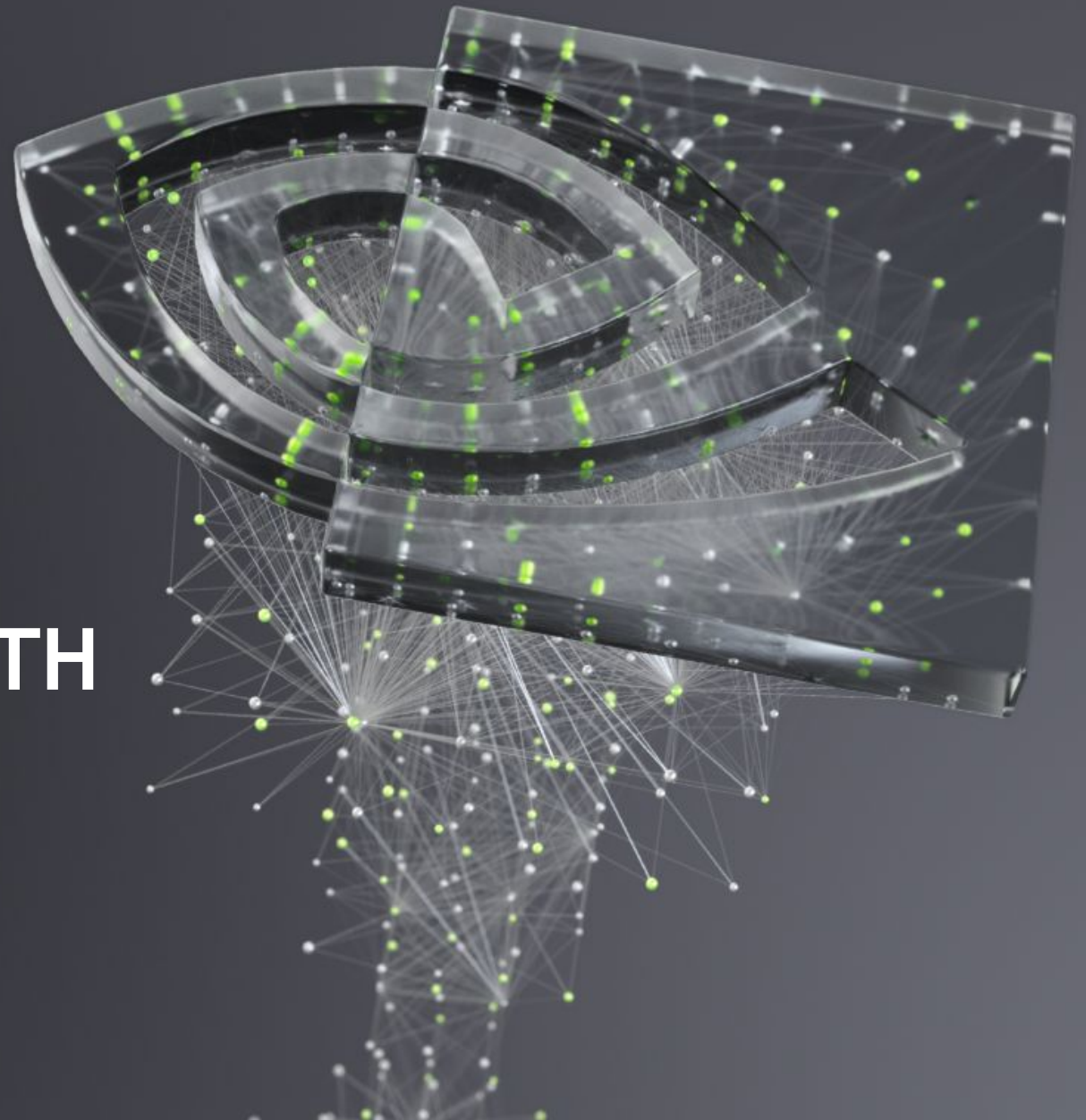


NVIDIA

FAST DENOISING WITH SELF-STABILIZING RECURRENT BLURS

Dmitry Zhdan (NVIDIA), 2020.03.18

AGENDA

Problem statement

Introduction

Ingredient #1 - recurrent blur

Ingredient #2 - accurate temporal accumulation

Ingredient #3 - hierarchical history reconstruction

Ingredient #4 - choosing sampling space

Ingredient #5 - choosing blur weights

Cooking diffuse & specular denoisers

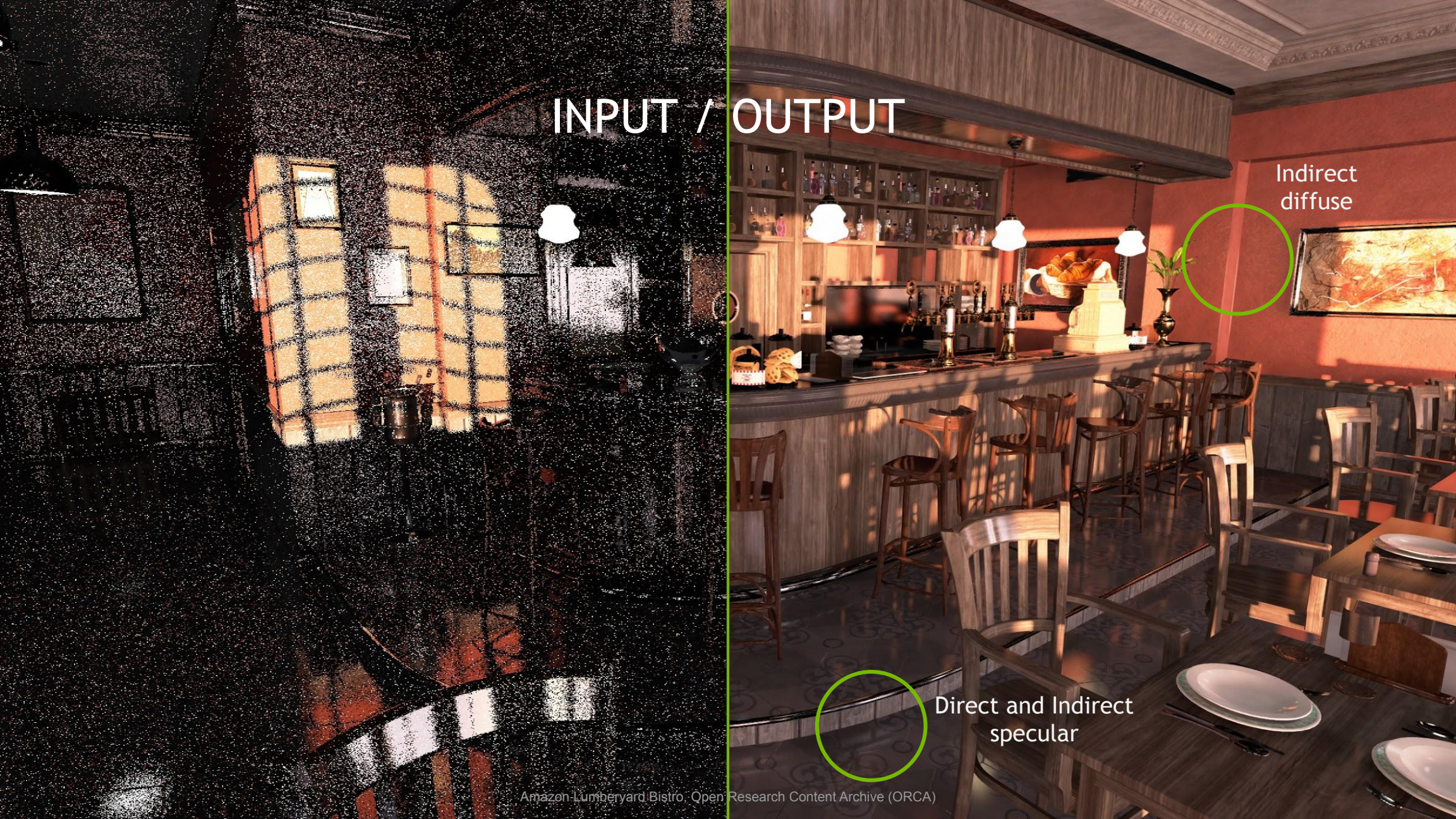
PROBLEM STATEMENT

TYPICAL DENOISER INPUT (RAY TRACING OUTPUT)

DENOISER OUTPUT

Amazon Lumberyard Bistro, Open Research Content Archive (ORCA)

INPUT / OUTPUT



Amazon Lumberyard Bistro, Open Research Content Archive (ORCA)

DENOISER IN ACTION (VIDEO)

Amazon Lumberyard Bistro, Open Research Content Archive (ORCA)

INTRODUCTION

USEFUL DEFINITIONS AND ACRONYMS

AO - ambient occlusion

SO - specular occlusion

Normalized hit distance - logically it's AO or SO, mathematically = *saturate(hitT / maxHitT)*

SH - spherical harmonics (first 4 coefficients in my case)

Roughness - linear roughness, m = roughness ^{2}

GGX-D - GGX specular lobe dominant direction [1]

RPP - rays per pixel

NoV - dot(pixelNormal, viewVector)

FUN THOUGHTS ABOUT DENOISING

Any blur works as a denoiser...

…but it's a slippery way

Anything except temporal accumulation makes the algorithm "biased"…

…but temporal accumulation introduces lag, how to preserve physical correctness?

For low RPP signals…

…denoising is a tradeoff between temporal lag and blurriness

If denoising is not DL/ML based…

…it's a bunch of blur and temporal accumulation passes

ALGORITHM: PREREQUISITES

Denoiser accepts as inputs only radiance (material information is not used)

Diffuse and specular signals must be separated

Better input signal - better denoising results:

- Diffuse - cos-distribution or explicit importance sampling
- Specular - Visible Normals Distribution Function sampling [2]

ALGORITHM: BASIC PRINCIPLES

A unified approach for diffuse and specular denoising

Based on recurrent blur → uses only a few samples per blur pass

Relies on temporal accumulation → but it's a tunable parameter

Does hierarchical signal reconstruction in regions with discarded history using mipmap chains → very stable if a disocclusion is detected

Adaptive blur radius depends on number of successfully accumulated frames → no temporal or spatial variance tracking

Method is biased → kinda "psycho-physical" where there is no chance to be physical

