Microsoft

Microsoft Game Stack Live

Microsoft

Denoising Raytraced Soft Shadows on Xbox Series X|S and Windows with FidelityFX

Dominik Baumeister, AMD

Agenda

- Building blocks of the denoiser
- Performance improvements

10,000 foot view

- Simulate area lights by choosing a random point on the light and tracing a ray towards it
- This inherently introduces noise that is tightly coupled with the sampling pattern
- We average over multiple rays (samples) to converge, but this may become prohibitively costly
- Reuse neighboring samples instead and accumulate samples over multiple frames
- For a small number of lights, it is feasible to drop the lights color and instead sample a mask

Based on NVIDIAs paper on Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination
https://research.nvidia.com/publication/2017-07_Spatiotemporal-Variance-Guided-Filtering%3A

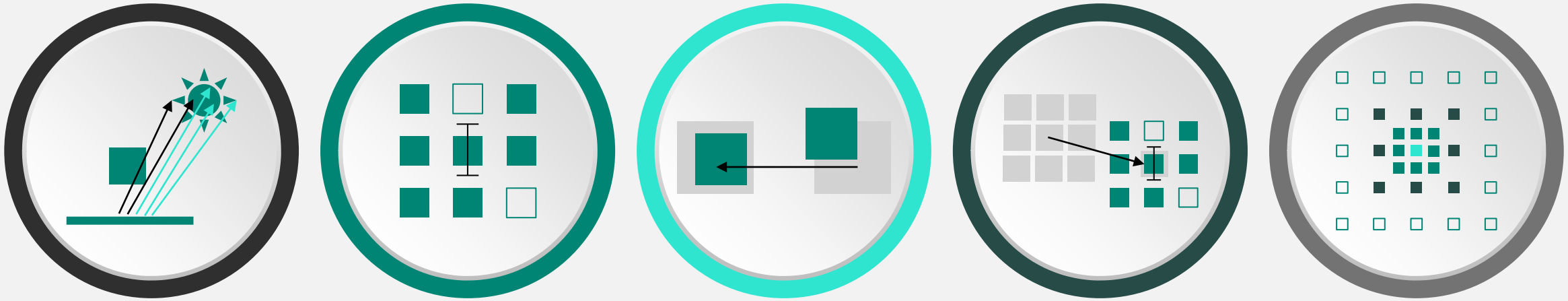
Disclaimer: I merely jumped in and made it faster.
All credit goes to Guillaume Boissé for making it work in the first place


DIRT 5


the RIFT BREAKER

Building blocks

Building blocks



Create Shadow Mask

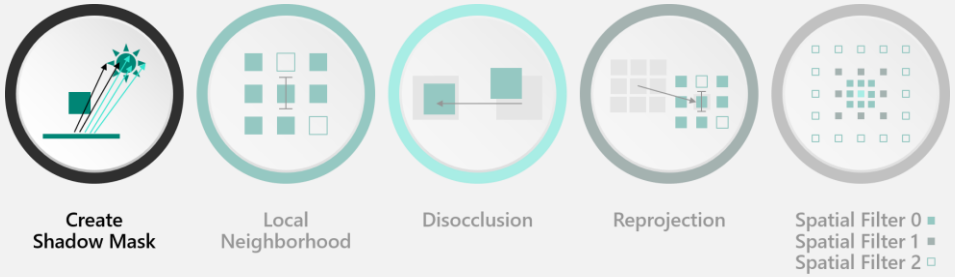
Local Neighborhood

Disocclusion

Reprojection

Spatial Filter 0
Spatial Filter 1
Spatial Filter 2

Building blocks

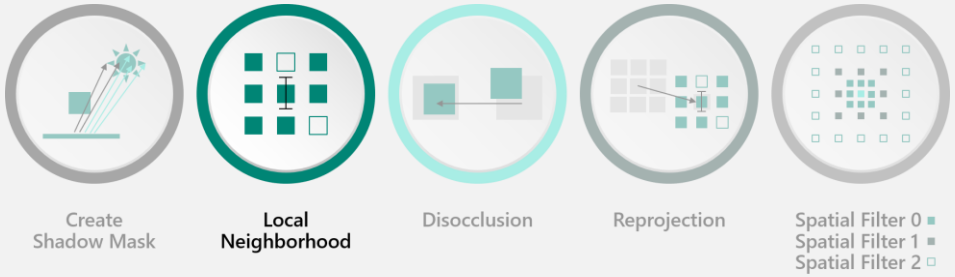


Create Shadow Mask

- Initialize a counter to 0
- Choose a fixed number of random points on the light source
- Shoot ray with offset along the normal to avoid self-intersections
- Increment the counter by 1 for every miss
  That means we missed geometry and thus, hit the light
- Store counter value in our shadow mask target

Noisy shadow mask

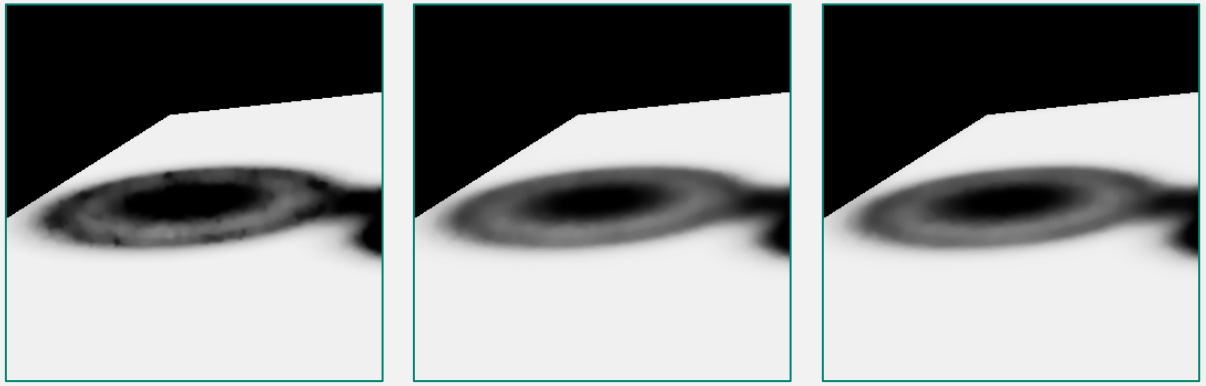
Building blocks



Estimate Local Neighborhood

- Estimate mean and variance in a neighborhood
- Variance tells how noisy the data is
- This can be used to create a window in which we would expect additional rays to lie in
- We settled with a 17x17 kernel. Usually, the larger the region, the better the results
- Important observation: We can separate horizontal and vertical filter kernels

