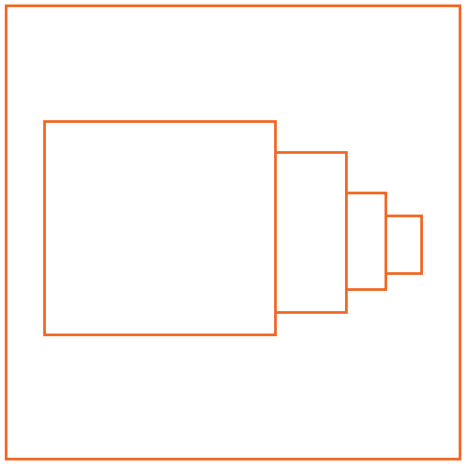
FFX SSSR

DOMINIK BAUMEISTER
TOBIAS FAST

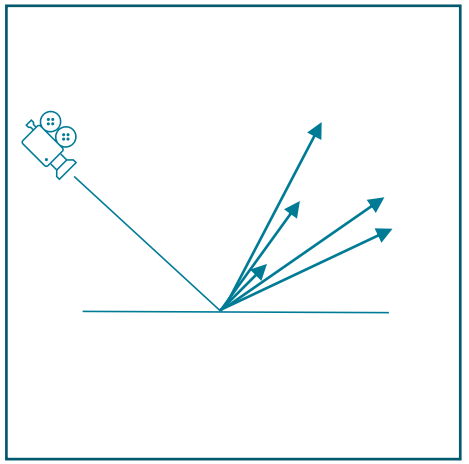
STOCHASTIC SCREEN SPACE REFLECTIONS

- Based on industry leading algorithm
- Hierarchical depth buffer traversal kernel
- Glossy reflections via ray jittering
- Variable Rate Traversal
  - from full rate for mirror reflections
  - down to quarter rate for glossy reflections
- Support for D3D12 and Vulkan
- Shaders written in HLSL utilizing SM 6.0 wave-level operations

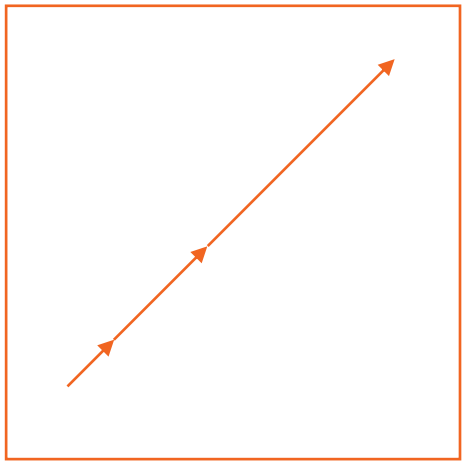
Create Depth Buffer Hierarchy (FidelityFX SPD)



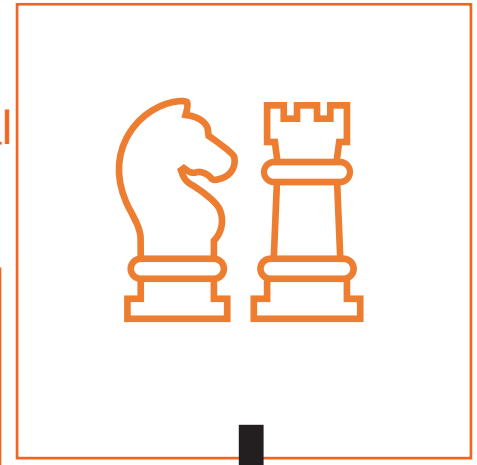
Ray Jitter

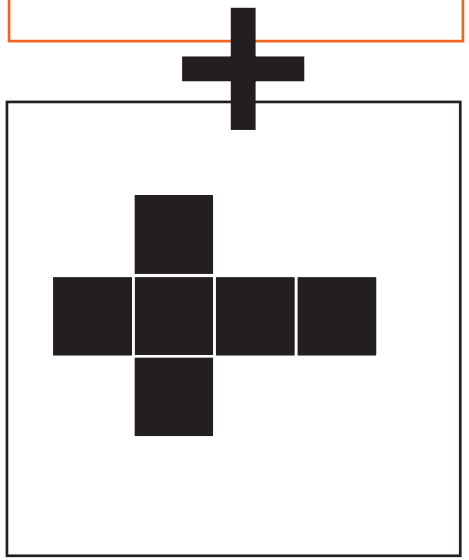


Hierarchical Depth Buffer Traversal (FidelityFX SSSR)



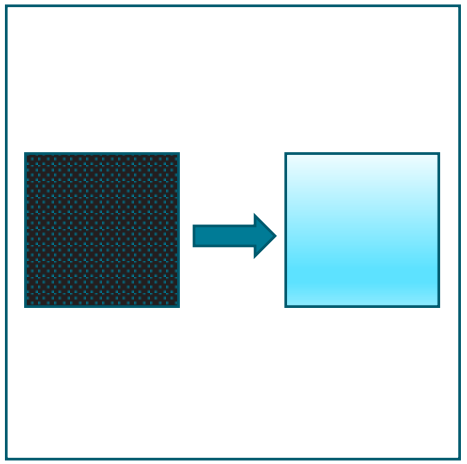
Screen Space Lookup





Environment Lookup

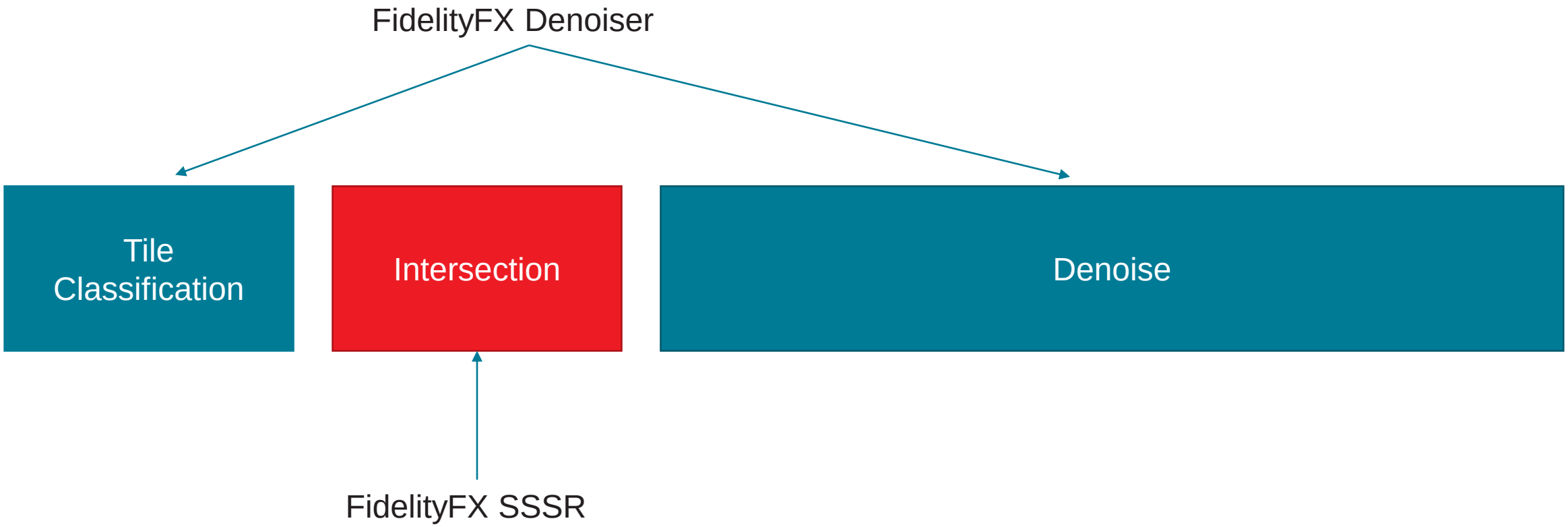
Denoise (FidelityFX Denoiser)