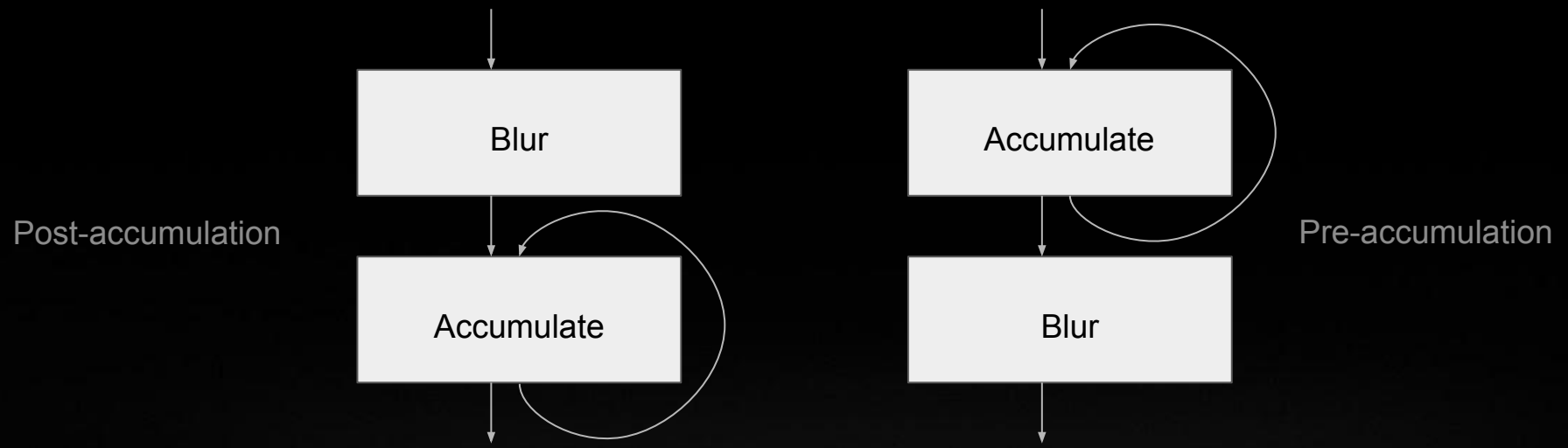
ALGORITHM: PERFORMANCE

	2060 @720p	2060 @1080p	2080 @1080p	2080 @1440p	2080 Ti @1080p	2080 Ti @1440p
Diffuse	1.12	2.44	1.40	2.48	0.99	**1.82**
Specular	0.87	1.73	1.22	2.08	1.04	**1.54**
Sun shadow	0.23	0.51	0.32	0.60	0.24	**0.44**
Total (ms)	**~2.2**	**~4.7**	**~3.0**	**~5.2**	**~2.3**	**~3.8**

Preliminary results

INGREDIENT #1 - RECURRENT BLUR

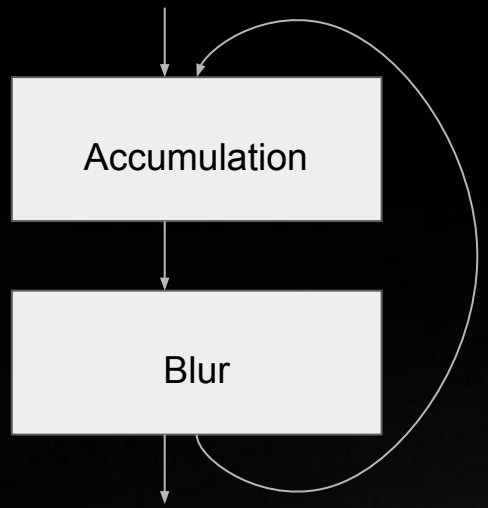
NON-RECURRENTLY BLURRED VARIANTS



Both variants don’t mix signals with different frequencies (noisy input and clean output)

BASIC PRINCIPLE OF RECURRENT BLUR [3]

Recurrent blur



Mixes signals with various frequencies (final - clean & input - noisy)

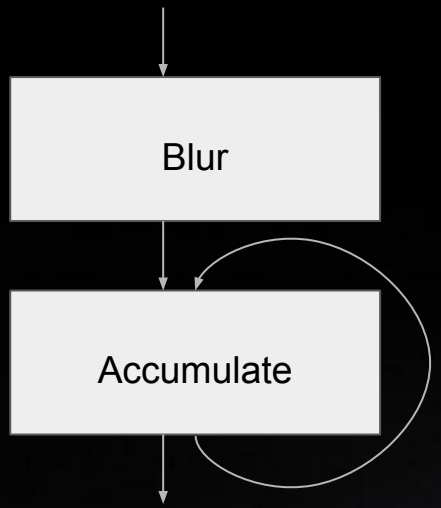
A blur with a “background” (a “clean” output from the previous frame)

Allows to redistribute spatial sampling in time due to recursive nature

30 FPS = 8 x 30 = 240 samples / sec (cumulative effect)

Using Poisson Disk distribution with only 8 samples works well

POST-ACCUMULATION: DOWNSIDES



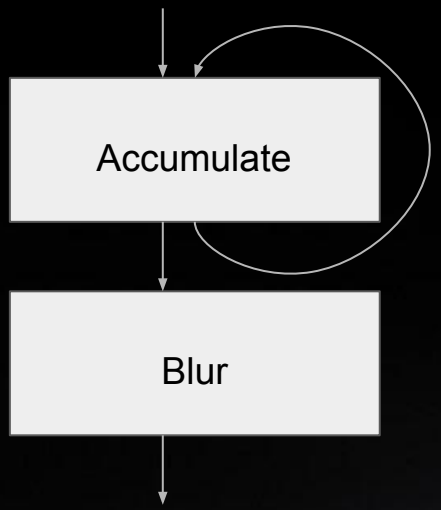
We blur the input signal without “background” 😞

If the history buffer is rejected… it becomes a nightmare, because at the moment of blur we don’t know which regions will be discarded (╯°□°)╯ ⌒ ┻━┻

Potentially more blur passes are needed

Potentially a wider blur is needed

PRE-ACCUMULATION: DOWNSIDES



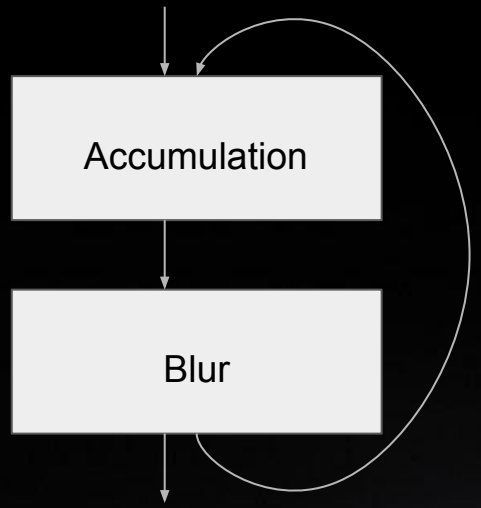
We have a “background” now, but it’s not the final image

We still blur the input signal without background if history buffer is rejected (╯°□°)╯ ‾ ┻━┻

Potentially more blur passes are needed

Potentially a wider blur is needed

RECURRENT BLUR



The previous frame is the best we have. Why not use it?

Looks odd because we cook high and low frequencies in one cauldron

If history is rejected we have the same problem, but there is a solution (see Hierarchical History Reconstruction section)

We have a clean background now

STABILIZATION OF RECURRENT BLUR

Constant blur radius leads to massive over-blurring (do you remember “Half Life 2” main menu?). How to avoid that? 🤔

Number of successfully accumulated frames can be used to control blur radius:

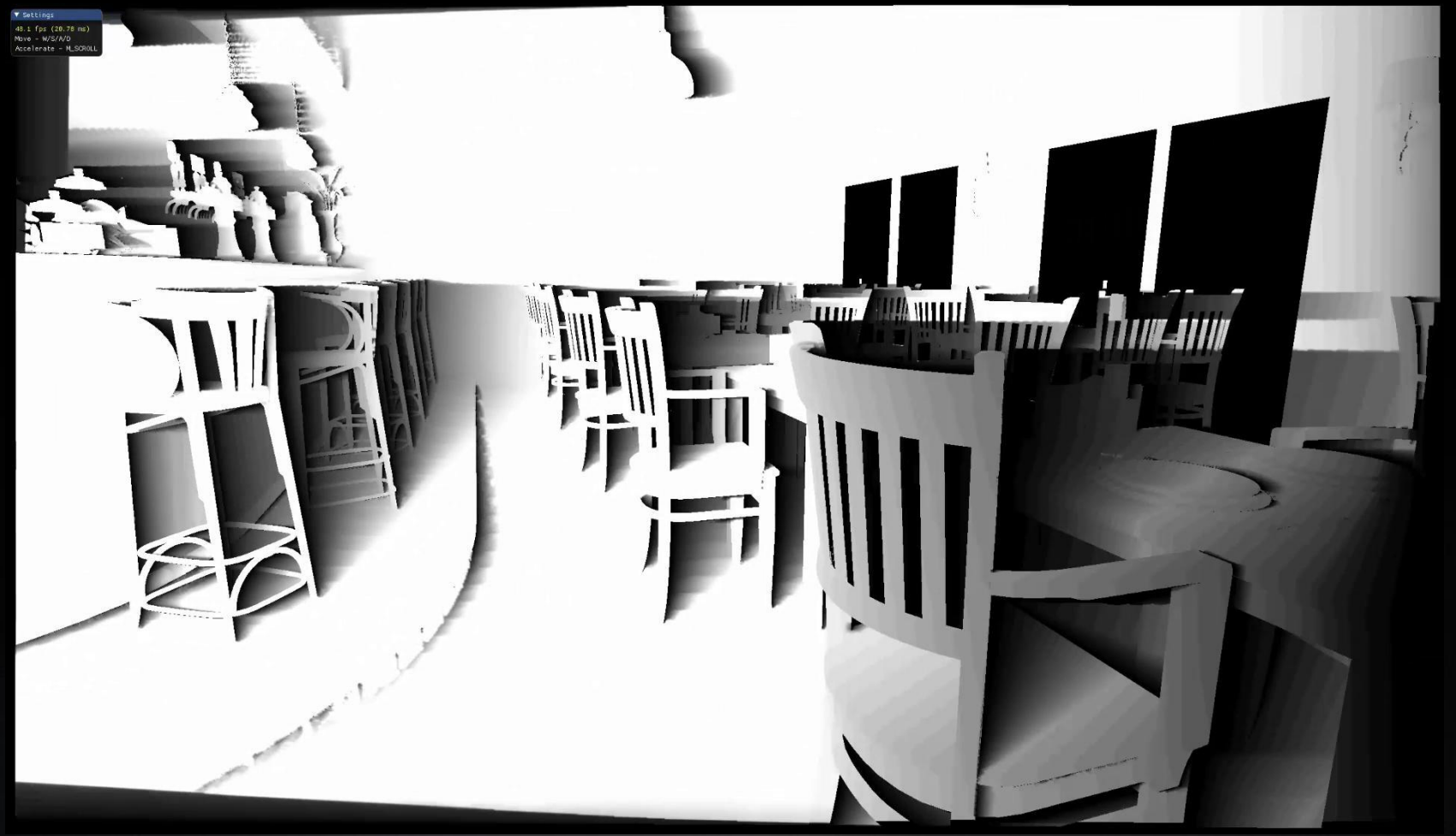
- accumulation has just started (after disocclusion) → Largest

- a lot of frames are accumulated (close to the maximum) → Smallest

Pseudo code [4]:

