SDFGI

Solving the accessible Global Illumination problem in Godot

Juan Linietsky (Godot Engine co-creator and technical lead)
juan@godotengine.org

Requirements

- Easy to use (no scene or object setup at import time, no setting up SDF, cards, lightmaps, etc). Ideally enable with one click, no set-up.
- Real-time (or at least fast updates).
- Good enough quality (no light leaks -or keep to minimum-).
- Supports both diffuse and reflected light.
- Supports light into transparent objects.
- Works as source of light for volumetric fog.
- Works in all hardware that supports Vulkan, even IGP.
- Can work in VR (so, using TAA is not required).

Sacrifices

- Not the best possible quality (high frequency GI missing, has to be compensated with screen space lighting).
- Poor dynamic object support (dynamic objects get light from environment, but don't contribute to it). Light blocking may be added to some extent in the future.
- Needs to use cascades.
- Limited amount of samples means small emissive objects are spotty.

Previous work

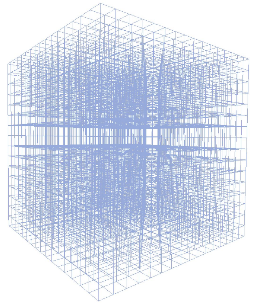
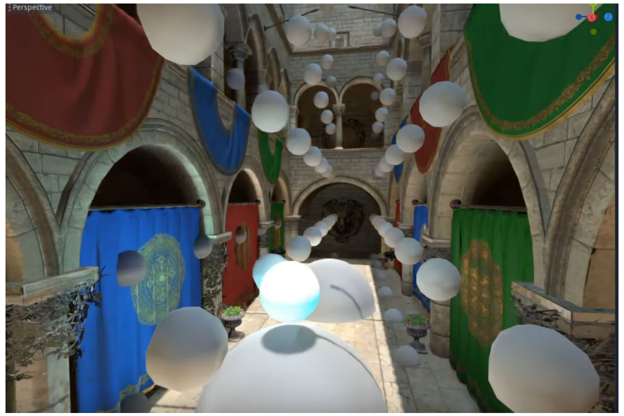
- Uses DDGI by Morgan McGuire as a base.
- Uses Signed Distance Fields generated with Jump Flood.
- The devil is in the details.

General Idea

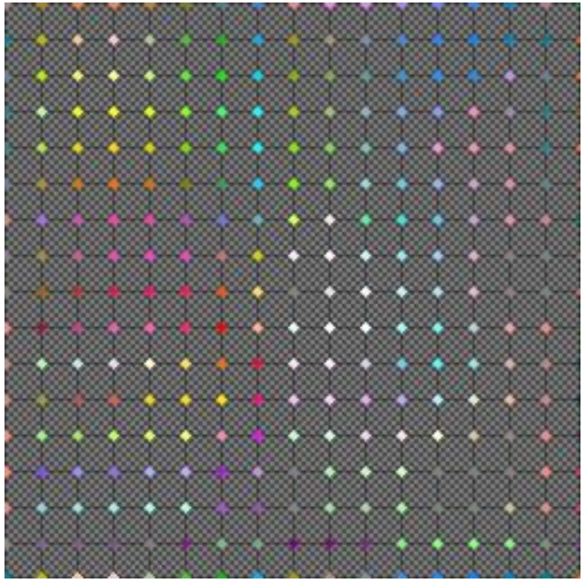
How light is stored and read

GOAL: Generate Irradiance and Occlusion Fields

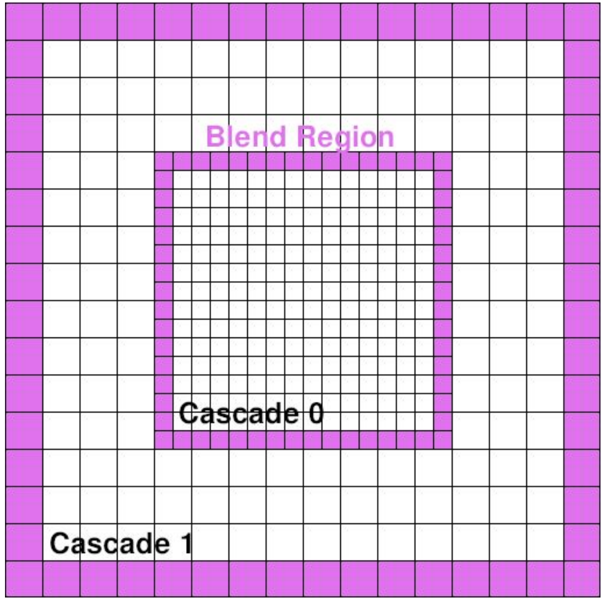
- Irradiance field allows computing diffuse light at a point in space.
- Fields are arranged in a 3D grid of light probes.
- To obtain the diffuse light at any point in space, the surface normal is probed into the eight surrounding probes, weighted against the closest ones.
- To avoid light leaks (probes across walls), the occlusion field is used.
- Cascades are used to blend GI between far and near distances (open world).

Cascade anatomy:



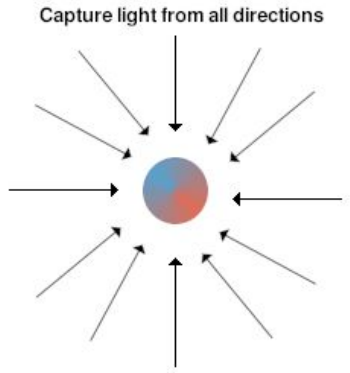
Cascade: 128^3 cells (Distance Field)
17^3 irradiance/occlusion probes
(must cover the volume)

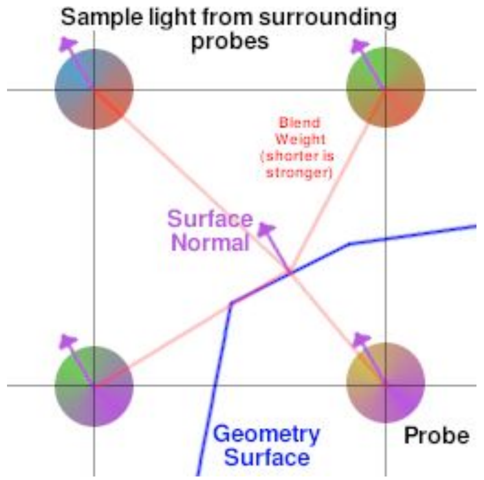


Cascade relationship and blend areas
(up to 8 cascades).

