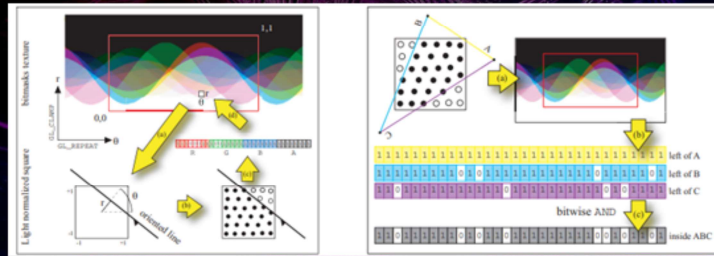


Coverage Bitmaps

for efficient Rendering algorithms



Electronic Arts

Martin Mittring
MMittring@EA.com
Nov 2023



SEED

Coverage Bitmaps

for efficient Rendering Algorithms

Thanks to Prof. Dr. Elmar Eisemann giving permission to use the image here. It's this image I have in mind when I talk about the topic.

Coverage Bitmasks

aka Visibility, Occupancy, Occlusion or Blocker [bit] masks

is a “tool to process binary decisions in bulk”

Intro

5 slides

Applications

7 slides

Deconstruct

4 slides

References

3 slides

Bonus

6 slides

This is a presentation where I want to show you something that I found in so many researchers work on rendering algorithms. It's nothing new for some but many don't even know they could benefit from it or how to use it.

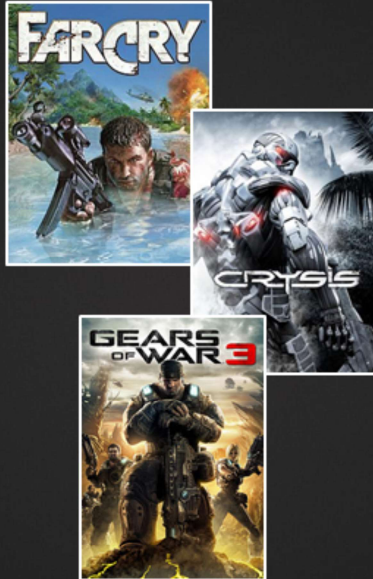
So to show you, I have to reference that existing work. The “Coverage Bitmask” is so versatile and have been used for so many application, in very creative ways. That is mostly on the purple slides. Often they used different names, like Visibility, Occupancy, Occlusion or Blocker masks. I stick to “Coverage” as a generalization.

Please check out those publications as I only can point out a few highlights each time. There are also more references with web links you can follow to deepen the topic.

Martin Mittring



Principal SW Engineer
at SEED Electronic Arts



Experience



Scripting Language, CSG,
Sparse Voxel Cone Tracing,
Particles, Post Processing,
Bloom, Shadows, Indirect
Lighting, Lightmaps, Normal
maps, Volumetric Fog, UGC,
Rasterization, Ray tracing, UI,
VR, ...

I am Martin Mittring and since January I work for Electronic Arts in the SEED team - more about that later.

I am a graphics engineer by heart and have many years experience working for multiple game companies like Crytek and Epic Games.

Most of my work was on the engine side implementing real-time rendering algorithms. One day I came across this curious image you've seen in the title slide..

I understood how clever and simple this was and I went through the full "Gartner Hype" cycle. After spending quite some time with this tool,

seeing it in a lot of other people's work but also using it myself, I can tell you all about it.



SEED
Search for Extraordinary Experiences Division

R&D team at Electronic Arts

- o Since ~2015
- o Team of a ~40
Stockholm, Montréal, Los Angeles,
Redwood Shores, London, Vancouver

Applied Research

- o Cross-disciplinary projects 6M - 5Y+

Collaborate

- o Games DICE, Madden, Respawn, ...
- o Central Groups Frostbite, ...
- o External Hardware, Software, Standards, Academia

Share

- o Present & Publish 50+ publications, Open Source



seed.ea.com
@SEED

In case you don't know what SEED is, we're a technical and creative research division of Electronic Arts Studios. Established in 2015 our team of around 40 is distributed in 6 locations across the world including Stockholm, Montreal, Los Angeles, Redwood Shores, London and Vancouver

You can find more info about us at seed.ea.com, or follow us on twitter

SEED exists at EA as a cross-disciplinary team where we combine art, engineering, creativity, and research to deliver disruptive innovation, for our games and our players.

We run fast towards the future and run in parallel to the current business constraints,
For the benefit of our games and all EA studios

By focusing on short, medium, and long term research portfolio gives us an opportunity to do research and always be delivering technology artifacts along the way

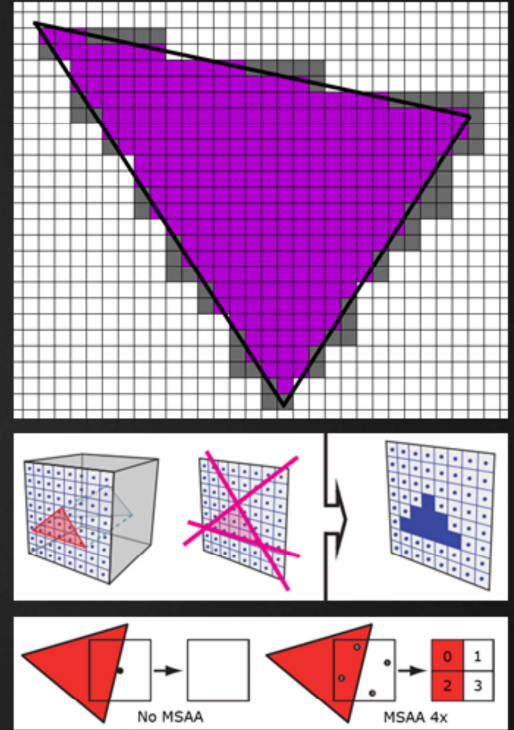
We collaborate with game teams, central groups, as well as many external partners in hardware, software, industry standards bodies, and academia.

Of course as an R&D group we have to present and publish. Over the years we've accumulated more than 50 publications and presentations

We also open source some of our work, so find us on github.

Recap, needed ?

- Classic triangle rasterization
 - Find pixels inside a triangle
Classify pixels relative to half spaces defined by edges
 - No holes or overdraw with neighbors
 - Multiple sample positions in a pixel for AA
- Fixed function unit has stages
 - Larger tile e.g. 1MB
 - Conservative coarse raster e.g. 8x8
 - 2x2 pixel block with helper lane mask
 - MSAA mask 2 / 4 / 8 bits per pixel
- Parallel hardware is fast
=> Software can do it too



Please show of hands: Who knows how triangle rasterization works?
Who wrote a rasterizer?
Do you want to know more about it ?

Triangle Rasterization is the process of turning 3 points in 3d space to colors in a 2D grid.
There are many methods but the fast (because parallelization friendly) method is based on using edge equations and generating blocks of work.

We need to test if a pixel center is inside a triangle. A point on an edge is ambiguous and without defining this right it can cause holes or double rendering. We can define some extra rules to ensure such cases are deterministic. If you want to use MSAA for anti aliasing the test is not made for the pixel center but for each sub sample in a pixel.

The GPU hardware does this efficiently and this is also tied to framebuffer blending and Z buffer rendering.

Modern rasterization can be also be conservative to find all blocks touching a triangle. Internally a 2x2 block is used by the GPU to provide 2x2 blocks called quads. This makes it easy to compute the mip level and anisotropic footprint during texture mapping.

Modern GPUs might have multiple levels of culling with conservative rasterization down to per MSAA sample.