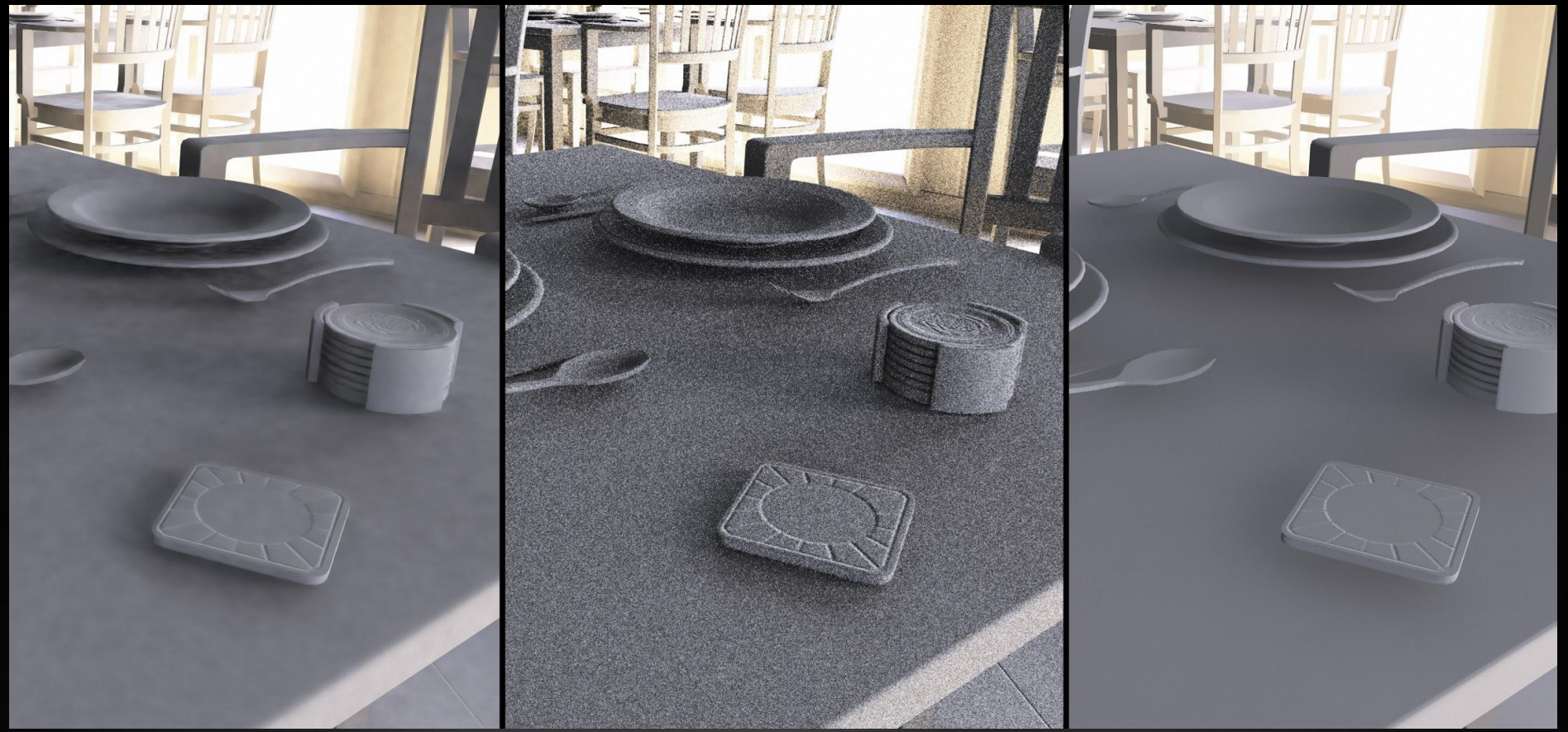
```
float oldCount = ReprojectFromPreviousFrame( t_AccumulatedFramesNum );
float newCount = min( oldCounts * MAX_FRAME_NUM + 1.0, MAX_FRAME_NUM );
newCount = occlusion ? 0 : newCount;

// later can be used for blur radius scaling
float blurRadiusScale = 1.0 / ( 1.0 + newCount );

// 5 < MAX_FRAME_NUM < 32 - we need to solve the integral but we don’t want to introduce significant temporal lag
```

NUMBER OF ACCUMULATED FRAMES VISUALIZATION



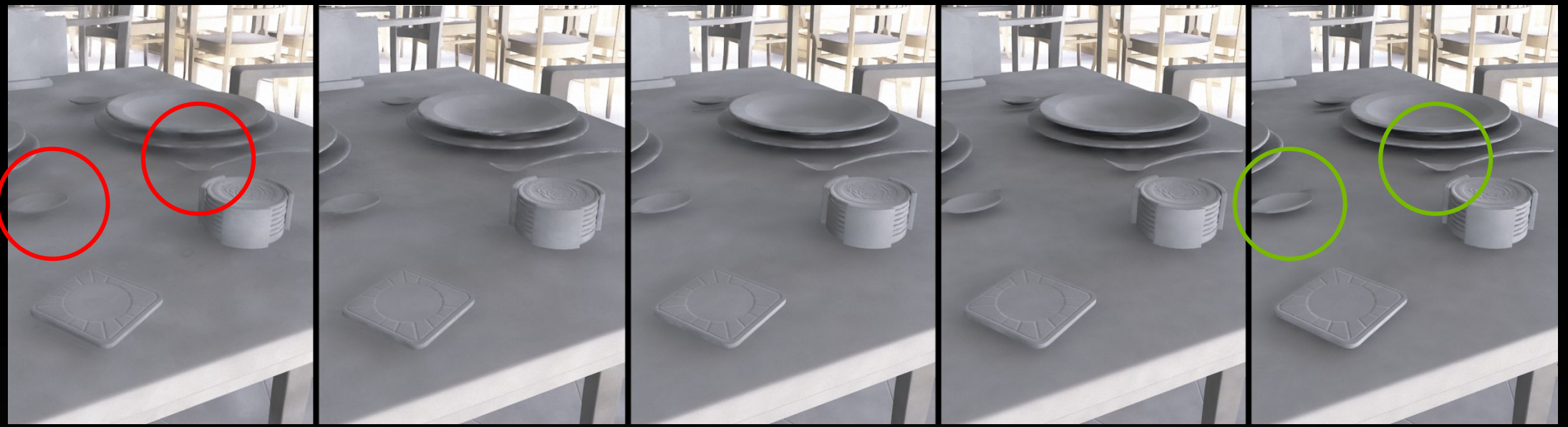
STABILIZATION OF RECURRENT BLUR

Stabilized (adaptive radius) ~Reference (no spatial filtering) Non-stabilized (const radius)



```
float blurRadiusScale = 1.0 / ( 1.0 + N ), where N - number of successfully accumulated frames
```

ADAPTIVE BLUR RADIUS



After 2 frames of accumulation | After 4 frames... | After 8 frames... | After 16 frames... | After 32 frames

If history gets discarded we see blur instead of noise

INGREDIENT #2 - ACCURATE TEMPORAL ACCUMULATION

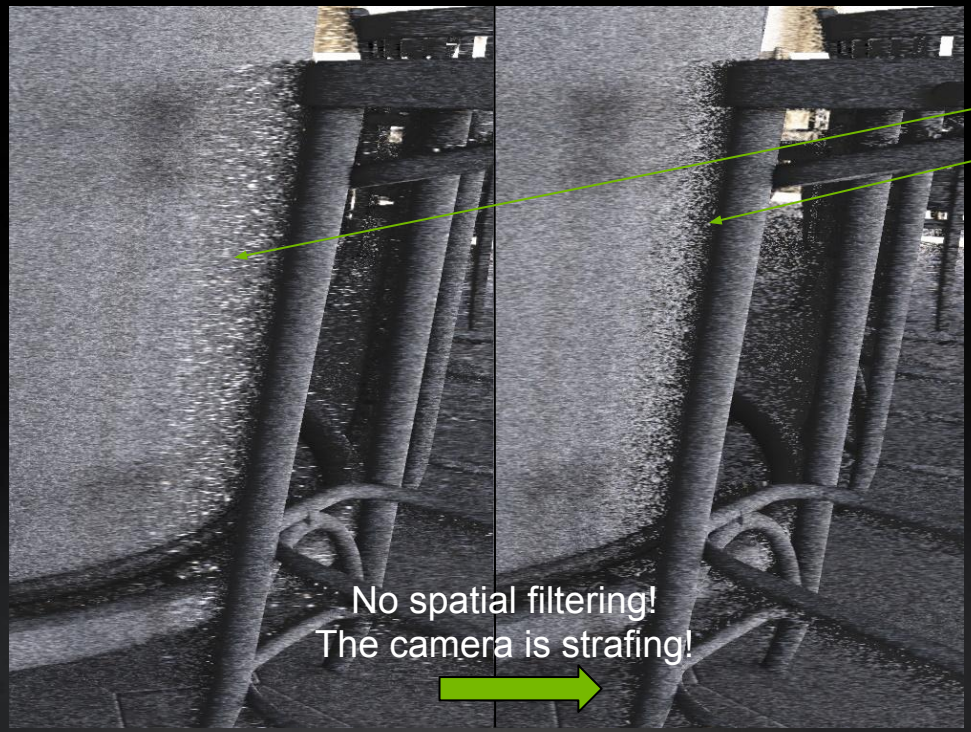
EXPONENTIAL VS LINEAR ACCUMULATION WEIGHTS

Exponential accumulation speed (an example) = { 1.0 if a disocclusion is detected, ~0.05 if everything is good }

But… it means that on 60 fps it will take some time to converge from scratch, because there are only two speeds – slow and “reset”. It can be improved by using linear weights!

Exponential weights

Linear weights



dis-occluded regions

While exponential weights force the history buffer to become stuck in the past...

...linear weights do averaging across frames

ACCUMULATION WITH LINEAR WEIGHTS - HOW?

Temporal accumulation:

history[n] = lerp(history[n - 1], curr, speed)

Linear accumulation speed:

speed = 1/(1 + N), N = number of accumulated frames

Proof (N = 2):

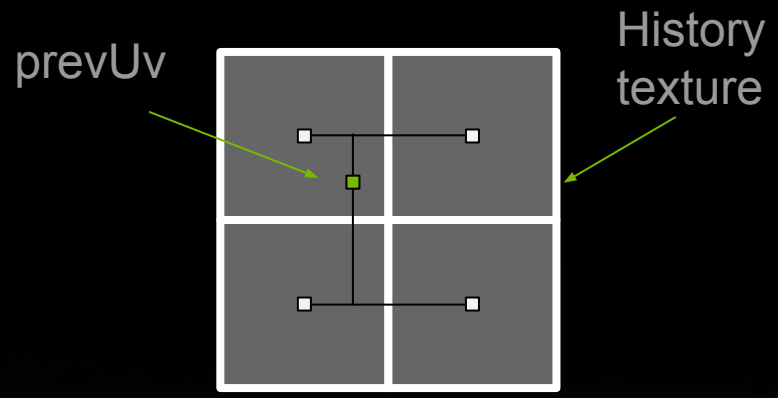
history[0] = curr0 * (1/1) = curr0 (disocclusion, aka history reset)

history[1] = history[0] * (1/2) + curr1 * (1/2) = curr0 * (1/2) + curr1 * (1/2) = (curr0 + curr1) / 2

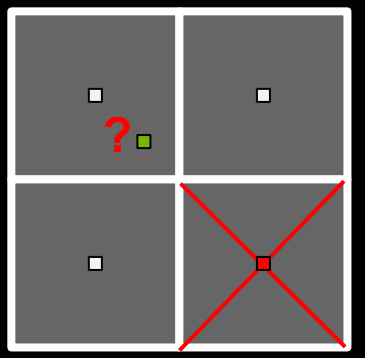
history[2] = history[1] * (2/3) + curr2 * (1/3) = [curr0 * (1/2) + curr1 * (1/2)] * (2/3) + curr2 * (1/3) =

= (curr0 + curr1 + curr2) / 3

GHOSTING FREE TEMPORAL REPROJECTION - INTRO



If all the samples of the bilinear footprint are not occluded → Bilinear filter = OK



If at least one sample is out due to disocclusion detection → Bilinear filter != OK

Bilinear filtering with custom weights can help!

