“My old dream was to see global illumination in an interactive application, which doesn't depend on any precomputation and works with 100% dynamic lighting conditions and a similarly dynamic environment.”

Oles Shyshkovtsov, 4A Games (GPU Gems 2, 2005)
AGENDA

1. Introduction
2. Implementation
3. Denosing
4. Artist point of view
INTRODUCTION
The Quest for the Holy Grail
MOTIVATION

RTX OFF
MOTIVATION

RTX ON
THE QUEST BEGINS
Know what you want to achieve

Everything?
Ok, fine. Global Illumination!
Ok, fine. A hybrid, indirect first-bounce, diffuse, Global Illumination and Deferred rendering pipeline.
Right. Off you go.
A standard deferred renderer

Calculate G-Buffer and lighting buffers, accumulate light and effects, TAA, post-process

Relies heavily on stochastic methods followed by TAA to reduce noise
The raytraced elements merge nicely with standard deferred renderer.

Added buffer to cache raytrace data for use in RTAO and RTGI passes.

Stochastic ray generation: Image is figuratively shouting at us, so a good denoiser is critical.
STOCHASTIC METHOD

Why do it 1000 times when once will do?

Monte Carlo Integration: the approximation of large data sets with few samples
   Rather than adding up every single ray/photon, pick a few as “representative”
   You may lose some specific details, but will get the big picture
   Desirable for a GI solution
   Results get to the point of diminishing returns at a tiny fraction of the total set

This doesn’t just apply to raytracing
   Shadows, volumetrics, reflections, hair
   Apply to any suitably complex effect
STOCHASTIC METHOD

The importance of samples

Beware of noise and aliasing, both are issues, aliasing is worse

You are going to produce a noisy data set, and you will run denoisers

Jitter, Importance sampling and Probability Density Function (PDF) provide leverage over sample distribution

Output data is buffered for analysis, filtering and use later on

Produces good general purpose (input agnostic) data on scene illumination

If your image is still noisy at the end of the frame (it will be) add TAA
STOCHASTIC METHOD
Know your noise

Noise breaks up patterns when sampling below input frequency
Must be repeatable, it is used later for re-construction of the hit location from stored distance value
Temporally and spatially uniform to avoid “clumping” and “swimming”
Sample small blue noise texture across the screen, oscillate across frames

```cpp
// Sample noise from screen position and frame index
float2 uv = t_blue_noise_64.Load(uint4((pixID.xy+32*(frmID&1))&63, frmID&63, 0)).xy;

// Generate ray on hemisphere
float3 vRay = HemisphereSample(uv);

// Transform to local space of the surface using surface normal
float3 T, B;
BasisFromDirection(N, T, B);
return normalize(FromLocal(ray, T, B, N));
```
RTAO
Why?

AO is a poor-man replacement for GI. We are doing real GI already so why bother?

We are running hybrid pipeline, which is smoothly blended into “old” pipeline

250m transition from foreground RTGI to “regular” pipeline

Regular pipeline expects AO available at different stages

All image-based lighting (light-probes) are directly multiplied by AO

Some “fake” lights use AO as their shadow approximation. Shame on us :)

Even sun shadow-map blends into AO at some distance

Searching for usage in shader-code finds 79 places…

Also, it’s cheap and helps guide the denoiser! :)
SSAO TO RTAO
Reuse and Improve

SSAO
Captures nearby details

RTAO
Recognise enclosed space

SSAO
Misses interior occlusion

RTAO
Progressively darkens
VOXEL-BASED GI TO RTGI

Night vs Day

Voxel GI
Broad directional light, insufficient detail for shadows

RTGI
Light bounce and contact shadows from nearby objects

Voxel GI
No sense of depth

RTGI
Gradual self-occlusion on object interiors
PUTTING IT ALL TOGETHER

Ok. Now it works. Just...