Light Probe Blending

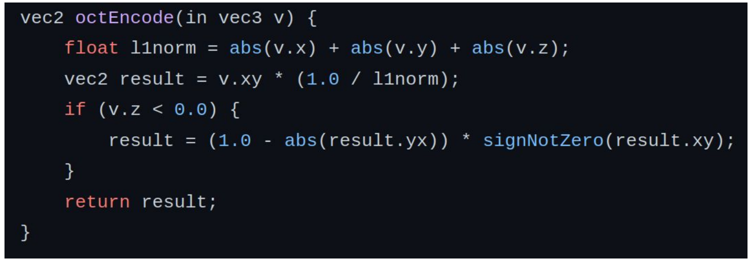
- Probes capture light from all direction (trace rays from probe, hit geometry or sky, read light at point and store it for that direction).
- When geometry is drawn, it finds the 8 probes surrounding it and blends the amount of lighting depending on the distance to each and the surface normal.




Sample light from surrounding probes
Blend Weight (shorter is stronger)
Surface Normal
Geometry Surface
Probe

Light Probe Storage

- Stored as a 5x5 octahedron encoded image.
- Similar to a 2x2 cubemap or a SHL2 in resolution, which is the standard.
- 25 pixels is a very good number for a Compute workgroup (more on this later).
- Cascades are light probes stored in a texture array layer (all probes of a cascade in a single layer), and need a 1px border for interpolation.



```
vec2 octEncode(in vec3 v) {
    float l1norm = abs(v.x) + abs(v.y) + abs(v.z);
    vec2 result = v.xy * (1.0 / l1norm);
    if (v.z < 0.0) {
        result = (1.0 - abs(result.yx)) * signNotZero(result.xy);
    }
    return result;
}
```

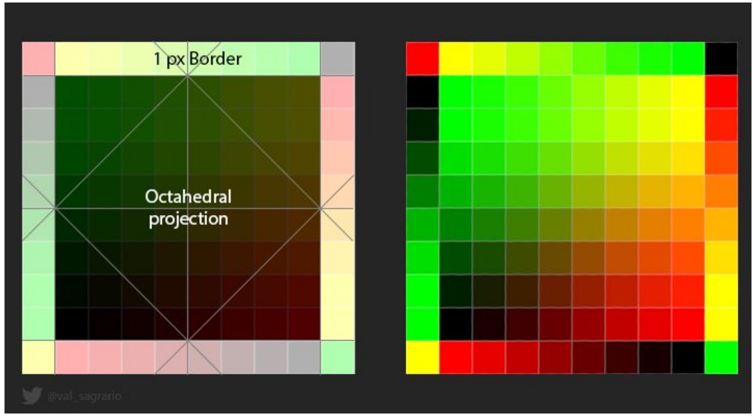


Image Source: Valentín Sagrario

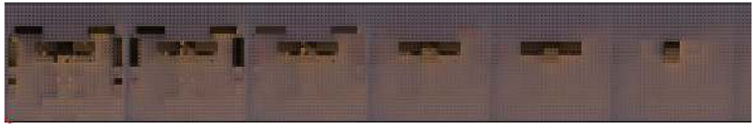
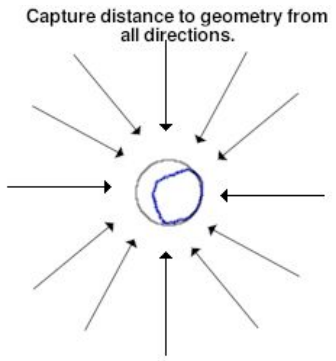
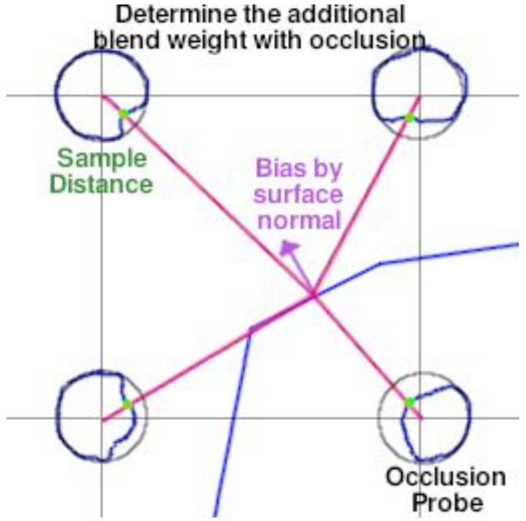


Image Source: Morgan McGuire

Occlusion Probes

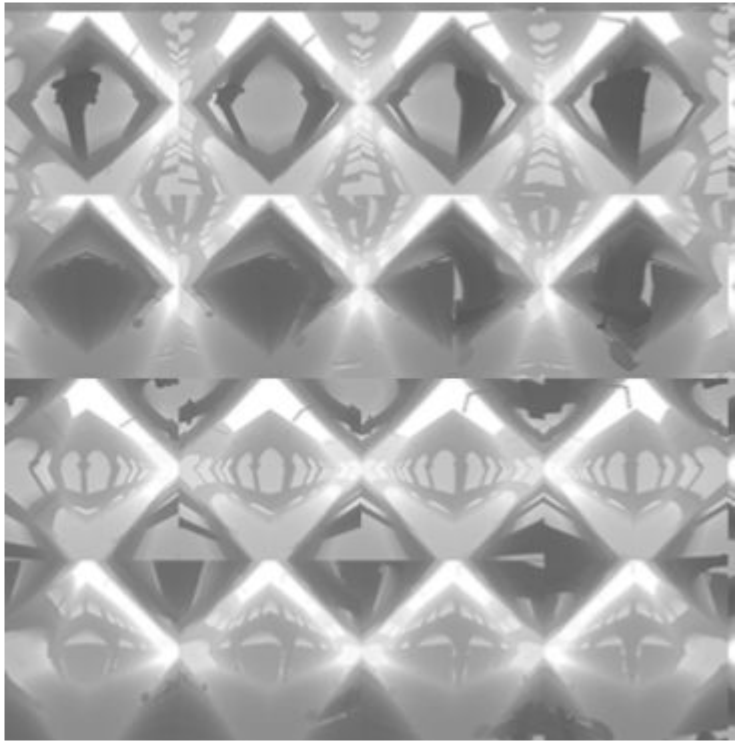
- Occlusion probes know the radial distance to geometry outwards the probe (within the 8 adjacent cells)
- When rendering, geometry must be biased a bit by the geometric normal and the resulting position is used for sampling.
- The direction to that position is used to sample depths, used to compute occlusion using the same algorithm as exponential shadow maps (Chevysheb inequality).
- Resulting value is added to the probe weight when color blending.