Stochastic Screenspace Reflections

PIPELINE

PIPELINE

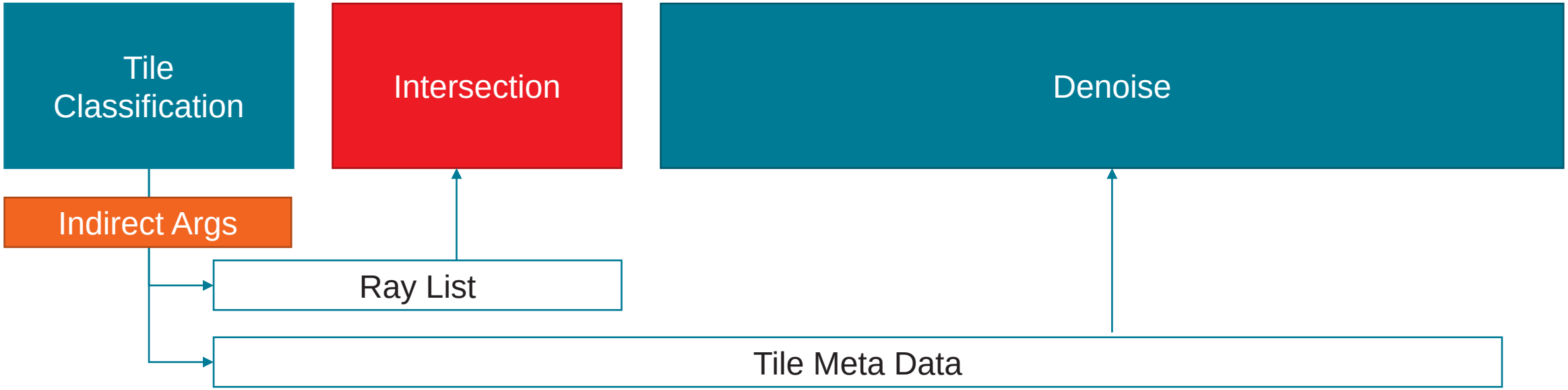

FidelityFX Denoiser
Tile Classification
Intersection
Denoise
FidelityFX SSSR

PIPELINE

PIPELINE


Tile Classification
Intersection
Denoise
Indirect Args
Ray List

PIPELINE

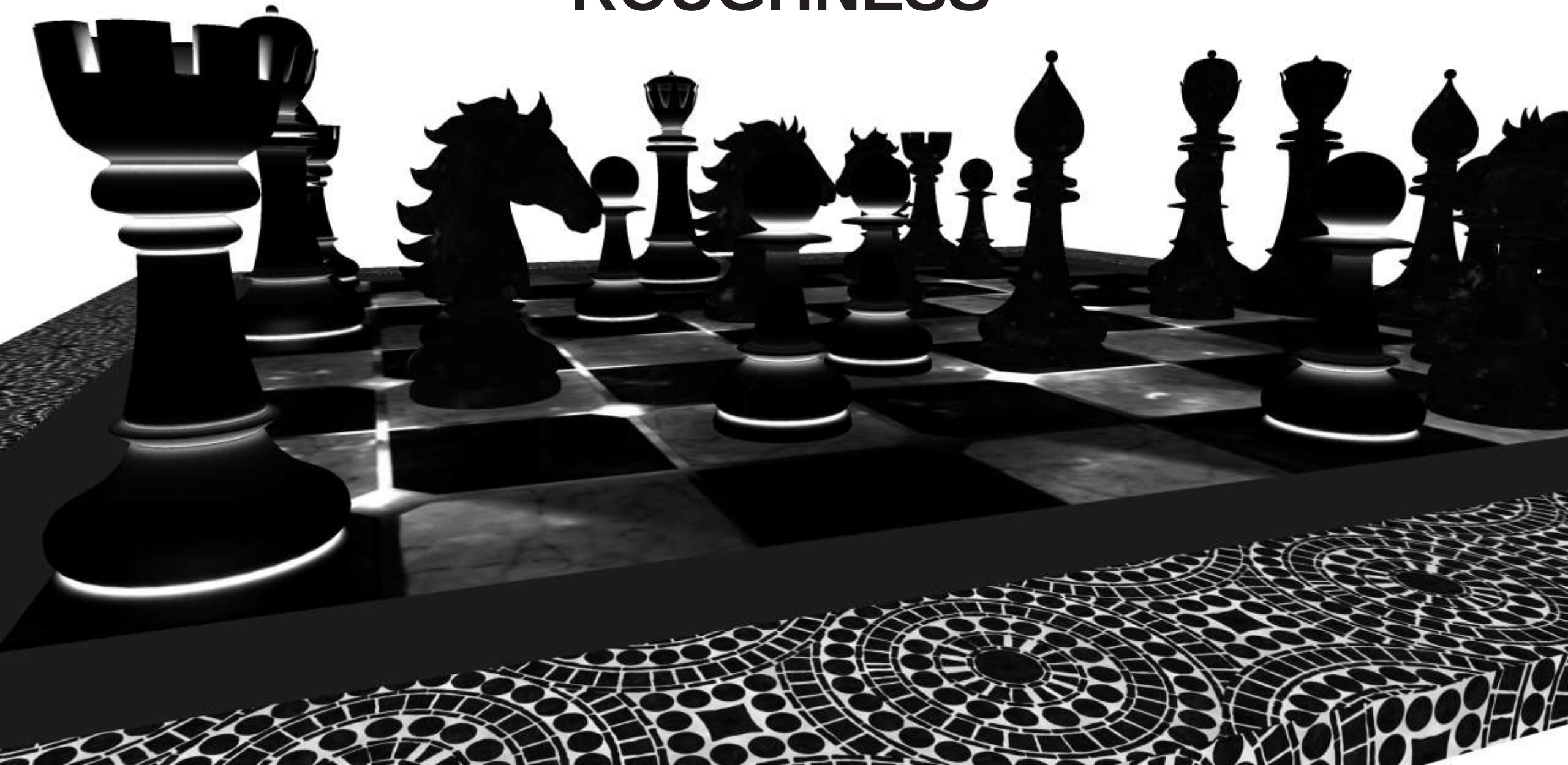

Tile Classification
Intersection
Denoise
Indirect Args
Ray List
Tile Meta Data

INPUTS

App side surfaces

- Depth Hierarchy (2x2 minimum)
  Check out FFX SPD on how to do this with a single compute pass
- Resolved scene
- Normals for current frame
- Roughness for current frame
- Environment map
- **Cleared** reflection target