RTX fitted in well with 4A engine

The game was balanced for the traditional pipeline, but RTX walked in made it its own

We want more “rays”: We generate as few as possible for performance, but we can always find as use for more them

Lots of options for the future...
IMPLEMENTATION
It WILL just work, if you work at it
IMPLEMENTATION (1)

Remove unnecessary pipeline stages

RSM rendering (replaced with cheaper depth-only shadow-map rendering)
Geometric ESM-AO (approximation of 16 rays)
SH-voxel-grid computation/gather
SH-voxel-grid temporal blending
SH-voxel-grid screen-space resolve
IMPLEMENTATION (2)
Modify some pipeline stages

SSAO-pass now computes accumulation weights and accumulates raytraced AO
  Velocity, depth disocclusion, etc.
  Weights used for both AO accumulation and GI

AO-filter pass
  Before:
    SSAO filtering, geometric ESM-AO sampling
    Blending with terrain AO, precomputed AO maps, per-vertex AO
  Now:
    Denoising and RTAO accumulation
IMPLEMENTATION (3)

Add new pipeline stages

Raytracing + screen-space pre-tracing
Geometry skinning and animation
Albedo updates/management
BLAS updates
TLAS rebuilds
Deferred shading of hit-positions
Denoising & accumulation
RT MODULE
Separate mini-engine from the rest of the pipeline

Handles skinning and geometric animation
Handles all BLAS updates/TLAS rebuilds
Separate instance-culling (expanded frustum, contribution)
Instance transforms, logical/game visibility
Separate memory manager
Separate command lists

Just 3 .cpp files, ~1500 lines DXR API, ~1100 lines logic, ~200 lines “glue”
BLAS = update only for skinned/animated instances; TLAS = rebuild only from scratch

- TLAS quality and compactness is extremely important
- TLAS selects those which are inside expanded frustum (+logical visibility, + contribution culling)
- Usually we have more than 100k potentially active instances; less than 5k will survive the culling

Relatively fast, but each update/rebuild is multi-pass, under utilizes GPU

Hide with async-compute!

- We hide it with pre-trace CS and SSR CS
- Alternatively run it from compute queue parallel to the gbuffer rendering
- We have both modes implemented, statistically insignificant perf difference
Every entity update increases priority of RT-instances

  Visible = higher priority, small and/or distant = lower priority

Sort instances based on accumulated (across frames) priority

Select a few (16 in our case) with highest priority

Select a few (4 in our case) randomly from the remaining set with non-zero priorities

  High priority objects should not block other stuff updating!

  Shrinks queue to "balanced" state in a matter of seconds

  "Balanced" state is just 5k-6k instances "outdated" :) out of 20k+

Additionally limit the vertex count as well

  Necessary to avoids rare "spikes" in processing
METRO IS EXTREMELY GEOMETRY HEAVY
20,000,000 polygons is a lot to render just for the sun shadow

Depth impostor cache / Simplified IB (separate position-only VB if shader allows)
Reuse those simplified "shadow" meshes for RT!

Result: BLAS meshes are about 4x smaller than the “real” geometry
- There are scenes where it translates into 30% perf gain in raytracing
- All vertex animation and skinning become cheaper
- Memory usage: ~1GB instead of ~4GB
- Zero or close to it difference in quality!
Shoot rays at every pixel in all directions (ok, according to BRDF lobe)
Gather lighting at the contact point; multiplied by albedo of that point
Accumulate that!
Hit distance gives us "free" RTAO
PIPELINE STAGES
Raytracing Specific GPU Pipeline

Screen-space pre-trace + all actual raytracing
Ambient Occlusion + Filtering
Global Illumination + Two pass denoiser
Initial implementation took around one person-month here.
PRE-TRACE
Ray tracing in screen space

Exactly the same ray-generation as the real raytrace
Ray-march against depth buffer
Runs as async-compute, parallel to BVH updates/rebuilds

Fixes missing "alpha-tested" geometry in most cases
   We aggressively filter it out whenever we can
Almost constant distance in screen-space (cache-friendly)
Outputs into UAV hit-distance and albedo (from g-buffer)
RAYTRACING

Real rays!

Only spawn the real ray if pre-trace failed to find intersection

  Leads to a small perf-boost

Ray-marches terrain’s heightmap inside the "raygen" shader

  Limit ray distance if intersection is found
  Almost free here (if done carefully) due to GPU latency hiding

Extremely simple pipeline config

  Only shader("closesthit") is necessary for us to get hit results
  Payload is a single UINT

Outputs to the same UAV, distance + albedo (packed into a single UINT)

  Needs to be careful with precision and tolerances
  Floating point precision hit us several times
DEFERRED LIGHTING
Hit-positions processing

Run exactly the same ray generation as in main trace
Reconstruct hit position (or indication of "miss") and albedo

MISS = sample skybox
HIT = compute lighting

Encode information, more on that later
Accumulate with history
DEFERRED LIGHTING
Why only the sun/moon and sky?