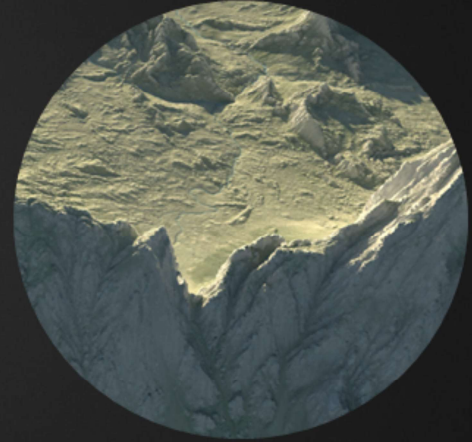
Image from <http://filmicworlds.com/blog/software-vrs-with-visibility-buffer-rendering/>

Image from https://cwyman.org/papers/hpg15_dcaa.pdf

Image from <https://therealmjp.github.io/posts/msaa-overview>

Real-time rendering of large detailed worlds

- Many objects
=> GPU based rendering
- Small geometry
=> avoid HW rasterizer, Mesh Level Of Detail
- Efficient GPU rasterization
=> conservative **Bitmask**
- Irregular sample positions for better AA
=> precomputed edge **Bitmask**
- Efficient compositing with low noise
=> stochastic **Bitmask**



In my new position at EA I am lucky that I can work on real-time rendering of large detailed worlds. Stochastic and ray tracing is still an option but why do we only get a binary intersection from the ray triangle intersection. While having the data in memory we could compute the result for lets say 32 rays in a narrow bundle. We could store those 32 bits in an unsigned integer value and return that for all visibility rays. This should result in much more efficient rendering or soft shadows. Obviously it's not simple but that was the idea.

So we want to render many objects. The CPU is too slow for the draw call count so doing GPU based rendering seems obvious.

With a lot of details we get a lot of small geometry and we know the hardware rasterizer get slow with small geometry. Nanite shows a 3x speed increase in some cases. So we might want to avoid the HW rasterizer and make sure a good view dependent Level Of Detail algorithm keeps the density within reason. Nanite manages clusters of triangles on the GPU and gets good quality and performance from that.

So if we roll our own we need to rasterize triangles ourselves and this is where you can use the "Coverage Bitmask"

You see the bitmask shows up in multiple applications:

There is a conservative coverage for a block of pixels to cull away areas that need no processing.

But also we can use a bitmask to test if a pixel or subpixel sample is inside a triangle.

There is also an application for efficient compositing of transparent materials with motion blur and depth of field.

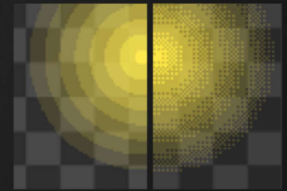
I will not go deeper into this work and focus on the "Coverage Bitmask" here instead.

Where is the connection to Mobile ?

- Draw call performance IS a mobile problem
- Bitmasks is bulk processing => efficient
- Might already use it e.g. A2C or LOD crossfade
- Eventually it comes to mobile e.g. fast GPU integer math
- Mobile has ML hardware => denoise stochastic output
- Immediate mode tiled only fits low triangle density
=> Ray tracing has scene on GPU



Nanite (UE5) is using SW Rasterization



A2C on NVIDIA / Metal

Now you might ask: Is this relevant for mobile or ML? Maybe, let me try to convince you in one slide. Performance on mobile is a problem and many draw calls cost CPU time so this IS a mobile problem. Nanite shows very good performance on desktop and console by moving the bulk of the work to GPU. They didn't yet on mobile as the hardware is not as programmable and features sets are fragmented across the many devices, often running with old drivers. If you are a "Mobile GPU Architect", a "Mobile Rendering Engineer" or a "ML Engineer" you should find some useful knowledge in my slides. It's just a tool to process binary decision is bulk, more efficiently.

You might even use it under a different name. MSAA rasterization happens in the GPU and you might not see the bitmask involved but you can adjust the mask already. "Alpha 2 Coverage" can add a dithering pattern to the MSAA sample mask to emulate partially see through surfaces. With Coverage Out and Coverage you can make your own better A2C in the shader. You can use this for crossfading LODs or fading out objects in the distance.

I believe most desktop / console features come to mobile, for sure GPU driving rendering would be a big one. Shadow mapping is also draw call heavy and ray tracing hardware is already available for mobile. Once you have the ray tracing data structures created you don't need the CPU to submit draw calls any more.

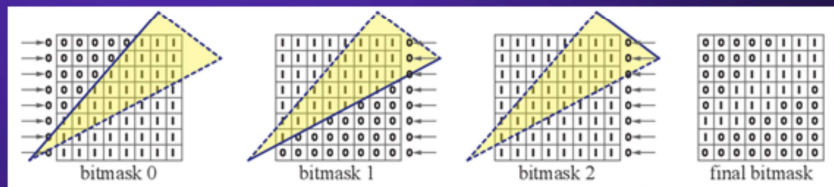
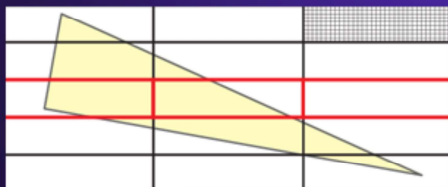
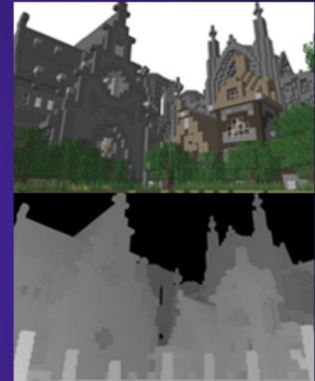
Mobile hardware is even leading with Machine Learning as smart phones make good use of ML inference. This might mean your upsampling and denoising passes can run for free or at least more power efficient. You might know that mobile hardware is often rendering the screen in "tiles". This saves memory traffic as the frame buffer memory can be kept more locally without affecting the memory bus to main memory. Usually the application doesn't see this trick as all draw calls are still submitted once per frame - this is called "immediate mode", in contrast to "retained mode" where the scene delta - the changes - per frame are submitted. Image mode tiled rendering is only efficient if the draw call count and mesh density is low. With more details this is no longer the case. Thankfully with ray tracing we have all the data structures to not require the per frame draw call submission. So Nanite on mobile seems possible but requires some different work from IHV and application side.

A2C Image from <https://twitter.com/kzr/status/1709844771974574564>

[Hasselgren16] Masked software occlusion culling

- **Goal:** Compute tight PVS from camera location on CPU
- 32x8 Tile => 256 bits in parallel using AVX2
- Regular grid of pixels per tile
- Rasterizer occluders and output depth for testing of occludees

=> CPU algorithm to cull draw calls



8

All slides with a purple background are referencing a specific work by others where someone cleverly used a bitmask. I try to point out the key takeaway that are relevant for the bitmask topic here.

I interleave some application of coverage bitmaps. This work is using SIMD and shows a pattern you can see in many other implementations.

This SIMD paper describes an algorithm that is similar to the one we used at Crytek to cull objects on the CPU. Here it uses a 32x8 tile stored and processed in the AVX2 SIMD unit. With 256 bits in parallel the CPU can process a high resolution frame buffer efficiently.

The rasterization outputs its data into a compressed form of a z buffer with z min, z max and the bitmask per tile. All occluders are getting rasterized into the buffer and occludees - objects you want to test if visible are getting tested against the buffer. Occluders use triangle rasterization but a more conservative test for occludees based on the bounding box can be made. This is culling objects on the CPU - early - allowing to cull draw calls and their setup cost. This is efficient under certain conditions. The test needs to be fast and the saved cost needs to make up for the cost spent on doing this.

The Crytek we found such a system was better than a precomputed PVS, worked in outdoor unlike portals, saved a lot of CPU and GPU performance.

We used manually simplified geometry for the occluders. On consoles we managed to implement a readback of the GPU z buffer (after downsampling with max) to the CPU and some reprojection and hole filling. So some of the work moved to GPU.

This trick saves a lot of CPU performance cost but it adds latency as the GPU runs after the CPU commands are issued and the result can arrive one or more frames late. On console this is less of a program than with desktop PCs so it's more suitable there.