BILINEAR FILTERING WITH CUSTOM WEIGHTS

```
struct Bilinear { float2 origin; float2 weights; };

Bilinear GetBilinearFilter( float2 uv, float2 texSize )
{
    Bilinear result;
    result.origin = floor( uv * texSize - 0.5 );
    result.weights = frac( uv * texSize - 0.5 );
    return result;
}

float4 GetBilinearCustomWeights( Bilinear f, float4 customWeights )
{
    float4 weights;
    weights.x = ( 1.0 - f.weights.x ) * ( 1.0 - f.weights.y );
    weights.y = f.weights.x * ( 1.0 - f.weights.y );
    weights.z = ( 1.0 - f.weights.x ) * f.weights.y;
    weights.w = f.weights.x * f.weights.y;
    return weights * customWeights;
}

float4 ApplyBilinearCustomWeights( float4 s00, float4 s10, float4 s01, float4 s11, float4 w, bool normalize = true )
{
    float4 r = s00 * w.x + s10 * w.y + s01 * w.z + s11 * w.w;
    return r * ( normalize ? rcp( dot( w, 1.0 ) ) : 1.0 );
}
```

Nothing unusual

Expand 3 lerp-s into separate weights for each sample of the 2x2 footprint. Multiply by user provided weights

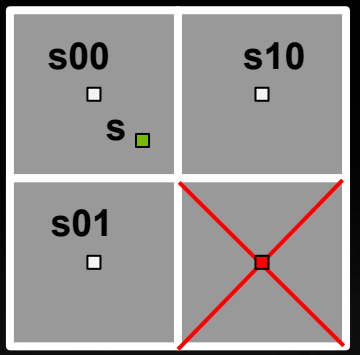
Apply 2x2 bilateral filter (it's just a weighted sum)

BILINEAR FILTERING WITH CUSTOM WEIGHTS - DERIVATIONS

Fully matches normal bilinear filtering if user provided weights = 1

It's a 2x2 bilateral filter by the definition

Using binary weights allows to get rid of ghosting in temporal accumulation passes



s = (s00 * f00 + s10 * f00 + s01 * f00) / (f00 + f10 + f01),

where fXY are corresponding bilinear weights

GHOSTING FREE TEMPORAL REPROJECTION

```
// Previous viewZ and accumulation speed
Bilinear bilinearFilterAtPrevPos = GetBilinearFilter( saturate( pixelUvPrev ), gScreenSize );
float2 gatherUv = ( float2( bilinearFilterAtPrevPos.origin ) + 1.0 ) * gInvScreenSize;
float4 viewZprev = gIn_Prev_ViewZ.GatherRed( gNearestClamp, gatherUv ).wzxy;
float4 accumSpeedPrev = gIn_Prev_AccumSpeed.GatherRed( gNearestClamp, gatherUv ).wzxy;

// Compute disocclusion basing on plane distance
float3 Xprev = X + motionVector;
float3 Xvprev = mul( gWorldToViewPrev, Xprev );
float NoXprev = dot( N, Xprev );
float NoVprev = NoXprev / Xvprev.z;
float4 planeDist = abs( NoVprev * viewZprev - NoXprev );
float4 occlusion = step( gDisocclusionThreshold, planeDist * invDistToPoint );
occlusion = saturate( float( isInScreen ) - occlusion );

// Sample history
float4 weights = GetBilinearCustomWeights( bilinearFilterAtPrevPos, occlusion );
// ... read s00, s10, s01, s11 using point filtering
float4 history = ApplyBilinearCustomWeights( s00, s10, s01, s11, weights );

// Accumulation speed
accumSpeedPrev = min( accumSpeedPrev + 1.0, MAX_ACCUM_FRAME_NUM );
float accumSpeed = ApplyBilinearCustomWeights( accumSpeedPrev.x, accumSpeedPrev.y, accumSpeedPrev.z, accumSpeedPrev.w, weights );
```

Reproject and read 2x2 footprints of previous viewZ and accumulation speed textures

Transform the current point to the previous view space and compute disocclusion factors using plane distance (see Slide 44), cut off values if relative delta >= 0.5-1.5%

Read the bilinear footprint, modify bilinear weights with occlusion info and apply the weights (Slide 28)

INGREDIENT #3 - HIERARCHICAL HISTORY RECONSTRUCTION

HIERARCHICAL HISTORY RECONSTRUCTION - IDEA

We have 1rpp signal, what we can get from it? We can trade-off resolution to RPP:

Mip #0 = 1 rpp, pixel = 1x1 (input)

Mip #1 = 4 rpp, pixel = 2x2

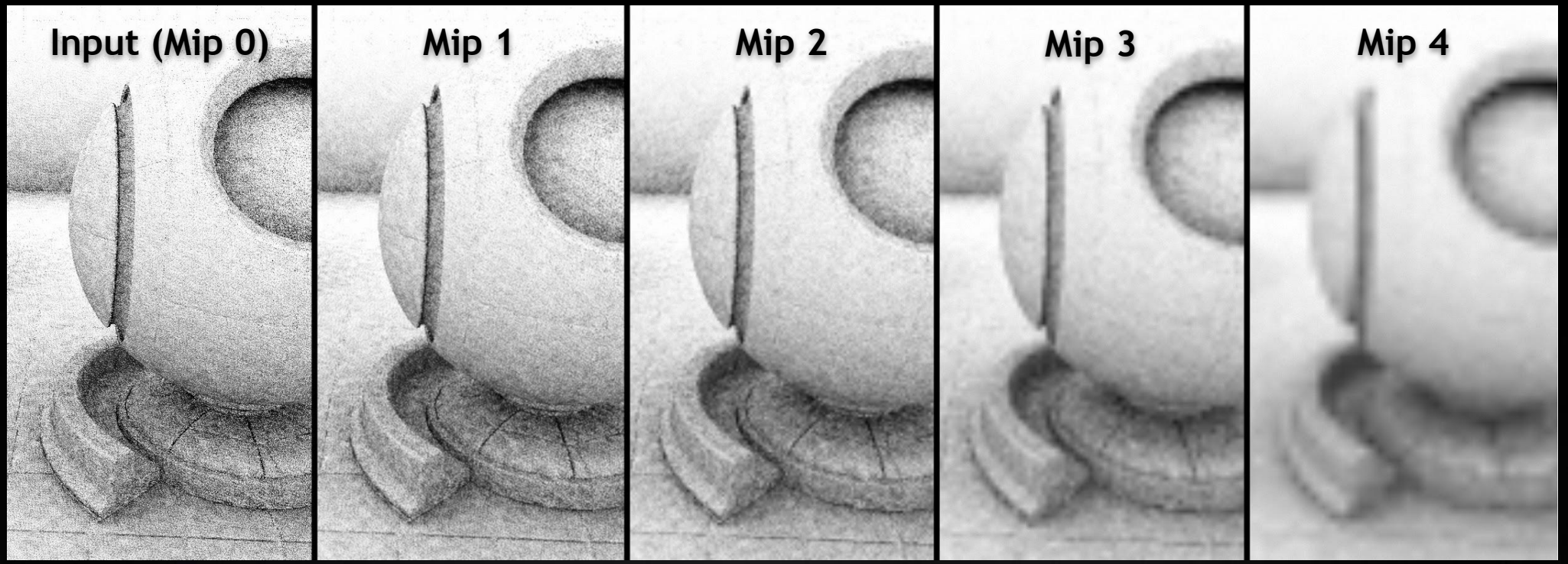
Mip #2 = 16 rpp, pixel = 4x4

Mip #3 = 64 rpp, pixel = 8x8

Mip #4 = 256 rpp, pixel = 16x16 (lower mips have significantly reduced visibility information)

Mips are generated for radiance and viewZ (viewZ delta is used to compute weights during upsampling)

HIERARCHICAL HISTORY RECONSTRUCTION - IDEA

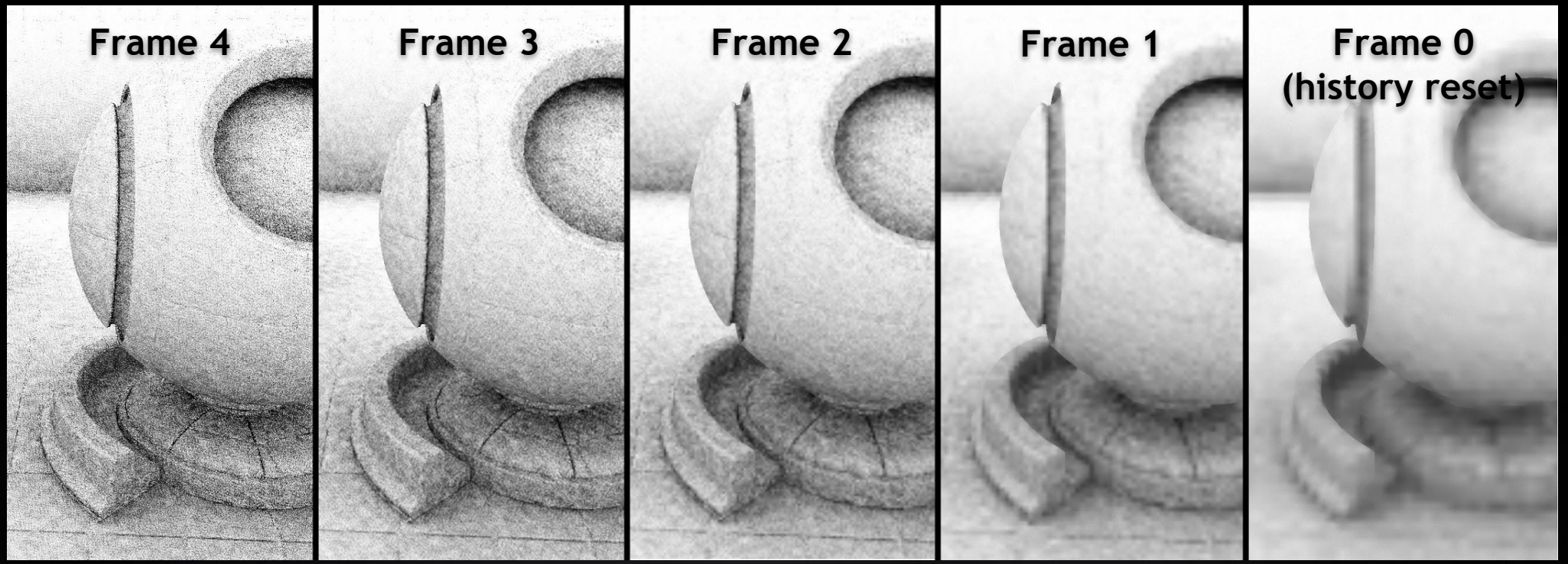


Just idea visualization with naive linear upsampling (the denoiser uses bilateral upsampling)!

Details

Stability of the image

HIERARCHICAL HISTORY RECONSTRUCTION - IDEA



Only bilateral upsampling for the data in selected mip level is demonstrated (no recurrent blurring)!