Tech stabilized quite late in the development cycle (late Q4/2018)

Content was mostly done and locked in at the time

Implemented 1st bounce contribution from all lights, out of curiosity
  - Lighting already computed in a deferred way? use it
  - In frustum, but occluded? Use precomputed lighting from atmosphere
  - Out of frustum - run real computation

Extremely cheap (~0.2ms on an RTX 2080ti), could be a big perf-boost if we managed to remove AO/IBL, but...
  - It conflicts with hand-crafted lighting and visuals :
  - It breaks the game, especially the stealth mechanic

Simply put: we were out of time to fix current content across the huge game
COLOR TRANSPORT

Where to get albedo for hit results?

Color bleeding is mostly visible on close to contact surfaces
  Usually those are found by initial screen space pre-trace
  Just sample albedo from gbuffer

Integration across the whole hemisphere is a low-pass filter in essence
It is a good idea to pre-filter signal to lower denoiser’s input noise level
We do that pre-filtering extremely aggressively - we store average albedo per-instance :) 
  Low input noise and extremely fast :)

COLOR TRANSPORT
COLOR TRANSPORT
Where to get albedo for hit results?

G-buffer (the pre-trace samples this)
Per-instance albedo (raytracing samples this)
COLOR TRANSPORT

A few problems

Usually average albedo color pre-calculated per-texture suffices

What to do with metals? Theirs albedo is essentially zero...

Solution: Albedo * (1 - F0) + F0

What if complex shading changes visible albedo?

Or maybe it is texture-atlas and average doesn’t make sense?

Solution: pre-render that exact combination of mesh-shader-textures-params!

Then average visible albedo from 6 directions

Store into sparse database/hash table

Still allow artists to “override” it

Database shipped in the first “hotfix”
Color bleeding - RTX ON
Decompose HDR-RGB into Y and CoCg

Encode Y as L1 spherical harmonics (world space), leave CoCg as scalars

- Human eye more sensitive to intensity, not color
- 4xFP16 for Y
- 2xFP16 for CoCg
- 96 bits per pixel in total

All the accumulation and denoising happens in this space
Denoisers could go really wide under certain conditions

- Loss of normal-map details
- Loss of "contact" details and general blurriness
- Loss of denoising quality if we weight heavily against normals of samples, less information could be "reused"

96 bits? Why not less?

- Tried to reduce it down to 64 bits - failed

Mostly because of "recurrent" nature of denoisers which could be extremely aggressive on temporal accumulation and thus precision

In case of LDR, Y would be in range of [0..1] and CoCg in [-1..1], in our case it is actually in [0..HDR] and [-HDR..+HDR]
This encoding is actually a low order approximation of cubemap.

But at each individual pixel!

This allows us to reconstruct indirect specular!

Crucial for metals where albedo is zero or close to it.

SPECULAR!

Important for PBR materials consistency

( Illustration from paper "Stupid Spherical Harmonics (SH) Tricks" by Peter Pike Sloan )
Resolve SH as usual against pixel's BRDF to get diffuse

Extract dominant direction out of SH

Compute SH degradation into non-directional/ambient SH
  
  If SH is non-directional - it means incoming light is uniform over hemisphere
  
  And if it is uniform - that’s the same as if material is “rough” -> recompute new roughness

Run regular GGX with (extracted_direction, recomputed_roughness)
SPECULAR GI ON!

Yay \( \text{•□•} / \)
THE POWER OF PIPELINE

Details

The BRDF importance sampling doesn't care what to integrate at all, it is "unbiased" in that sense.

Be it 1st, 2nd or 3rd bounce indirect lighting or "direct" lighting or whatever.

What if we put something emissive in the scene?

DEMO TIME!
Yes, that's arbitrary shaped and textured polygonal lights.

I saw a lot of research on that...

But nobody does shadows, right? 😃

It is free!
WRAPPING THINGS UP
“Holy Grail” cracked!

Game-scale realtime 1st bounce indirect lighting from any analytic light

Not limited to 1st bounce at all, but... Xms trace Yms light per bounce

Even 2nd bounce gives diminishing returns compared to cost

Direct lighting and shadowing from arbitrary shaped polygonal area lights

Or sky, or whatever... Artistic freedom...