$$mean \mathrel{+}= value * weight$$
$$variance \mathrel{+}= value^2 * weight$$

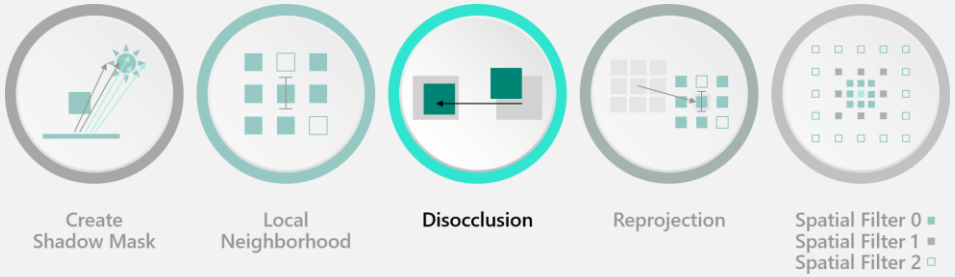


5x5 kernel

17x17 kernel

29x29 kernel

Building blocks



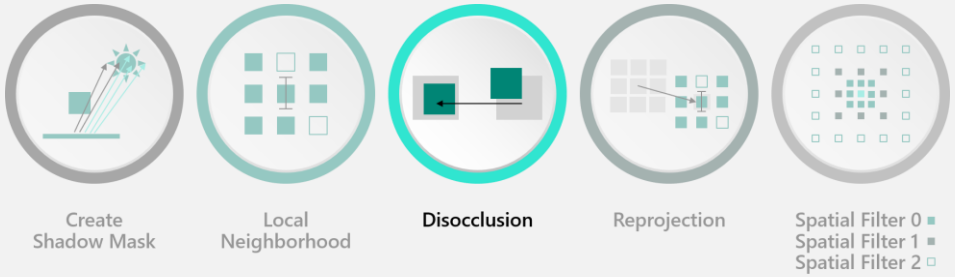
Create Disocclusion Mask

- Identify regions of the screen without history data
- Calculate the expected depth in the previous frame using a reprojection matrix (intermediate matrix calculations should be ideally performed using double precision):

$$Reprojection = ViewProjection^{-1} * PreviousViewProjection$$

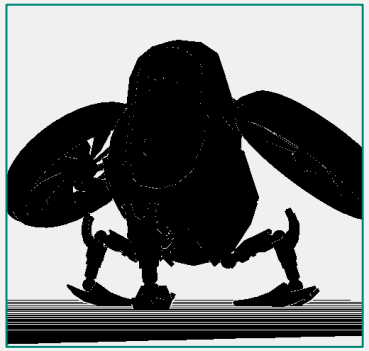
- Compare that with the depth buffer value from last frame, reprojected using per pixel motion vectors
- Check if the reprojection to the last frame belongs to the same surface by comparing the depth values

Building blocks



Create Disocclusion Mask

- Mismatching depth values are treated as a disocclusion
- Use adaptive depth thresholds to counter false disocclusion at grazing angles

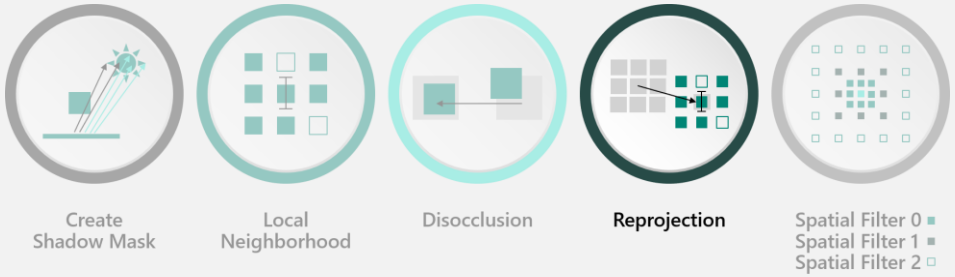


Constant depth threshold

Adaptive depth threshold

Disocclusion Mask

Building blocks



Reprojection

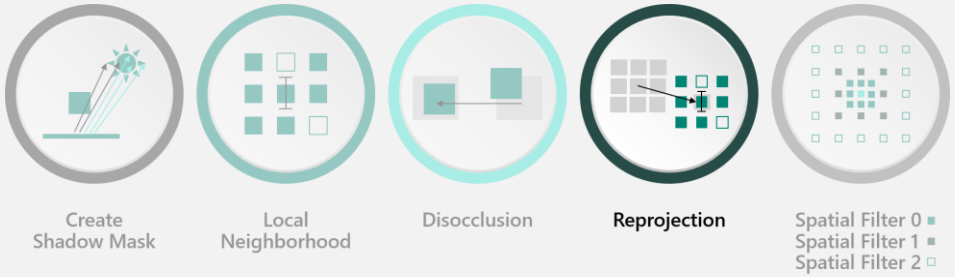
- Use disocclusion information to determine if we can reuse the history moments
- Reproject using per pixel motion vectors and sample the history moments
- Use those to update the current temporal variance which is used to guide the spatial filtering
- Additional variance boost for more aggressive spatial filtering when dealing with low temporal sample counts