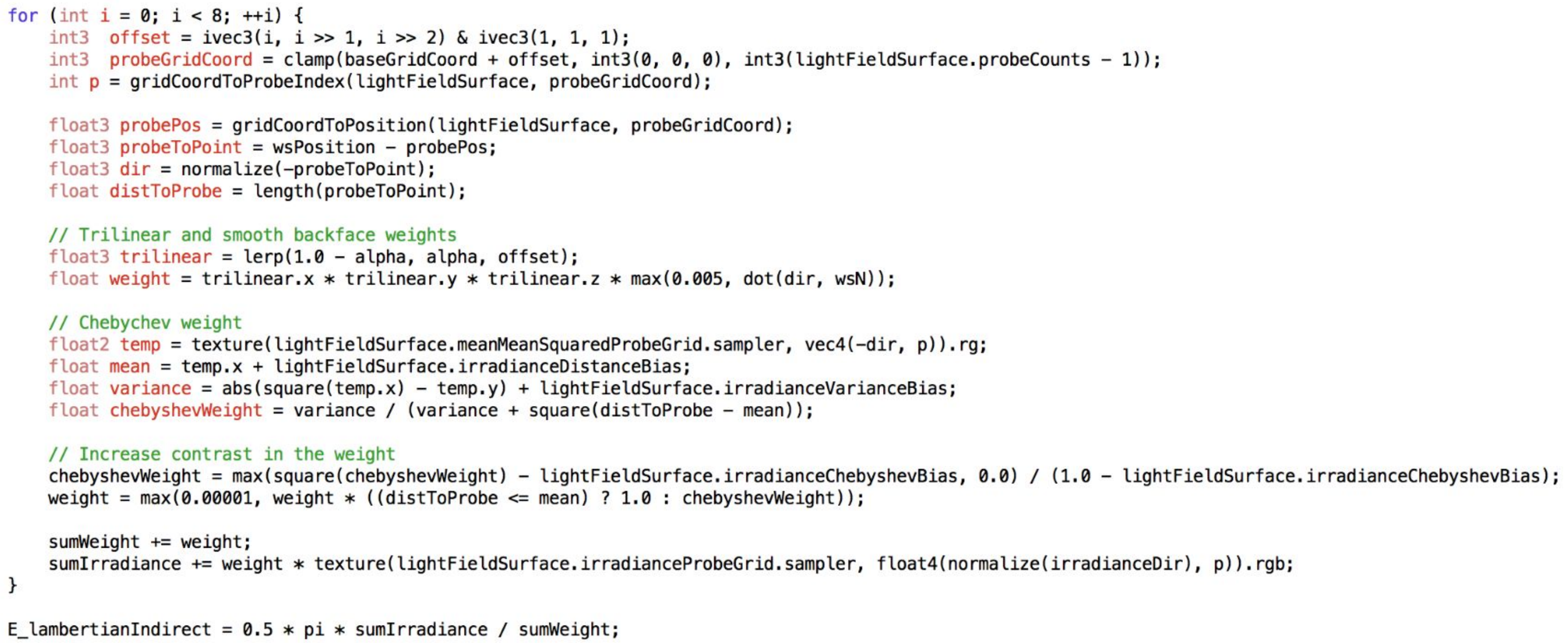


Occlusion probe storage.

- Stored as a 16x16 octahedron encoded depth image.
- Stores a radial distance from within the probe to geometry in all directions. Max distance is the max diagonal distance between the cube formed by 8 probes.
- Used to probe occlusion (walls). If a probe is occluded, it has less influence on the final average. This significantly reduces light leaks.
- Filtered a bit to smooth out discontinuities.



Code used for probing (McGuire):



```
for (int i = 0; i < 8; ++i) {
    int3  offset = ivec3(i, i >> 1, i >> 2) & ivec3(1, 1, 1);
    int3  probeGridCoord = clamp(baseGridCoord + offset, int3(0, 0, 0), int3(lightFieldSurface.probeCounts - 1));
    int p = gridCoordToProbeIndex(lightFieldSurface, probeGridCoord);

    float3 probePos = gridCoordToPosition(lightFieldSurface, probeGridCoord);
    float3 probeToPoint = wsPosition - probePos;
    float3 dir = normalize(-probeToPoint);
    float distToProbe = length(probeToPoint);

    // Trilinear and smooth backface weights
    float3 trilinear = lerp(1.0 - alpha, alpha, offset);
    float weight = trilinear.x * trilinear.y * trilinear.z * max(0.005, dot(dir, wsN));

    // Chebychev weight
    float2 temp = texture(lightFieldSurface.meanMeanSquaredProbeGrid.sampler, vec4(-dir, p)).rg;
    float mean = temp.x + lightFieldSurface.irradianceDistanceBias;
    float variance = abs(square(temp.x) - temp.y) + lightFieldSurface.irradianceVarianceBias;
    float chebyshevWeight = variance / (variance + square(distToProbe - mean));

    // Increase contrast in the weight
    chebyshevWeight = max(square(chebyshevWeight) - lightFieldSurface.irradianceChebyshevBias, 0.0) / (1.0 - lightFieldSurface.irradianceChebyshevBias);
    weight = max(0.00001, weight * ((distToProbe <= mean) ? 1.0 : chebyshevWeight));

    sumWeight += weight;
    sumIrradiance += weight * texture(lightFieldSurface.irradianceProbeGrid.sampler, float4(normalize(irradianceDir), p)).rgb;
}

E_lambertianIndirect = 0.5 * pi * sumIrradiance / sumWeight;
```

Rendering DF and occlusion

Preparing the data structures for tracing

No raytracing, all sorts of problems

- In order to run on every hardware, raytracing is not supported.
- Closest approximation is sphere tracing signed distance fields.
- Generating signed distance fields is expensive, and for a cascaded approach they need to be generated often.
- The resolution of SDFs is not enough for occlusion probes, solid blocks can occlude them easily.
- Creative solutions are required for all these problems.

