GDC

Real-Time Reflections in MAFIA III and Beyond

Martin Sobek
Lead Rendering Engineer

GAME DEVELOPERS CONFERENCE® | MARCH 19–23, 2018 | EXPO: MARCH 21–23, 2018 #GDC18

UBM

Martin Sobek has been passionate about making games since 1992. Martin studied computer science at Masaryk University in Czech Rebublic with a specialization in computer graphics. He joined Illusion Softworks in 2007 and worked on 'Mafia II'. He then moved to Hangar 13 in California in 2013 and led the rendering team toward a successful release of Mafia III. Now he is lead rendering engineer at Hangar 13 Brno, Czech Republic.

Mafia III is running on custom engine, which is an evolution of engine used in Mafia II.

Agenda

Motivation

Existing solutions

Ray casting on GPU

Reflection rendering

Reflections on rough surfaces

Timings, Results, Conclusion

Future work

Motivation

With PBR, reflections are an essential part of material shading

Having proper reflections is a major step towards photorealism

Not happy with any of the existing solutions

Obvious case – reflection from wet road

Example 1 – without reflections

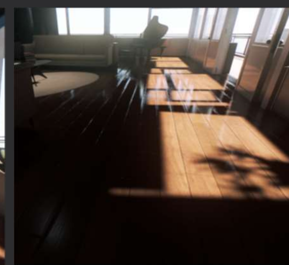
Doesn’t even look wet without reflections.

Most of the surfaces are quite rough, reflections still play major role.


Example 2 – without reflections

Existing solutions

Screen-space tracing



- PROS
  - Doesn't require content authoring
  - Good performance
  - Low memory cost
- CONS
  - Only captures what's on screen
    - → Lots of missing information (especially for high roughness)
    - → Unstable with movement (camera or dynamic objects)

Existing solutions

Pre-filtered cube-map look-up

- PROS
  - Simple to implement
  - Great performance
- CONS
  - Floating reflections
  - Problems with transitions between CMs
  - Iteration issues (if pre-rendered)
  - Missing dynamic objects
  - Isotropic

To achieve anisotropy, we would need to pre-filter the CM with multiple kernel configurations that would make it much less practical.

GDC GAME DEVELOPERS CONFERENCE® | MARCH 19–23, 2018 | EXPO: MARCH 21–23, 2018 #GDC18

Existing solutions

Combination of SSR + Pre-filtered cube-maps

- PROS
  - Simple to implement
  - Good performance
- CONS
  - Partially: Floating reflections
  - Partially: Missing dynamic objects
  - Isotropic
  - Problems with transitions between CMs
  - Development iteration issues (if pre-rendered, need to re-render every time scene changes)
  - Stability issues (with camera movement)

UBM

Bad issues around main character in 3rd person games.

Existing solutions

SSR + Parallax-corrected cube-maps (pre-filtered)

- PROS
  - Good performance
  - No floating reflections
  - Better transitions between CMs
- CONS
  - Only works well for environments with certain shapes
  - More content authoring (scene approximation)
  - Partially: missing dynamic objects
  - Isotropic
  - Iteration issues (if pre-rendered)

Multiple variants exist. E.g.:

Kevin Bjorke: sphere approximation

Bartosz Czuba: box approximation

Seb Lagarde: convex approximation

Existing solutions

Cone tracing

- PROS
  - No floating reflections
  - Dynamic objects can be included
  - Robust
  - Doesn't require authoring
- CONS
  - Requires run-time scene voxelization (difficult to implement)
  - Huge memory requirements
  - High GPU cost (scene update, tracing)
  - Isotropic

Existing solutions summary

None of the existing solutions fulfilled all requirements:

- Stability with camera movement
- Good performance and memory cost
- Working seamlessly in all environments (indoor, city, landscape)
- Reasonable content authoring cost
- Real-time update (scene changes)

Problem breakdown

Problem #1

- General GPU-friendly ray casting
- Find ray intersection with scene
- Achieve mirror reflections (roughness=0)

Problem #2

- Proper BRDF on all materials
- What rays to cast?
- How to process the results

Ray casting on GPU

Mesh/BVH

- Branching
- Non-coherent memory accesses
- How to compute shading?

Voxels

- Memory heavy
- Non-trivial implementation

Depth texture

- GPU-friendly
- Trivial implementation
- Not perfect coverage of the space

Update on mesh/BVH: New API (DX12 DXR) and HW has been announced that is supposed to address some of the issues.

Covering space with 2D projections

Cube-map covers space perfectly from a given POV

- 6 2D views
- Add depth
- Works well if ray start position is close to CM origin
- Efficiency decreases with distance from origin

Tracing height-fields seems to be the right direction for nowadays GPUs.

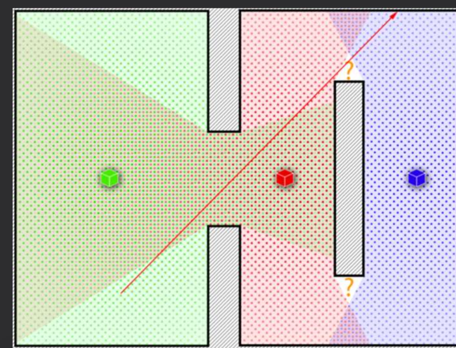
We like the small implementation cost (we already have 2D rasterization implemented), low memory footprint and good performance.

Cube-map placed in camera covers reflection on vast majority of the pixels on the screen. Has been proven on a prototype.

But can’t render a cube-map every frame! Sparse updates (like 1 side every frame) would result in reflections popping and latency.

Multiple cube-maps

- Pick best CM for ray start position
- Switch to a different CM when ray enters “shadow region”
- Use cube-map array



We’ve got 3 manually placed cube-maps on the right image.

Ray starts tracing the green CM, at some point gets to shadow region, red CM takes over. Ray reaches area without any coverage (implausible result) and blue CM takes over to finally find a hit.

Cube-map array: to be able to run single tracing pass.

The more complex the environment is, the less efficient the CM coverage is. Would be terrible for fractals but works well for typical environment that we live in.

GDC GAME DEVELOPERS CONFERENCE® | MARCH 19–23, 2018 | EXPO: MARCH 21–23, 2018 #GDC18

Cube-maps placement

- Hand-placed CMs
  - Indoors: 1 CM for each room/hallway
  - City: Crossroad and every about 50 m on straight roads
  - Landscapes: Sparsely placed CMs (approx. every 100x100 m)
- Automatic backup CMs
  - Automatically placed CM in camera, if no hand-placed CM is around
  - Mainly used during development

UBM

Manually placed CM is always better than the automatic backup probes.
It was used on open water areas for example.

Cube-map coverage issues

Manual placement: Need good tools

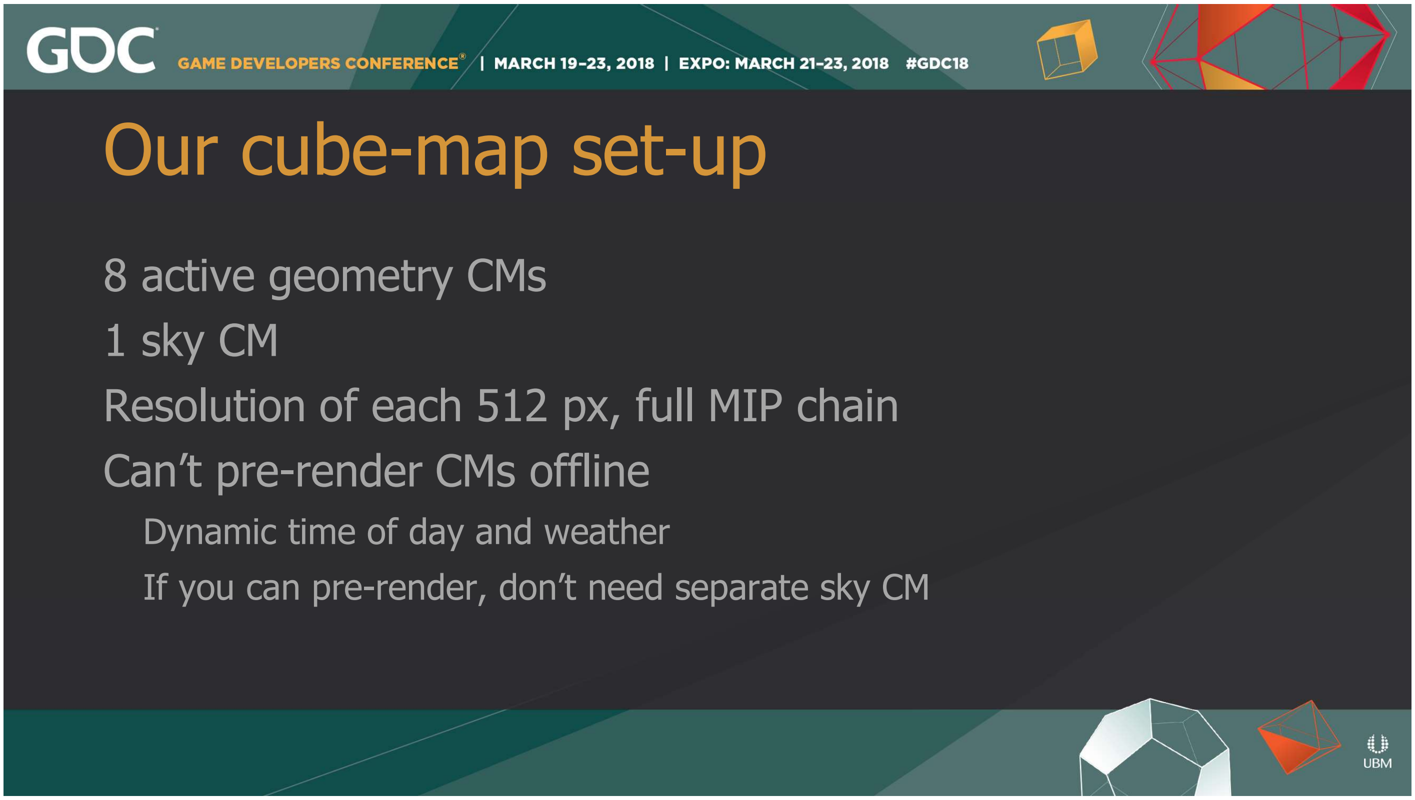
Dynamic objects: costly update → rely on SSR