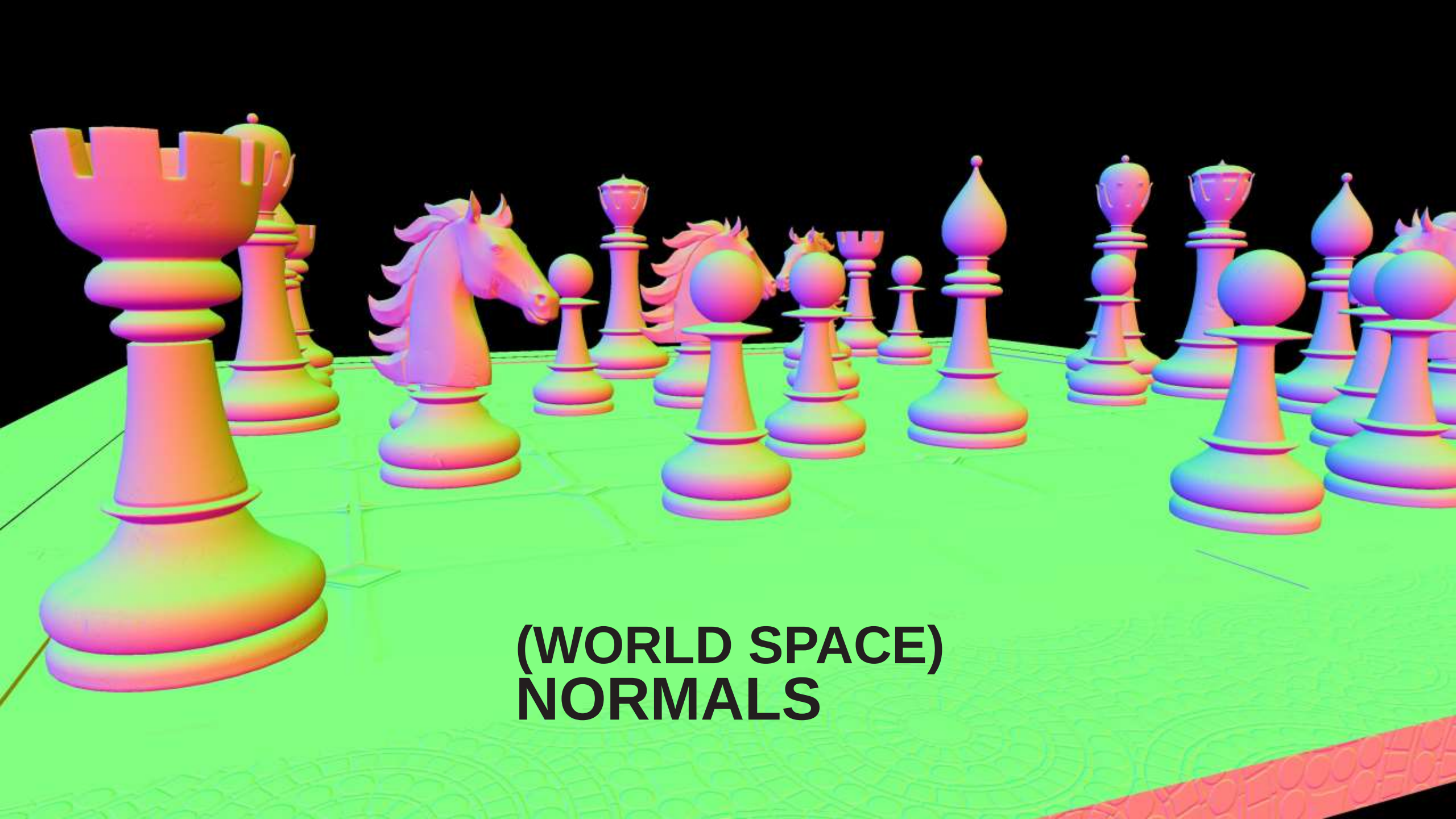
ROUGHNESS



MIN DEPTH HIERARCHY

MIN DEPTH HIERARCHY

MIN DEPTH HIERARCHY


(WORLD SPACE)
NORMALS


LIT SCENE

ENVIRONMENT RADIANCE

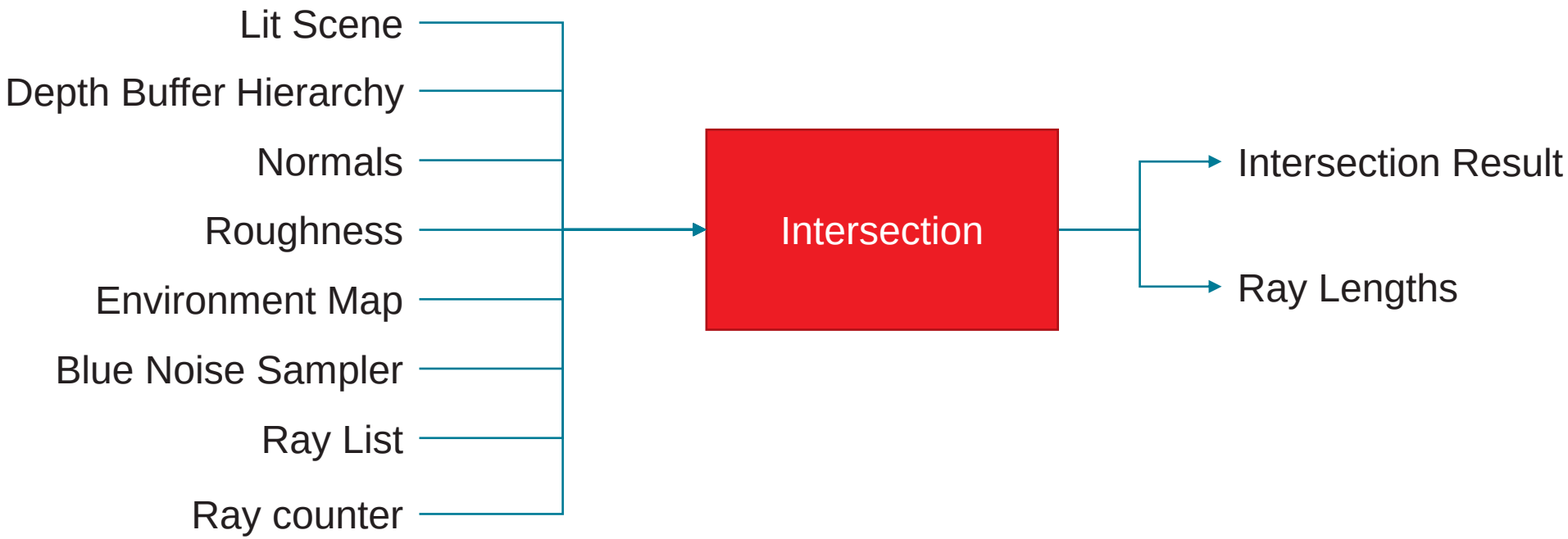
```
[numthreads(1, 1, 1)]
void main()
{
    uint ray_count = g_ray_counter[0];

    g_intersect_args[0] = (ray_count + 63) / 64;
    g_intersect_args[1] = 1;
    g_intersect_args[2] = 1;

    g_ray_counter[0] = 0;
    g_ray_counter[1] = ray_count;
}
```