Note:

PVS: Potentially Visibility Set, can be precomputed but precomputation time / storage and more conservative and cannot handle dynamic objects well

[Laine11] High-Performance Software Rasterization on GPUs

- Goal: Performant GPU rasterizer
- CUDA
- Precomputed 8x8 coverage mask
- Not as fast as fixed function unit but more flexible

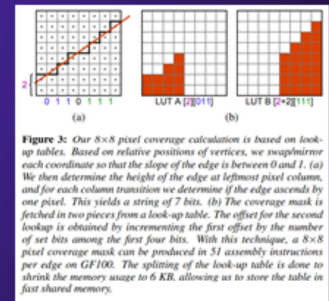
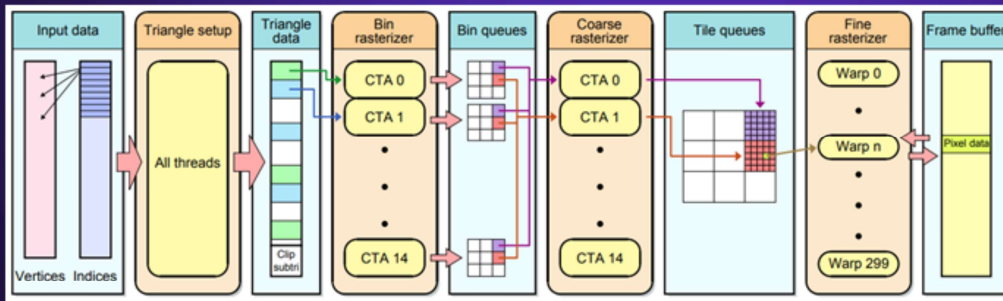
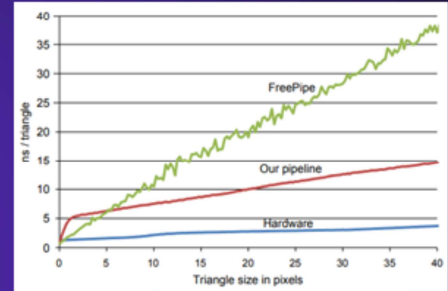


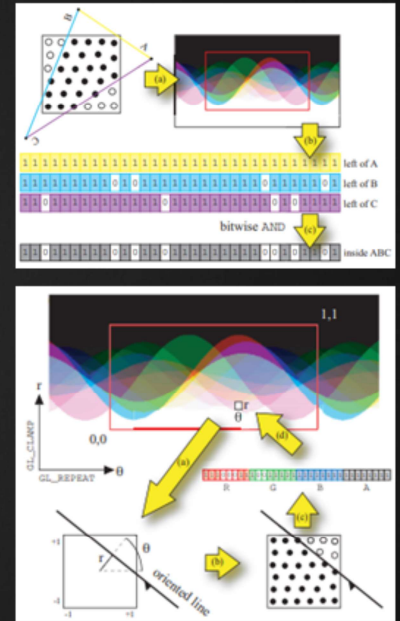
Figure 3: Our 8x8 pixel coverage calculation is based on look-up tables. Based on relative positions of vertices, we assign minor each coordinate so that the slope of the edge is between 0 and 1. (a) We then determine the height of the edge at leftmost pixel column, and for each column transition we determine if the edge ascends by one pixel. This yields a string of 7 bits. (b) The coverage mask is fetched in two pieces from a look-up table. The offset for the second lookup is obtained by incrementing the first offset by the number of set bits among the first four bits. With this technique, a 8x8 pixel coverage mask can be produced in 51 assembly instructions per edge on GF100. The splitting of the look-up table is done to shrink the memory usage to 6 KB, allowing us to store the table in fast shared memory.

This work is also rasterized triangles but not on CPU, here it is on GPU using the NVidia CUDA API. The goal is to emulate the hardware rasterizer with the compute units. This is difficult as the fixed function unit in the GPU is made for that purpose and shader units time is spent instead. A precomputed 8x8 coverage bitmask is used and with some clever mirroring the algorithm saves memory allowing the precomputed data to stay in efficient cache memory. Multiple stages of culling make the method fast but you can see in the graph the hardware rasterizer is still after, especially with larger triangles. The result may not be as fast as the hardware but it allows to run more flexible frame buffer blending to customize the rasterization.

The Precomputed Edge Bitmask

[Eisemann04] Visibility Sampling GPU and Applications
 [Waller00] Efficient Coverage Mask Generation for AA

- [Hough60] Hough Transformation edge detection patent
 $\text{tri}(\text{EdgeAB}) = \text{tex2d}(\text{angle}, \text{distance})$
- Triangle mask
 $\text{tri}(\text{ABC}) = \text{edge}(\text{EdgeAB}) \ \& \ \text{edge}(\text{EdgeBC}) \ \& \ \text{edge}(\text{EdgeCA})$
- Small 32 bit uint texture for performance e.g. 16x16
- Any **convex** shape e.g. triangle 3 edges



I mentioned the precomputed edge mask in the former slide and that is different from the math CPU solution I mentioned. The title slide shows the precomputed edge bitmask. It's important to understand this concept as it's a bit trick that makes the coverage bitmask fast.

You can find more about it in the Eisemann reference, That one references also mentioned the Waller reference as an earlier source.

In order to make a lookup into a 2d texture or 2d array you need 2 values. You can compute the 2 values from an edge and later combined the 3 triangle edges from the 3 edge bitmaps.

The edge defines a half space in 2D and can be parametrized with an angle and a distance to the center of the pixel. This is called the Hough Transformation - see the patent reference.

Once you have the bitmaps for the 3 edges you can binary AND the result and get the triangle bitmask.

A small 16x6 texture is enough for quality, the small approximation error from the limited resolution can be mitigated.

With this method you can implement any convex shape.

As an optimization you can also implement a quad made out of 2 triangles with only 4 edge lookups.

Image from <https://graphics.tudelft.nl/Publications-new/2007/ED07c/PreprintEG2007.pdf>

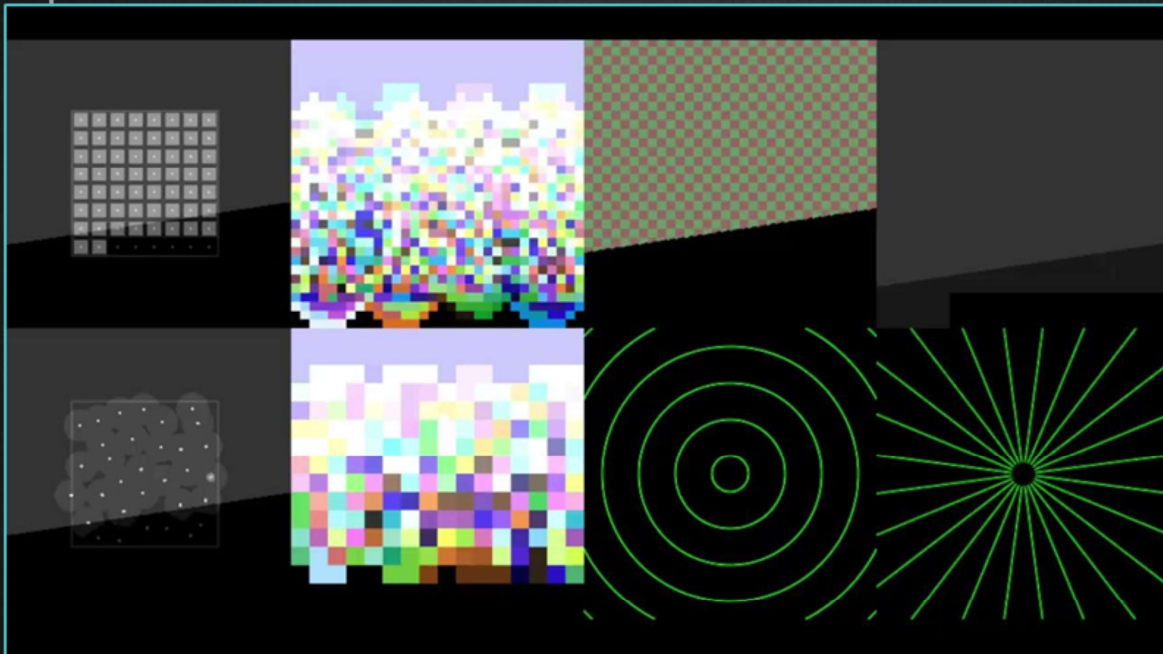
Video 1/3

Antialiased Triangles
2D Rasterization (no occlusion)



11

The "Visibility Bit Mask" prototype works well for a small number of triangles. It's meant to work on tiny triangles but here you can see it also works on larger ones. The video shows the 2D rasterization of an animated mesh. It has high quality anti-aliasing and by scaling the AA filter kernel it's even possible to blur the geometry. This could be useful to render cone traces, approximated by 32 rays. The method relies on a precomputed 2d texture.