Not all pixels are covered

Inconsistent resolution (depends on distance from CM origin)

Thin objects (rails, poles, signs, …) interrupt rays

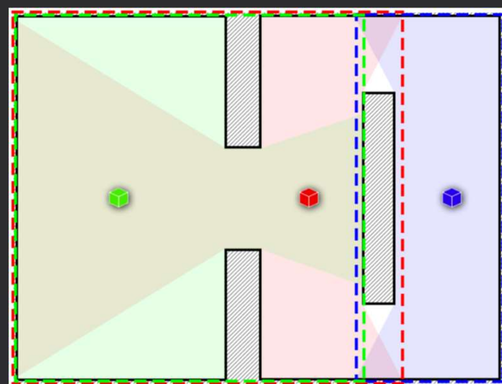
Thin objects create aforementioned shadow regions that interrupt ray tracing.



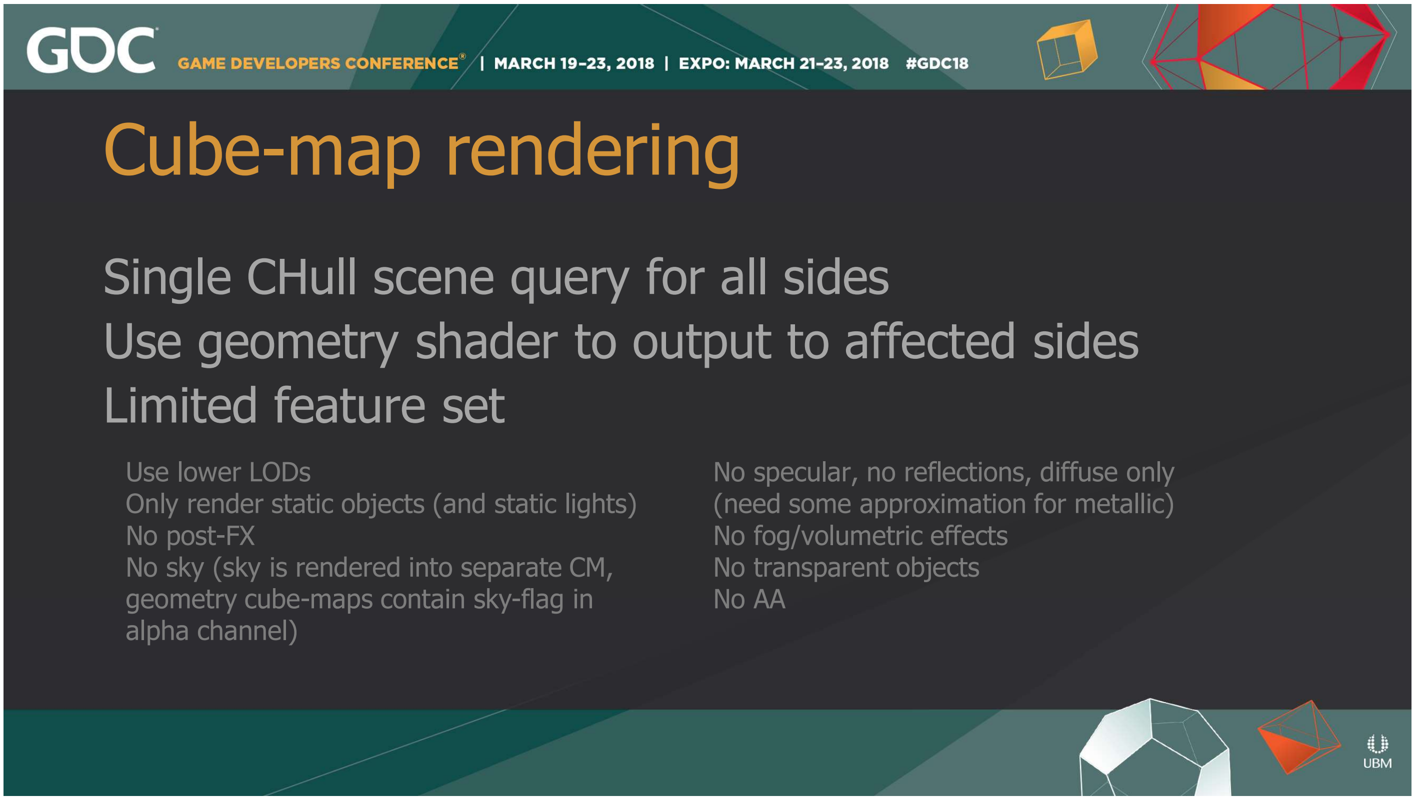
CM array slightly larger to be able to prepare new CM.

Cube-map rendering

Pre-compute max view distance offline (for each side)



Only consider objects in the pre-computed CM range for rendering the CM.



We want to submit as few draw-calls as possible. Many static objects are large (terrain, buildings) and intersect more than 1 cube-map view frustum (end up in more sides). So we collect all objects (for all sides) and then only test, which sides are affected (fill to CB from CPU). Submit just 1 draw-call that outputs the object into multiple sides using geometry shader.

We have learned that Geometry Shaders aren't the most optimal way of attacking multi-viewport rendering, however is supported on all our platforms and is least intrusive from the shader combinations point of view.

We are rendering simpler LODs – these don't have many vertices, so in the end this is not an issue and we will stick to this solution.

No specular in CMs: not only it's an optimization but it also dramatically reduces noise in the result – specular has high intensity and frequency. Having specular baked in CMs isn't correct either since specular is view dependent – reflection in a mirror has different specular.

Cube-map updating

- Sky CM
  - Update every few frames (clouds, ToD)
- Geometry CMs
  - Update dynamic lighting regularly (round robin)
  - Cache G-Buffer and static lighting
  - Render new when better CM has been found

Because of dynamic time of day and moving clouds, we need to update sky CM very often (several times per second). Sun is considered dynamic light.

Active cube-maps selection

- Might differ per project
- We use 8 closest to the player, with 2 special cases
  - At least one outdoor CM
  - Penalty in vertical axis to separate floors
- Possible improvements
  - Use bounding boxes (in/out, distance)
  - Use occlusion queries
  - Pre-compute best CM set for volumes

Indoors are typically more populated with CMs, so if player is standing in front of indoor location, all 8 closest might be inside. Outdoor would have no CM at all, so we always force at least one outdoor.

Reflection rendering

Algorithm overview

Down-sample G-Buffer, apply NDF

Trace screen, output distance

Trace cube-maps, output distance & index

Resolve to color

Upscale

G-Buffer down-sampling and jittering

- Can't afford tracing at full resolution
  - Trace at half resolution
- Bilinear down-sample not recommended
  - Incorrect depth on edges
  - Lost detail in normal & roughness buffers

G-Buffer down-sampling and jittering

- Detect depth discontinuities
  - If edge is detected, discard “minor samples”
- Pick random sample (exploit temporal filter)
- Jitter normal (apply NDF)
- Output (all at half-res)
  - RT0: Depth
  - RT1: Jittered normal and roughness
  - RT2: Original normal and roughness

Random sample: we actually alternate pixels in 2x2 block

Screen-space tracing

- Trace screen-space depth
  - Output: traveled distance, “finished” flag
- Stencil mask for “finished” flag

Traveled distance:

Stencil mask (white means finished):

Best cube-map selection – CPU

Generate 8 cube-map index chains

- For each starting CM, estimate best 3 consecutive CMs
- Based on distance only
- Output: 8 4-item CM chains
- Encoded to global CB

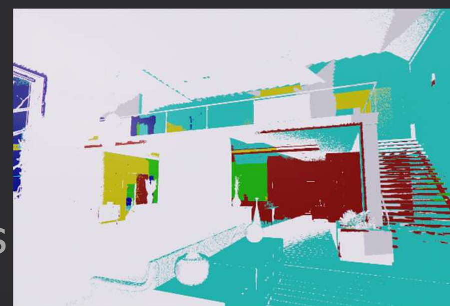
0	1	5	2
1	0	3	4
2	1	7	3
3	2	4	1
4	5	6	7
5	4	6	7
6	4	5	7
7	2	4	5

This is something to be improved. We currently only find 3 closets CMs to each CM. It doesn’t even take visibility into account.