INDIRECT ARGUMENTS

Yes, numthreads(**1**, **1**, **1**) …

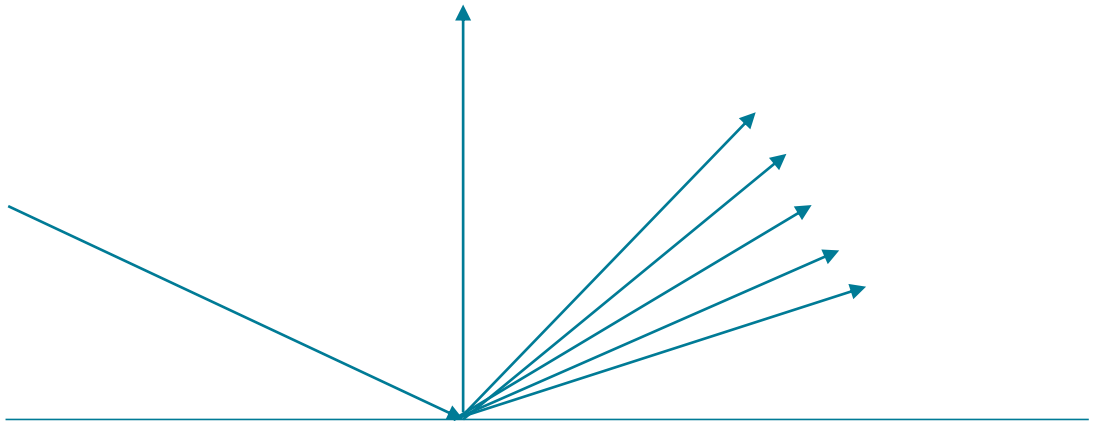
Indirect Args

INTERSECTION

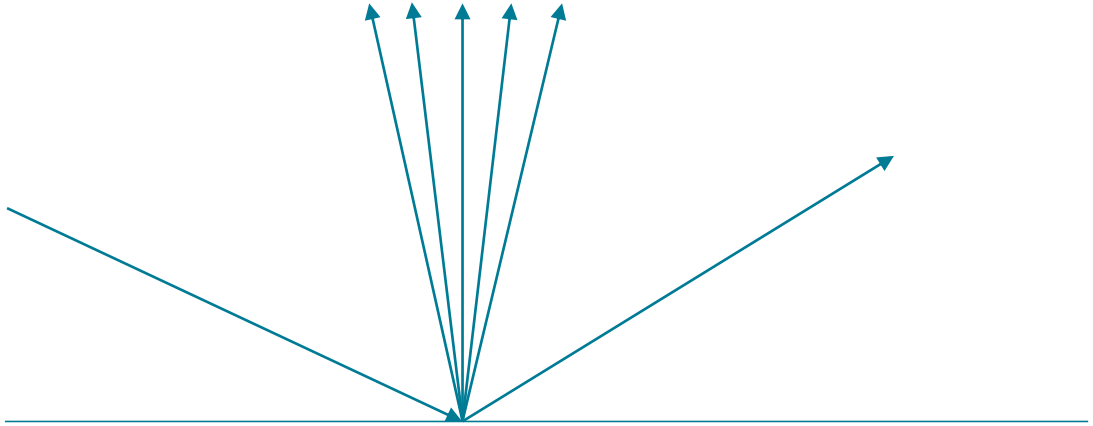


INTERSECTION – RAY CREATION

- Visible Normal Distribution Function (VNDF) - Sampling the GGX Distribution of Visible Normals
- Jitter normal and then reflect ray
- Use blue noise sampler and create permutations using golden ratio