Computes both diffuse BRDF (Disney) and specular BRDF (GGX)

Everything is fully dynamic, both the geometry and lighting (no precomputation!)

In fact 4A-Engine doesn’t really have a concept of something static (prebaked)

Massive scenes

~150 000 000 triangles on a typical Metro level in TLAS before culling
DENOISING
Trapping the beast in 15 mins
DENOISING

What is it?

Denoising (or noise reduction) is the process of removing noise from a signal. Can be convolution or Deep Learning based. DL-based solution is barely explored in real-time graphics. Our approach is convolution-based and has spatial and temporal components.
EXAMPLE
Denoised vs Noisy input
EXAMPLE

Noisy input vs Denoised
DENOISING IS NOT A FUN...

...but casting rays is :)

Keeps you sad - IQ is always lower than it needs to be

Friendship is very fragile - a small change can ruin IQ completely

Small gifts don’t help - tiny tunings here and there turn the algorithm into Frankenstein’s creature

Demands too much of attention - single pass denoising works badly or inefficiently
DENOISING

Problem decomposition

Spatial component:

Sampling space, distribution and radius?

Sample weight?

Number of samples?

Temporal component:

Feedback link or links?

Feedback strength and ghosting?
DENOISING: SPATIAL COMPONENT (1)

As a single-pass blur

Take a lot of samples around current pixel

Accumulate weighted sum

The weight depends on the signal type (AO or GI, reflections, shadows)

Same as Monte Carlo integration:

\[ \int f(x) \, dx \approx \frac{1}{N} \sum_{n=1}^{N} f(x_i) \]

- Final reconstructed signal (GI, AO)
- Weighted sum (N samples)
- \( f(x) \) - noisy input
DENOISING: SPATIAL COMPONENT (2)

Screen- vs world- space sampling

Screen space problems:
- thin objects
- surfaces at glancing angle
- lots of samples are wasted due to anisotropy caused by perspective
DENOISING: SPATIAL COMPONENT (3)

Importance sampling

$$\int f(x) \, dx \approx \frac{1}{N} \sum_{n=1}^{N} \frac{f(x_i)}{p(x_i)}$$

Final reconstructed signal (GI, AO)

Weighted sum (N samples)

$$f(x)$$ - noisy input

$$p(x)$$ - Probability Distribution Function (PDF) allows to replace uniform distribution with something more relevant...
DENOISING: SPATIAL COMPONENT (4)

Sampling distribution & distance weight

Moving distance falloff math to the distribution and simplifying weight calculation to “step” function leads to output noise reduction!

Uniform

Weight = non_linear_F(d)

Quadratic

Weight = linear_F(d) or step(d, R)
Most important samples are on tangent plane

Use plane distance to calculate falloff

Use absolute value, otherwise denoising will skip all rounded objects
DENOISING: SPATIAL COMPONENT (6)

Normal weight

Using `pow` is incorrect because it explicitly contradicts lighting theory. It makes your result very oriented.

Using `x` instead of `pow(x, 8)` is a good idea.

```
// Please, don't use `pow`!
float NormalWeight(float3 Ncenter, float3 Nsample) {
    float f = dot(Ncenter, Nsample);
    return pow(saturate(f), 8.0);
}
```
DENOISING: SPATIAL COMPONENT (7)

Per pixel kernel rotations

NO!

Leads to 2x-5x slowdown!

Input signal is already noisy (applying noise on top of noise isn’t worth it)

Use per frame random rotation to improve quality of temporal accumulation!
DENOISING: SPATIAL COMPONENT (8)

Radius of denoising

Needs to be large, but can be scaled with distance

Compute **variance** of the input signal, blur less if variance is small

Blur less in “dark corners”, i.e. multiply by AO

**Signal-to-noise ratio** - blur less where direct lighting is strong

\[ R = \text{BaseRadius} \cdot F(\text{viewZ}) \cdot F(\text{variance}) \cdot F(\text{AO}) \]
DENOISING: SPATIAL COMPONENT (8)

Number of samples

A lot of samples are required! 32? 64? 128? (depending on number of passes)

Compute variance of the input signal, adaptively reduce number of samples if variance of the input signal is small...