Solution: sphere-tracing distance fields

Advantages:

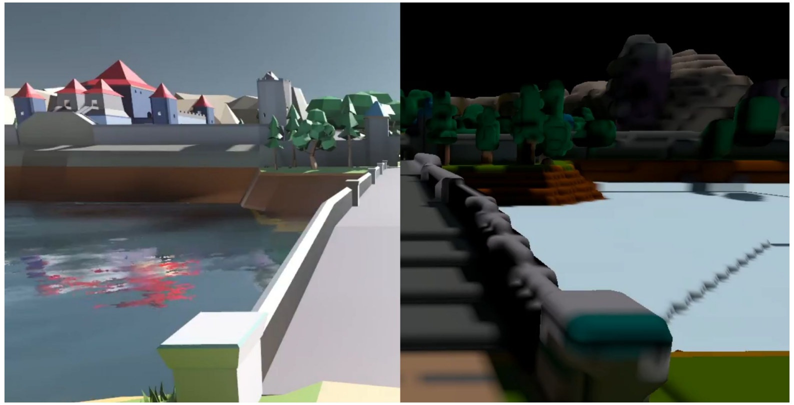
- Works without raytracing support.
- Good performance, even if very old hardware.

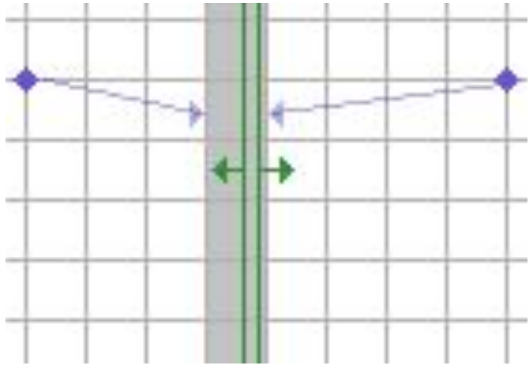
Disadvantages:

- Costly to generate, needs optimizations (see optimization section).
- Lacks resolution, which causes problems (see next slide).

Main problems with distance fields:

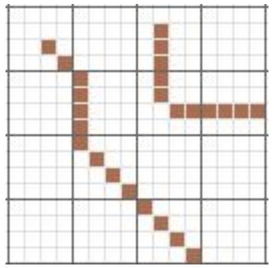
- Lack of resolution (rendering to cascaded voxels) results in bad quality.
- Voxel sized-geometry results in worse quality occlusion probes.
- As such, thin walls that fit within a voxel either leak light or require a voxel-sized bias (due to chebyshev inequality).
- Bias helps to some extent, but large values cause different types of problems.





Solution:

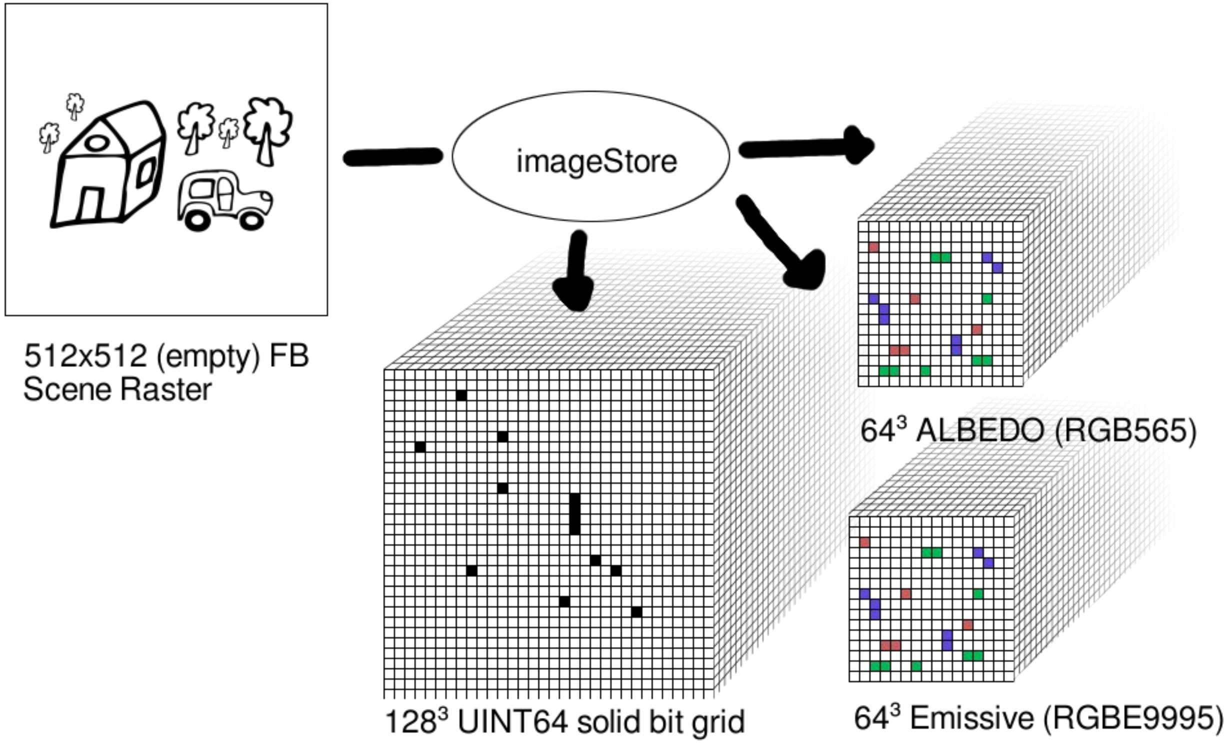
- Render solid geometry to a grid 16x times the resolution.
- Use bit-fields and raster using imageStoreAtomicOr (which is very fast).
- For 128^3 distance field, a 512^3 cascade can be used for rendering (just 16mb).
- More precise occlusion probes and more precise (sub-voxel) distance fields can be built.



Overview: Create the distance field and occlusion maps

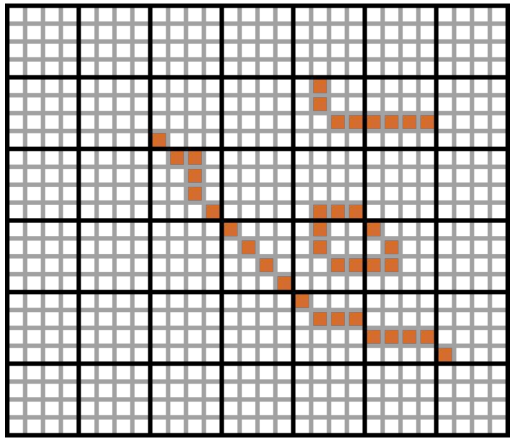
- Cascade SDF size is 128^3 grid, contains 16 bit distances.
- Irradiance and occlusion probes are placed every 16 cells, hence a grid of 17^3 probes.
- Render (raster) into an empty framebuffer (no attachments), 4x the size in 2D (512^2), either from all sides or use a geometry shader to choose side depending on triangle facing.
- Use imageStore atomic OR to "blit" to a 128^3 "solid bits" grid containing 2xUINT32 buffers (64 bits), each is a 4^3 bitfield containing 64 bits more solid resolution on the geometry.
- Use subgroup operations to condense the 4x4 2D pixels rastered for *albedo* and *emissive*, use imageStore to put then in 64^3 grids, since we don't need a lot of detail for sampling.