HIERARCHICAL HISTORY RECONSTRUCTION

```
float normAccumulatedFrameNum = saturate( gIn_AccumulatedFrameNum[ pixelPos ] / MAX_FRAME_NUM_WITH_HISTORY_FIX );
if ( normAccumulatedFrameNum == 1.0 )
    return; // No output

uint mipLevel = 4.0 * ( 1.0 - normAccumulatedFrameNum ) * roughness;
float2 mipSize = gScreenSize / ( 1 << mipLevel );
Bilinear filter = GetBilinearFilter( pixelUv, mipSize );
float4 bilinearWeights = GetBilinearCustomWeights( filter, 1.0 );

float realZ = gIn_ViewZ[ pixelPos ]; // it's an SRV with 5 mips, base level = 0

float4 z = ...; // read unfiltered Z data from mip = mipLevel
float4 bilateralWeights = GetBilateralWeight( z, realZ, UPSAMPLING_VIEWZ_SENSITIVITY );
float4 w = bilinearWeights * bilateralWeights;

float4 s00, s10, s01, s11;
... ; // read unfiltered radiance data (an SRV with 4 mips, base level = 1, base level = 0 - UAV)

float4 blurry = ApplyBilinearCustomWeights( s00, s10, s01, s11, w );

gOut_Radiance[ pixelPos ] = blurry;
```

No disocclusion = no work (read/write into the same UAV)

Mip level = F(number of accumulated frames, roughness)

Process a block of 2x2 (or more) pixels using a bilateral filter (bilinear filtering with custom weights in my case, but the same idea can be applied to cubic filter)

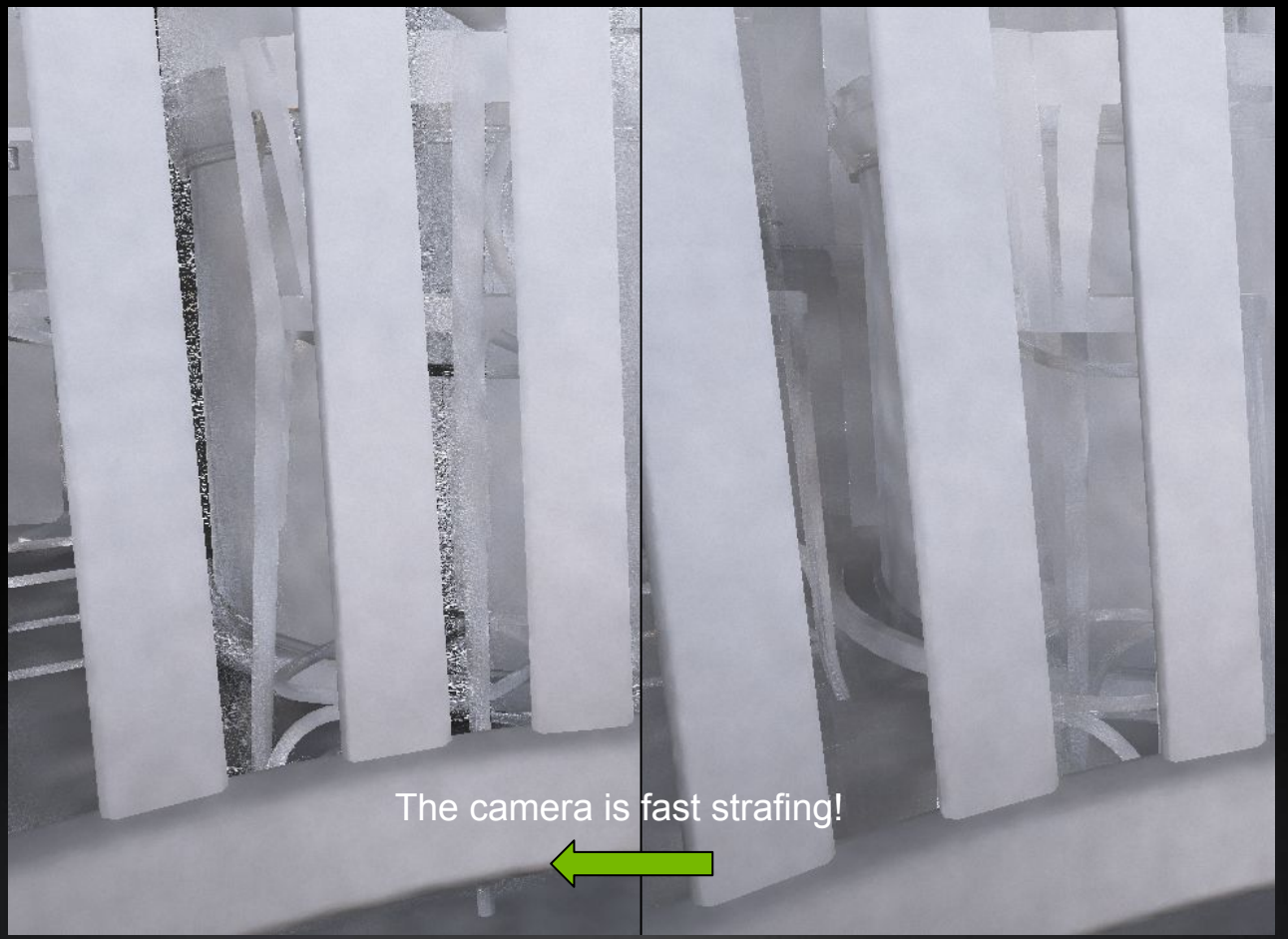
Progression:

- Frame 0 - 2x2 (mip 4) = 32x32 real pixels touched
- Frame 1 - 2x2 (mip 3) = 16x16 real pixels touched
- Frame 2 - 2x2 (mip 2) = 8x8 real pixels touched
- … (current results go into the feedback loop on each iteration)

HIERARCHICAL HISTORY RECONSTRUCTION

History fix is OFF

History fix is ON

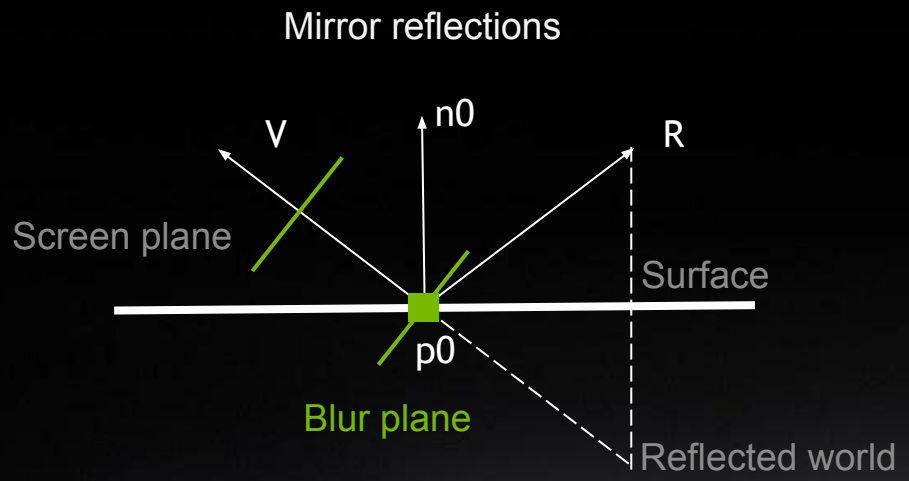


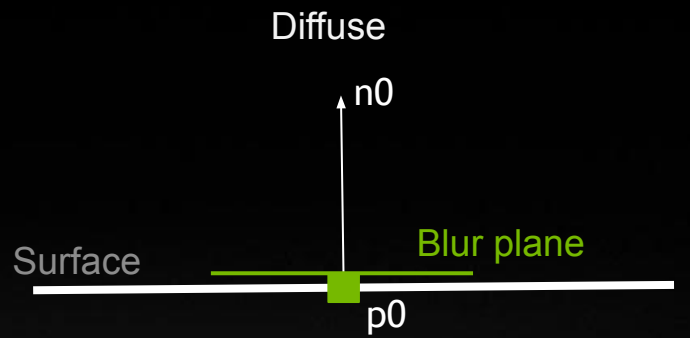
INGREDIENT #4 - CHOOSING SAMPLING SPACE

DIFFUSE AND SPECULAR SAMPLING - OBVIOUS EXAMPLES

Diffuse - want to blur in the tangent plane to preserve tiny details under glancing angles [3]

Almost mirror specular - want to blur in a plane which is perpendicular to the view vector (screen space for simplicity) to match its nature





How to generalize all the cases into a single sampling model?

GGX DOMINANT DIRECTION

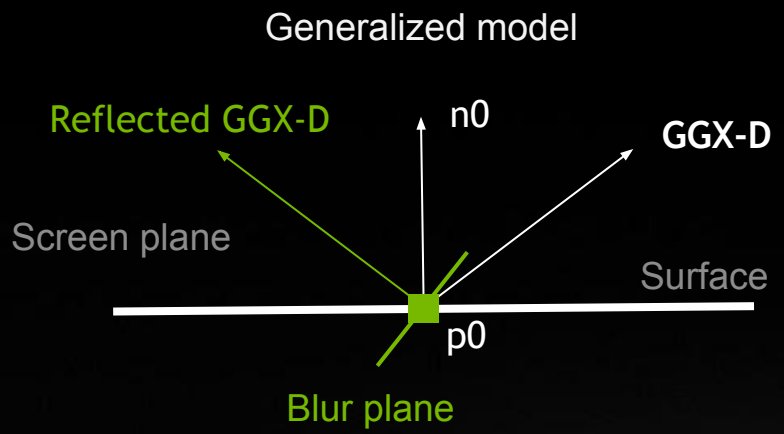

```
float3 GetSpecularDominantDirection( float3 N, float3 V, float roughness )
{
    // See [5]
    float f = ( 1.0 - roughness ) * ( sqrt( 1.0 - roughness ) + roughness );
    float3 R = reflect( -V, N );
    float3 dir = lerp( N, R, f );

    return normalize( dir );
}
```

Same effect can be achieved by “trimming”: VNDF importance sampling:

```
// See [2]
// Section 4.2: parameterization of the projected area
// trimFactor: 1 - full lobe, 0 - true mirror
float r = sqrt( saturate( rnd.x * trimFactor ) );
float phi = rnd.y * Math::Pi( 2.0 );
float t1 = r * cos( phi );
float t2 = r * sin( phi );
```

UNIFYING DIFFUSE AND SPECULAR SAMPLING

We can blur in a plane which is perpendicular to the reflected GGX dominant direction!




```
void GetKernelBasis( float3 X, float3 N, float roughness, float worldRadius, out float3 T, out float3 B )
{
    float3 V = -normalize( X ); // Assuming view space
    float3 D = GetSpecularDominantDirection( N, V, roughness );
    float3 R = reflect( -D, N );

    // Such basis construction is needed for anisotropy (see Slide 62)
    T = normalize( cross( N, R ) ); // IMPORTANT: doesn't handle the case when N = R!
    B = cross( R, T );

    T *= worldRadius;
    B *= worldRadius;
}
```

p0 - point of interest (kernel center)
n0 - normal at kernel center

UNIFYING DIFFUSE AND SPECULAR SAMPLING

Specular (roughness = 1)