The video shows two precomputed bitmaps. The top line shows a conservative bitmap to accelerate the per pixel method. You can see how the regular grid is conservative, only if a pixel is clearly behind the plane it can switch the bit to 0.

The second square shows the precomputed bitmap is 32x32. The lookup position on the half space is visualized as a white circle. The third square is reconstructing the bitmap to 8x8 pixel blocks and you can see a much better than 8x8 blocky images as the bits hold more information.

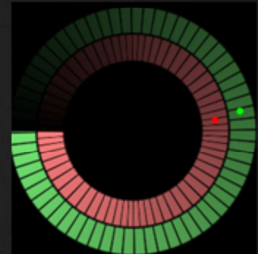
The bottom line shows the antialiased sampling mask used per pixel. In this early prototype a rotated grid was used but later this became a carefully crafted tiling sample pattern that is optimized for edges in any direction.

With 3 texture lookups, one per edge of a triangle you get 3 bitmaps and combining those results in the triangle bitmap.

The second square shows the precomputed bitmap is 16x16. The next two squares show antialiased circles and lines to validate the antialiasing quality.

Some Learnings from GPU Software Rasterization

- 32 Bit per pixel for HQ Anti-aliasing => 32 shades
- 32 Bit per 2x2 quad for LQ Anti-aliasing => 8 shades per pixel
- Needed for water tight rasterization
Flip edge to always point down and invert mask
- atan2() is expensive, see [Waller00] or approximate



```
// correct for 0, 90/360, 180/360, 270/360 and 360/360 but inbetween a bit distorted
// @return unitAngle 0..1 for -PI .. PI
float atan2Approx(float y, float x) { return (y > 0) - sign(y) * (x / (abs(x) + abs(y)) * 0.25f + 0.25f); }
// @param unitAngle 0..1 for -PI .. PI
// @return approximates float2(sin(unitAngle * PI * 2 - PI), cos(unitAngle * PI * 2 - PI))
float2 sincosApprox(float unitAngle) { return normalize(float2(abs(abs(3 - 4 * unitAngle) - 2) - 1, 1 - abs(unitAngle * 4 - 2))); }
```

13

With 32bit you get 32 shades or 30 if you don't count the fully see through and fully opaque as a shade. The HLSL function `countbits()/31` gives you a value between 0 and 1. But you can also use the 32bits and distribute them over 2x2 pixel blocks. This gives only 8 shades per pixel but you have only a quarter number of texture lookups - for larger triangles anyway. This might be a good low quality opinion. This is like 8x MSAA which looks quite good for most content.

Or watertight rasterization "no holes or double rendering" you can flip edges pointing downwards and invert the bitmask. This disambiguates the edge cases .

I found the atan needed for the Hough Transformation to be quite costly. I made a simple approximation. The angle I defined to be in 0..1 range, not in -PI to +PI for easier texture lookups. You can see the red slices (approximation) is a bit more dense every 90 degree but it's minor.

The SinCos approximation is computing a position on a 45 degree rotated square and normalized the vector to a circle and the atan2 approximation is doing the inverse.

This can be optimized further but I solved by bottleneck, see Waller00 in the reference a alternative.

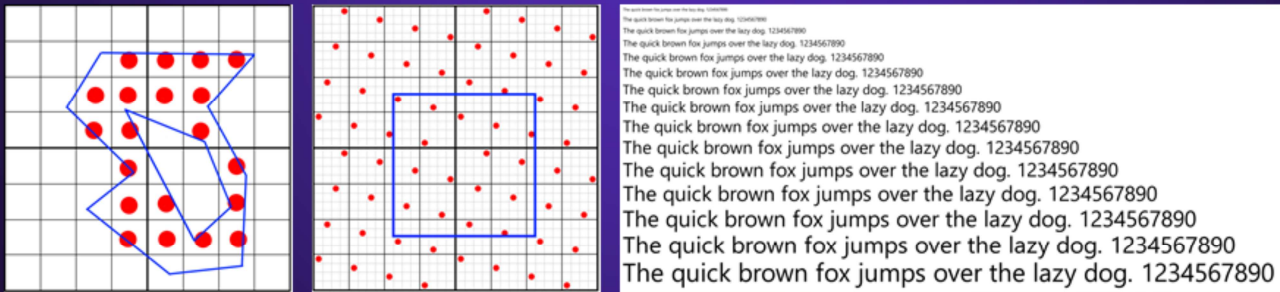
This is more readable version of the code:

```
// diamond shape, see sincosApprox()
// correct for 0, 90/360, 180/360, 270/360 and 360/360 but inbetween a bit distorted
// @return unitAngle 0..1 for -PI .. PI
float atan2Approx(float y, float x)
{
    // diamond shape in -0.5 .. 0.5 range
    float temp = x / (abs(x) + abs(y));
    // 0..0.5
    float squareUnitSide = temp * 0.25f + 0.25f;
    return (y > 0) - sign(y) * squareUnitSide;
}

// diamond shape, see atan2Approx()
// @param unitAngle 0..1
// @return approximates float2(sin(unitAngle * PI * 2 - PI), cos(unitAngle* PI * 2 - PI))
float2 sincosApprox(float unitAngle)
{
    float2 vec;
    // cos() approximation, zigzag pattern around 0.5f, scaled to -1..1 range
    vec.y = 1 - abs(unitAngle * 4 - 2);
    vec.x = abs(abs(3 - 4 * unitAngle) - 2) - 1;
    // this make the crude zigzag pattern much smoother
    return normalize(vec);
}
```

[Beek18] 16x AA font rendering using coverage masks

- **Goal:** font rendering with AA and glyph cache
- 4x4 bitmask per pixel for 16 shades, rotated grid more shades where needed
- Store bitmask in cache for sub pixel blitting
- Two 32 entry 1D tables instead of one 1024 entry 2D table



This is a blog post about efficient font rendering and caching of glyphs

Coverage Bitmask quite naturally solve 2D font rendering with anti-aliasing. Many implementations cache all characters (glyphs) with 8 bit alpha value and blit those to pixel snapped positions.

This work keeps the glyphs with the coverage mask allowing for sub pixel positioning. With small fonts or slowly moving text this can make a big difference.

A 4x4 mask is enough for 16 shades.

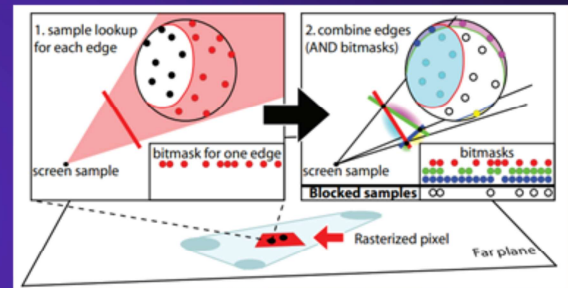
The blog post shows a CPU and a GPU implementation.

The sample points are not in an axis aligned grid but in a rotated grid giving more quality for near vertical and near horizontal edges. It also helps with the sub pixel offset in font rendering. Those are common for font rendering.

Two small 1D tables are used instead of a larger 2D table.