Best cube-map selection – GPU

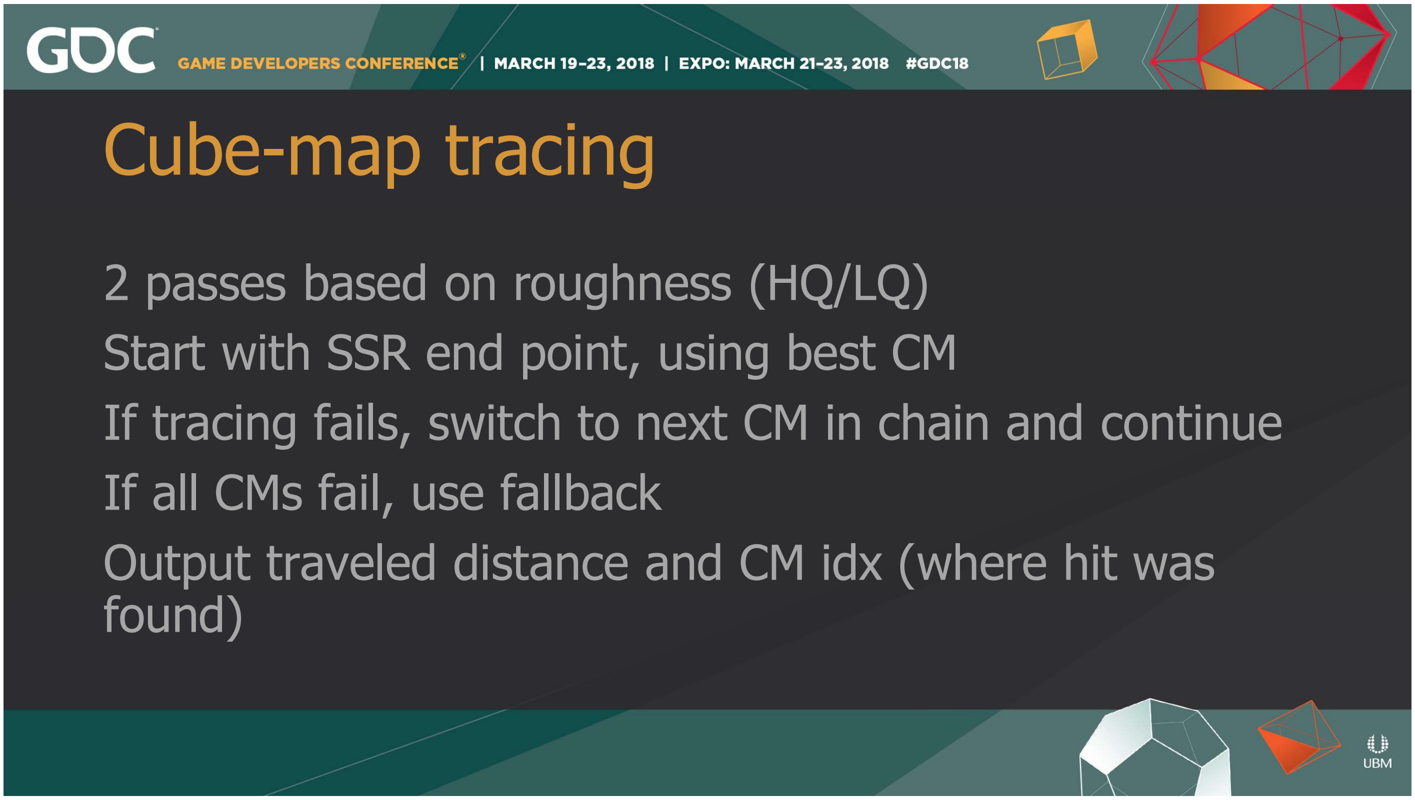
Select best starting CM per pixel

- Use stencil (unfinished pixels)
- Start at SSR end position
- Assign score to each of 8 active CMs
- Output CM index with best score



Score per pixel is assigned based on:

- Visibility (is that pixel visible from CM origin?)
- Distance from CM origin
- Ray direction vs. origin→point vector
- CM fade value (when adding/removing CM)



Roughness > 0.1: 16 steps, 100 m, scale 1.17 – 1.25, 3 refine iterations

Roughness <= 0.1: 24 steps, 300 m, scale 1.18 – 1.22, 4 refine iterations

Tracing fallback solution in Mafia III

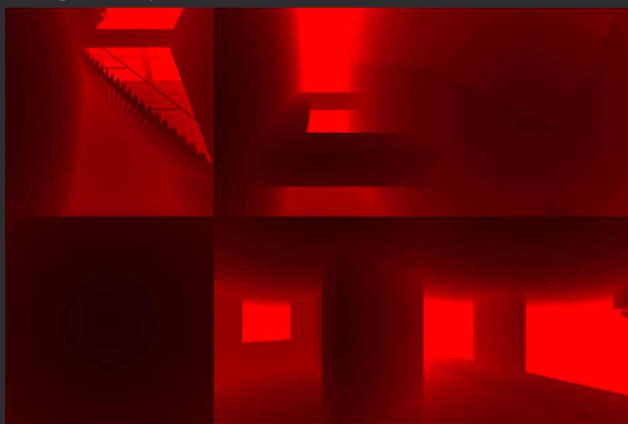
- Black reflection
  - Mostly OK
  - Really bad on very reflective surfaces (water, metals)
- Simple lookup of best CM
  - Very different results when best CM was changing
  - Eliminate popping using temporal filter

Current tracing fallback solution

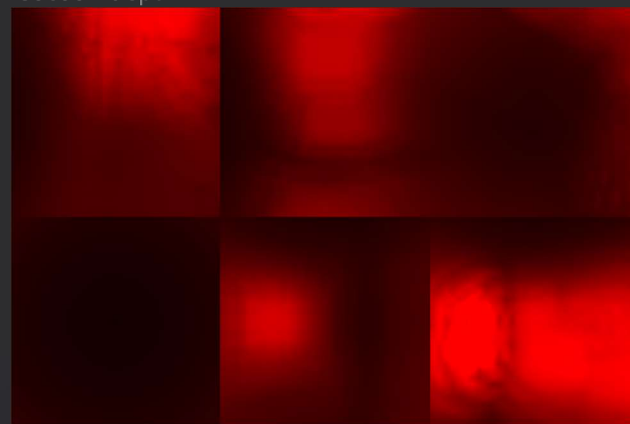
- "Cocoon" cube-map depth MIP
  - Use 1 MIP (e.g. MIP#4 – 32x32) to store very smooth approximation of space
  - Large blur kernel with MAX filter ignoring sky
  - Pushing thin geometry away
  - Removing all edges
  - Caps windows
  - Tracing never fails
  - Preserves space but removes details
  - Similar idea to parallax corrected cube-maps, automatically generated

Cocoon MIP example

Original depth:



Cocoon depth MIP:



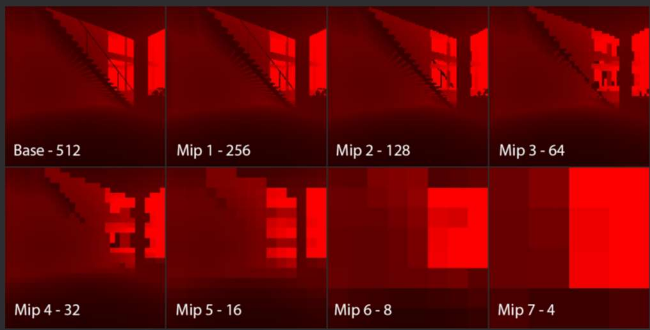
Note how the stairway, columns and flower-pot is pushed to the background but windows are still at their correct location.

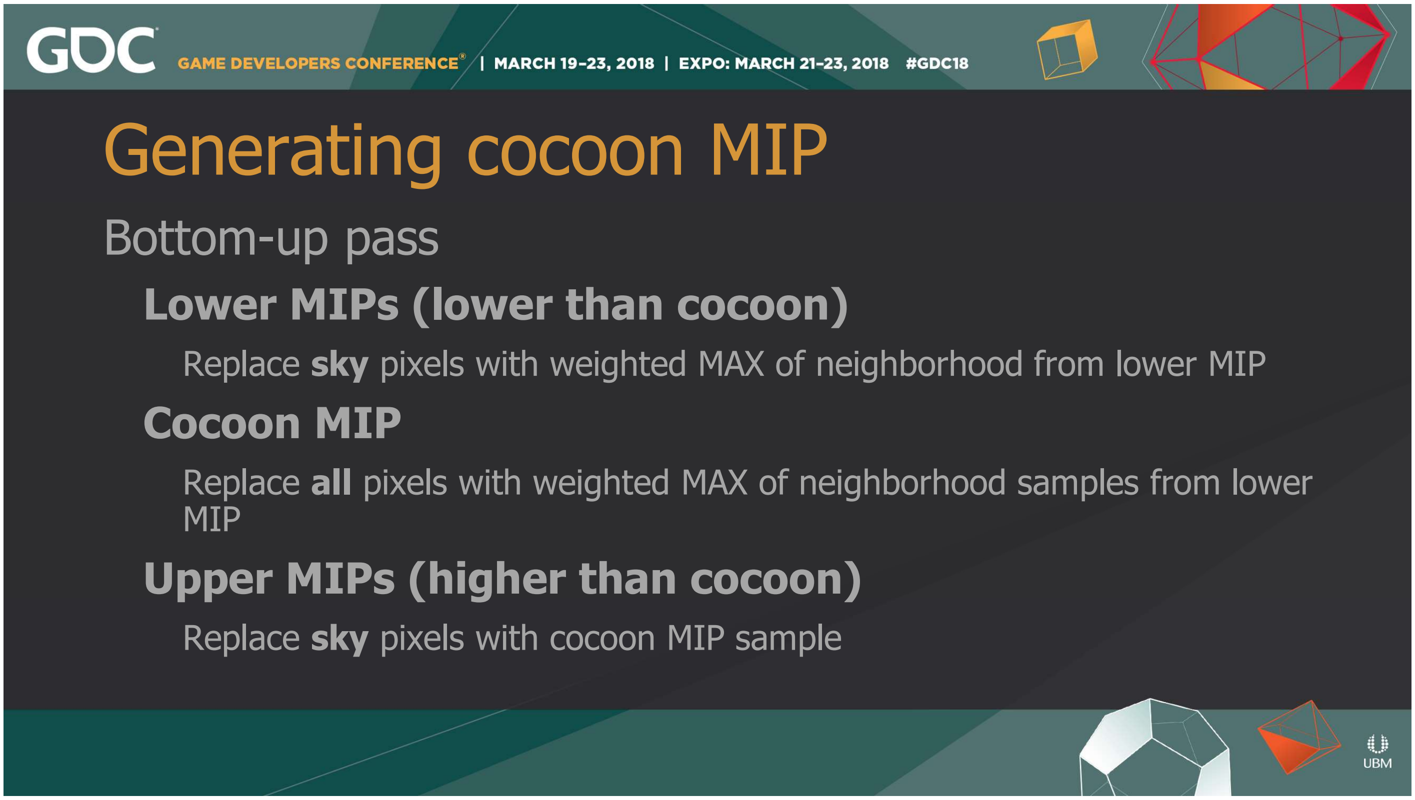
Compare to simple look-up, where the windows would be on wrong place.