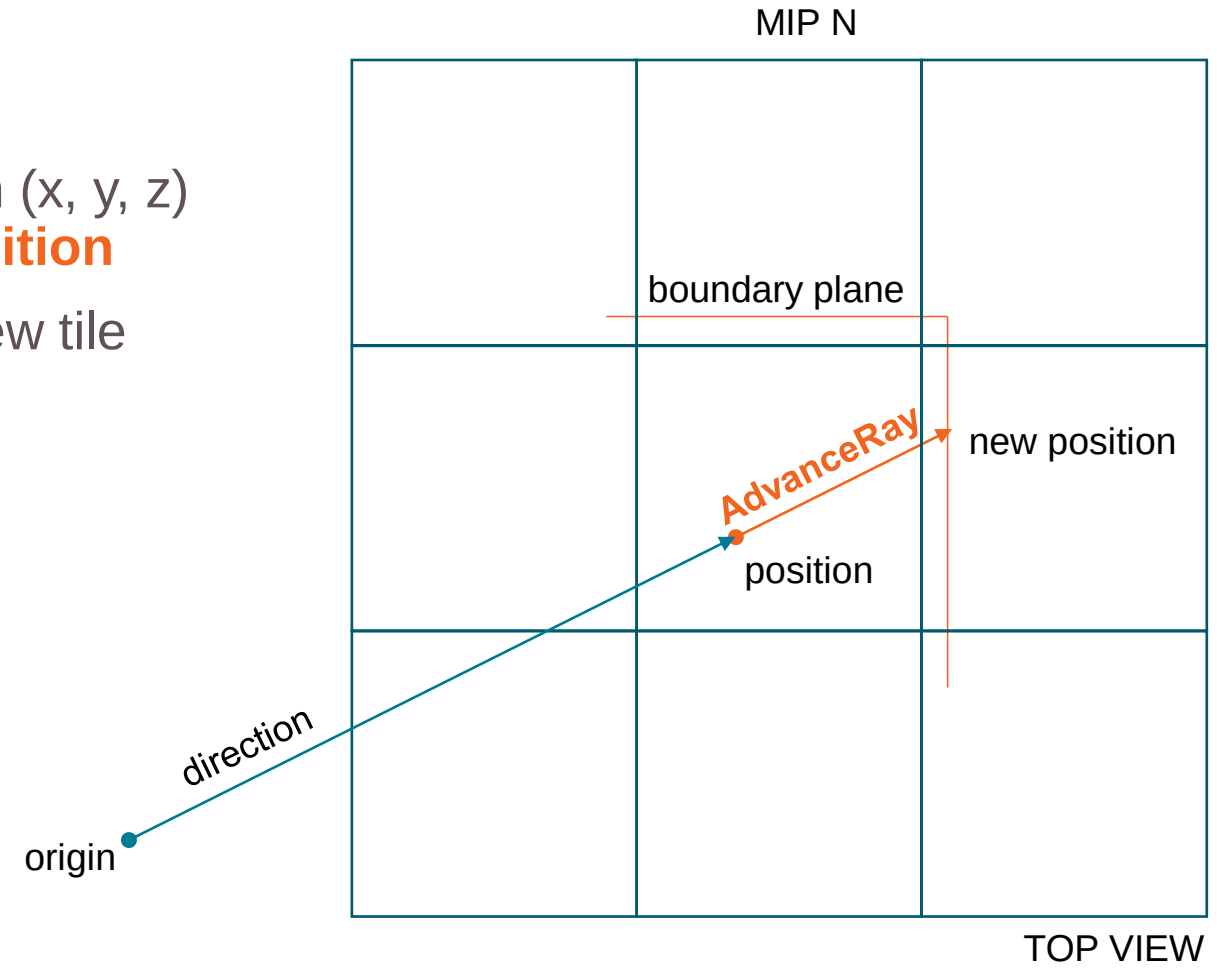
jitter ray directly



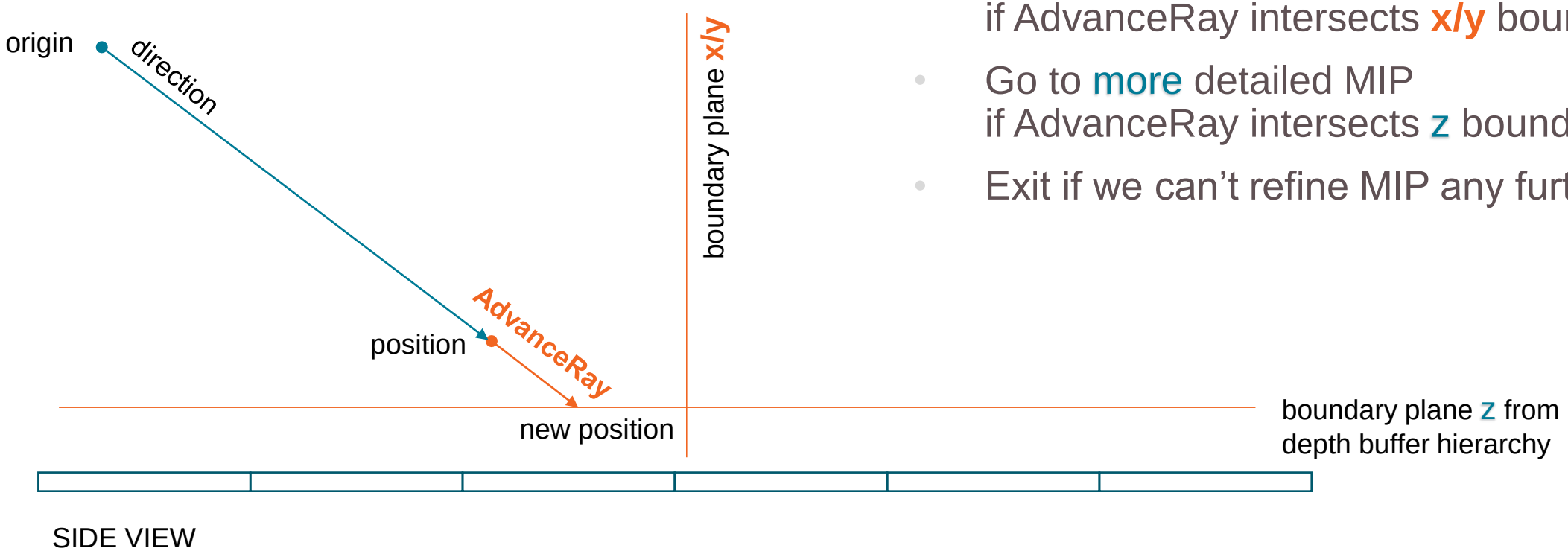
jitter normal (VNDF)

INTERSECTION – RAY MARCH

- Hierarchical ray march in screen space
- At each iteration create bounding planes in (x, y, z) depending on current **MIP** and current **position**
- Additional sub-pixel **offset** to guarantee new tile

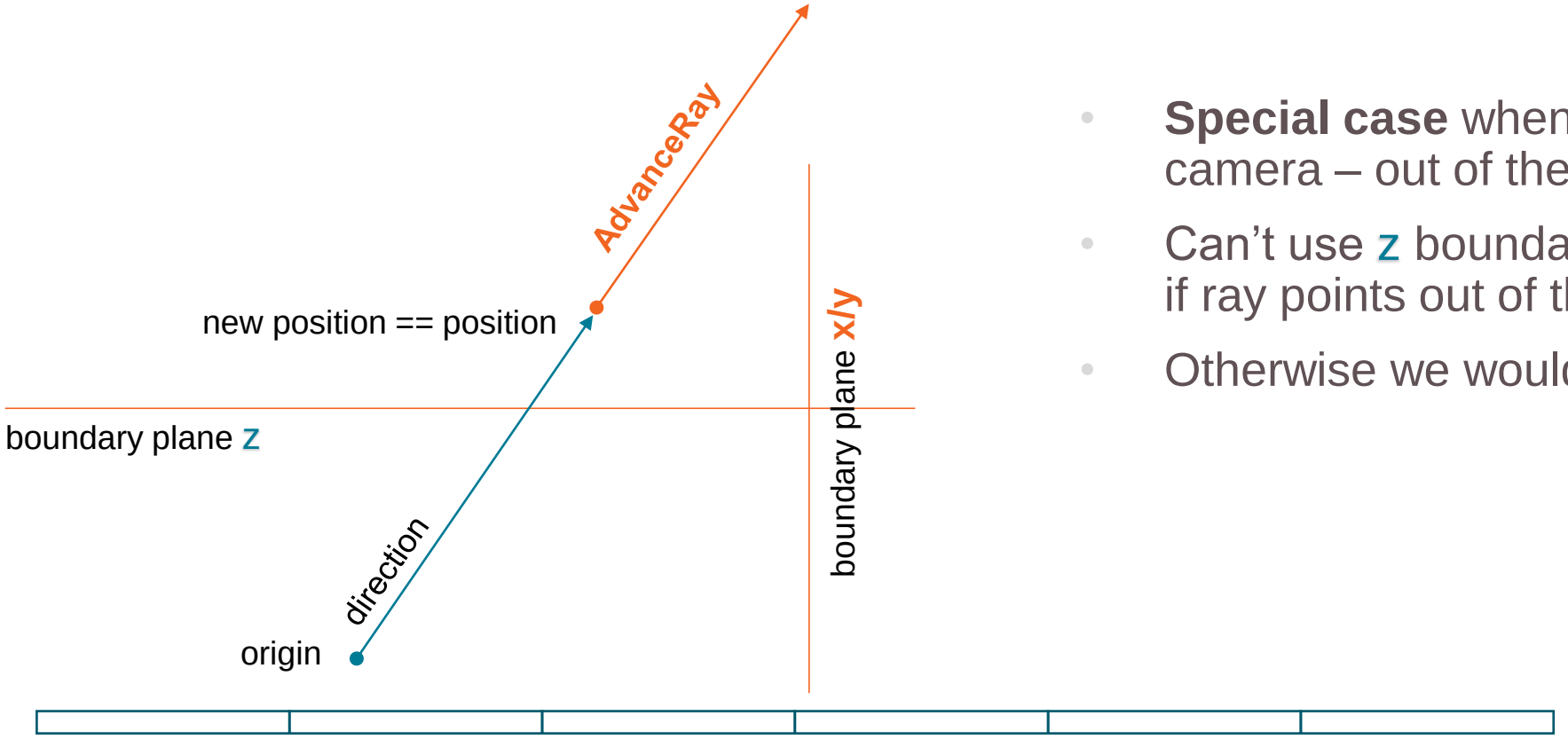


INTERSECTION – RAY MARCH



- Go to **less** detailed MIP
  if AdvanceRay intersects **x/y** boundary
- Go to more detailed MIP
  if AdvanceRay intersects z boundary
- Exit if we can't refine MIP any further