...but variance computed for the current frame is always big! Solution - add temporal component \O/

Obviously, accumulated signal will get less and less variance over time!
DENOISING: TEMPORAL COMPONENT (1)

Common ideas

- GI/AO
- DENOISING
- TEMPORAL ACCUMULATION

A

- Better Low frequencies
- Less ghosting
- Better High frequencies

B

- GI/AO
- TEMPORAL ACCUMULATION
- DENOISING
DENOISING: TEMPORAL COMPONENT (2)

Our idea

More frequencies over time (mixture of low and high)

Requires less samples per frame

Less ghosting (denoising smoothes out reprojection artefacts)

(AO denoising uses this scheme, adaptive sampling with up to 64 samples, processes 2 pixels per thread sharing results between them if no edges)
DENOISING: LITTLE MONSTER (1)

GI denoiser

Hit distances -> GI -> Temporal accumulation -> Denoiser #1

Denoiser #1 -> Temporal accumulation -> Combiners

Combiners -> Denoiser #2

Denoised diffuse GI and indirect specular

Temporal feedback

Signal pass-through
DENOISING: LITTLE MONSTER (2)

Denoiser block

Computes variance of the input signal (3x3 pixels)

Computes radius scale as “F(viewZ) \cdot F(variance) \cdot F(AO)”

Computes adaptive step N = F(scaleRadius) (small radius = bigger step)

Processes each Nth sample from a poisson disk (up to 32 samples per pass)

The combiner just mixes up denoised and noisy input signals as:

\[
\text{Combiner} = \text{lerp}(\text{denoisedSignal}, \text{inputSignal}, 0.5 \ast \text{accumSpeed})
\]

(accumSpeed = 0.93 if no motion)
**DENOISING: LITTLE MONSTER (3)**

GI denoiser

**Temporal accumulation** always happens before denoising to eliminate ghosting and reprojection artefacts.

History is always rejected if out-of-screen sampling or z-occlusion are detected.
The output of each denoiser is always a combination of denoised and noisy input signals!

It helps to preserve tiny details.
First pass of denoising doesn’t take normals into account
It has wider base radius (6m)
Second pass of denoising takes normals into account

It has smaller base radius (3m)

Physically it’s same denoiser which applies “normal weight” on top of geometry weight
DENOISING

Tips & tricks

Use **NSIGHT GRAPHICS** GPU Trace utility to understand your limiters

**Fetch** heavy data only if weight is non-zero

**TAA** is your friend - it’s a free pass of denoising

**SH irradiance** is your friend - solves “blurriness” problem

**Know your noise** - perfection in image “cleanness” is not needed
## PERFORMANCE
RTX 2080 at 2560x1440

<table>
<thead>
<tr>
<th>Stage</th>
<th>HIGH</th>
<th>ULTRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pretrace</td>
<td>~0.4 ms</td>
<td>~0.8 ms</td>
</tr>
<tr>
<td>BLAS/TLAS (completely hidden by async)</td>
<td>~0.5 ms</td>
<td>~0.5 ms</td>
</tr>
<tr>
<td>Raytracing</td>
<td>1 to 3 ms</td>
<td>2 to 6 ms</td>
</tr>
<tr>
<td>AO Denoising</td>
<td>~0.6 ms</td>
<td>~0.9 ms</td>
</tr>
<tr>
<td>GI computation</td>
<td>~0.6 ms</td>
<td>~1.0 ms</td>
</tr>
<tr>
<td>GI denoising</td>
<td>~1.6 ms</td>
<td>~2.1 ms</td>
</tr>
<tr>
<td><strong>Total Frame Time Overhead (vs RTX OFF)</strong></td>
<td>~20%</td>
<td>~30%</td>
</tr>
</tbody>
</table>
ARTIST POINT OF VIEW

Just make it work for us
OUR FIRST RTAO SHOT

...Which one is RT ON? 🤔
DEFENDING THE CHOICE OF RTGI

Why not do reflections instead?

There were not many people who believed RTGI was a good direction of research.