Generating cocoon MIP

Top-down pass

- Build a MIP chain using MAX filter ignoring sky
- If all (4) pixels are sky, result is sky, otherwise sky pixels are discarded





Caps windows/sky – also works as an optimization for rays ending up at sky. Instead of burning all steps towards sky, ray hits the sky proxy sooner.

Weighted MAX:

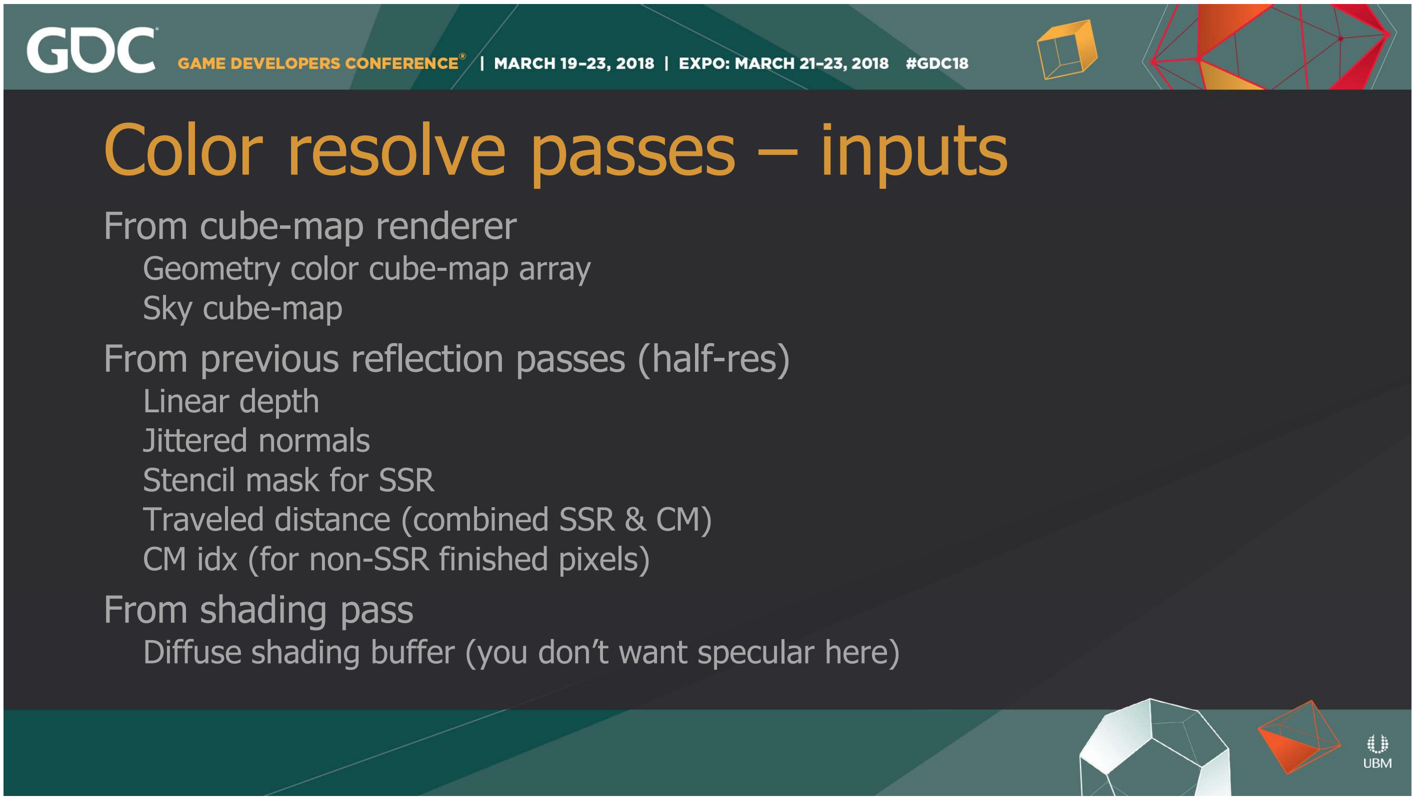
```
float pivotSample = SAMPLE_4D_LOD( srcTex, srcSampler, float4( dir, srcArrayIdx ), srcMip ).r;
float depth= 0;

for each sample
{
    float smp = SAMPLE_4D_LOD( srcTex, srcSampler, float4( vec, srcArrayIdx ), srcMip ).r;
    float weight = pow( dot( vec, dir ), specPow );
    float currVal = pivotSample + ( smp - pivotSample ) * weight;

    depth = max( depth, currVal );
}
```

Cube-map tracing optimizations

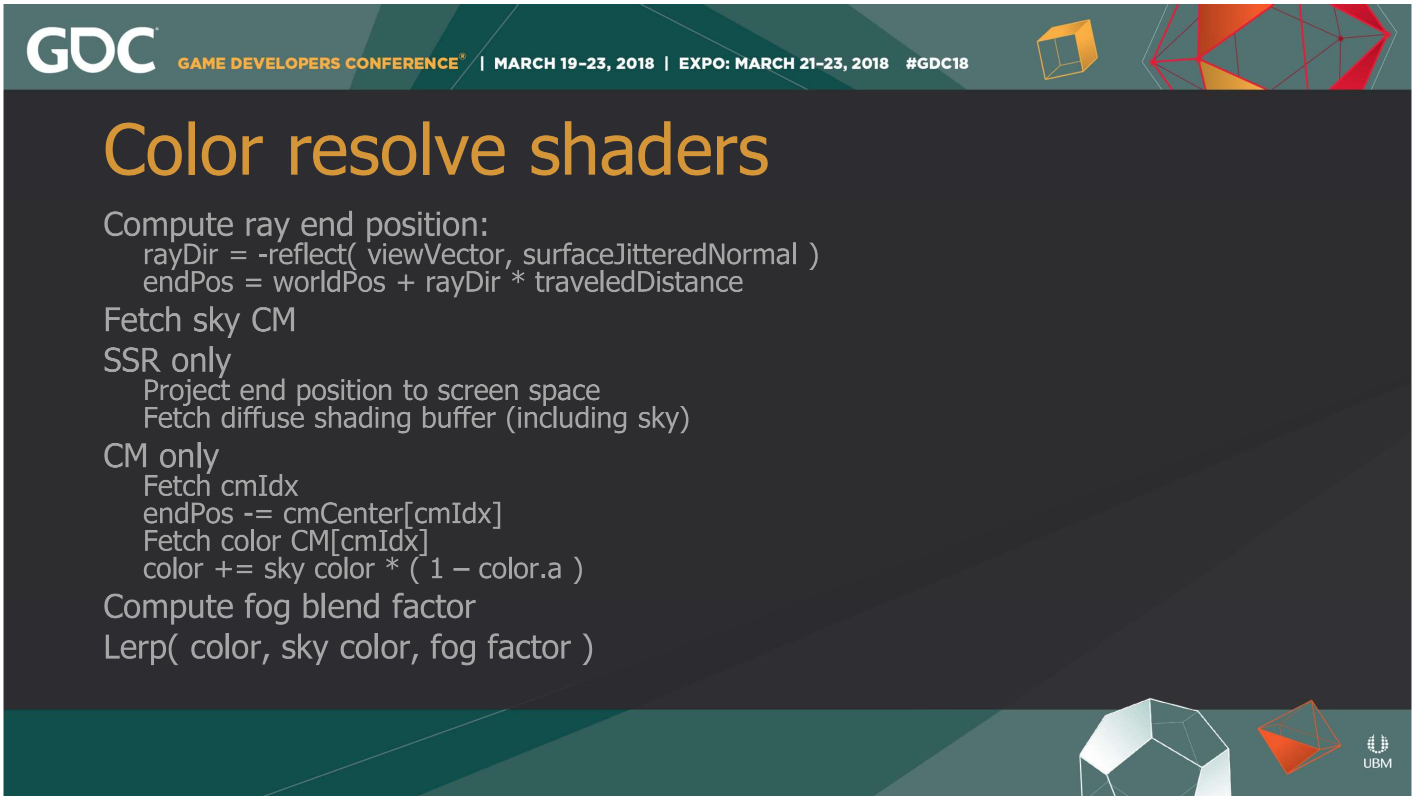
- Use lower depth MIPs for higher roughness
- Pre-compute internal volume (AABB/sphere/convex hull)
- Run as async compute shader (lose stencil)



When tracing is finished (got traveled distance, stencil mask, possibly CM index per pixel), it can be resolved to color using the mentioned inputs.

Color resolve passes

- Half-res passes
  - Resolve SSR color
  - Resolve CM color
- Full-res passes
  - Upscale half-res resolved buffer, generate low-roughness stencil mask
  - Resolve SSR on low-roughness pixels
  - Resolve CM on low-roughness pixels



A little hack to add fog to reflections (fog is included neither in CM nor is SS diffuse shading buffer): because we have volumetric fog, which is non-trivial to compute for other rays than from camera, we simply fade towards sky color – which in fact is fog integrated over long distance.