[Sintorn08] Sample Based Visibility for Soft Shadows using Alias-free Shadow Maps

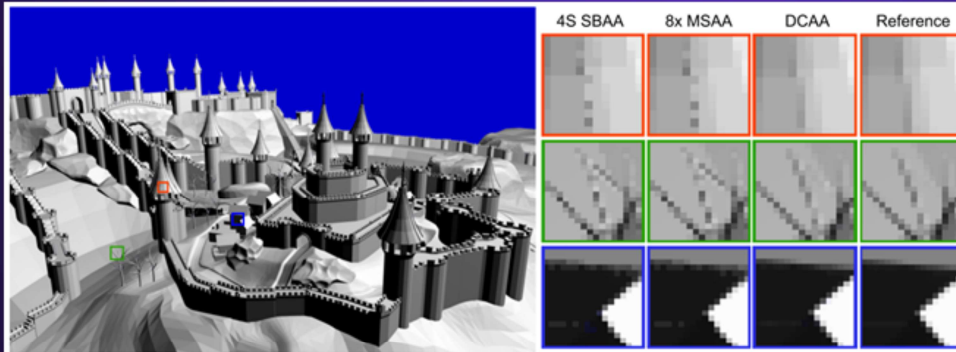
- **Goal:** Area light shadows
- Bits represent occlusion of rays to the light
- Extruded triangle formed by 4 3D planes
- 3D Half space => 2 angles and distance
- Precomputed 128x128x128 texture
- 4 channel x 32 bits = 128 bits for less noise
- Jitter to trade banding for noise
- Also look at [Kalbertodt23]



Compared to anti-aliasing we need more bits to avoid banding or noise. The 3D volume used here is to implement a 3d half space. A triangle shadow can be defined by 4 3D planes and
In this methods a single lookup provides a 128 Bit mask for an 3D plane. This requires a rather large 128x128x128 texture. This is trashing the cache but I believe you can optimize this quite a bit.
As soft shadows can be large on screen this method uses 128 bits and further adds random rotation to the pixels trading noise for banding. It looks better and noise can be blurred in a postprocess.
I think this work has the seed for spending up ray traced shadows and a lot of potential.

[Wang15] Decoupled Coverage Anti-Aliasing

- **Goal:** Immediate Mode Rendering with Antialiasing
- Bitmask: ordered grid, 64 bit, 3 edge lookup, binary AND
- **Store 4 per pixel:** GBuffer + Mask + triangle plane equation



R8	G8	B8	A8
Depth		Stencil	
Normal U		Normal V	
Diffuse Albedo RGB		Emissive	
Metal		Roughness	

(a) A typical G-Buffer layout (per sample)

R8	G8	B8	A8
Depth		Stencil	
Normal U		Normal V	
Face Equ U		Face Equ V	
Coverage Mask 1			
Coverage Mask 2			
Diffuse Albedo RGB		Emissive	
Metal		Roughness	

(b) DCAA G-Buffer layout (per surface)

This work solves the primary visibility. Geometry can be rendered in the classic image mode and occlusion is solved during rasterization.

It's basically a deferred renderer that maintains 4 GBuffer values per pixel. Each of those 4 surfaces also has a coverage mesh and a plane equation for depth. The mask allows to have surface merging with very little occlusion error.

The plane equation is stored to allow for z buffer intersection.

The paper is from NVidia and it seems to me with special hardware support this could be much faster.

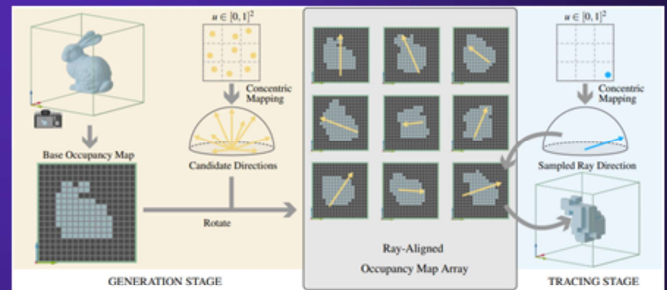
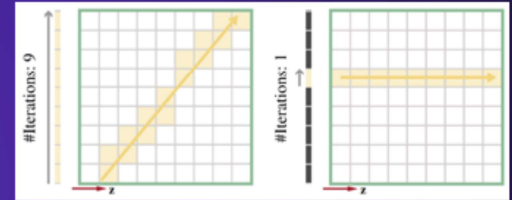
This paper is interesting but rather high quality and expensive.

The 64 bit coverage mask could have been 32 bit but even better someone could use the 4 samples for 4 pixels making this much faster at the cost of quality.

As you have the coverage mask you have to distribute the colors to the 4 pixels after shading.

[Zeng23] Ray-aligned Occupancy Map Array for Fast Approximate Ray Tracing

- **Goal: Approximate Ray** intersection
 - 1D Bitmask stores geometry coverage of a ray
 - First bit is first ray hit
- Steps per frame
 - Voxelize to BOM bitmask
 - Rotate BOM to multiple RAO bitmask
 - Ray trace by lookup in RAO bitmask
- Approximation improvements
 - Jitter each frame for resolution
 - Denoise pass for quality
 - Cascades / per object for larger scenes



This work is using a 3d grid bitmask to accelerate intersection with geometry. As the limited resolution is causing approximation errors the result is not perfect.

You can implement shadows and Ambient Occlusion this way and if you additionally store geometry attributes (not part of bitmask) and implement lighting you also can implement glossy reflections or indirect lighting.

The method has 3 steps happening each frame:

- 1) All geometry is rendered into a voxel representation and a single bit is enough to store occlusion. Multiple methods are possible

TODO

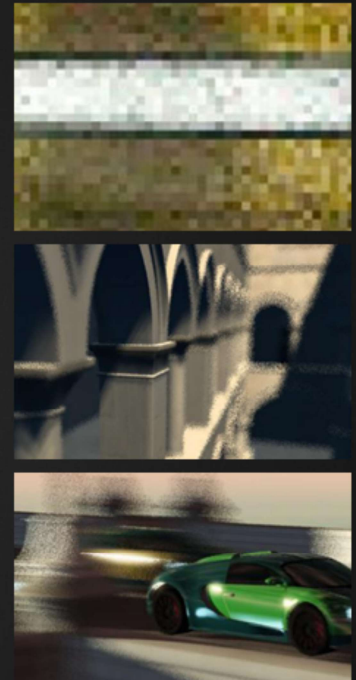
The method recreates the data every frame with some jitter making it less approximate and smooth when the result is combined over multiple frames.

hit position in $O(1)$ from bitmask AND and getting first bit

No hierarchy, no knowledge what was hit, cascades possible

The Usage Axis

- **Spatial** (position)
 - Domain 1D line light / 2D AA / 2D DOF / 3D Object / 4D Portal
 - Primitive Line / Triangle / Quad / Disc
constructed from Edge / 3D Half-space
- **Temporal** (time)
e.g. motion blur
- **Probability** (random decision)
 - Approximate transparency => Alpha 2 Coverage
 - Ray casting volumes



Images from [Enderton12]

Looking at the mentioned applications there are some patterns to observe. To categorise the different methods you can group the using of the coverage maske in spatial (position), temporal (time) and probability (random decision). **Spatial** can be a n-dimensional domain. 2D is good for pixel anti-aliasing (see top image) but also for Bokeh Depth-Of-Field (see middle image where stochastic rasterization is used). With 3D coverage mask you can store the coverage of a 3d objects in a axis aligned bounding volume to accelerate ray tracing [Yoshimura23]. A 4D space is formed by two 2D portal surfaces and there you can store the visibility between the points on the portals. 1D can be used to compute shadows from a line light.

You can render different primitives like lines, triangles, Spheres. You can render any convex object with half spaces or you can render concave objects with holes with another method (more about that later)

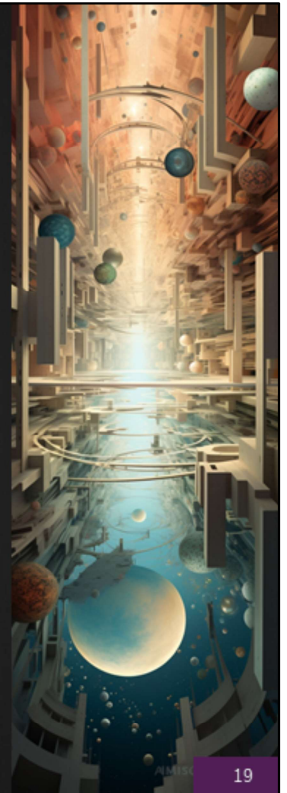
The **temporal** axis can be used for motionblur, see stochastic rasterization on the bottom image.

A bitmask also can be used to express multiple random binary decision - it can express a **probability**. This way you can express translucent objects. Hardware Alpha To Coverage is a simple ordered grid but you can craft your own and permute for each object to get better compositing of many layers [Enderton11].