Without variance boost

With variance boost

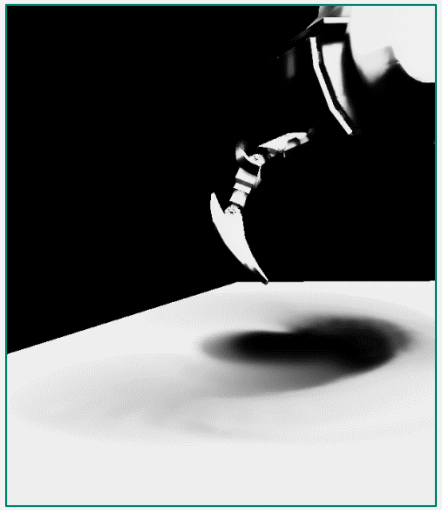
Temporal Variance

Building blocks



Reprojection

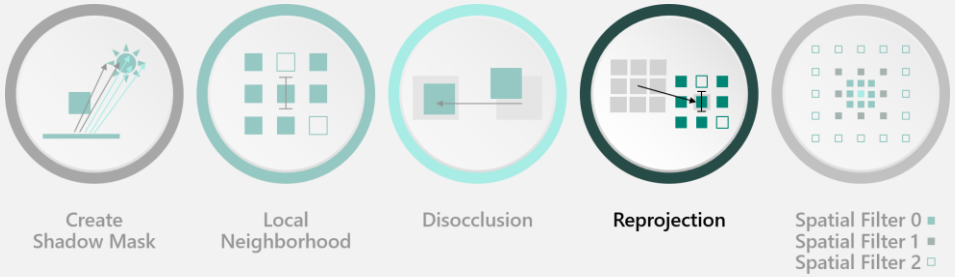
- Moving shadows are a challenging problem
- Notice how the shadow pretty much disappears as many temporal samples get incorrectly blended
- Neighborhood clamping the history values helps with obtaining a much more responsive filter and preserves the shadow details under motion



Naïve blending

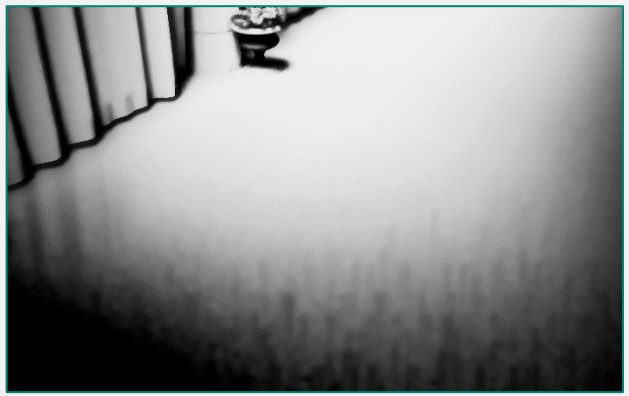
Neighborhood clamping

Building blocks



Reprojection

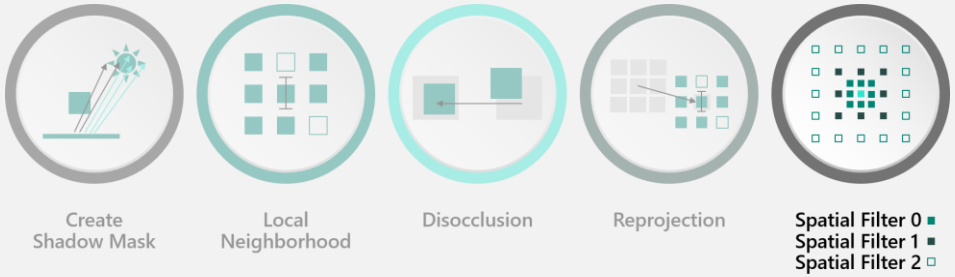
- The clamped history information is merged with the current frame using an exponential moving average blend
- A low blend factor heavily weights the current frame value resulting in a responsive but unstable filter,
- A high blend factor heavily weights the clamped history value resulting in a stable but unresponsive filter
- Our implementation adjusts the blend factor based on the number of accumulated history samples



Constant
blend factor

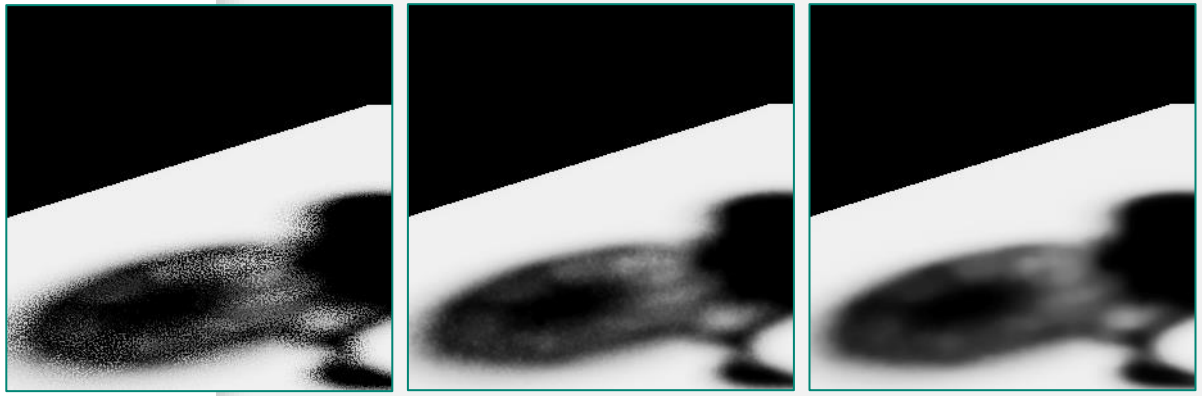
Adaptive
blend factor

Building blocks



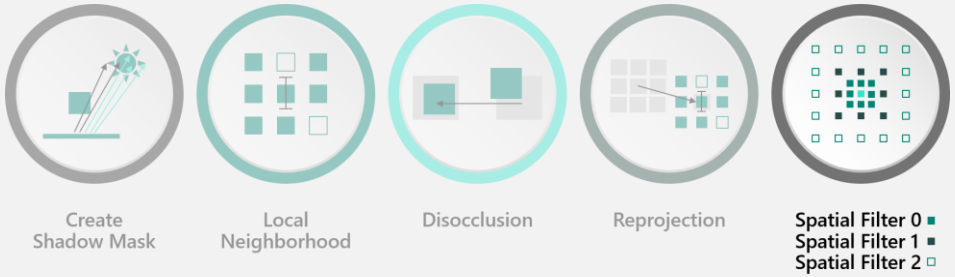
Spatial Filter

- We're left with a (mostly) temporally stable image
- Use consecutive spatial filters—edge aware á-trous wavelet (EAW) filters—to remove the remaining spatial noise
- Spatial Filter 0: EAW with 1-pixel stride
- Spatial Filter 1: EAW with 2-pixel stride
- Spatial Filter 2: EAW with 4-pixel stride
- The results of Spatial Filter 0 are reused as history input for the next frame