Upscale

Inputs:

- Half-res color
- Half-res unjittered normals
- Half-res depth
- Full-res normals
- Full-res depth

Outputs:

- Full-res color (high roughness pixels)
- Stencil mask

Picks 1 sample from half-res color that best matches full-res normal & depth

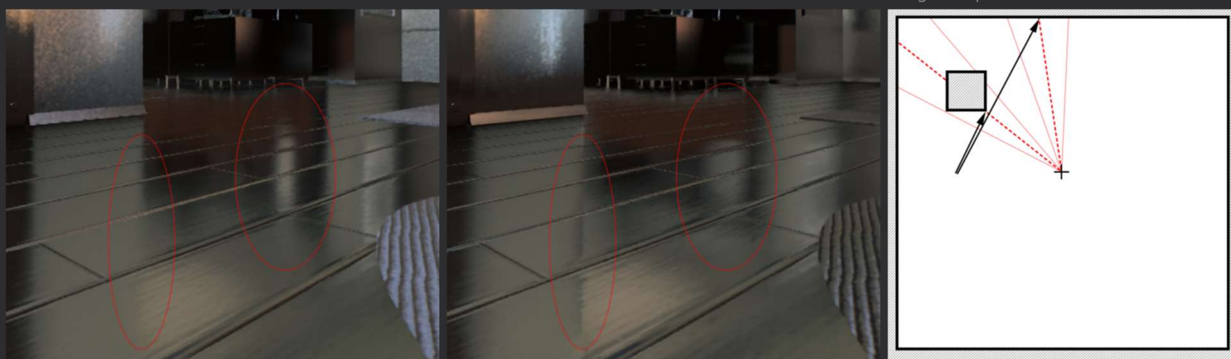
Reflections on rough surfaces

Possible approaches

Using pre-filtered MIPs

- Visible edges
- Isotropic

Reference: Pre-filtered MIPs: Edges explanation:



Check out the edge artifact and missing elongation on the left image.

Diagram shows, how two neighbor pixels rays end up in a completely different location in the CM, the results are vastly different. CM is pre-filtered from the point of view of its origin, not from the point of view of reflecting pixel.

On rough surfaces, the kernel is really large – would be very costly for real-time. That’s why MIPs are used, so the blur can’t be depth/normal/roughness aware. Note the big loss of normal map detail but also how it leaks across edges.

Possible approaches

- Importance sampling
  - Noise vs. performance
  - Need hundreds of samples to get noise-free result

1: 8: 32: 128:

We are shooting 1, 8, 32, 128 rays for every 4 pixels (still tracing at half resolution).

Mafia III approach

- Combination of screen-space blur and importance sampling
  - 50 % SS blur
  - 50 % importance sampling
  - Trade-off between leaking and noise
- Large blur kernel (up to 25 % of screen)
  - Need to use MIPs
  - Can't be depth-aware

Compute approximate reflection cone angle.

Halve the angle and jitter normal within this cone.

Output the ray traveled distance along with the reflection color.

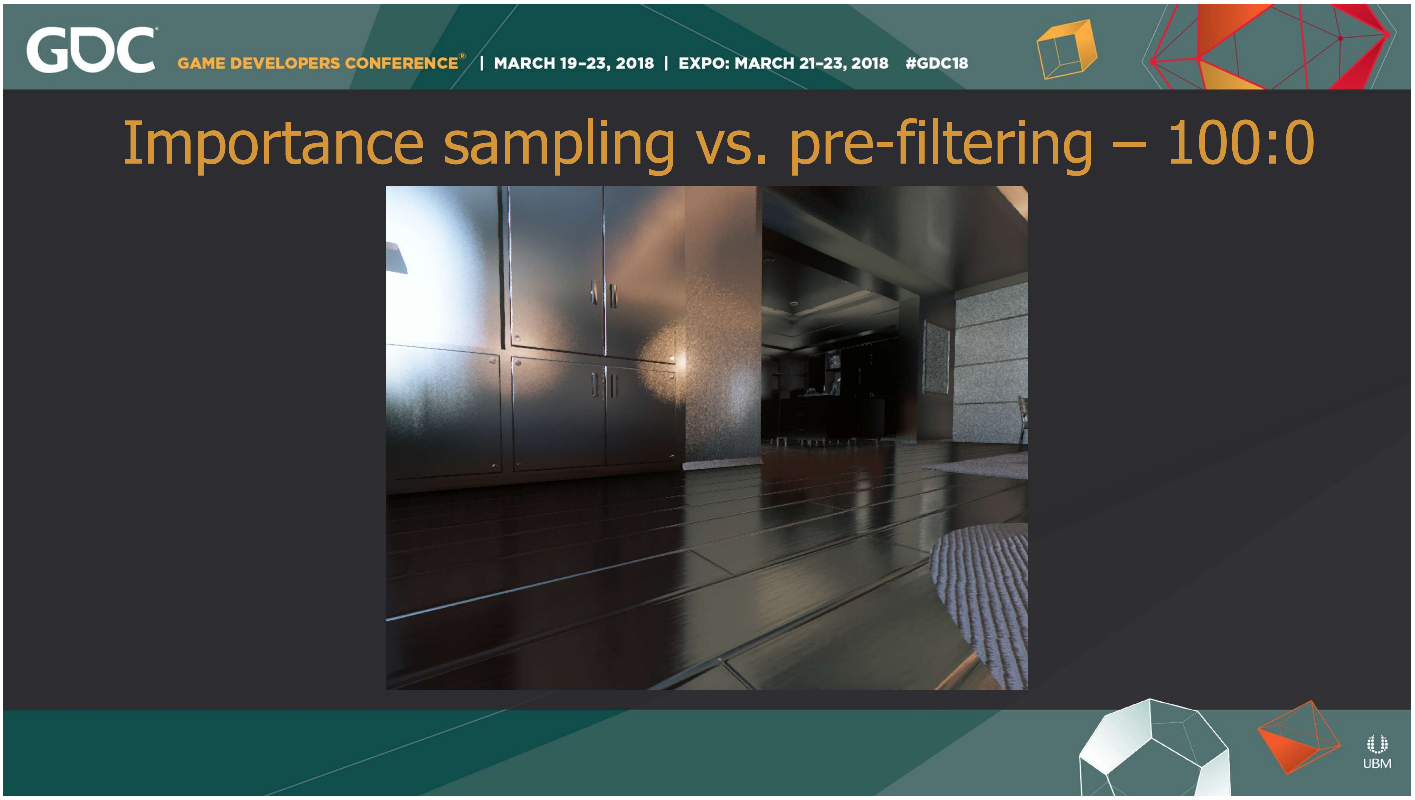
Build color MIP chain.

For each pixel, estimate the MIP level to be used, based on traveled distance.

Current approach

- Mix of all 3 + some tricks
  - 50 % importance sampling
  - 50 % using pre-filtered MIPs (both SSR and CM)
  - 5-sample BRDF-weighted screen-space blur
  - Modified sample distribution
  - Temporal filter
- Math is based on Blinn-Phong (not converted to GGX yet)

Note the leaking and loss of normal map detail.

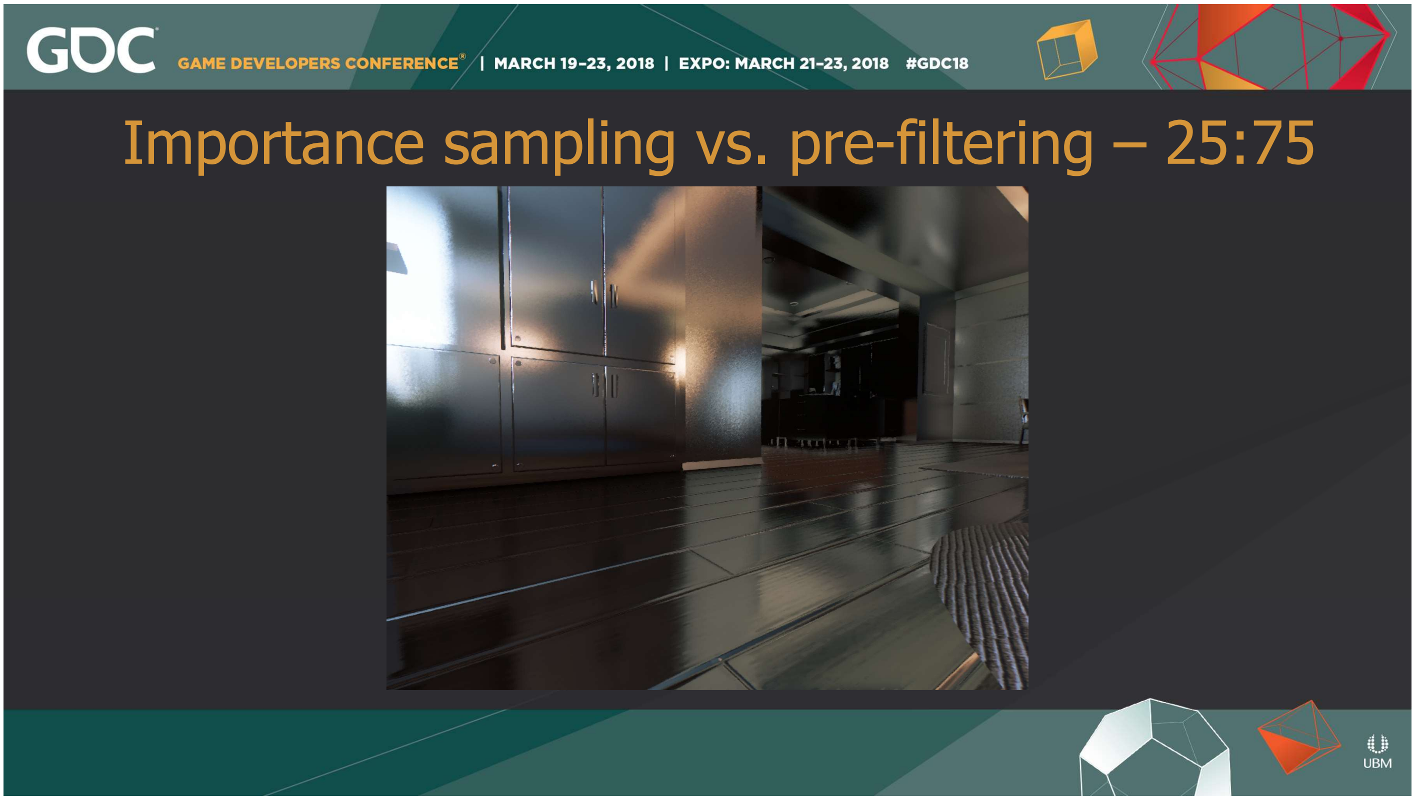


Compare several mixtures of importance sampling vs. pre-filtering.