In detail: Render scene to solid bits, albedo and emissive



Creating the distance field

- Use jump flood based on proximity to the closest sub-voxel (set bit). This allows more precise distance fields than just to solid voxels.
- Create the 128^3 distance field 3D texture.

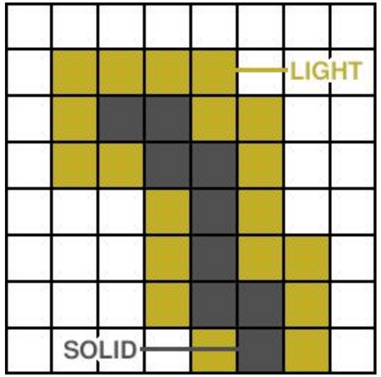


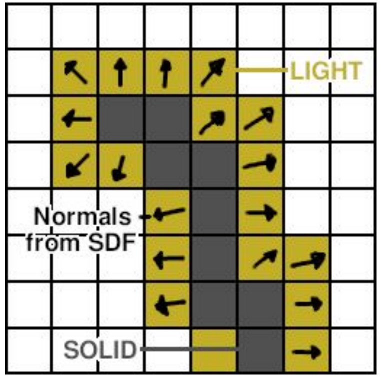
Creating the occlusion probes

- For the occlusion probes, the maximum distance is 28 voxels (max distance in the 16^3 voxel space between probes).
- For each pixel in the octahedral encoded texture for occlusion probes, get the ray direction. If the SDF determines that the distance is > 28, then just place 28 as distance.
- If not, trace the bitfield (solid bid grid) using a DDA until a hit is found. This will result in a precise distance (sub-voxel).

Creating the light texture and buffer

- This is an array of empty voxels that are next to solid voxels that will be used for computing the light information, then bliting it to the 3D light texture.
- The 3D light texture is used for sampling light after a ray is traced and hits something.
- The light buffer is just an 1D array containing positions, normals (generated from the SDF), an “extra outline bitmask”, and indirect dispatch command memory (with amount of elements). It is generated at the time the SDF is stored.
- The light textures exist in RGBE9995 format and are blitted from the light buffer processing.



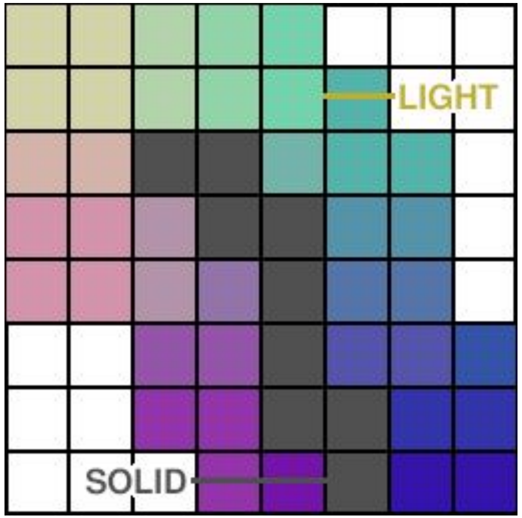


Tracing light into the probes

Not raytracing, but sphere tracing

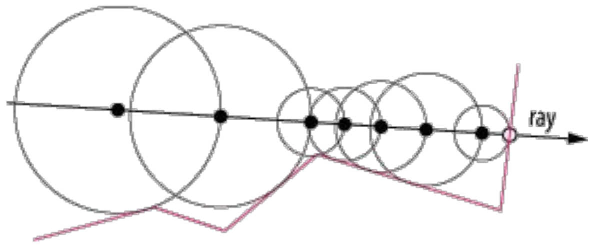
Filling the light texture with light

- All the elements of the light buffer array are processed.
- All the lights are iterated for each element.
- Shadows are sphere traced.
- Indirect light is added here by reading from surrounding light probes.
- The normal saved on the previous step is used to see the effect. Lights that face away from the normal are not processed.
- The light value is combined into a single RGBE9995.
- The value is plotted into the 3D light texture, the surrounding bits that were saved in the preprocess steps are used to plot extra voxels around the outline. This ensures that the 1-voxel traceback after spheretracing will most likely find the proper light information.



Overview of tracing light

- As mentioned before, each probe information is stored in a 5x5 texture.
- The compute shader will trace the whole probe in a single workgroup.
- Each ray direction is picked by running octahedral map in reverse, picking a jittered position within the pixel.
- Biasing is not generally needed, since we have the occlusion probes to do a bit of an extra initial boost in the distance we want the tracing to go.
- The ray is sphere traced from the current cascade to upper cascades.
- If it hits something, it reads albedo and emission from those 3D textures, and computes a surface normal from the SDF texture.
- Because the hit position may not contain light information, retracing back a bit until a cell containing light information is found may be needed.
- Light information is merged with the computed normal based on the anisotropy weights.
- Finally, after a group barrier (wait until all rays done), integrate the probe pixels by averaging neighbours in 180 degrees.
- Use the typical temporal function (lerp with a partial delta) to converge.



Pseudocode

```
layout(local_size_x = 5, local_size_y = 5, local_size_z = 1) in;

shared vec4 light_cache[25];

void main() {

        vec2 octa_pos = vec2(gl_LocalInvocationID.xy);
        vec3 ray_dir = octahedron_decode((octa_pos + jitter_func(frame)) / 5.0);
        vec3 hit_pos;
        uint hit_cascade;
        vec4 light;
        if (trace_ray(ray_dir,hit_pos,hit_cascade) {
                // Hit something, compute light from it.
                light = compute_light(hit_pos,hit_cascade);
        } else {
                // No hit, read from the sky.
                light = compute_light_from_sky(ray_dir);
        }

        // Store in shared light cache, to later allow averaging
        uint light_cache_index = gl_LocalInvocationID.x + gl_LocalInvocationID.y * 5;
        light_cache[light_cache_index] = light;

        groupMemoryBarrier();
        barrier();
        // Average with neighbours to ensure light reaches from 180 degrees.
        light = average_light(light_cache_index);

        save_to_lightfield(light);

}
```

All is nice but..