No EAW pass | Single EAW pass | 3 EAW passes

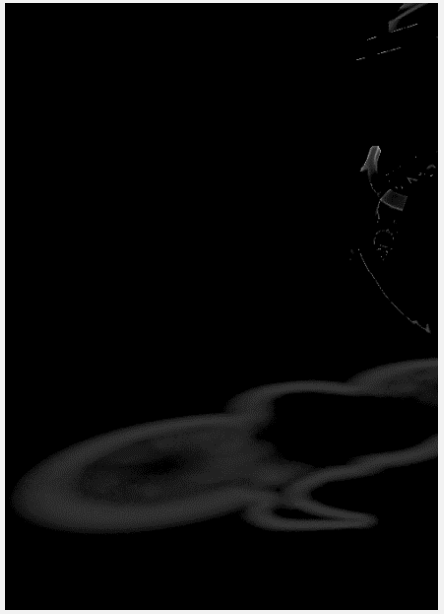
Building blocks



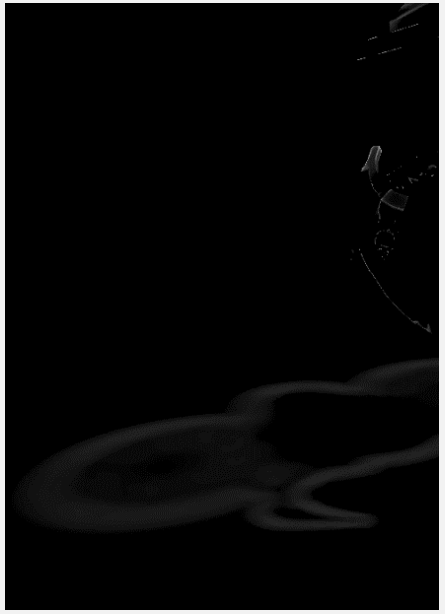
Spatial Filter

- Track variance to avoid over blurring in regions where variance is already low enough
- With each successive pass the variance cools off

Variance after 1 blur pass



Variance after 2 blur passes



Variance after 3 blur passes

Building blocks

TAA

- TAA is an important step to finish stabilizing and cleaning up the image after the denoiser has run and is recommended to get the best results both in terms of image quality and stability
- Notice how the render without TAA exhibits artifacts at the edges where the spatial blur fails to find matching samples
- TAA helps eliminate those artifacts and overall improves the image stability

Without TAA

With TAA

Performance improvements

Key observations

- For performance reasons we likely only ever shoot one ray per pixel anyway
- There may be large areas of the screen where the denoiser does not perform any work

Single ray per pixel

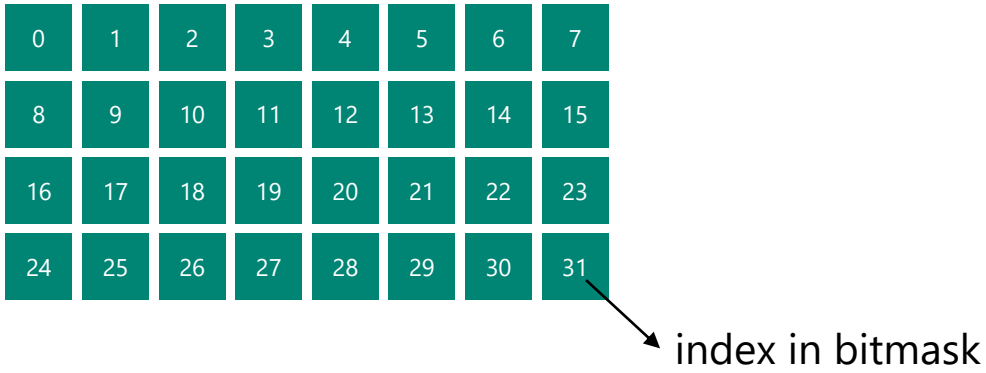
- At 1 sample per pixel the shadow values become either 0 or 1
- Thus, calculating the second order moment (used for variance) becomes indifferent from calculating the first order moment (mean)
- As a result, we can drop half of the calculations in the local neighborhood pass

$$mean \mathrel{+}= value * weight$$
$$variance \mathrel{+}= value^2 * weight$$

becomes:

$$moments \mathrel{+}= value * weight$$

Create Shadow Mask



Single ray per pixel

Compress the occlusion data to 1 bit per pixel

- A single 32-bit unit can store the data of an entire 8x4 region of pixels
- This nicely matches the native SIMD size of 32 on RDNA
- The ray traversal shader creates a single bool for “in light” or “in shadow”
- Naturally, that flag is kept in a scalar 32bit uint anyway
- In theory, this idea may be extended to multiple rays per pixel and just use one bit per sample. But this makes some of the later optimizations more complex