100 % importance sampling is our reference.

Importance sampling vs. pre-filtering – 75:25

Importance sampling vs. pre-filtering – 50:50



- Lost elongation
- Visible Edges
- Less correct – some surfaces look a lot different

Combining importance sampling with pre-filtering

- NDF produces vectors with angle *[0, π/2)* from normal
- Find angle, where probability drops below threshold (in our case 0.1)
  - Ignore all vector beyond this angle
- Split angle among NDF and pre-filering
  - Modify NDF to produce vectors *[0, angle/2)*
  - Compute cone base radius and MIP level for angle/2

We lose a bit of the tail by ignoring all vectors, where “cos(angle)^specPow < 0.1” but on the other hand that helps reducing the noise quite a bit.

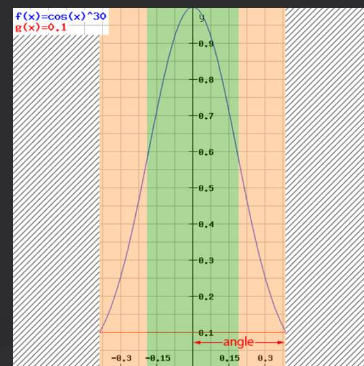
Combining importance sampling with pre-filtering

Example:

- roughness = 0.5 → specPow = 30
- angle = acos(threshold $^{1/\text{specPow}}$) = 0.387

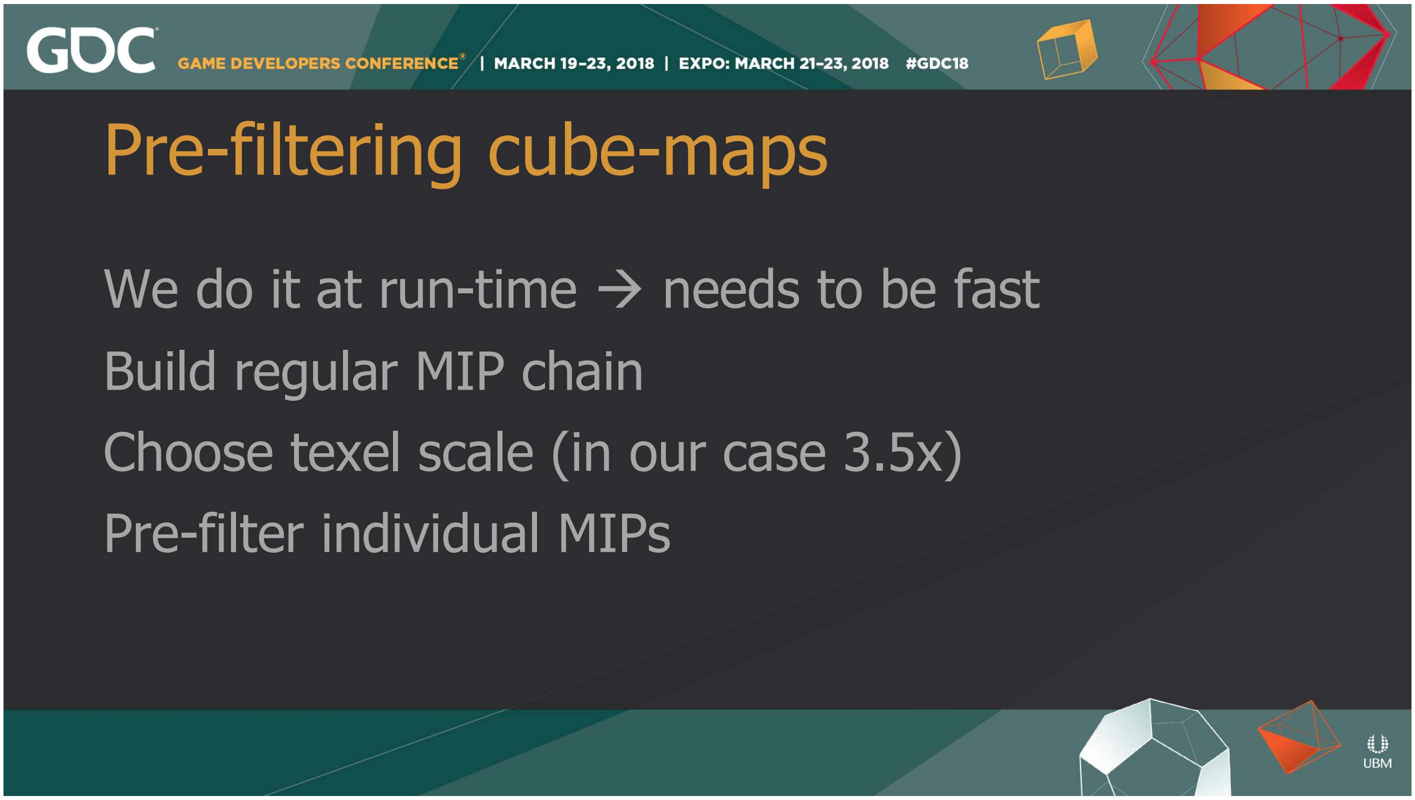
Changing importance sampling vs. pre-filtering ratio:

- More importance sampling → more noise
- More pre-filtering → less anisotropy



NDF MIP

Blue graph is target NDF. Red line is threshold (0.1). We ignore regions, where blue is below red. Compute corresponding (cone) angle. Half of the cone is delivered using NDF (green), second half using pre-filtering (yellow).



Simple English: once we know our cone angle, we find cube-map MIP, where cone base radius is texelScale texels (3.5 texels).

Setting texel scale to 1 would cause pre-filtering of only 1 texel -> no pre-filtering at all.

Setting texel scale too high would increase the cost of pre-filtering (you need to add more taps) but also force sampling of higher MIP levels, which will cost additional performance in resolve pass.

When playing with this, cross-check with reference (1000+ taps from upper MIPs or base level).

Found more advanced run-time pre-filtering later – want to have a look at that:

http://research.nvidia.com/publication/real-time-global-illumination-using-precomputed-light-field-probes

Pre-filtering cube-maps

MIP pre-filtering (in our case 29 taps):

numPixels = 2^{mipIdx} * texelScale
angle = atan(numPixels / cmSize / 2)
specPow = $\log_{\cos(\text{ angle })}$ threshold

Computing MIP level in resolve shader:

angle = AngleFromSpecPow(specPow) // see previous slides
radius = tan(angle) * traveledDist
cmRadius = radius / length(hitPosCM) / texelScale
numPixels = max(1, cmSize / 2 * cmRadius)
mipLevel = log2(numPixels)

GDC GAME DEVELOPERS CONFERENCE® | MARCH 19–23, 2018 | EXPO: MARCH 21–23, 2018 #GDC18

Modified NDF

Input: 2 random values *[0, 1)*, uniform distribution

Default Phong distribution:

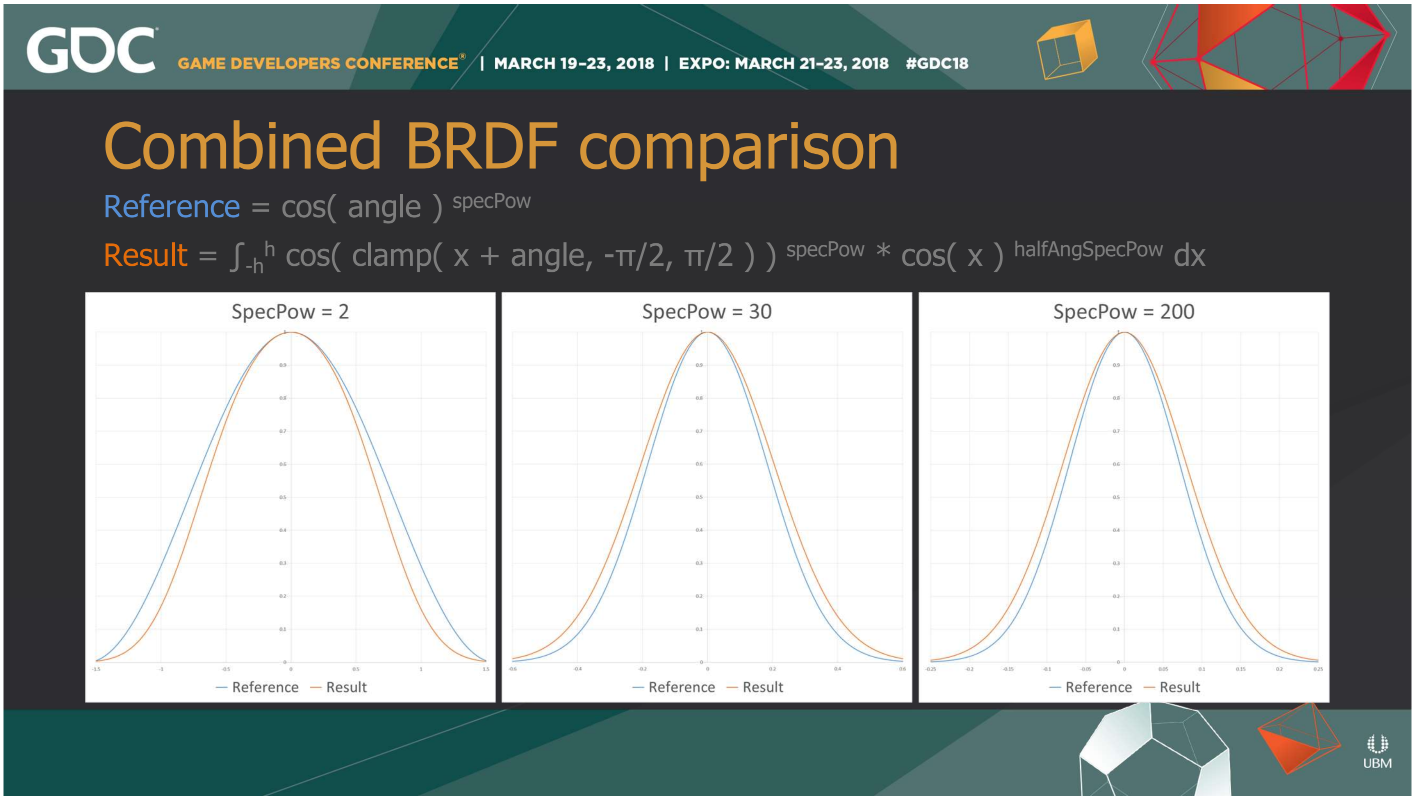
- $\theta = \text{acos}(rnd^{1/(specPow+1)})$
- $\varphi = 2 * \pi * rnd2$