Screen space sampling

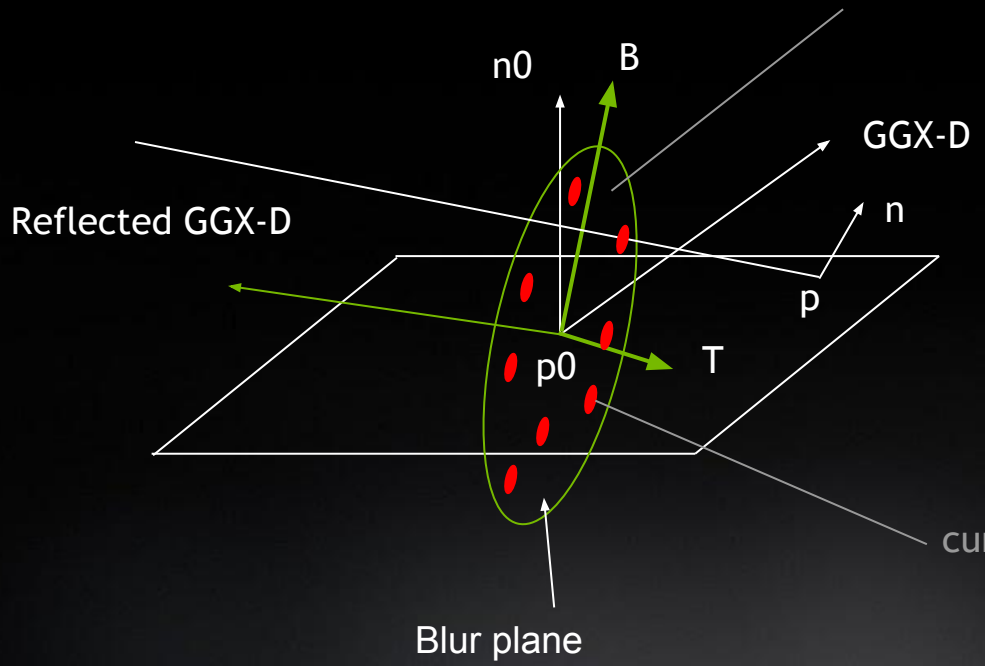
Generalized sampling

INGREDIENT #5 - CHOOSING SAMPLE WEIGHTS

SAMPLE WEIGHT

Sample weight = Geometry weight • Normal weight • Roughness Weight

Poisson disk distribution (8 samples)



p0 - point of interest (kernel center)
n0 - normal at kernel center
p - real sample position
n - sample normal
T and B - define sampling plane

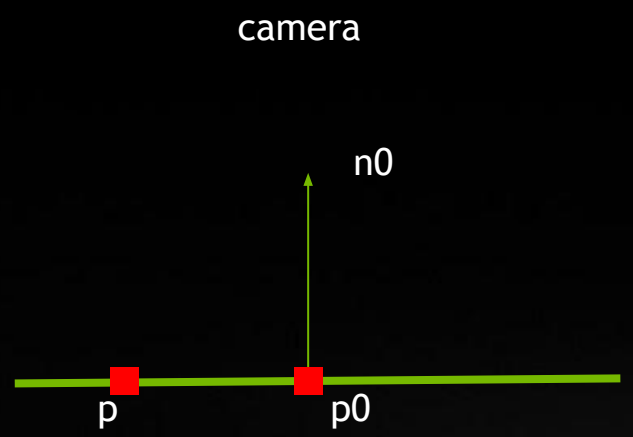
Reflected GGX-D = reflect(-GGX-D, n0)
cross(T, B) = Reflected GGX-D
dot(T, n0) == 0
cross(GGX-D, n0) gives T direction

current sample with properties:
p - position
n - normal
roughness - roughness

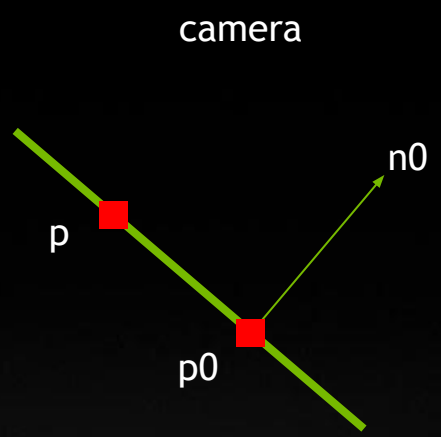
SAMPLE WEIGHT - GEOMETRY [3]

Based on plane distance - distance between the current sample “p” and the tangent plane {p0, n0} (see [3], slide 59)

Why plane distance works better than viewZ comparison?



Plane T = {p0, n0}



p.z == p0.z and PlaneDist(T, p) == 0 (p is accepted)

p.z < p0.z, while PlaneDist(T, p) == 0 (under glancing angles Z comparison leads to rejection of valid samples)

SAMPLE WEIGHT - GEOMETRY

Based on plane distance - distance between the current sample “p” and the tangent plane {p0, n0} (see [3], slide 59)

```
// View space positions can be reconstructed from “screen coordinates” and “viewZ”
float GetGeometryWeight( float3 p0, float3 n0, float3 p, float planeDistNorm )
{
    // where planeDistNorm = accumSpeedFactor / ( 1.0 + centerZ );
    // It represents { 1 / "max possible allowed distance between a point and the plane" }
    float3 ray = p - p0;
    float distToPlane = dot( n0, ray );
    float w = saturate( 1.0 - abs( distToPlane ) * planeDistNorm );
    return w;
}
```

SAMPLE WEIGHT - NORMAL

The idea is to reject samples if the angle between two normals is higher than the specular lobe half angle, which can be computed as:

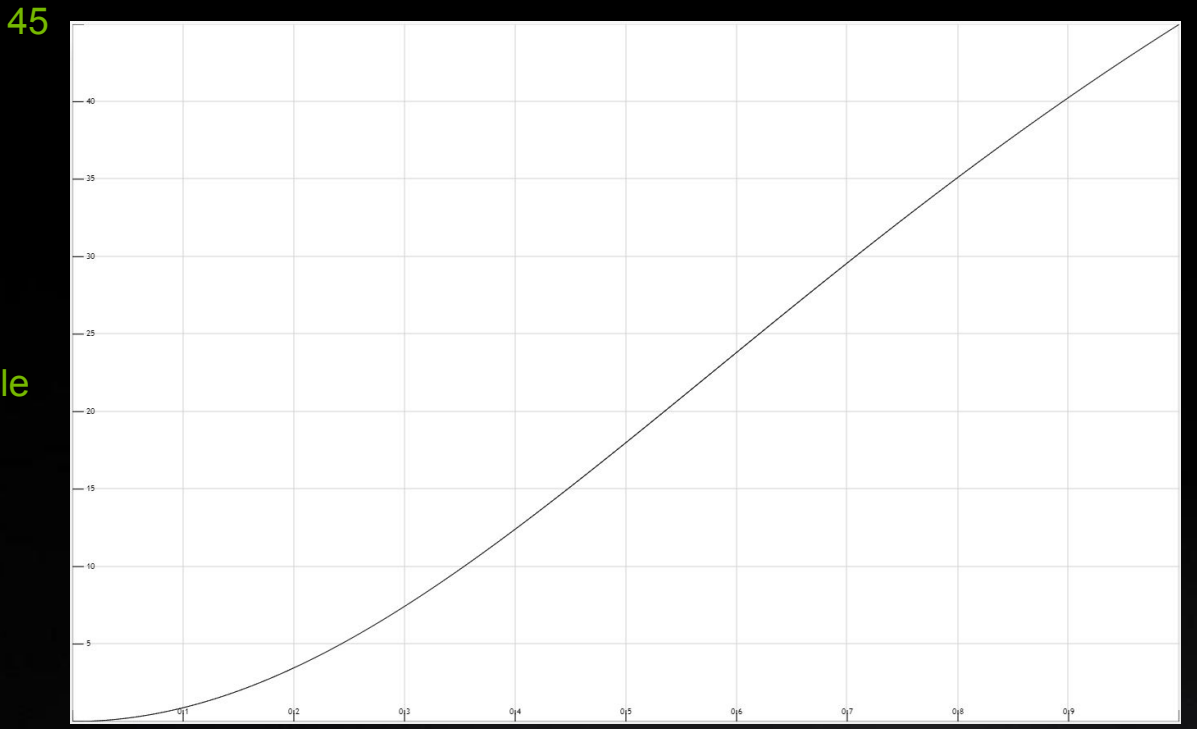
```
halfAngleInDegrees = 90.0 * m / ( 1.0 + m ), where m = roughness * roughness
```

```
float GetNormalWeight( float3 n0, float3 n, float roughness )
{
    float a0 = STL::ImportanceSampling::GetSpecularLobeHalfAngle( roughness );
    a0 = a0 * CloseToZeroIfAccumulationGoesWell( numberOfAccumulatedFrames ) + STL::Math::DegToRad( 0.5 ); // Optional

    float cosa = saturate( dot( n0, n ) );
    float a = STL::Math::AcosApprox( cosa );
    float w = STL::Math::LinearStep( a0, 0.0, a );

    return w;
}
```

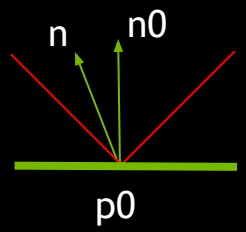
Make the angle more narrow over time

a < a0 ? 0 : linear rescale to [0; 1]

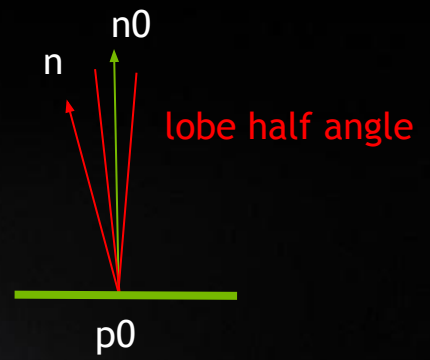
SAMPLE WEIGHT - NORMAL



half angle = 90 * roughness ^{2} / (1 + roughness ^{2})



Roughness = 1.0, angle = 45°,
sample “n” is accepted



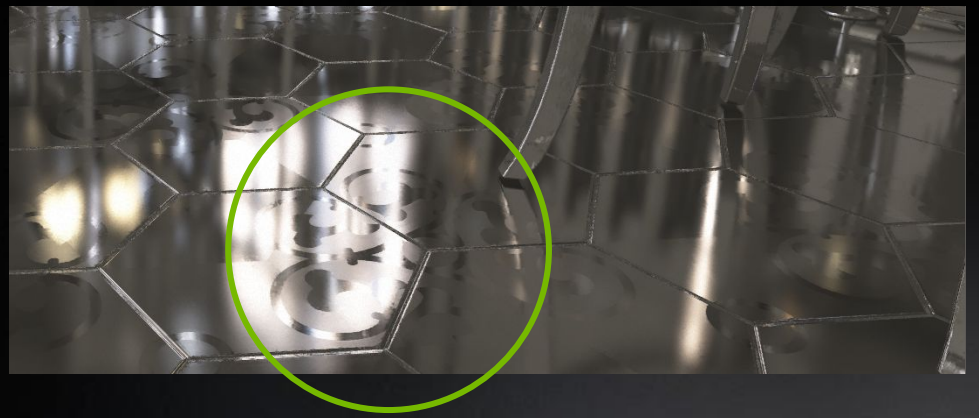
Roughness = 0.4, angle = 12°,
sample “n” is rejected