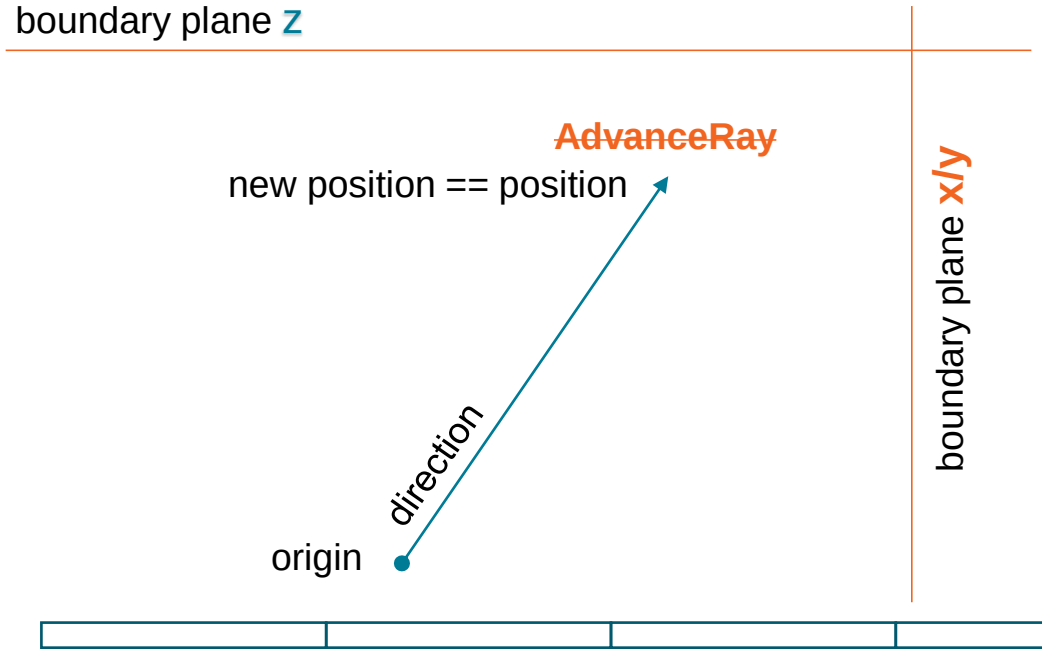
INTERSECTION – RAY MARCH



SIDE VIEW

- **Special case** when ray shoots towards the camera – out of the depth buffer
- Can't use z boundary
  if ray points out of the depth buffer
- Otherwise we would clamp too soon

INTERSECTION – RAY MARCH



SIDE VIEW

- **Special case** when ray shoots towards the camera – out of the depth buffer
- Can't advance ray due to risk of tunneling if position is already below z boundary
  → Go to more detailed MIP

INTERSECTION – RAY MARCH

```
while (i < max_traversal_intersections
    && current_mip >= most_detailed_mip
    && !exit_due_to_low_occupancy) {

    float2 current_mip_position = current_mip_resolution * position.xy;
    float surface_z = LoadDepth(current_mip_position, current_mip);
```

```
// Load from depth buffer hierarchy to
retrieve z boundary
```

INTERSECTION – RAY MARCH

```
while (i < max_traversal_intersections
    && current_mip >= most_detailed_mip
    && !exit_due_to_low_occupancy) {

    float2 current_mip_position = current_mip_resolution * position.xy;
    float surface_z = LoadDepth(current_mip_position, current_mip);
    bool skipped_tile = AdvanceRay(origin, direction, inv_direction,
        current_mip_position, current_mip_resolution_inv,
        floor_offset, uv_offset, surface_z, position, current_t);
```

```
// Advance current position and current t
    Also builds x/y planes, more later
```

INTERSECTION – RAY MARCH

```
while (i < max_traversal_intersections
    && current_mip >= most_detailed_mip
    && !exit_due_to_low_occupancy) {

    float2 current_mip_position = current_mip_resolution * position.xy;
    float surface_z = LoadDepth(current_mip_position, current_mip);
    bool skipped_tile = AdvanceRay(origin, direction, inv_direction,
        current_mip_position, current_mip_resolution_inv,
        floor_offset, uv_offset, surface_z, position, current_t);

    current_mip += skipped_tile ? 1 : -1;
    current_mip_resolution *= skipped_tile ? 0.5 : 2;
    current_mip_resolution_inv *= skipped_tile ? 2 : 0.5;
    ++i;

                                    // Handle MIP changes
```

INTERSECTION – RAY MARCH

```
while (i < max_traversal_intersections
    && current_mip >= most_detailed_mip
    && !exit_due_to_low_occupancy) {

    float2 current_mip_position = current_mip_resolution * position.xy;
    float surface_z = LoadDepth(current_mip_position, current_mip);
    bool skipped_tile = AdvanceRay(origin, direction, inv_direction,
        current_mip_position, current_mip_resolution_inv,
        floor_offset, uv_offset, surface_z, position, current_t);

    current_mip += skipped_tile ? 1 : -1;
    current_mip_resolution *= skipped_tile ? 0.5 : 2;
    current_mip_resolution_inv *= skipped_tile ? 2 : 0.5;
    ++i;

    exit_due_to_low_occupancy = !is_mirror &&
        WaveActiveCountBits(true) <= min_traversal_occupancy;
}       // Small optimization: Exit loop if only a few threads still run it
```

```
bool AdvanceRay(…) {
    // Create boundary planes
    float2 xy_plane = floor(current_mip_position) + floor_offset;
    xy_plane = xy_plane * current_mip_resolution_inv + uv_offset;
    float3 boundary_planes = float3(xy_plane, surface_z);
```

```
bool AdvanceRay(…) {
    // Create boundary planes
    float2 xy_plane = floor(current_mip_position) + floor_offset;
    xy_plane = xy_plane * current_mip_resolution_inv + uv_offset;
    float3 boundary_planes = float3(xy_plane, surface_z);
    // Intersect ray with the half box that is pointing away from the ray origin.
       o + d * t = p' => t = (p' - o) / d
    float3 t = (boundary_planes - origin) * inv_direction;
```

```
bool AdvanceRay(…) {
    // Create boundary planes
    float2 xy_plane = floor(current_mip_position) + floor_offset;
    xy_plane = xy_plane * current_mip_resolution_inv + uv_offset;
    float3 boundary_planes = float3(xy_plane, surface_z);
    // Intersect ray with the half box that is pointing away from the ray origin.
       o + d * t = p' => t = (p' - o) / d
    float3 t = (boundary_planes - origin) * inv_direction;

    // Prevent using z plane when shooting out of the depth buffer.
    t.z = direction.z > 0 ? t.z : SSR_FLOAT_MAX;
```

```
bool AdvanceRay(…) {
    // Create boundary planes
    float2 xy_plane = floor(current_mip_position) + floor_offset;
    xy_plane = xy_plane * current_mip_resolution_inv + uv_offset;
    float3 boundary_planes = float3(xy_plane, surface_z);
    // Intersect ray with the half box that is pointing away from the ray origin.
       o + d * t = p' => t = (p' - o) / d
    float3 t = (boundary_planes - origin) * inv_direction;

    // Prevent using z plane when shooting out of the depth buffer.
    t.z = direction.z > 0 ? t.z : SSR_FLOAT_MAX;

    // Choose nearest intersection with a boundary.
    float t_min = min(min(t.x, t.y), t.z);
```