Create Shadow Mask

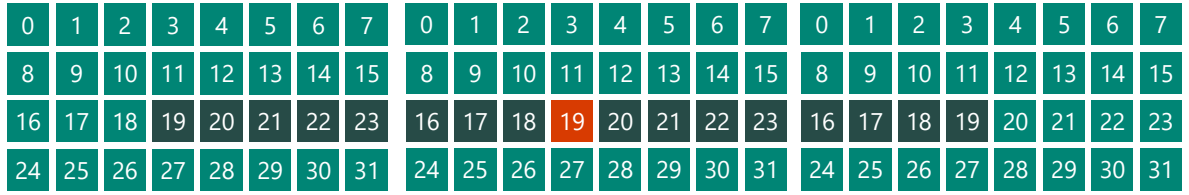
```
uint GetBitMaskFromPixelPosition(uint2 pixel_pos) {
    int lane_index = (pixel_pos.y % 4) * 8 + (pixel_pos.x % 8);
    return (1u << lane_index);
}

const uint lane_mask = hit_light
    ? GetBitMaskFromPixelPosition(dispatch_thread_id) : 0;

ShadowMask[tile_index] = WaveActiveBitOr(lane_mask);
```

Single ray per pixel

Creating the mask:

- All that would be left to do is writing that uint into a buffer
- Can not guarantee this on PC, as the mapping from WaveGetLaneIndex() to SV_DispatchThreadID is not enforced
- So, we must manually calculate the bit location on PC and use a WaveActiveBitOr()
- The thread group size of a RayGen shader is not defined on DXR1.0. Thus, we write to a R8 target and compress the mask in a separate pass

Local Neighborhood



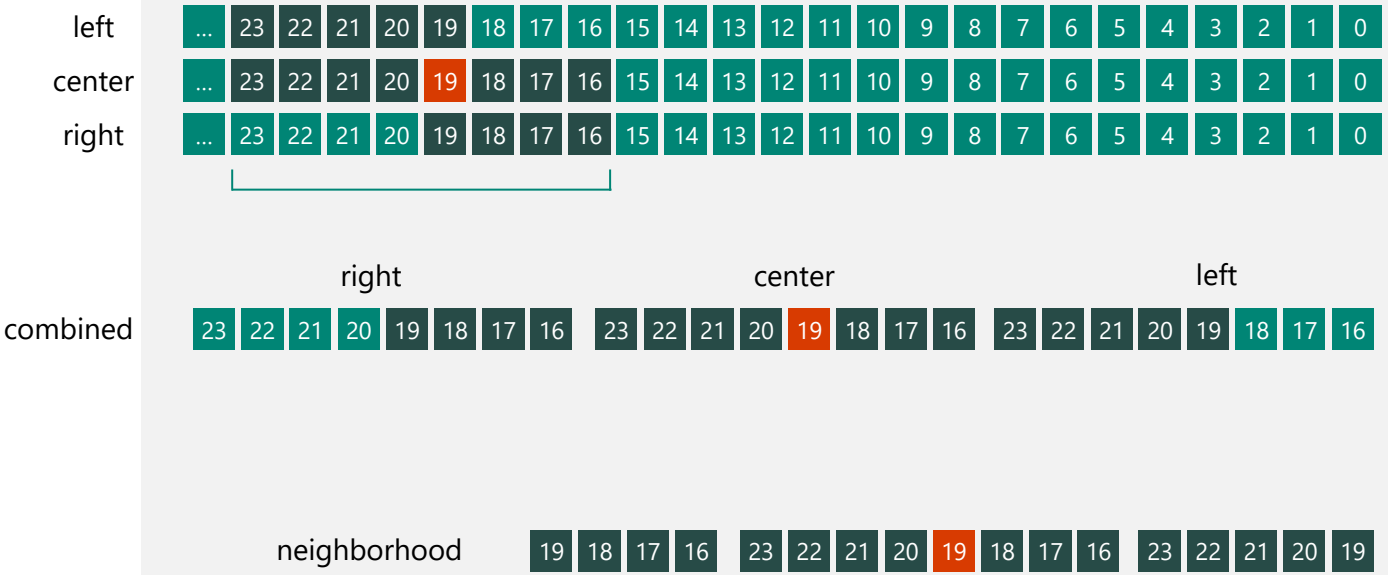
Single ray per pixel

- Compute the horizontal neighborhood from that bit mask
- Brief example on how this is done around the pixel at bit position 19 colored in orange, which is part of the center bit mask

Local Neighborhood

```
// First extract the 8 bits of our row
// in each of the neighboring tiles
const uint row_base_index = (did.y % 4) * 8;
const uint left = (left_tile >> row_base_index) & 0xFF;
const uint center = (center_tile >> row_base_index) & 0xFF;
const uint right = (right_tile >> row_base_index) & 0xFF;

// Combine them into a single mask containing
// [left, center, right] from least significant
// to most significant bit
const uint combined = left | (center << 8) | (right << 16);

// Make sure our pixel is at bit position 9
// to get the highest contribution from the filter kernel
const uint bit_index_in_row = (did.x % 8);

// Shift out bits to the right,
// so the center bit ends up at bit 9.
const uint neighborhood = combined >> bit_index_in_row;

moment += (mask & neighborhood) ? KernelWeight(i) : 0;
```

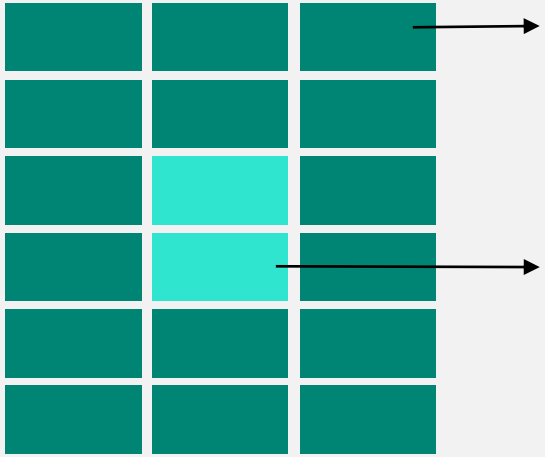
Single ray per pixel



- Iterate over these 17 bits to combine the horizontal local neighborhood
- Utilizes scalar loads and requires relatively low VGPR usage for some high ALU counts