SAMPLE WEIGHT - ROUGHNESS

```
float GetRoughnessWeight( float roughness0, float roughness )
{
    float norm = roughness0 * roughness0 * 0.99 + 0.01;
    float w = abs( roughness0 - roughness ) * rcp( norm );

    return saturate( 1.0 - w );
}
```

This normalization forces to ignore samples with significantly higher roughness than at the kernel center

Roughness weight



Roughness weight ignored

COOKING DIFFUSE DENOISER

INPUTS / OUTPUTS

Inputs:

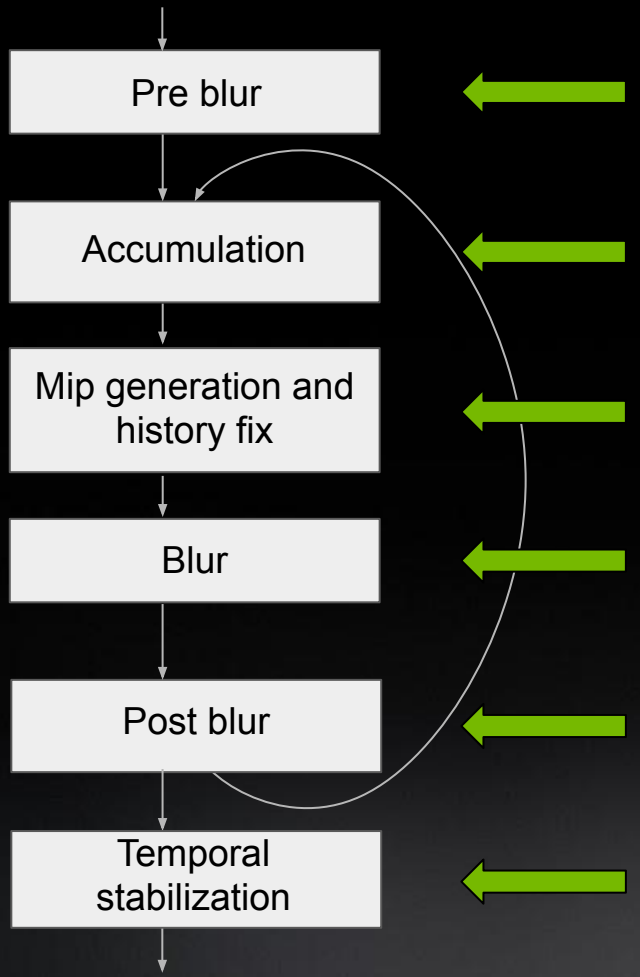
- Incoming radiance in SH (2 x RGBA16f)
  - RGBA16f - SH.c1, AO
  - RGBA16f - SH.c0, chroma, viewZ (fp16)
- ViewZ (R32f)
- Normal (RGBA8)
- Motion vector (2D or 3D - RG16f or RGBA16f)

Nothing unusual, all these textures are parts of G-buffer

Outputs:

- Denoised radiance in SH and AO (same layout as input)

PIPELINE



Uses constant radius, pass is needed to fix outliers (see next slide)

Linear weights, up to 32 frames (input parameter)

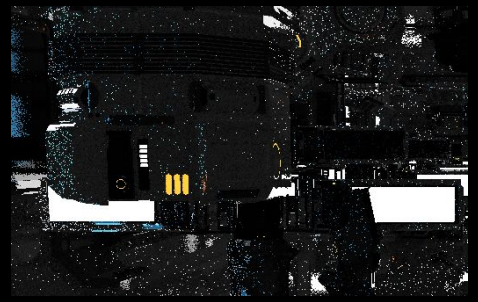
Ultra fast single pass generation of first 4 mips in shared memory (averaging) + history reconstruction in discarded regions

Adaptive radius (depends on number of accumulated frames)

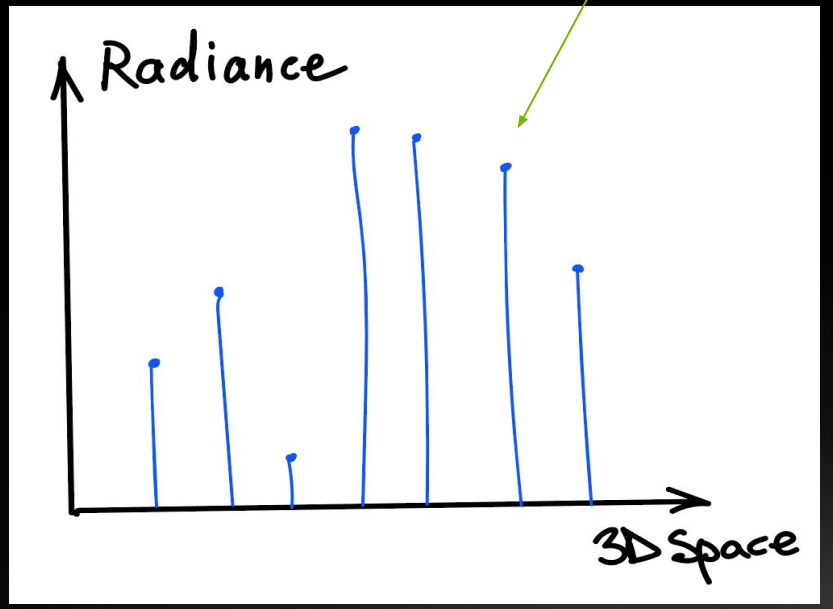
Adaptive radius (depends on number of accumulated frames + adaptively scales if intensity delta between reprojected history and the final value is high)

Temporal stabilization != temporal accumulation (no additional lag, TAA like filter but uses wider variance clamping if possible)

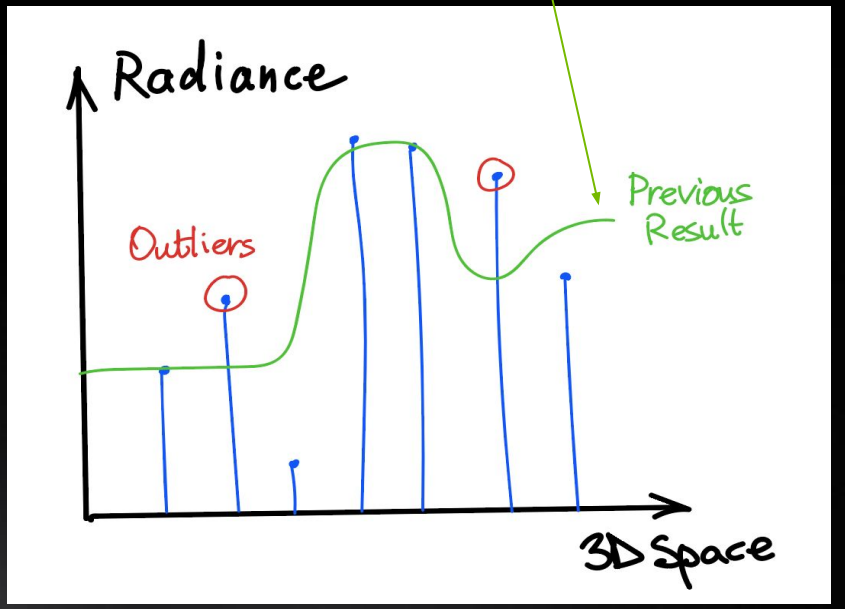
PRE-BLUR - WHY? (WITH NICE CHARTS)



Input signal

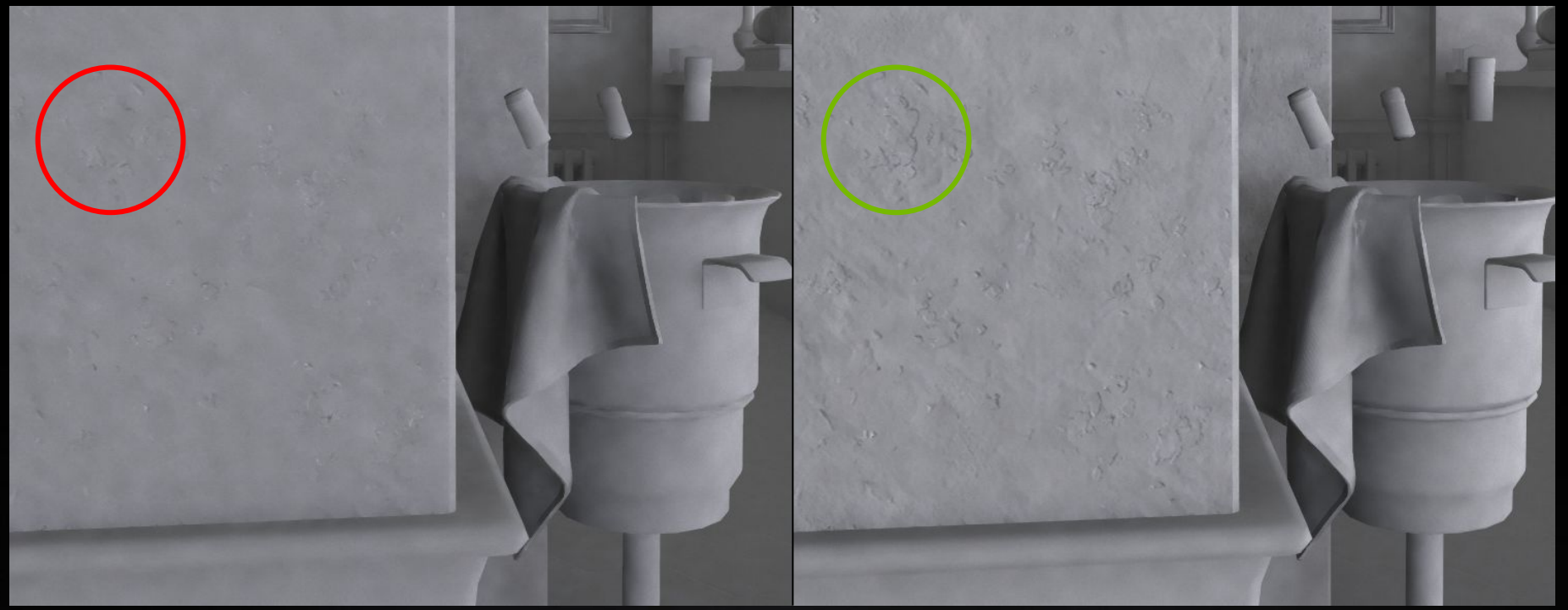


Reprojected previous frame



Pre-blur allows to blur out pixels with significantly changed radiance

SPHERICAL HARMONICS - WHY?



