis of the material graph.

We think that MSAA could still be used if artists were made aware that using masked materials and/or non-linear material nodes can slow down performance due to the incurred super-sampling at runtime.



Limitations

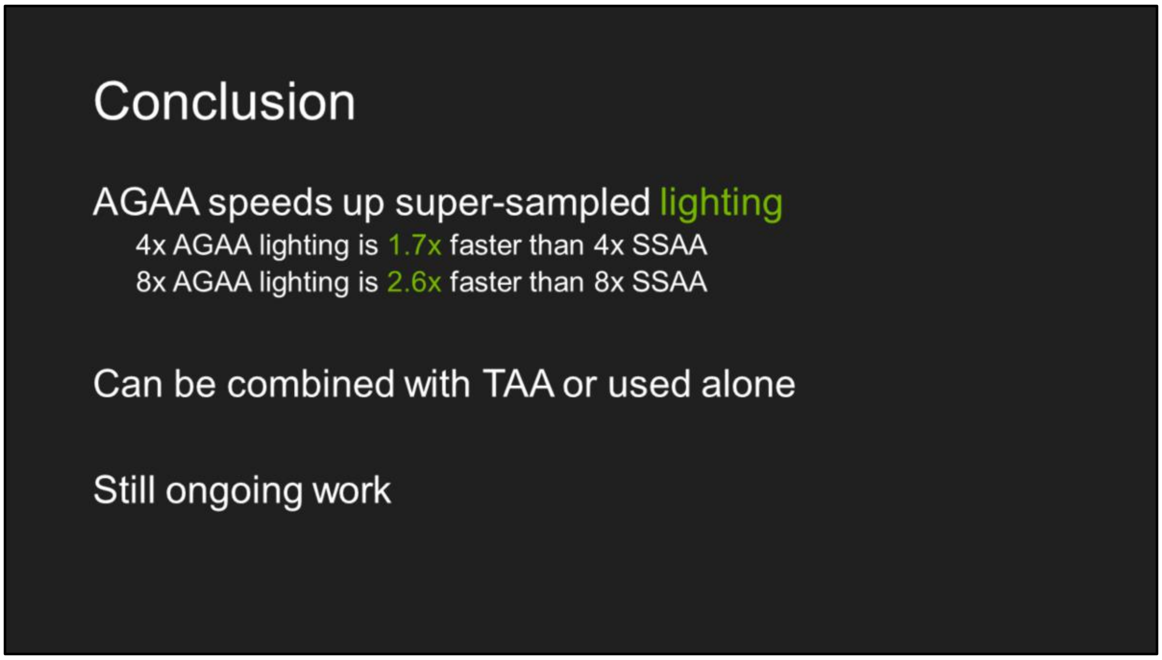
AGAA is speeding up only the lighting pass

Non-standard UE shading models not fully tested yet

More than 2 shading model IDs / pixel untested

Although the original paper proposed a solution to speedup the Gbuffer Fill using Maxwell features, it raises more issues in the context of UE4 and we decided not to do it.

Tiled-deferred shading is the only light pass that is optimized so far in our UE4 implementation.