From audience to stakeholders (oops)

Especially when convincing solutions already exist:

- SSAO and geometric ESM-AO for world space AO
- Super-lazy-realtime grid of probes for GI
- Voxel GI (which we already have nicely integrated with PBR in Exodus)
LIMITATIONS ARE ALSO WELL KNOWN

And we accepted them for years

Reflection probes or lightmaps for GI?

not a realtime solution

SSAO for AO?

suffers from its screen-space nature

limited to 1m tracing (good for features of… <1m in size)
SIZE MATTERS

1m is not enough (°่น°)

In large scenes short rays produce no more than an ‘edge trace’ effect
NEW INSANE POSSIBILITIES

Literally insane

50m ray tracing
Billions of rays per second
Per-pixel details at any scale:
  pencils on table 1mm scale
  ships 20m scale
  canyons, skyscrapers 100m+ scale

And at no cost!.. Well, almost
1m vs 50m
LEGACY AO
SSAO NO MORE
G1 replaces the need for it

Legacy AO:
- Tons of AO sources mixed
- Multiplied directly on shadows
- Effectively a patch

RTGI:
- Solves it all
SKYLIGHT SHADOWS
No direct lights involved

Single frame took several minutes of rendering in ‘99

Mesmerizing to watch
GI FROM LIGHT SOURCES

Interiors fully lit by sun
GI BY LIGHT PROBES
PER-PIXEL RTGI
Specular GI

Specular lighting contributes up to 50% of light on rough surfaces

Color bleeding

The most prominent feature in GI
COLOR BLEEDING OFF
CLOSE TO RELEASE
Content fixes and polishing

Making content work well in both modes

- Revert fake artsy lights
- Adjust non-RTX mode content to match RTX in extreme cases

Both versions must look good!

- There cannot be a loser
- It's Exodus vs Exodus
NEW MEASUREMENT OF ‘BETTER’

Enough of concerns

We do not expect RT-lighting to be exactly 'better'

Especially in an art-directed game

Results are clearly different

Mathematically stable solution makes them believable and natural

Or just convincing
HOW RT MAKES US HAPPY
A tool to play with

An achievement

Fully dynamic solution - 4A’s pillar

Lighting reference tool

Emergent results
OUR NEXT DREAMS
What would Oles dream of next?

AO and GI are nailed
Area lights with soft shadows
Caustics. Magic in real life
Raytracing as one unified solution
  - Light-based gameplay logic
  - Deferred-Forward
  - Volumetrics
RT on consoles
USEFUL LINKS


http://orlandoaguilar.github.io/sh/spherical/harmonics/irradiance/map/2017/02/12/SphericalHarmonics.html


Спасибо!

Slides at bit.ly/4agames

Oles Shyshkovtsov | oleksandr.shyshkovtsov@4a-games.com.mt
Sergei Karmalsky | sergei.karmalsky@4a-games.com.mt
Benjamin Archard | benjamin.archard@4a-games.com.mt
Dmitry Zhdan | dzhdan@nvidia.com
BONUS SLIDE!
Color to spherical harmonics

```c
float4 ConvertToIrradianceSH(float3 color, float3 dir, out float2 CoG)
{
    float Co = color.r - color.b;
    float t = color.b + Co * 0.5;
    float Cg = color.g - t;
    float Y = max(t + Cg * 0.5, 0.0);
    CoG = float2(Co, Cg);

    float L00 = 0.282095;
    float L1_1 = 0.488603 * dir.y;
    float L10 = 0.488603 * dir.z;
    float L11 = 0.488603 * dir.x;
    float4 shY = float4 (L11, L1_1, L10, L00) * Y;

    return shY;
}
```
BONUS SLIDE!
Spherical harmonics to color (no resolve)

```c
float3 GetColorFromIrradianceSH(float4 shY, float2 CoCg)
{
    float Y = shY.w / 0.282095;
    float T = Y - CoCg.y * 0.5;
    float G = CoCg.y + T;
    float B = T - CoCg.x * 0.5;
    float R = B + CoCg.x;

    return max(float3(R, G, B), 0.0);
}
```
BONUS SLIDE!

Spherical harmonics resolve

```c
float3 ResolveIrradianceSHToDiffuse(float4 shY, float2 CoCg, float3 N)
{
    float d = dot(shY.xyz, N);
    float Y = 2.0 * (1.023326 * d + 0.886226 * shY.w);
    Y = max(Y, 0.0);

    // correct color-reproduction
    CoCg *= 0.282095 * Y / (shY.w + 1e-6);

    // YCoCg -> RGB
    float T = Y - CoCg.y * 0.5;
    float G = CoCg.y + T;
    float B = T - CoCg.x * 0.5;
    float R = B + CoCg.x;

    return max(float3(R, G, B), 0.0);
}
```