Half-angle:

- halfAngle = 0.5 * AngleFromSpecPow(specPow)
- $minRnd = \cos(halfAngle)^{specPow + 1}$
- $\theta = \text{acos}((minRnd + (1-minRnd)*rnd)^{1/(specPow+1)})$

UBM

We don’t care about the PHI angle for now but want to modify THETA, to get only angle/2 instead of angle. We inverse the function, find minimum random value and then scale the input random value to be in range [minRnd,1). Don’t clamp the value, it needs to be linear operation to preserver the relative probabilities.



"Result" is what you get, if you modify NDF to half angle and sample MIP corresponding to half-angle.

h – half-angle

halfAngSpecPow – specular power corresponging to half-angle

angle = acos(threshold $^{1/specPow}$)

halfAngSpecPow = $\log_{\cos(0.5 * angle)}$ threshold

It's not 100 % the same but it's pretty close

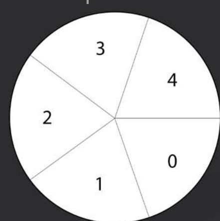
Modified NDF

- Concentrate as much variance as possible to neighborhood
  - The best pattern we found was a "+" pattern – assign each pixel a value of 0-4
  - Every pixel has all 5 "classes" around that it can sample in blur pass
- Map class ID to ray direction
  - $\varphi = 2 * \pi * (0.2 * rnd2 + GetSSJitterPlus(ssPos, frameCounter))$
- Shuffle temporarily

SS pixels:

0	1	2	3	4	0	1
2	3	4	0	1	2	3
4	0	1	2	3	4	0
1	2	3	4	0	1	2
3	4	0	1	2	3	4
0	1	2	3	4	0	1

Hemisphere slices:



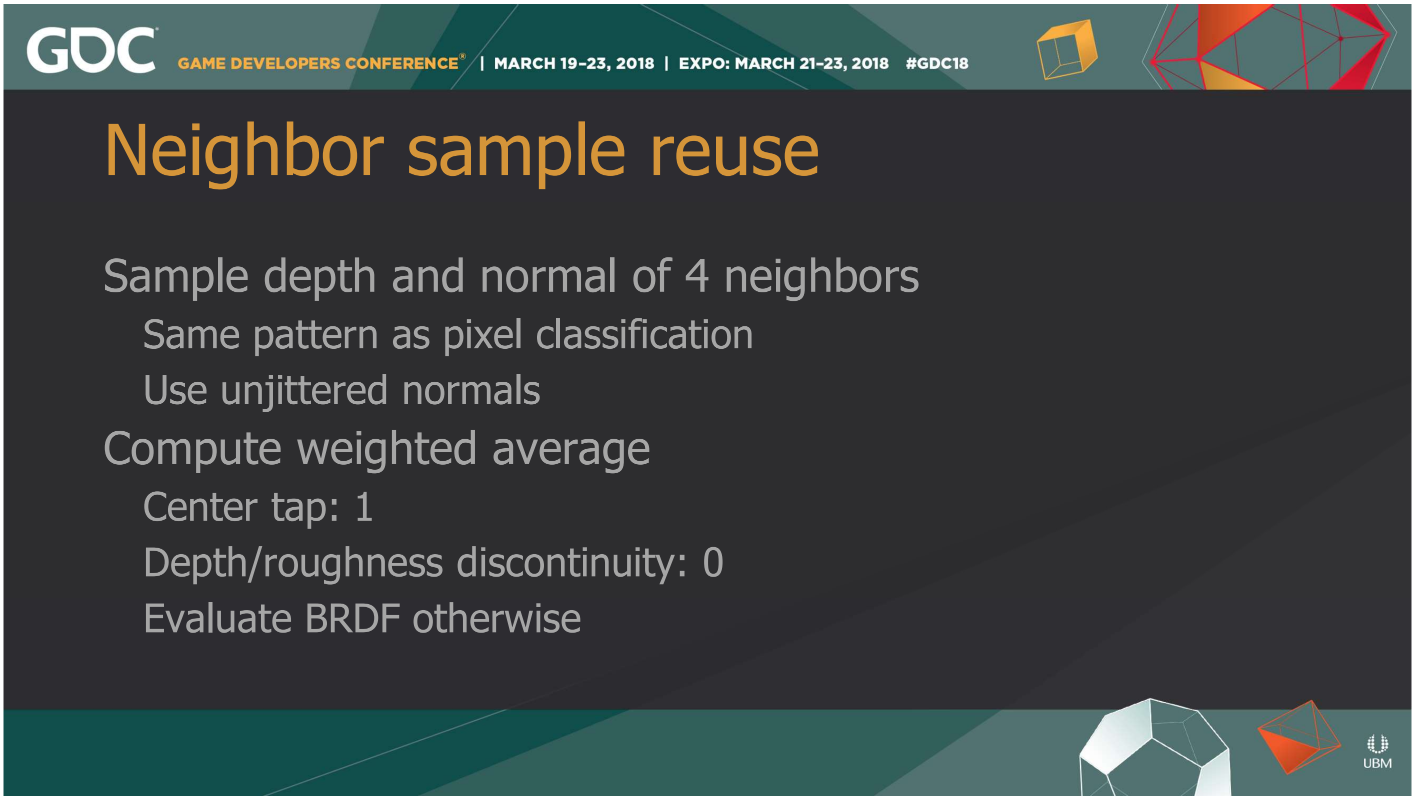
Pixel class ID from screen-space position and frame ID:

```
float GetSSJitterPlus( in const uint2 ssPos, in const uint txaaFrameCounter )
{
    const uint  SAMPLES_Y_OFFSET = 2;
    const uint  SAMPLES_COUNT = 5;
    const uint  sampleIdx = ( ssPos.x + SAMPLES_Y_OFFSET * ssPos.y + txaaFrameCounter ) % SAMPLES_COUNT;

    return 1.0 / SAMPLES_COUNT * sampleIdx;
}
```


2nd modification of NDF is to concentrate color variance to a small neighborhood, to be able to blur that in SS blur pass and remove the noise. The assumption is that rays going in similar direction are more likely to result in similar color and vice versa. Focus direction variance to neighbor pixels. We found that shifting "+" pattern works pretty well for this purpose.

Blue noise might be a good alternative. Will try that later and compare the results.



If all the pixels have the same roughness and normal (flat, rough surface), you can look at it as multiple (temporal) samples. Just average them (assuming there is no discontinuity).

If roughness is very different, we haven't found a way, how to combine these samples.

With changing normals, the BRDF using unjittered normal seems to be a good metric.

For very small roughness, we would have to consider also view vector divergence between neighbor pixels. Instead of that (extra cycles), we simply fade this blur out.

Temporal filter

- We use up to 15:1 previous frame blend ratio
- Reflections view dependent
  - Compute view vector divergence (previous vs. current frame)
  - Compute divergence threshold based on roughness
    - Mirror reflections: zero divergence threshold but no issues with noise!
    - Rough reflections: high divergence threshold but not so much view dependency!
- Invisible in last frame (or discarded due to divergence)
  - Evaluate extra 4 samples in centers of neighbor "+" elements
  - Effectively up to 25 samples (5x5)

We use variance clamping for mirror reflections (roughness = 0) and we gradually increase the clamping window with growing roughness. Variance clamping is fully disabled when roughness > 0.1.

Extra 4 samples: look at it as separable blur. But instead of 2-pass horizontal/vertical, we do "+" and tilted "x" that is sampling the neighbor "+" centers.