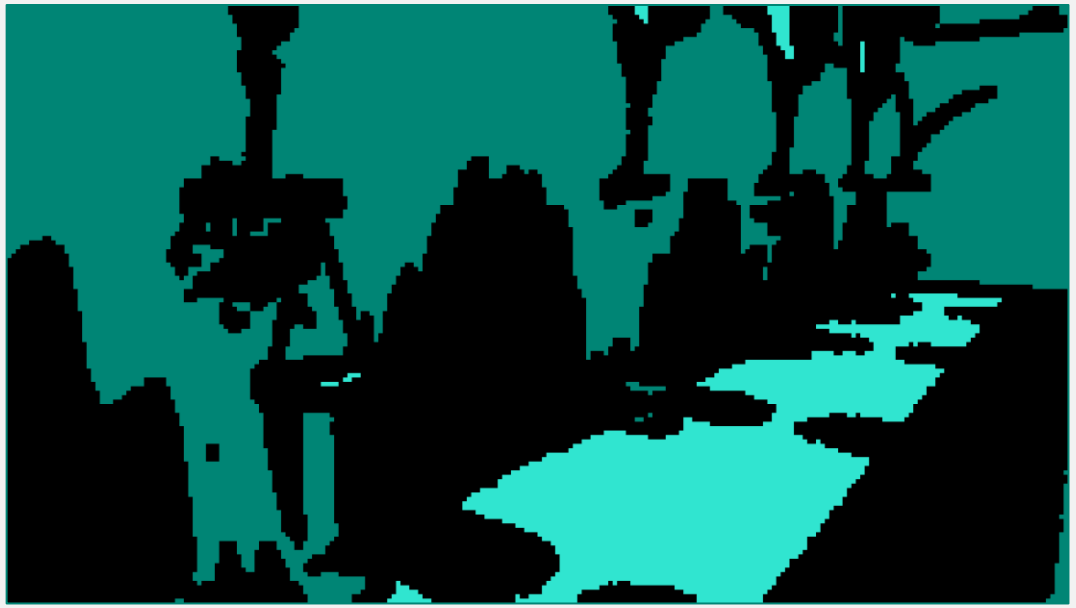
Tile classification

- **Idea:** quickly identify regions that we may skip in the denoiser
- Seeing gains in the 25–30% range, highly dependent on the scene
- The denoisers run on 8x8 tiles
- The neighborhood stretches another 8 in each direction
- Thus, we check a 24x24 region if it is entirely in light or entirely in shadow using 18 scalar loads:



32-bit bitmask covering 8x4 pixel tile

8x8 pixel center tile = target of the classification



Tile classification

```
label_01F8:
  s_add_i32     s59, s51, s0
  s_add_i32     s51, s51, 3
  s_max_i32     s57, s59, 0
  s_min_i32     s14, s57, s1
  s_add_i32     s57, s59, 1
  s_add_i32     s59, s59, 2
  s_mul_i32     s60, s27, s14
  s_max_i32     s57, s57, 0
  s_max_i32     s59, s59, 0
  s_add_i32     s58, s60, s74
  s_add_i32     s62, s60, s75
  s_min_i32     s61, s57, s1
  s_lshl_b32    s57, s58, 2
  s_lshl_b32    s58, s62, 2
  s_min_i32     s62, s59, s1
  s_add_i32     s59, s60, s90
  s_mul_i32     s63, s27, s61
  s_buffer_load_dword  s57, s[20:23], s57
  s_mul_i32     s60, s27, s62
  s_lshl_b32    s59, s59, 2
  s_add_i32     s61, s63, s74
  s_add_i32     s62, s63, s75
  s_add_i32     s63, s63, s90
  s_add_i32     s66, s60, s90
  s_add_i32     s64, s60, s74
  s_add_i32     s65, s60, s75
  s_buffer_load_dword  s60, s[20:23], s58
  s_buffer_load_dword  s59, s[20:23], s59
  s_lshl_b32    s58, s61, 2
  s_lshl_b32    s61, s62, 2
  s_lshl_b32    s67, s64, 2
  s_lshl_b32    s78, s65, 2
  s_lshl_b32    s62, s63, 2
  s_buffer_load_dword  s65, s[20:23], s58
  s_buffer_load_dword  s64, s[20:23], s61
  s_lshl_b32    s66, s66, 2
  s_waitcnt     lgkmcnt(0)
  s_or_b32      s56, s57, s56
  s_and_b32     s15, s57, s15
  s_buffer_load_dword  s63, s[20:23], s62
  s_buffer_load_dword  s62, s[20:23], s67
  s_buffer_load_dword  s61, s[20:23], s78
  s_buffer_load_dword  s58, s[20:23], s66
  s_or_b32      s56, s60, s56
  s_and_b32     s15, s60, s15
  s_or_b32      s56, s59, s56
  s_and_b32     s15, s59, s15
  s_or_b32      s56, s65, s56
  s_and_b32     s15, s65, s15
  s_or_b32      s56, s64, s56
  s_and_b32     s12, s64, s15
  s_waitcnt     lgkmcnt(0)
  s_or_b32      s56, s63, s56
  s_and_b32     s12, s63, s12
  s_or_b32      s56, s62, s56
  s_and_b32     s57, s62, s12
  s_or_b32      s15, s61, s56
  s_and_b32     s57, s61, s57
  s_or_b32      s56, s58, s15
  s_and_b32     s15, s58, s57
  s_cmp_eq_u32  s51, 4
  s_cbranch_scc0  label_01F8
```

Tile Classification

```
int2 tile_dimensions = int2(
        RoundedDivide(BufferDimensions.x, 8),
        RoundedDivide(BufferDimensions.y, 4));

// Load the entire region of masks in a scalar fashion
uint combined_or_mask = 0;
uint combined_and_mask = 0xFFFFFFFF;
for (int j = -2; j <= 3; ++j) {
    for (int i = -1; i <= 1; ++i) {
        int2 tile_index = base_tile + int2(i, j);
        tile_index = clamp(tile_index, 0, tile_dimensions- 1);
        const uint linear_tile_index = Flatten(tile_index);
        const uint shadow_mask = RaytracerResult[linear_tile_index];

        combined_or_mask = combined_or_mask | shadow_mask;
        combined_and_mask = combined_and_mask & shadow_mask;
    }
}

all_in_light = combined_and_mask == 0xFFFFFFFFu;
all_in_shadow = combined_or_mask == 0u;
```