Images from <https://casual-effects.com/research/Enderton2012Stochastic/Enderton2012Stochastic.pdf>

Deconstructing the Components

- Value
 - Conservative / Stochastic / Approximate / Exact
 - Bit count 8 / 32 / 128
 - Bit order unused / randomized / sorted by an axis or distance
- Location of samples
 - Ordered grid / rotated grid / stratified / blue noise / density distributed
- Instance variation
 - e.g. per pixel random rotation or temporal offset
- Math vs Memory tradeoff



19

Using the Coverage Bitmask you can make conservative decisions like: should I process a triangle in this grid cell? You can make multiple stochastic decision to approximate a non binary value with reduced noise, 2x more bits means $\frac{1}{2}$ the noise.

You can make some exact decision like which pixels a triangle covers in a 2d block.

You can vary the bit count and you can make use of the bit order. Remember [Zeng23] where the first 1 bit gives you the approximate hit location for a ray.

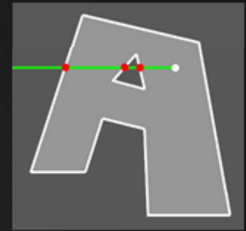
With precomputed bitmasks it's easy to place the sample point at any location. This can be used to implement a filter kernel that is not a box filter, much cheaper a math solution, at least a software math solution.

If you want to approximate a value and the number of bits is not enough to give a result without banding (e.g. area light shadows), you can add per instance variation like Adding a random rotation or shift. The result is some high frequency noise, spatial or temporal or both. This might be acceptable or something a following denoise pass can remove easily. Denoise for binary decision are common but quality and performance can be better if you have many binary decision (e.g. 32 bits).

Coverage masks can be generated with the help of memory and/or math. Depending on the available resources either one can be the right choice.

Fill from Boundary

- When compared to half spaces also supports concave and holes
- Regular Math based 1D mask bit shift
- Irregular precomputed 2D mask sort samples by Y, mask after lookup
- 1D Inside test e.g. stencil shadows
- 2D Rasterization e.g. font rendering
- 3D Solid Voxelization e.g. HW rasterizer with XOR frame buffer blend
- Single pass XOR mask from outside to the point
- Two passes like Amiga Blitter polygon fill
- Output bits XOR bits
- Output weight add area for front facing, subtract back facing => Analytical AA



Inside = hitCount % 2

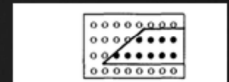


Figure 7. Polygon edge mask.

[Captenter84]



A classic graphics programming solution to the point in polygon test is to draw a line from the point to some point outside and if you count the number of intersections. If this is a odd number you are inside.

Remember using the binary AND of edges tests only works for convex objects.

This in contrast works with concave objects, with holes and objects in objects. But you need to have a closed mesh and the intersection test should not double count in some corner cases.

When doing this on a ordered grid you can use math to implement bitmask updates. It's easiest to think about each line separately. You want a bitmask that has all bits 0 left to the edge and all bits 1 right to the edge.

This can be implemented in **math** with a shift of a all-1 bitmask. This bitmask can be xor-ed with the existing bitmask (all former edges). The result is a bitmask that has 1 for inside and 0 for outside.

With SIMD multiple lines can be processed in parallel.

Some of the mentioned papers did this with a **precomputed mask**. To reduce the number of dimensions you can lookup a halfspace and implement the masking of where the edge starts and ends with math.

Here it's best to make use of the order of bits in the bitmask. This even allows for **irregular** distributed sample points.

Again the domain can be multi dimensional.

2D for font rastization - concave shapes would be very limiting here.

3D can be used to implement efficient voxelization (xor binary frame buffer blend, output up to 128 bit mask from triangle depth at this pixel, someone might say this is 1D but it comes with a 2D grid so it's 3D)

A single bit **1D** was used for stencil shadows in Doom. The lack of anti-aliasing, soft shadows and unpredictable performance characteristics made this short lived. Shadow maps replaced stencil shadows.

But this can be (and has been) extended to use a 2D coverage mask to implement high quality area light shadows.

The amiga coprocessor Blitter had an interesting feature. It was able to fill polygons quickly by rendering (xor-ing) all polygon edges with 1 bit per line.

After all edges have been drawn a full screen pass was filling in all bits between the edges. This **2 pass** method can be faster than the **1 pass** method in some conditions. Nowadays it would be simple to implement this pass vertically using integer math on the CPU or GPU.

Note that if you don't need the bits but only need the area / **weight** (e.g. font AA) you can implement a analytical rasterizer that simply adds / subtract areas.

Amiga Blitter polyfill explained: https://www.youtube.com/watch?v=H7QB_pslX10

References 1/3

- [Bloom70] Space/Time Trade-offs in Hash Coding with Allowable Errors
https://en.wikipedia.org/wiki/Bloom_filter
- [Captenter84] The A-buffer, an antialiased hidden surface method
<https://dl.acm.org/doi/10.1145/964965.808585>
- [Akeley93] Alpha To Coverage
https://en.wikipedia.org/wiki/Alpha_to_coverage
- [Waller00] Efficient Coverage Mask Generation for Antialiasing
https://www.researchgate.net/publication/262245332_Efficient_Coverage_Mask_Generation_for_Antialiasing
<https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=29456d27f8d1d439e6ae44d3433ba4998ace642f>
- [Kilgard01] Robust Stencil Shadow Volumes
https://developer.download.nvidia.com/assets/gamedev/docs/StencilShadows_CEDEC_E.pdf
- [Eisemann04] Visibility Sampling on GPU and Applications
<https://graphics.tudelft.nl/Publications-new/2007/ED07c/PreprintEG2007.pdf>
- [Kayalin07] Screen Space Ambient Occlusion
https://en.wikipedia.org/wiki/Screen_space_ambient_occlusion

References 2/3

- [Sintorn08] Sample Based Visibility for Soft Shadows using Alias-free Shadow Maps
<https://maverick.inria.fr/Publications/2008/SEA08/softshadows2008EGSR.pdf>
- [Enderton11] Stochastic Transparency
https://research.nvidia.com/publication/2011-08_stochastic-transparency
- [Enderton12] Stochastic Rasterization
<https://casual-effects.com/research/Enderton2012Stochastic/Enderton2012Stochastic.pdf>
- [Heitz12] Representing Appearance and Pre-filtering Subpixel Data in S. V. Octrees
<https://inria.hal.science/hal-00704461>
- [Liu13] A Micro 64-Tree Structure for Accelerating Ray Tracing on a GPU
<https://graphicsinterface.org/proceedings/gi2013/gi2013-22>
- [Wang15] Decoupled Coverage Anti-Aliasing
https://cwyman.org/papers/hpg15_dcaa.pdf
- [Hasselgren16] Masked software occlusion culling
<https://www.intel.com/content/dam/develop/external/us/en/documents/masked-software-occlusion-culling.pdf>
- [Crassin18] Correlation-Aware Semi-Analytic Visibility for Antialiased Rendering
https://research.nvidia.com/sites/default/files/pubs/2018-08_Correlation-Aware-Semi-Analytic-Visibility//CorrelationAwareVisibility_AuthorsVersion.pdf

References 3/3

- [Beek18] 16x AA font rendering using coverage masks
<https://superluminal.eu/16x-aa-font-rendering-using-coverage-masks-part-ii>
- [Bako22] Deep Appearance Prefiltering 16x16
http://civc.ucsb.edu/graphics/Papers/TOG2022_DAP/PaperData/TOG2022_DAP.pdf
- [Zeng23] Ray-aligned Occupancy Map Array for Fast Approximate Ray Tracing
<https://zheng95z.github.io/assets/files/egsr2023-roma.pdf>
- [Therrien23] Screen space indirect lighting with visibility bitmask
<https://arxiv.org/abs/2301.11376>
- [Sannikov23] Radiance Cascades: A Novel Approach to Calculating GI
https://drive.google.com/file/d/1L6v1_7HY2X-LV3Ofb6oyTlxgEaP4LOI6/view
- [Yoshimura23] Subspace Culling for Ray-Box Intersection
<https://arxiv.org/pdf/2305.08343.pdf>
- [Kalbertodt23] Tiled per-triangle soft shadow volumes
<https://lukaskalbertodt.github.io/2023/11/18/tiled-soft-shadow-volumes.html>



Thanks

Everyone from
referenced work

Colin Barré-Brisebois
Henrik Halén
Jon Greenberg
Chris Lewin
William Donnelly
Alan Wolfe
Jim Royal

seed.ea.com
Electronic Arts


SEED



Thanks to ...