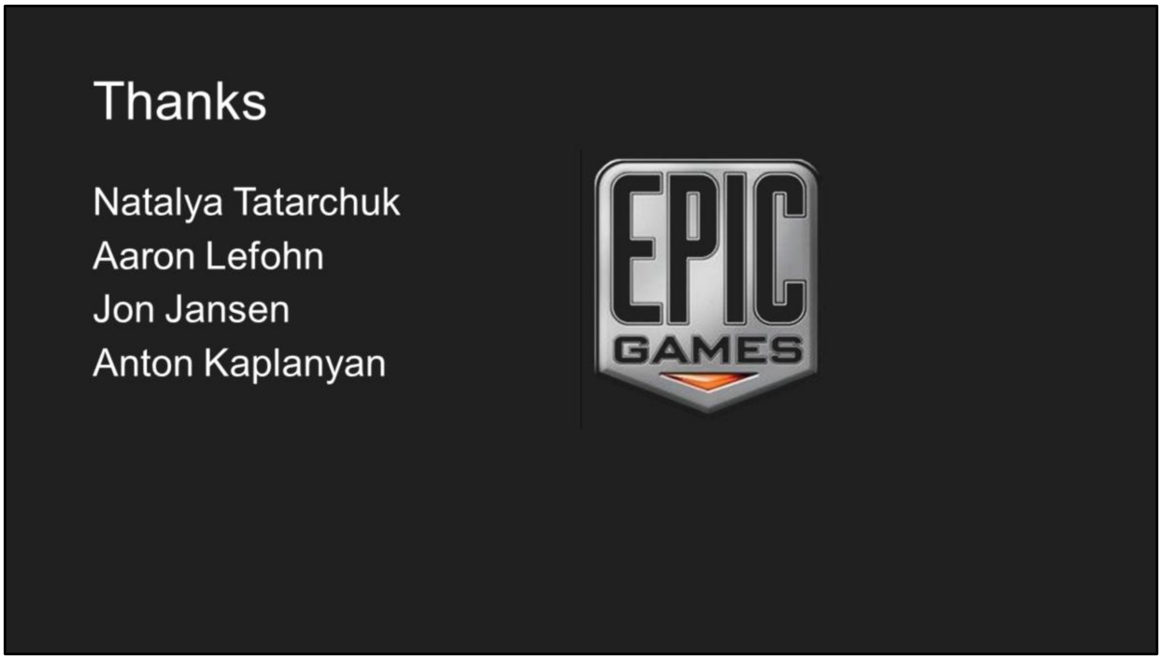
Conclusion

AGAA speeds up super-sampled lighting

- 4x AGAA lighting is 1.7x faster than 4x SSAA
- 8x AGAA lighting is 2.6x faster than 8x SSAA

Can be combined with TAA or used alone

Still ongoing work

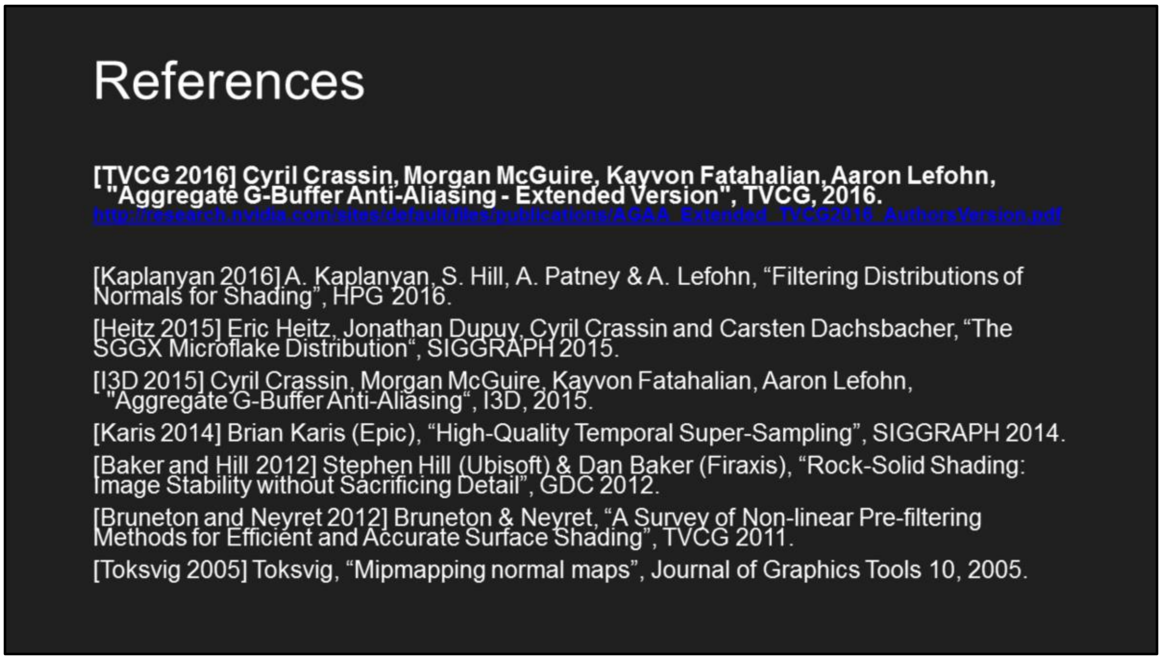


Thanks

Natalya Tatarchuk
Aaron Lefohn
Jon Jansen
Anton Kaplanyan

EPIC GAMES

Questions?



References

[TVCG 2016] Cyril Crassin, Morgan McGuire, Kayvon Fatahalian, Aaron Lefohn, "Aggregate G-Buffer Anti-Aliasing - Extended Version", TVCG, 2016.
http://research.nvidia.com/sites/default/files/publications/AGAA_Extended_TVCG2016_AuthorsVersion.pdf

[Kaplanyan 2016] A. Kaplanyan, S. Hill, A. Patney & A. Lefohn, "Filtering Distributions of Normals for Shading", HPG 2016.

[Heitz 2015] Eric Heitz, Jonathan Dupuy, Cyril Crassin and Carsten Dachsbacher, "The SGGX Microflake Distribution", SIGGRAPH 2015.

[I3D 2015] Cyril Crassin, Morgan McGuire, Kayvon Fatahalian, Aaron Lefohn, "Aggregate G-Buffer Anti-Aliasing", I3D, 2015.

[Karis 2014] Brian Karis (Epic), "High-Quality Temporal Super-Sampling", SIGGRAPH 2014.

[Baker and Hill 2012] Stephen Hill (Ubisoft) & Dan Baker (Firaxis), "Rock-Solid Shading: Image Stability without Sacrificing Detail", GDC 2012.

[Bruneton and Neyret 2012] Bruneton & Neyret, "A Survey of Non-linear Pre-filtering Methods for Efficient and Accurate Surface Shading", TVCG 2011.

[Toksvig 2005] Toksvig, "Mipmapping normal maps", Journal of Graphics Tools 10, 2005.