SH off

SH on

SH helps to preserve normal map details and increases “readability” of diffuse in general…
… but adds overhead
(signal to SH conversion can be found here [3])

DIFFUSE DENOISER - TAKEAWAYS

Recurrent blur allows to use low sample count in sparse blur passes

Free AO denoising (one texture channel is dedicated to AO, no additional logic)

Pre- and post- blur passes are optional, but real signals dictate the opposite

SH increases “readability” of diffuse

Chaining temporal accumulations can significantly increase temporal lag, hard to tune

Hierarchical history reconstruction allows to avoid using very wide blurs in dis-occluded regions

COOKING SPECULAR DENOISER

INPUTS / OUTPUTS

Inputs:

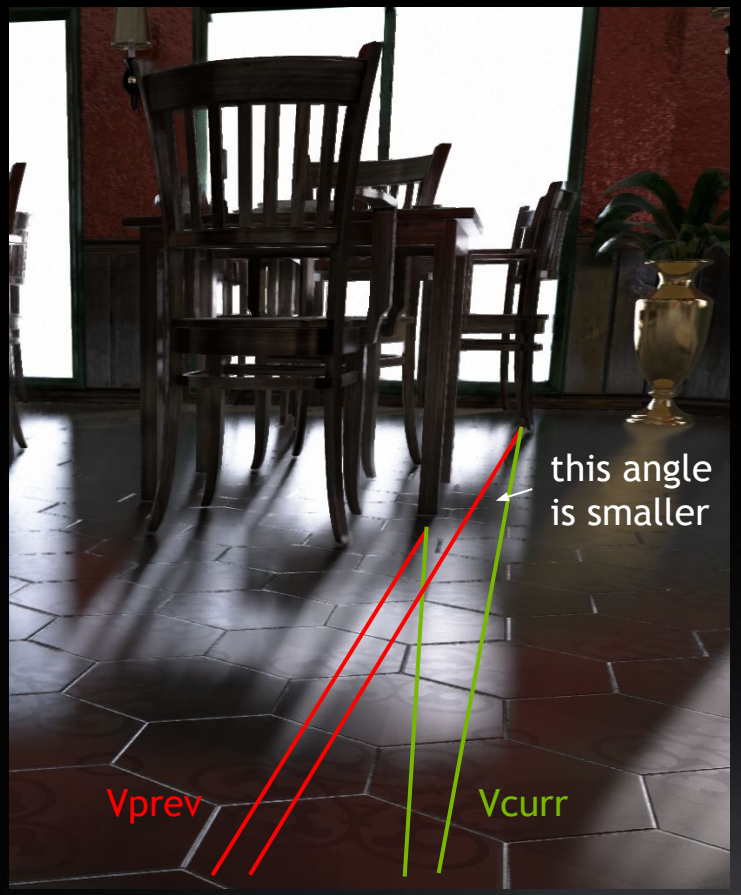
- Incoming radiance and SO (RGBA16f)
- ViewZ (R32f)
- Normal and roughness (RGBA8)
- Motion vector (2D or 3D - RG16f or RGBA16f)

Nothing unusual, all these textures are parts of G-buffer

Outputs:

- Denoised radiance and SO (same layout as input)

SPECULAR ACCUMULATION USING SURFACE MOTION



Naive accumulation using surface motion doesn't work!

We can introduce a new entity - parallax

Parallax - represents angle between previous and current view vectors for the same surface point

```
// Coordinates in world space
float3 movementDelta = X - ( Xprev - gCameraDelta );

// ~Sine of angle between old and new view vector in world space
float parallax = length( movementDelta ) / ( distToPoint * frameTime );
```

NoV = dot(N, V) represents "sensitivity" to parallax - more sensitive under glancing angles (accumulation speed becomes faster)

Accumulation speed = F(parallax, NoV, roughness)

In other words, we don't touch MVs we adopt accumulation speed!

SPECULAR ACCUMULATION USING SURFACE MOTION

Psychological model for specular accumulation - accumulate using surface motion where our eyes are not sensitive

SPEC_ACCUM_CURVE controls aggressiveness of history rejection depending on viewing angle

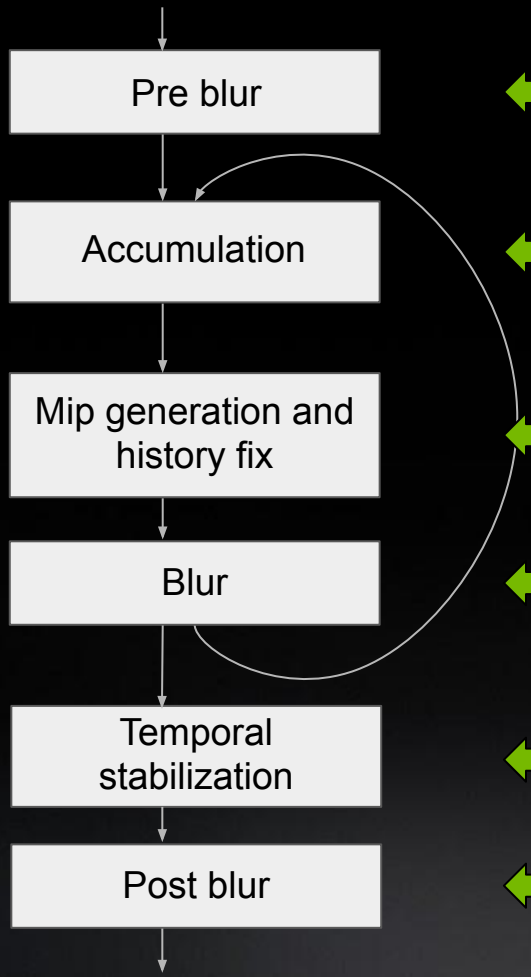
- Smaller values - less accumulation under glancing angles

SPEC_ACCUM_BASE_POWER controls sensitivity to parallax in general

- Smaller values - more aggressive accumulation

```
float GetMaxAllowedAccumulatedFrameNum( float roughness, float NoV, float parallax )
{
    float acos01sq = saturate( 1.0 - NoV ); // ~ ”normalized acos” ^ 2
    float a = pow( acos01sq, SPEC_ACCUM_CURVE );
    float b = 1.001 + roughness * roughness;
    float angularSensitivity = ( b + a ) / ( b - a );
    float power = SPEC_ACCUM_BASE_POWER * ( 1.0 + parallax * angularSensitivity );

    return MAX_ACCUM_FRAME_NUM * pow( roughness, power );
}
```

PIPELINE



Uses constant radius, pass is needed to fix outliers

Linear weights, up to 32 frames (input parameter), accumulation speed = F(parallax, NoV, roughness)

Ultra fast single pass generation of first 4 mips in shared memory (averaging) + history reconstruction in discarded regions

Adaptive radius (depends on number of accumulated frames)

Temporal stabilization != temporal accumulation (no additional lag, TAA like filter but uses wider variance clamping if possible)

Adaptive radius (depends on number of accumulated frames)

USING HIT DISTANCE TO CONTROL BLUR RADIUS

```
float d = length( Xv );
float f = hitDist / ( hitDist + d );
blurRadius *= lerp( K * roughness, 1.0, f);
```

Reflections are sharper when the reflected world is closer to the ray origin

Hit distance can be used to control blur radius:

- hitDist - hit distance (denoised in all passes except pre-blur)
- Xv - pixel position in view space
- f - always in range [0; 1]
- K - constant in range (0; 1)

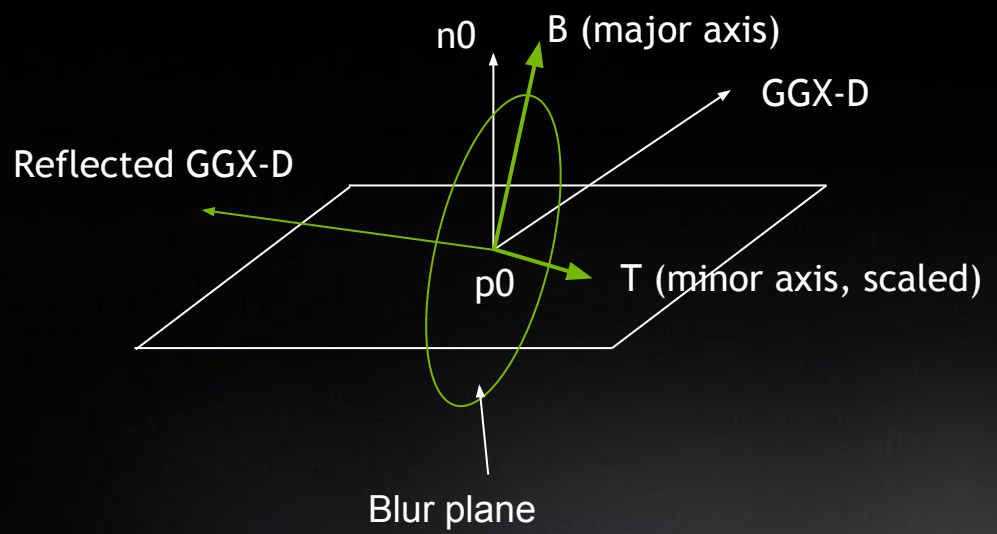
ANISOTROPIC SAMPLING

Isotropic sampling

Anisotropic sampling

ANISOTROPIC SAMPLING

```
void GetKernelBasis( float3 X, float3 N, float roughness, float worldRadius, float normalizedAccumFrameNum, out float3 T, out float3 B )
{
    // Same as on Slide 40

    float angle = saturate( STL::Math::AcosApprox( abs( N.z ) ) / STL::Math::Pi( 0.5 ) ); // assuming normal is in view space...
    float skewFactor = lerp( 1.0, roughness, angle );
    T *= lerp( 1.0, skewFactor, normalizedAccumFrameNum ); // unnormalized!
}
```



T gets scaled more under glancing angles

Skew factor depends on roughness

If accumulation goes badly kernel shape moves towards isotropic (in the world space, still anisotropic in the screen space)

SPECULAR DENOISER - TAKEAWAYS

Free SO denoising (one texture channel is dedicated to SO, no additional logic)

Pre- and post- blur passes are optional, but real signals dictate the opposite

Temporal stabilization is a must have (stabilization doesn't increase temporal lag)

Hierarchical history reconstruction allows to avoid using very wide blurs in dis-occluded regions

SO only denoising = the simplest poor-man solution for IBL visibility:

FUTURE WORK

Avoiding constant radius usage in pre-blur pass

Improving temporal lag

Reducing extra blurriness of reflections

Research possibility of dropping SH and still preserving IQ (seems to be possible, but harder in motion)

Separating reprojection and accumulation, doing reprojection first allows to estimate temporal variance in pre-blur pass

Playing with low discrepancy sampling

Using experience from more game integrations to improve denoiser!

More optimizations and ideas!

THANKS!
Evgeny Makarov (NVIDIA)
Ivan Fedorov (NVIDIA)
Oles Shyskovtsov (4A Games)
Alexey Panteleev (NVIDIA)
Maksim Eisenstein (NVIDIA)

QUESTIONS?

dzhdan@nvidia.com

NVIDIA

REFERENCES

[1] - https://seblagarde.files.wordpress.com/2015/07/course_notes_moving_frostbite_to_pbr_v32.pdf

[2] - http://jcgt.org/published/0007/04/01/paper.pdf

[3] - https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9985-exploring-ray-traced-future-in-metro-exodus.pdf

[4] - https://developer.nvidia.com/sites/default/files/akamai/gamedev/files/gdc12/GDC12_Bavoil_Stable_SSAO_In_BF3_With_STF.pdf (slide 39)

[5] - https://seblagarde.files.wordpress.com/2015/07/course_notes_moving_frostbite_to_pbr_v32.pdf (page 69)