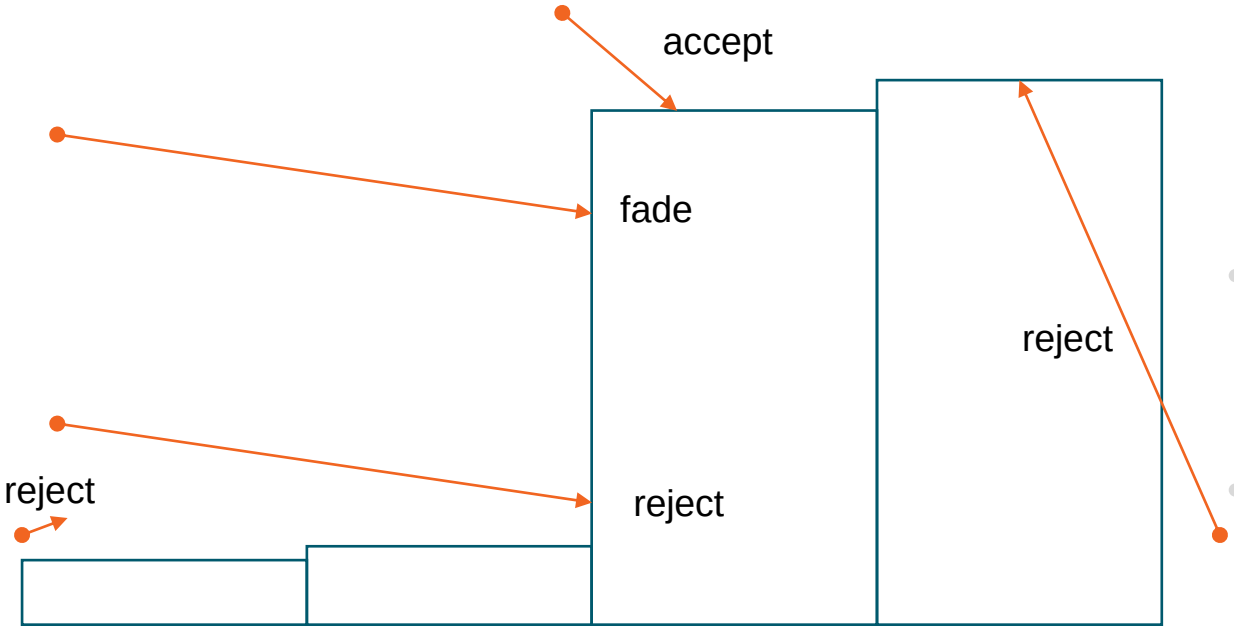
```
bool AdvanceRay(…) {
    // Create boundary planes
    float2 xy_plane = floor(current_mip_position) + floor_offset;
    xy_plane = xy_plane * current_mip_resolution_inv + uv_offset;
    float3 boundary_planes = float3(xy_plane, surface_z);
    // Intersect ray with the half box that is pointing away from the ray origin.
       o + d * t = p' => t = (p' - o) / d
    float3 t = (boundary_planes - origin) * inv_direction;

    // Prevent using z plane when shooting out of the depth buffer.
    t.z = direction.z > 0 ? t.z : SSR_FLOAT_MAX;

    // Choose nearest intersection with a boundary.
    float t_min = min(min(t.x, t.y), t.z);

    // Smaller z means closer to the camera.
    bool above_surface = surface_z > position.z;
```

```
bool AdvanceRay(…) {
    // Create boundary planes
    float2 xy_plane = floor(current_mip_position) + floor_offset;
    xy_plane = xy_plane * current_mip_resolution_inv + uv_offset;
    float3 boundary_planes = float3(xy_plane, surface_z);
    // Intersect ray with the half box that is pointing away from the ray origin.
       o + d * t = p' => t = (p' - o) / d
    float3 t = (boundary_planes - origin) * inv_direction;

    // Prevent using z plane when shooting out of the depth buffer.
    t.z = direction.z > 0 ? t.z : SSR_FLOAT_MAX;

    // Choose nearest intersection with a boundary.
    float t_min = min(min(t.x, t.y), t.z);

    // Smaller z means closer to the camera.
    bool above_surface = surface_z > position.z;
    // Decide if we had to clamp the ray at the surface.
    bool skipped_tile = t_min != t.z && above_surface;
```

```
bool AdvanceRay(…) {
    // Create boundary planes
    float2 xy_plane = floor(current_mip_position) + floor_offset;
    xy_plane = xy_plane * current_mip_resolution_inv + uv_offset;
    float3 boundary_planes = float3(xy_plane, surface_z);
    // Intersect ray with the half box that is pointing away from the ray origin.
       o + d * t = p' => t = (p' - o) / d
    float3 t = (boundary_planes - origin) * inv_direction;

    // Prevent using z plane when shooting out of the depth buffer.
    t.z = direction.z > 0 ? t.z : SSR_FLOAT_MAX;

    // Choose nearest intersection with a boundary.
    float t_min = min(min(t.x, t.y), t.z);

    // Smaller z means closer to the camera.
    bool above_surface = surface_z > position.z;
    // Decide if we had to clamp the ray at the surface.
    bool skipped_tile = t_min != t.z && above_surface;

    // Make sure to only advance the ray if we're still above the surface.
    current_t = above_surface ? t_min : current_t;
```

```
bool AdvanceRay(…) {
    // Create boundary planes
    float2 xy_plane = floor(current_mip_position) + floor_offset;
    xy_plane = xy_plane * current_mip_resolution_inv + uv_offset;
    float3 boundary_planes = float3(xy_plane, surface_z);
    // Intersect ray with the half box that is pointing away from the ray origin.
       o + d * t = p' => t = (p' - o) / d
    float3 t = (boundary_planes - origin) * inv_direction;

    // Prevent using z plane when shooting out of the depth buffer.
    t.z = direction.z > 0 ? t.z : SSR_FLOAT_MAX;

    // Choose nearest intersection with a boundary.
    float t_min = min(min(t.x, t.y), t.z);

    // Smaller z means closer to the camera.
    bool above_surface = surface_z > position.z;
    // Decide if we had to clamp the ray at the surface.
    bool skipped_tile = t_min != t.z && above_surface;

    // Make sure to only advance the ray if we're still above the surface.
    current_t = above_surface ? t_min : current_t;
    position = origin + current_t * direction; // Advance ray

    return skipped_tile;
}
```