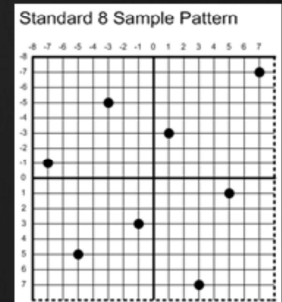
Bonus Slides

Electronic Arts

 SEED

Anti-aliasing

- Is expected quality despite high DPI
- **Supersampling** too slow and costly
- **MSAA** is the hardware solution
 - Does not fix all aliasing
 - Does not scale well after 8x
- Analytical AA => ok for 2D, too complex for 3D visibility
- Better than box filter => emissive objects in sub pixel motion
- Real-time Ray-tracing => 1 ray per pixel, jitter per pixel
- TemporalAA => whole frame jitter, has artifacts
- ML hardware => AA, Upscaling, frame generation



Relevant Hardware Features

- **SISD**: Single instruction, single data (CPU 32 / 64 bit in parallel)
 - 32 / 64 bit CPU Atomics
- **SIMD**: Single instruction, multiple data (CPU MMX / SIMD / AVX 4/8 wide)
- **SIMT**: Single instruction, multiple threads (GPU)
 - 32 / 64 bit GPU Atomics, fast when no collision
 - 32 / 64 Bit Bitmask is used to mask out threads in a threadgroup
`WaveActiveBallot()`, `WaveReadLaneAt(WaveGetLaneIndex)`
- Stencil buffer
- Raster Order View
- Binary Frame Buffer Blending (OR, AND, XOR)

CPUs and GPUs have hardware feature to make a HW or SW version of this efficient.

CPUs are normally Single instruction, single data but they can process 32 or 64 bits in parallel. Mostly for multi core thread synchronization they also have atomics which can do bitmask math without having to use expensive critical sections.

Modern CPUs also have Single instruction, multiple data like SIMD or AVX where multiple machine words (8 / 16/ 32 / 64 bit wide) are processed in parallel (e.g. 4 / 8 / 16 elements at once).

GPUs use a close relative of SIMD, SIMT, something that emerged out of vector processors. Here the processor has more flexibility to plan many thread in flight to improve throughput, at the cost of latency. When you have a lot of the same units (like pixels, triangles or vertices) to process this is much more efficient.

GPUs also have atomics but as the latency in a GPU is higher anyway they don't cause as much of a performance issue. In comparison they are more lightweight to the CPUs atomics. Also there is no guarantee about execution order between atomic operations for different memory locations making them more efficient when working on different (no collision) data.

GPUs also allow for some bitmask usage within the thread group and this is even faster but also harder to use.

A old GPU feature is the stencil buffer with 1 or 8 bits.

Raster Order View is a modern GPU feature that ensures memory operations are executed in the order rasterization would do. This can be slower than frame buffer blending but it's more flexible.

If you only need AND/OR/XOR operation you are better off using the new binary frame buffer blending feature. The hardware can implement this faster or they might optimize it in the future when enough developers us this functionality.

Orthogonal Masks for Coarse Culling

Input: 64 quads, Output: grid of pixels

Phase 1: ThreadGroup=Quads, Loop over bbox
One or Two AtomicOr in group shared

memory

Phase 2: ThreadGroup=Pixels, Loop over all 1 bits

```
while(mask) {  
    uint bit = firstbitlow(mask);  
    mask &= ~(1u << bit);  
    work(bit * 2); work(bit * 2 + 1);  
}
```

Option a) Brute Force

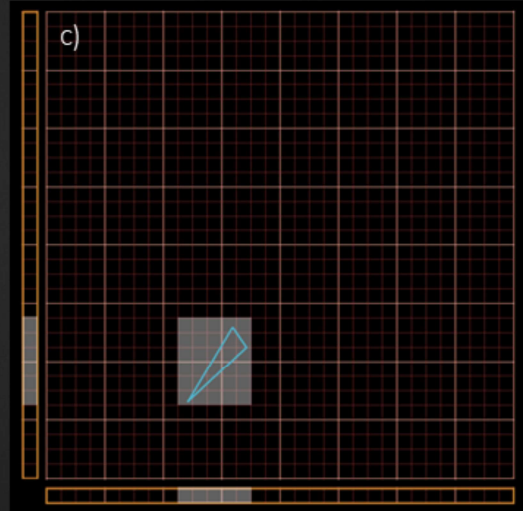
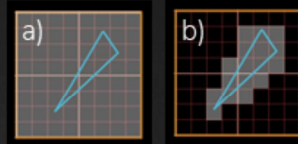
64 quads x 32 x 32

Option b) One 2D Bitmask (8x8 pixel grid)

mask = groupshared[x,y]

Option c) Two 1D Bitmasks (32x32 pixel grid)

mask = groupshared[x] & groupshared[y]



28

If you use bitmasks for culling work - you can use the bitmask in different ways. Let's say you want to rasterize a triangle and find all pixels triangle intersections.

A conservative result is acceptable - meaning we accept false positives.

Then you can work on all those pixels and do more work, like doing antialiasing with multiple samples per pixel.

Like in tiled rendering two phases are needed. If we run on a GPU we can choose a thread group size. In Phase 1 we process 64 Quads in parallel. For each quad (two triangles) we loop over all affected grid elements and set the right bit with an atomic OR. For very small quads (micropoly) this is fast.

In Phase 2 we process pixels in parallel. Using the mask we know which quads we need to consider. Here is the code to do that.

We have some options on how we use the bitmask.

a) as a comparison you can brute force the work, consider all triangles for each pixel. It will work but is not very fast.

b) Better is to use a 2D bitmask for each pixel in some block. A 64bit bitmask works for 8x8 pixel block

c) Much more efficient for small quads is to use two 1D bitmasks and AND those. Now we can process a 64x64 pixel block with the same bitmask memory.

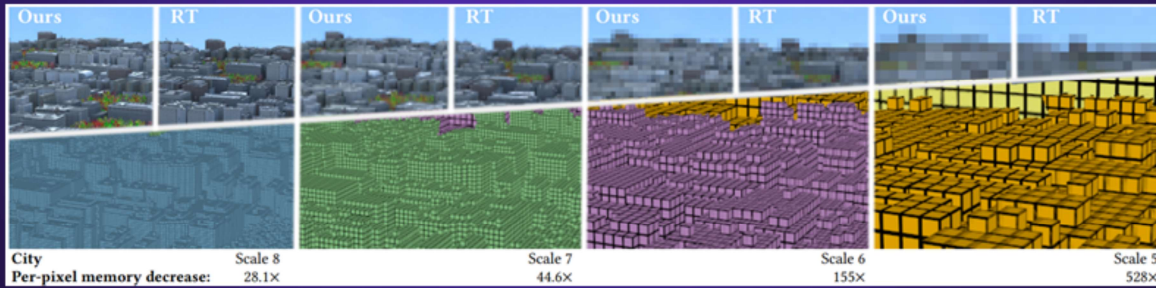
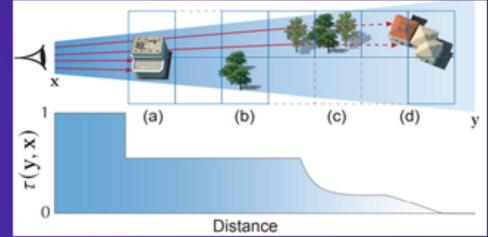
This memory is in group shared memory and option b) and c) need the same amount of memory (8x8 uint = 2 * 32 uint).