Spatial Filter

```
bool is_cleared;
bool all_in_light;
ReadTileMetaData(group_id, is_cleared, all_in_light);

[branch]
if (is_cleared) {
    Clear(all_in_light, …)
}
else {
    ApplyFilter(…);
}
```

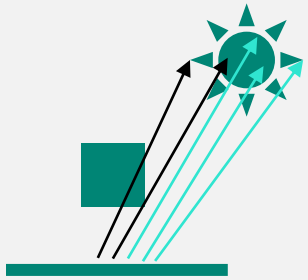
```
s_buffer_load_dword  s0, s[32:35], s14
s_waitcnt     lgkmcnt(0)
s_and_b32     s1, s0, 1
s_cmp_eq_u32  s1, 0
s_cbranch_scc0  label_0914
```

Tile classification

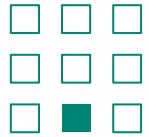
The three Spatial Filter passes now run off a tile meta data mask

- Scalar loads for clears
- Scalar branch for clear/not clear

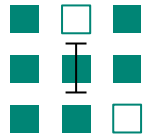
Merge passes



Create Shadow Mask

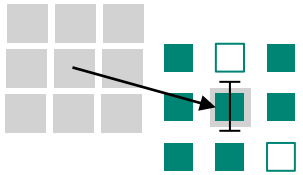


Classify Tiles



Local Neighborhood

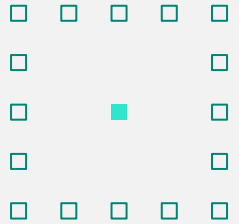
Disocclusion



Reprojection

Spatial Filter 0

Spatial Filter 1



Spatial Filter 2

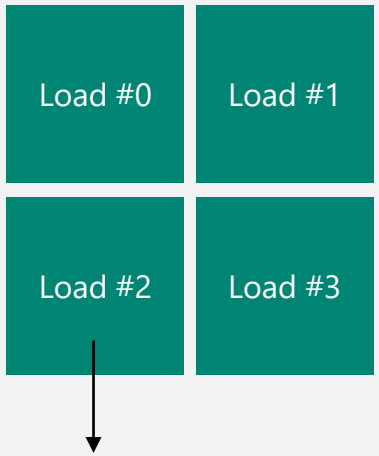
- Using the bit mask makes this feasible for local neighborhood—store the intermediate values in groupshared memory
- Allows to remove a few transient resources, including a depth buffer copy, the local neighborhood and the tile classification target
- Resulted in around 10% performance improvement

Spatial Filter

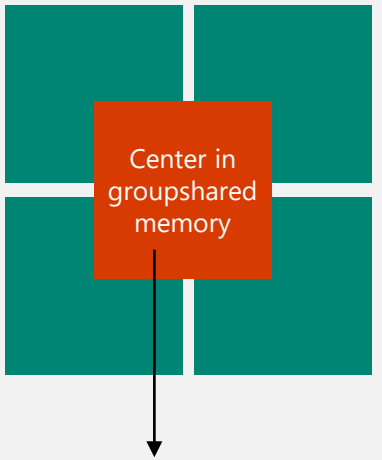
```
int2 offset[4] = {
    int2(0, 0), int2(8, 0),
    int2(0, 8), int2(8, 8)
};

min16float3 normals[4];
min16float2 input[4];
float depth[4];

did -= 4;
for (int i = 0; i < 4; ++i) {
    LoadWithOffset(dispatch_thread_id,
        offset[i], normals[i], input[i], depth[i]);
}

for (int j = 0; j < 4; ++j) {
    StoreWithOffset(group_thread_id,
        offset[j], normals[j], input[j], depth[j]);
}

// Center threads in groupshared memory
group_thread_id += 4;
DenoiseFromGroupSharedMemory(…);
```

Groupshared memory

- Need 16x16 region in groupshared memory
- Requires 4 loads in total
- Each loads an 8x8 region for best cache utilization



Load 8x8 pixels into groupshared memory



Offset indices by 4 to center the work tile in groupshared memory

Spatial Filter

```
Texture2D<float16_t2>  InputBuffer; // first and second order moments

uint PackFloat16(float16_t2 v) {
    uint2 p = f32tof16(float2(v));
    return p.x | (p.y << 16);
}

groupshared uint g_shared_input[16][16];

…
g_shared_input[idx.y][idx.x] = PackFloat16(input);
```

Packed math

- Two 16-bit values from the same texture fetch will share a register
  (if the **image_load** uses the **d16** flag)
- **v_pk_** instructions can work on both values in that register at once
- That register will be written to groupshared memory directly → **PackFloat16** will be a no-op