Step-by-step recap – tracing

Down-sample G-Buffer depth, normal (add jitter), roughness to half-res buffers

Stencil mask based on roughness (different tracing quality for high/low roughness)

2-pass (high/low roughness) SSR trace outputting traveled distance and FIN flag

Stencil mask for SSR finished pixels

Best CM select

2-pass (high/low roughness) CM trace outputting traveled distance and CM idx

Step-by-step recap – post-tracing

Resolve to color (SSR + CM)

Neighbor sample reuse (screen-space blur)

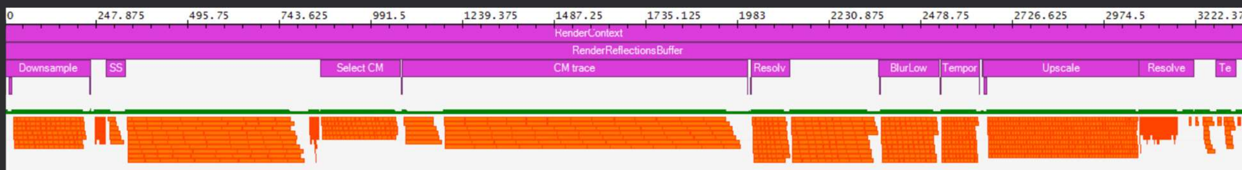
Temporal filter for high roughness

Depth & normal aware upscale to full res

Resolve low roughness at full res (using half res traveled distance)

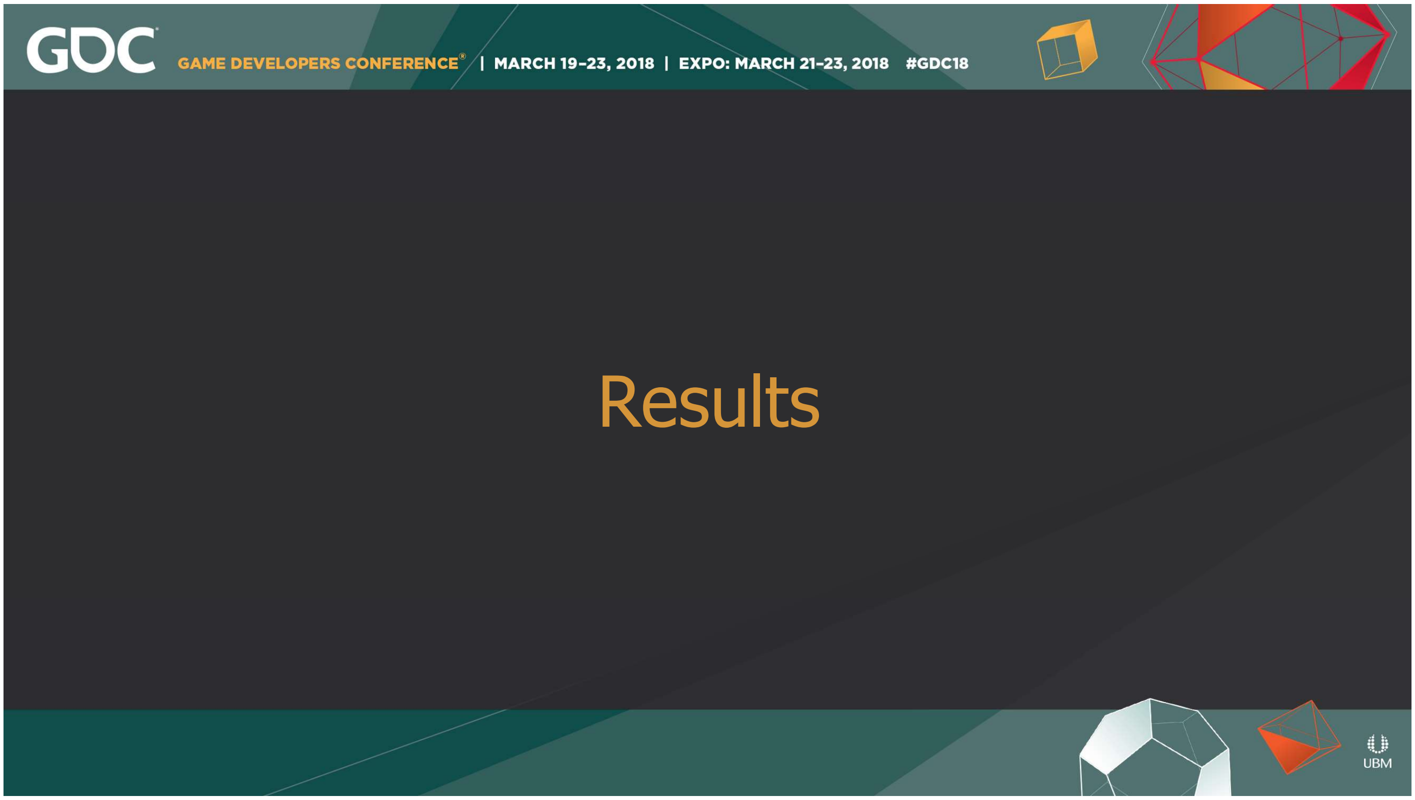
Temporal filter for low roughness (with variance clamping)

Timings (1080p @ PS4)



Down-sample G-Buffer	0.25
SSR trace	0.55
Select best starting CM	0.25
CM trace	0.9
Half-res resolve	0.35
SSR	0.1
CM	0.25
Half-res blur	0.17
Half-res temporal	0.1
Upscale	0.41
Resolve	0.22
Temporal	0.1
Sum	3.3 ms

Captured before porting to async CS. Slightly above budget of 3.0 ms.



All the screenshots have been captured using Mafia III assets and the new tech.

Note that the new tech has NOT been shipped in Mafia III.


BIG BREAK
So light and mild
what a relief!


Bourbon City Jewelers

Conclusion

- Stable reflections when camera/dynamic objects move
- Reasonable amount of manual work
- Little pre-compute (max view distance, inner volume)
- Real-time on nowadays gaming hardware
- Scalable in terms of:
  - Lighting changes: re-light cube-maps
  - Geometry changes (destruction): re-render affected cube-maps
  - Scene complexity: adjust amount of cube-maps

Future work

Convert to GGX

Temporal re-projection using reflection depth

Improve upscaling pass

Pre-compute optimal starting CM and chain

Investigate automatic probe placement

Investigate better handling of off-screen dynamic objects

Thanks

Petr Smílek
First implementation

Naty Hoffman
Consulting

Rinaldo Tjan
Testing and feedbacking

Tianli Bi
Optimizations

Jiří Štempin
Code support

Eva Tajovská
Help with presentation

Jan Marvánek
Help with presentation

Radim Doleček
Help with presentation

Petr Záveský
Help with presentation

Sebastien Lagarde
Proof review

References

Umenhoffer, Patow, Szirmay-Kalos. 2007. GPU Gems 3 – Chapter 17. Robust Multiple Specular Reflections and Refractions
https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch17.html

Stachowiak. SIGGRAPH2015. Stochastic Screen-Space Reflections
http://advances.realtimerendering.com/s2015/Stochastic%20Screen-Space%20Reflections.pptx

Robinson, Shirley. 2009. Image-Space Gathering
http://www.nvidia.com/object/nvidia_research_pub_015.html

Valient. GDC2014. Taking Killzone Shadow Fall Image Quality into the Next Generation
https://www.guerrilla-games.com/read/taking-killzone-shadow-fall-image-quality-into-the-next-generation-1

Lagarde. SIGGRAPH2012. Parallax Corrected Cube-Maps
https://seblagarde.files.wordpress.com/2012/08/parallax_corrected_cubemap-siggraph2012.pdf

Bjorke. 2004. GPU Gems – Chapter 19. Image-Based Lighting
http://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch19.html

Czuba. 2011. Box Projected Cube Environment Mapping
https://blenderartists.org/forum/archive/index.php/t-209688.html

Manson, Sloan. 2016. Fast Filtering of Reflection Probes
https://www.ppsloan.org/publications/ggx_filtering.pdf

McGuire, Mara, Nowrouzezahrai, Luebke. 2017. Real-Time Global Illumination using Precomputed Light Field Probes
http://research.nvidia.com/publication/real-time-global-illumination-using-precomputed-light-field-probes

.....questions?

Martin Sobek
martin.sobek@2kgames.com
https://hangar13games.com/

Bonus slides

GDC GAME DEVELOPERS CONFERENCE® | MARCH 19–23, 2018 | EXPO: MARCH 21–23, 2018 #GDC18

Cube-map tracing pseudo-code

```
stepScale = Rand( stepScaleMin, stepScaleMax )
currStep = ComputeInitialStep( maxRayLength, stepScale )
bestCMIdx = FetchBestCMIdx
currCMIdx = bestCMIdx
usedCMs = 0
currPos -= cmCenter[currCMIdx] // currPos is always in CM space
for each step
    currPos += currStep
    cmDepth = FetchCMDepth( bilinear, currPos )
    cmDepthPoint = FetchCMDepth( point, currPos )
    cmDepth = clamp( cmDepth, cmDepthPoint – threshold, cmDepthPont + threshold )
    cmDist = length( currPos ) // Note: sqrt can be avoided
    if cmDist > AddBias( cmDepth )
        if ComputeMassDepth( currPos, currStep ) + cmDepth > cmDist
            // Hit has been found
            numRefineSteps++
            if numRefineSteps >= maxRefineSteps
                success = true
                break
            else
                currPos -= currStep
                currStep *= 0.5
        else
            // Entered shadow region → need to switch CM
            usedCMs++
            if usedCMs >= maxTracedCMs
                // Tracing failed, need fallback
                success = false
                Break
            else
                currPos += cmCenter[currCMIdx]
                currCMIdx = cmOrder[bestCMIdx][usedCMs]
                currPos -= cmCenter[currCMIdx]
    currStep *= stepScale
if success
    Secant refine
else
    Use fallback solution

Output:
    RT0: length( currPos + cmCenter[currCMIdx] – rayStartPos )
    RT1: currCMIdx
```

UBM

CM depth texture contains distance from CM origin instead of linear depth

- Simpler math
- Eliminate pre-filtering issues on CM edges