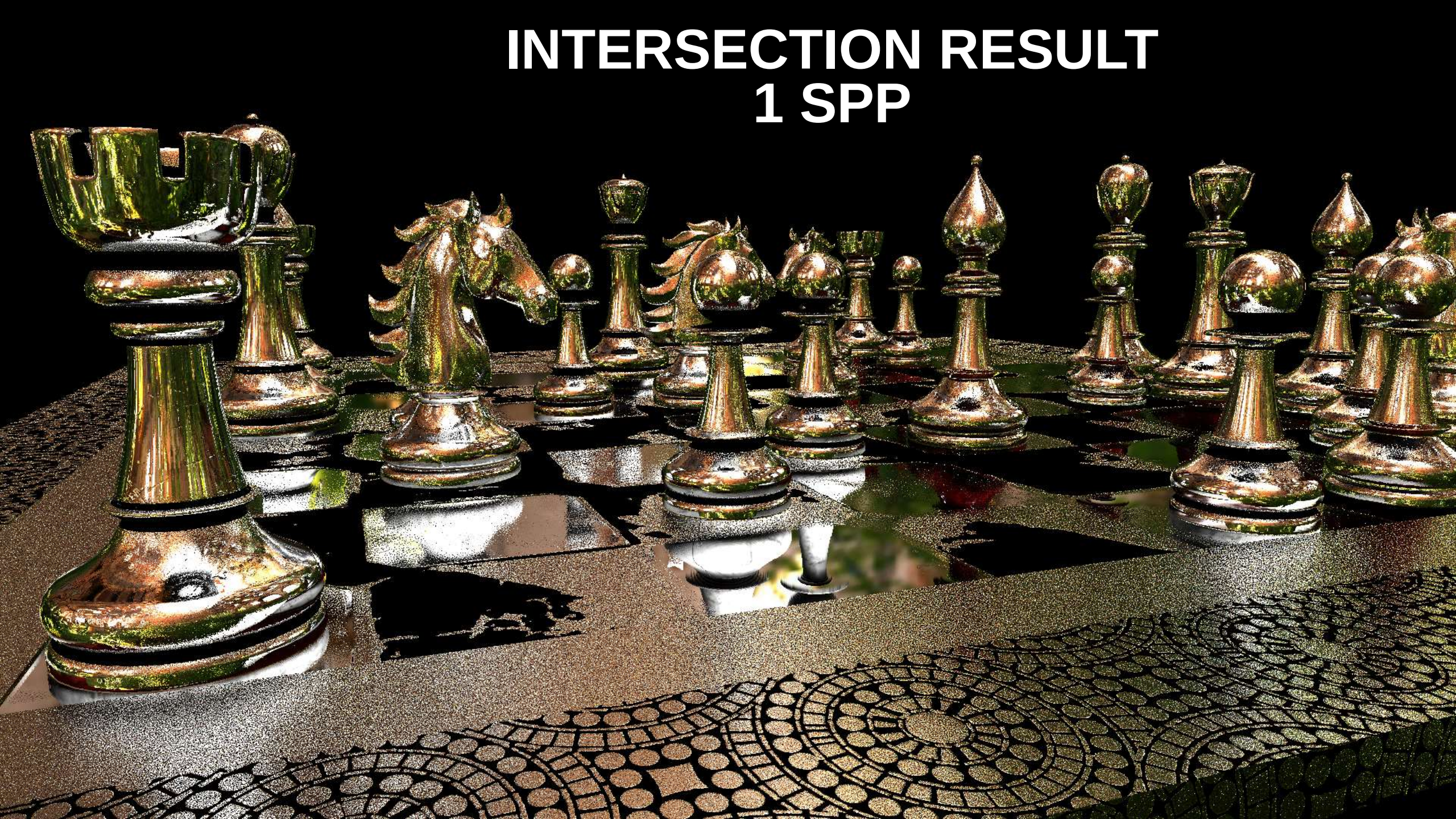
INTERSECTION – HIT VALIDATION



DEPTH BUFFER SIDE VIEW

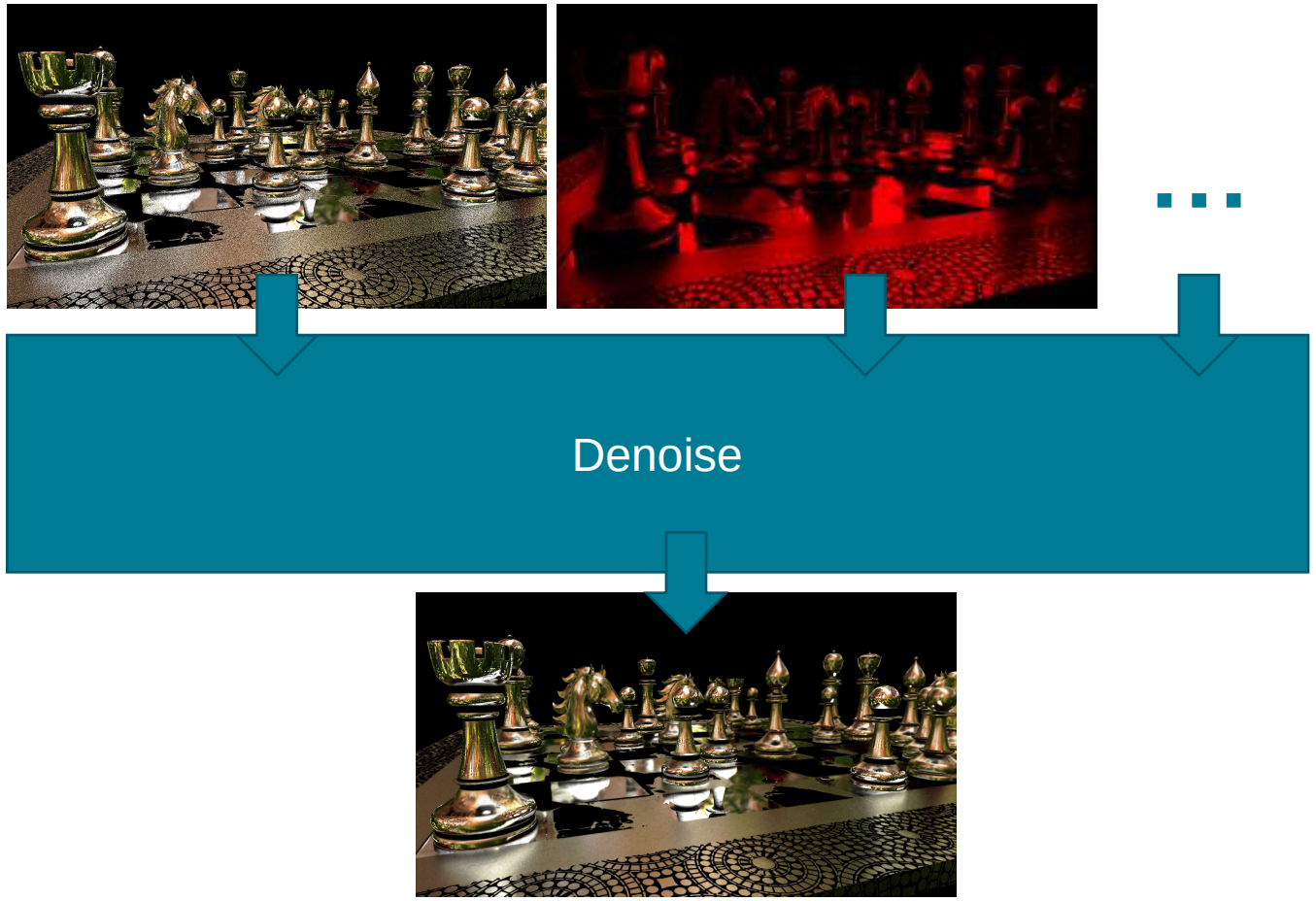
- Reject if
  - Hit outside the view frustum
  - Hit background
  - Hit the back of the surface
  - Ray didn't travel far enough
- Confidence based on
  - Hit distance below depth buffer
  - Hit closeness to border (vignette)
- Fade into environment map sample based on confidence

INTERSECTION RESULT
1 SPP



RAY LENGTHS
1 SPP

DENOISER




APPLIED

SOURCE

- GPUOpen Product Page
  https://gpuopen.com/FidelityFX-SSSR
- GitHub
  https://github.com/GPUOpen-Effects/FidelityFX-SSSR
- GPUOpen FFX Denoiser Product Page
  https://gpuopen.com/FidelityFX-Denoiser
- GPUOpen FFX SPD Product Page
  https://gpuopen.com/FidelityFX-SPD

REFERENCES

- Frostbite presentations on Stochastic Screen Space Reflections
  https://www.ea.com/frostbite/news/stochastic-screen-space-reflections
- EA Seed presentation on Hybrid Real-Time Rendering
  https://www.ea.com/seed/news/seed-dd18-presentation-slides-raytracing
- Eric Heitz` paper on VNDF
  http://jcgt.org/published/0007/04/01/
- Eric Heitz` paper on Blue Noise sampling
  https://eheitzresearch.wordpress.com/762-2/