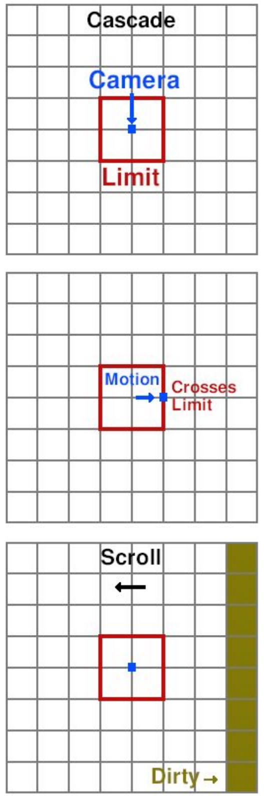
- Doing the whole process every frame (rastering, generating SDF, generating occlusion and sphere tracing) is *very* slow.
- Computing all lights (specially shadows) for all ray hits is slow.
- To make this algorithm useful, it needs to be sped up.
- The devil is in the details.

Optimizing 3D texture gen

Speeding up the 3D texture generation

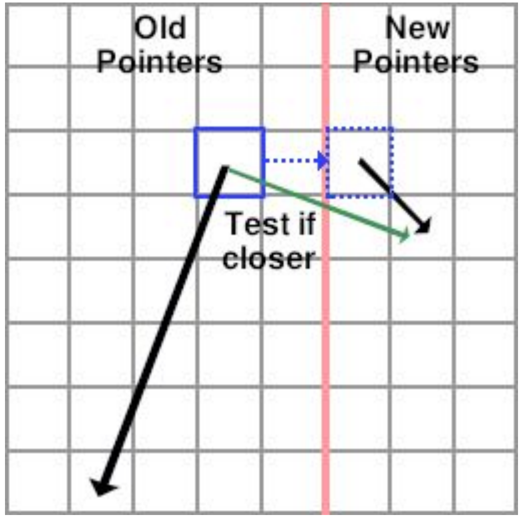
Optimizing the distance and occlusion field generation

- For now, we deal only with static objects.
- We try to keep the camera centered within the cascades.
- If the camera leaves the center (moves beyond the closest probe to the center in a direction), we must re-render the cascade.
- To optimize, we can just "scroll" the cascade and re-rendering the slice near the side that changed.



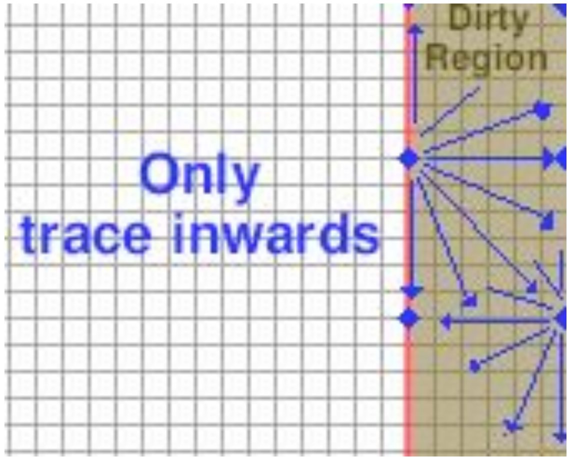
Merging the distance fields

- Distance fields keep closest distance, hence just running jump flood on the dirty region is not enough. Both regions (old and re-rendered) need to be merged.
- Keep, for both sides, the final step of the jump flood (pointers before they are converted to distance).
- For each cell, check the pointer vs the position pointed by the closest cell on the other side. If closer, replace the pointer.
- Re-generate the distance field.



Merging the occlusion probes

- The probes in the new region need to be re-generated. Use the pre-blurred versions to work with.
- Those at the new edge of the cascade only trace towards the inside.
- Those at the division between the new regions need only to only trace the pixels (of the octahedral map) towards the new region, but the ones tracing towards the old one can be left intact.
- Re-blur the occlusion probes.



Merging the light buffers.

- The light buffer array (positions) need to be re-generated from scratch in the distance field merging step (though this is not an expensive process).
- The light texture can be safely scrolled and the new area marked dirty (more on this later).

Conclusion:

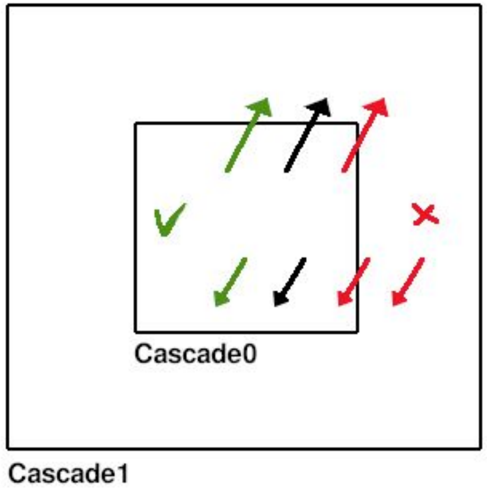
- Only re-generating the slice that moved is very cheap.
- When camera moves, it never goes in a pure diagonal direction (as in: vec3(1,1,1)), so it only pushes the limit and causes scrolling once at a time. No need to implement re-rendering of all 3 sides at once.

Optimizing sphere tracing

Only trace what needs to be re-traced

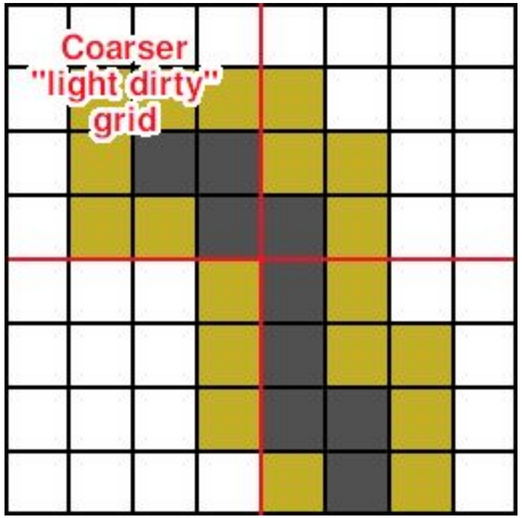
Caching rays

- As geometry is static, ray hits (cascade, position in light texture) can be cached.
- The normal is expensive to compute from the distance field, so it can be cached too.
- Scrolling will invalidate caches, but not all of them.
- If the hit is local and the hit point remains within the same cascade, it does not need invalidation.
- If the hit is in an upper cascade and the side the ray exits from the cascade remains the same, it does not need invalidation.