Spatial Filter

```
image_load v[8:9], v[4:5], s[24:31] dmask:0x7 dim:SQ_RSRC_IMG_2D unorm d16 // normals - 4x FP16
image_load v[10:11], [v6, v5], s[24:31] dmask:0x7 dim:SQ_RSRC_IMG_2D unorm d16
image_load v[12:13], [v4, v7], s[24:31] dmask:0x7 dim:SQ_RSRC_IMG_2D unorm d16
image_load v[14:15], v[6:7], s[24:31] dmask:0x7 dim:SQ_RSRC_IMG_2D unorm d16
image_load v16, v[4:5], s[32:39] dmask:0x3 dim:SQ_RSRC_IMG_2D unorm d16  // input moments - 2x FP16
image_load v17, [v6, v5], s[32:39] dmask:0x3 dim:SQ_RSRC_IMG_2D unorm d16
image_load v19, [v4, v7], s[32:39] dmask:0x3 dim:SQ_RSRC_IMG_2D unorm d16
image_load v18, v[6:7], s[32:39] dmask:0x3 dim:SQ_RSRC_IMG_2D unorm d16
image_load v23, v[4:5], s[24:31] dmask:0x1 dim:SQ_RSRC_IMG_2D unorm // depths - FP32
image_load v22, [v6, v5], s[24:31] dmask:0x1 dim:SQ_RSRC_IMG_2D unorm
image_load v26, [v4, v7], s[24:31] dmask:0x1 dim:SQ_RSRC_IMG_2D unorm
image_load v7, v[6:7], s[24:31] dmask:0x1 dim:SQ_RSRC_IMG_2D unorm
image_load v4, v[2:3], s[24:31] dmask:0x1 dim:SQ_RSRC_IMG_2D unorm
s_waitcnt vmcnt(0)
v_pk_fma_f16 v6, v8, 2.0, -1.0 op_sel_hi:[1,0,0]
v_pk_fma_f16 v8, v8, 2.0, -1.0 op_sel_hi:[1,0,0]
v_pk_fma_f16 v5, v10, 2.0, -1.0 op_sel_hi:[1,0,0]
v_pk_fma_f16 v10, v10, 2.0, -1.0 op_sel_hi:[1,0,0]
v_pk_fma_f16 v21, v14, 2.0, -1.0 op_sel_hi:[1,0,0]
v_pk_fma_f16 v14, v14, 2.0, -1.0 op_sel_hi:[1,0,0]
v_pk_fma_f16 v20, v12, 2.0, -1.0 op_sel_hi:[1,0,0]
v_pack_b32_f16 v24, v5, v6
v_pack_b32_f16 v6, v5, v6 op_sel:[1,1,0]
v_pack_b32_f16 v5, v15, v13
v_pk_fma_f16 v13, v12, 2.0, -1.0 op_sel_hi:[1,0,0]
v_pack_b32_f16 v25, v21, v20
v_pack_b32_f16 v30, v21, v20 op_sel:[1,1,0]
v_pack_b32_f16 v9, v11, v9
v_pk_mul_f16 v11, v24, v24
```

Packed math

```
v_pk_mul_f16 v27, v25, v25
v_pk_fma_f16 v24, v9, 2.0, -1.0 op_sel_hi:[1,0,0]
v_pk_fmac_f16 v11, v6, v6
v_pk_fma_f16 v6, v5, 2.0, -1.0 op_sel_hi:[1,0,0]
v_pk_fmac_f16 v27, v30, v30
v_pk_fmac_f16 v11, v24, v24
v_pk_fmac_f16 v27, v6, v6
v_rsq_f16 v5, v11
v_rsq_f16 v9, v27
v_rsq_f16_sdwa v5, v11 dst_sel:WORD_1 dst_unused:UNUSED_PRESERVE src0_sel:WORD_1
v_rsq_f16_sdwa v9, v27 dst_sel:WORD_1 dst_unused:UNUSED_PRESERVE src0_sel:WORD_1
v_pk_mul_f16 v11, v24, v5
v_pk_mul_f16 v10, v10, v5 op_sel_hi:[1,0]
v_pk_mul_f16 v8, v8, v5 op_sel:[0,1]
v_pk_mul_f16 v6, v6, v9
v_pk_mul_f16 v15, v14, v9 op_sel_hi:[1,0]
v_mov_b32_sdwa v21, v11 dst_sel:DWORD dst_unused:UNUSED_PAD src0_sel:WORD_0
v_mov_b32_sdwa v11, v11 dst_sel:DWORD dst_unused:UNUSED_PAD src0_sel:WORD_1
v_pk_mul_f16 v27, v13, v9 op_sel:[0,1]
v_mov_b32_sdwa v14, v6 dst_sel:DWORD dst_unused:UNUSED_PAD src0_sel:WORD_0
v_mov_b32_sdwa v6, v6 dst_sel:DWORD dst_unused:UNUSED_PAD src0_sel:WORD_1
ds_write2st64_b32 v1, v16, v23 offset1:4 // depth and input moments
ds_write2st64_b32 v0, v17, v22 offset1:4
ds_write2st64_b32 v0, v18, v7 offset0:2 offset1:6
ds_write2st64_b32 v1, v19, v26 offset0:2 offset1:6
ds_write2st64_b32 v1, v8, v11 offset0:8 offset1:12 // normals
ds_write2st64_b32 v0, v10, v21 offset0:8 offset1:12
ds_write2st64_b32 v1, v27, v6 offset0:10 offset1:14
ds_write2st64_b32 v0, v15, v14 offset0:10 offset1:14
```

Miscellaneous

Use single channel mask per light instead of a combined 4-channel format for 4 lights

- Avoids a read-modify-write when writing the final output (unused channels would still be fetched)
- Yielded roughly another 5%

```
image_load v0, v[34:35], s[16:23] dmask:0x1 dim:SQ_RSRC_IMG_2D unorm
s_waitcnt vmcnt(0) // this can stall for hundreds of cycles
v_mov_b32 v1, v0
v_mov_b32 v2, v0
v_mov_b32 v3, v0
image_store v[0:3], v[34:35], s[0:7] dmask:0xf dim:SQ_RSRC_IMG_2D unorm glc


v_mov_b32 v54, v57
v_mov_b32 v55, v57
v_mov_b32 v56, v57
image_store v[54:57], v[34:35], s[16:23] dmask:0xf dim:SQ_RSRC_IMG_2D unorm glc
```

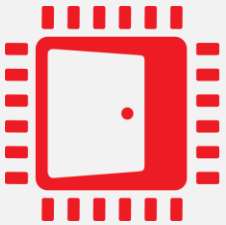
Closing notes

Future ideas

- Trace and denoise at half resolution then upsample
- Guide raytracing
  - Include information from a VRS image to reduce number of rays
  - Compare history frames, shoot fewer rays where denoiser converged
  - Reuse tile meta data
- Denoiser may as well be applied to AO

Acknowledgements

- Guillaume Boissé
- Marcus Rogowsky
- Matthäus Chajdas
- Rys Sommefeldt
- Colin Riley
- Kenneth Mitchell
- Marco Weber
- Gareth Thomas
- Jonas Gustavsson
- Nicolas Thibieroz



AMD GPUOpen | Let's build everything...

Soon to be found here—next to our reflection denoiser:
https://gpuopen.com/fidelityfx-denoiser/

Disclaimer and attribution

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION

Microsoft