Option c) allows to fill a larger grid of pixels and scales linearly and not quadratically in size for Phase 1.

```
while(mask) {  
    bit=firstbitlow(mask);  
    mask &= ~(1u << bit);  
    work(bit*2); work(bit*2+1);  
}
```

[Bako22] Deep Appearance Prefiltering

- **Goal:** More correct visibility than directional alpha
- Precomputed with ray tracer
 - 16x16 2D spatial coverage mask
 - Per 2D direction (spherical coordinates)
 - Per voxel in hierarchy
- Radiance between voxel A and B
 - Positively correlated (A&B=0) => Unblocked
 - Negatively correlated => Occluded



[Therrien23] Screen Space Indirect Lighting with Visibility Bitmask

- **Goal:** Efficient **high quality** Ambient Occlusion
- [Kayalin07] SSAO, [Sannikov23] SSGI
- Input is depth buffer and normal
- Hemisphere is split into slices
- During tracing it accumulates occlusion bitmask

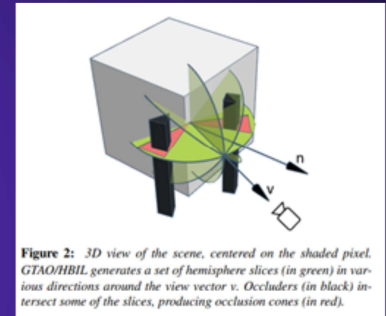
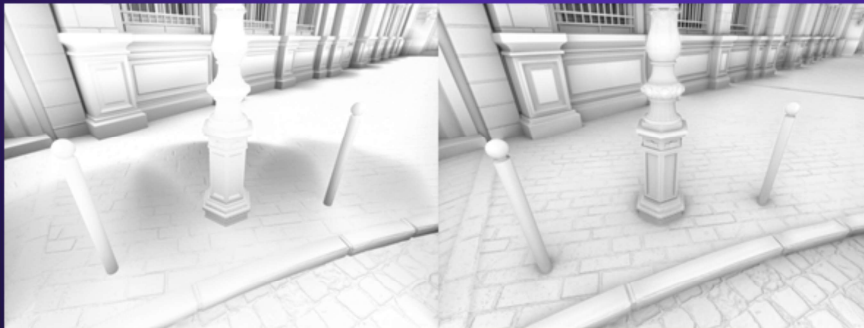


Figure 2: 3D view of the scene, centered on the shaded pixel. GTAO/HBIL generates a set of hemisphere slices (in green) in various directions around the view vector v . Occluders (in black) intersect some of the slices, producing occlusion cones (in red).

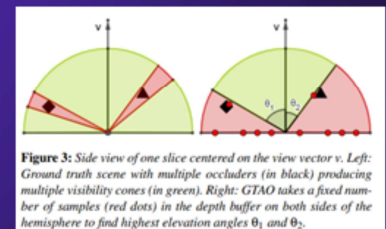


Figure 3: Side view of one slice centered on the view vector v . Left: Ground truth scene with multiple occluders (in black) producing multiple visibility cones (in green). Right: GTAO takes a fixed number of samples (red dots) in the depth buffer on both sides of the hemisphere to find highest elevation angles θ_1 and θ_2 .

30

Screen Space Ambient Occlusion is a well known method to compute Ambient Occlusion - invented by Vladimir Kayalin.

Ambient Occlusion is a percentage that expresses how much the nearby geometry is occluding a specific surface point. You can compute it with ray tracing but

That can be costly. SSAO is cheaper as it uses only a depth buffer (and a normal buffer, if available). Visually AO approximates indirect lighting but at a much smaller cost.

SSAO wants to integrate the occlusion of the depth buffer samples in a specific direction - this can be computed with a min and a max angle.

This paper chooses to use a bitmask and it results in better quality.

I personally implemented SSAO and was able to avoid the artifact without a bitmask but I think this method is better.

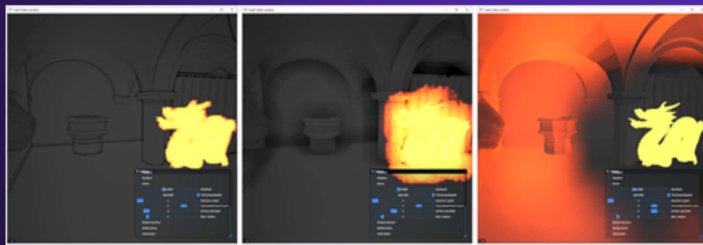
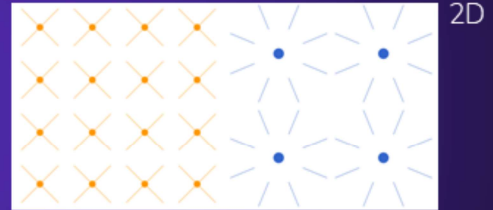
I also seen the bitmask for tracing ScreenSpace depth buffer uses in [Sannikov] - more about this later.

Image from <https://www.arxiv-vanity.com/papers/2301.11376>

GTAO: Ground Truth AO is treating depth buffer like heightmap with some falloff (the example image might have been tweaked for illustration)

[Sannikov23] Radiance Cascades: A Novel Approach to Calculating Global Illumination

- **Goal:** Indirect Lighting from Screen Space Ray-Marching
- 4 ray casts in parallel
- Bitmask: 128 bit
- Visibility
 - Nearby: more position resolution
 - Far: more directional resolution



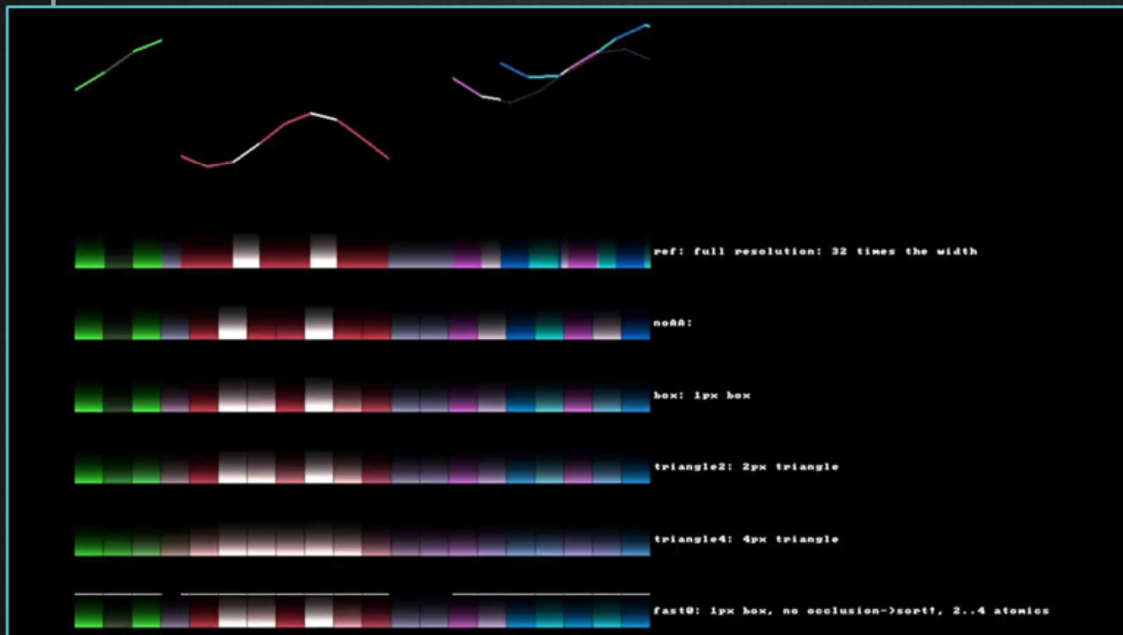
This indirect lighting method is very performant and quite high quality. It uses a coverage bitmask during the screen space ray marching step like the SSAO paper a few slides ago.

Multiple cascades are used to approximate near and far lighting. Take a look at the 2D example, it's similar to a 3D version:

The near cascades have high spatial resolution, the orange shows 16 samples with 4 directions.

The one level further cascaded have lower spatial resolution but high directional resolution, here 8 directions.

This is stored in the same amount of memory. The result is a combination of many cascades and you get good quality at good performance.



This video shows a side view of a single 20 pixel line in a 2d renderer. On the top you can see multiple patches with different colors. They overlap, the viewer is on the top, looking down. This is why geometry can be occluded (dark grey).

One patch is animated, another one is moving.

Below you can see multiple 20 pixel lines that show different methods. A color pixel is shown as gradient to express the HDR ness of a color. You see the red patch has some very bright white pixels. Those could be specular highlights.

As they are sub pixel in size and all geometry is pixel size we should see extreme aliasing if nothing special is done. You can see this in the noAA line.

The other lines shows different methods and visualizations that helped during development to assess the quality of different rasterization methods.