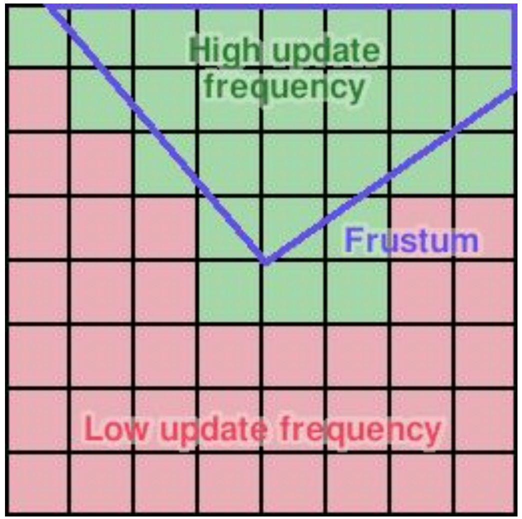
Caching light hits

- To avoid computing the light reaching every cell in all the cascades, ray hits can mark areas from a (coarser) 3D texture as dirty (set a bit).
- This way, when iterating the light array to compute the lighting, regions not needed by ray hits do not need to be computed.
- This has a frame of delay, but high temporal coherency.
- Having a separate static light texture (for static lights) which is computed only once (and updated on dirty regions) can ensure they work at no cost.



Reducing update frequency on invisible probes

- One large optimization possible is to reduce the update frequency of probes not contained in the frustum.
- Processing of some probes can simply be skipped every some frames if far away from the frustum. If they take longer to converge, it's not a problem.
- As the camera is always centered, more than half of the probes will see significantly reduced update frequency.

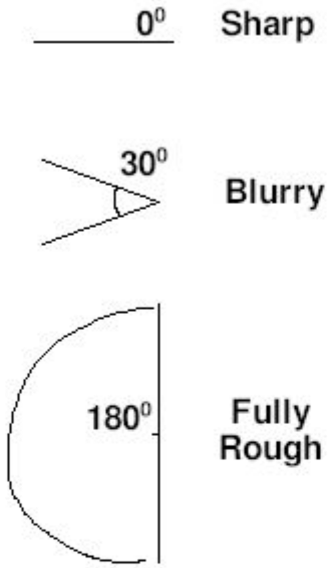


Reflected Light

It's not really global if you can't properly do reflections

SDFGI excels at reflections!

- Fully rough reflections are possible, as well as sharp SDF reflections.
- Combined with Screen Space Reflections, it should allow having top notch reflections.
- There are three stages for roughness:
  1. Tracing rays (sharp)
  2. Using the reflection field (medium roughness)
  3. Using the irradiance field (full roughness).
- Depending the roughness level, 2 of the above 3 are used and blended.



Sharp Reflections

- For every pixel on the screen, reconstruct the position based on depth and projection matrix.
- Find the smallest cascade containing it and trace a the reflected ray until it hits something.
- Reconstruct the lighting using the light texture.
- **NOTE**: Tracing the same way as the light probes will not look right, because cascade switches just “pop” in reflections and become distracting. Combine cascades by tracing 2 levels together and use minimum distance of both with a blend function.

Medium roughness

- In the “Overview of tracing light” section, before doing the barrier, save the unfiltered value to another buffer (with interpolation average based on time too).
- This will result in a similar texture array as the irradiance probe field, but less blurry.
- This works excellent for intermediate rough reflections.

Fully rough reflections

- Use the irradiance field probes to sample the light coming in the reflection vector direction.
- This gives you fully free fully rough reflections, which are very difficult to obtain with regular raytracing.

Dynamic Objects

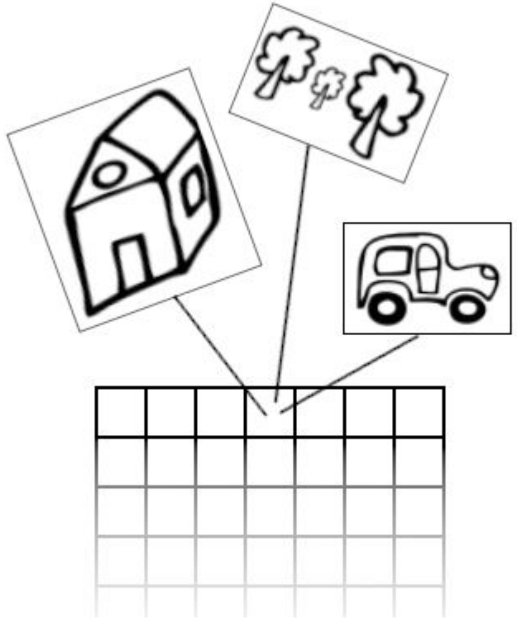
Dynamic object support

The logic should be simple..

- For dynamic objects (that move), cache a local SDF (generated via Jump Flood once) for each. Can be done dynamically.
- Take the 128^3 SDF, for each cell iterate all the objects and find the closest distance to a solid.
- Store it.

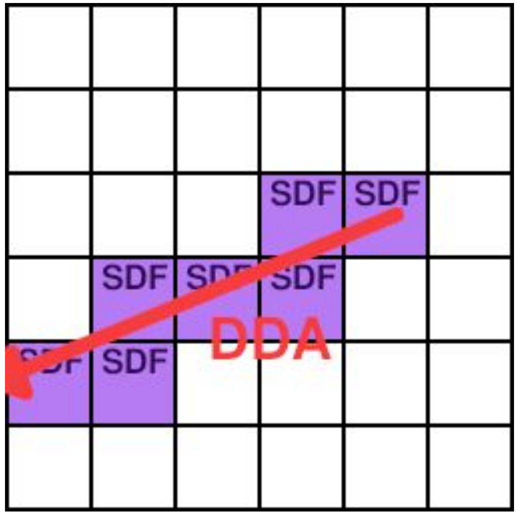
But its not.

- Many objects result in high iteration for all cells in the grid, resulting in $O(N^3 * M)$ complexity.



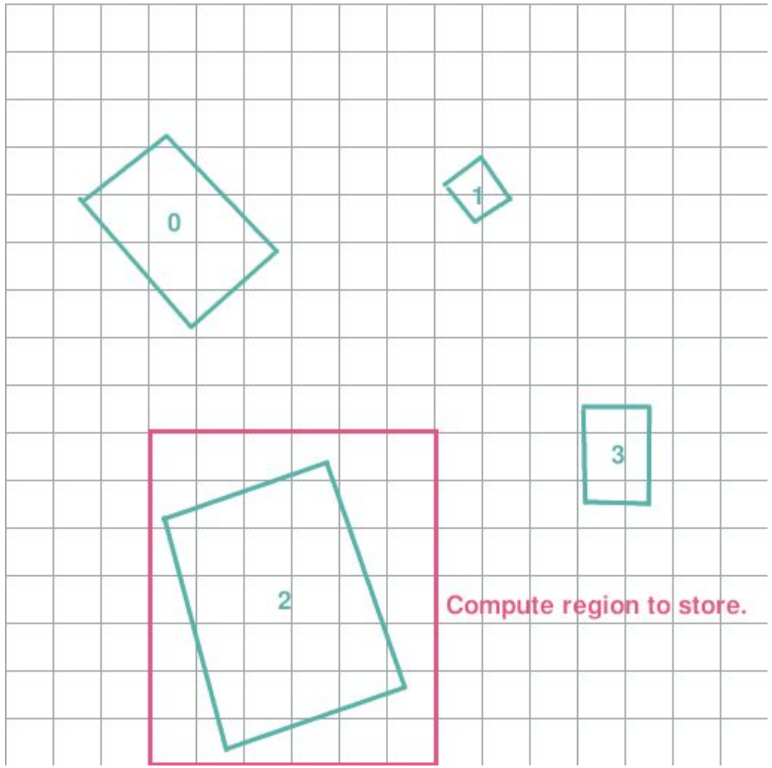
Solution: Split into smaller SDFs and combine with DDA

- Tentatively, use macro regions of 8^3, containing SDFs of 16^3.
- Only dynamic objects overlapping with that region are scanned.
- Traversal is done with a mix of DDA and SDF. Possibly bit slower, but should not be significantly so.
- A 64-bytes long bitfield could be used for looking up empty cells beforehand to avoid the DDA tapping into empty SDF space.



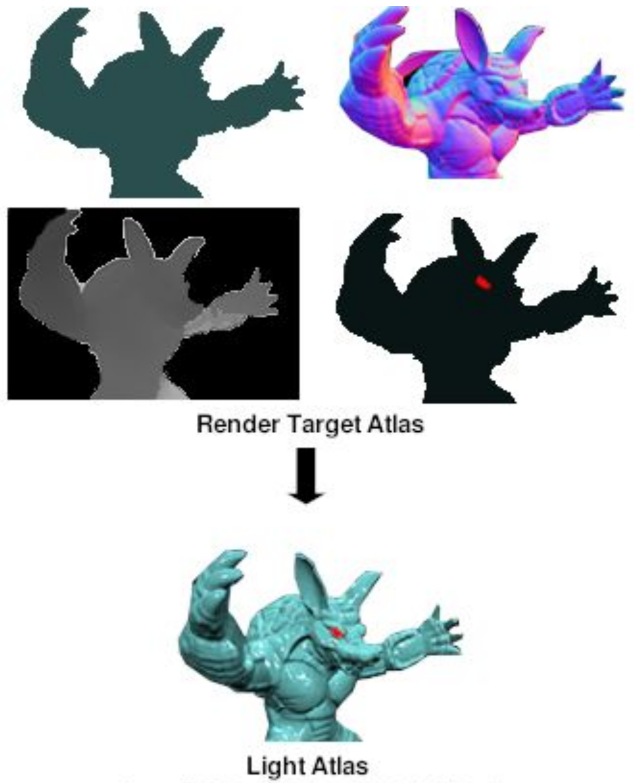
Storing the object list per region

- In a storage buffer, use a bitfield (in uints) to mark the presence of an object, depending on the amount of objects.
- Example, for 64 objects, 2x**uint**s are needed.
- For every dynamic object, run a compute shader for the region to store, a thread per cell.
- Cull the cell (may be outside the OOBB of the object).
- Use Atomic OR to store the right bit to mark the object presence in the cell bitfield.



Storing lighting information.

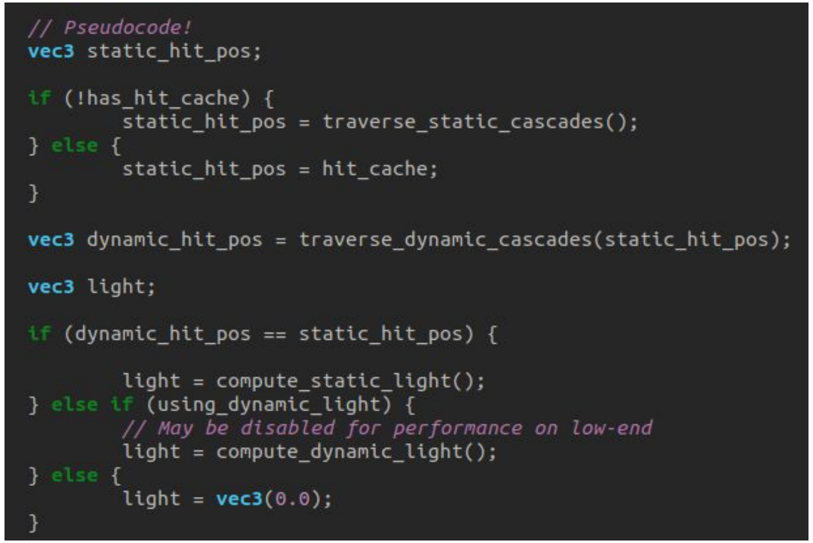
- Light for these objects is likely more expensive to compute, so it could be disabled in low end (and they will only occlude light).
- Create a large set of textures, used as "card atlas", containing *albedo*, *normal*, *emission*, *depth* render targets and a separate *light* atlas of same size.
- Render the 6 views of the object (X+,X-,Y+ .. Z-) to this atlas, to all the render targets. Use a card size (in px) where 1px is roughly 1 voxel.
- In a second pass, use a compute shader to read the render targets, compute lighting (can use the existing SDF cascades to check for shadowing), write to light atlas.
- When filling each cell distances as described before, check the depth atlas to see distance to the outline of the card intersects the voxel. If so, plot the light from the light atlas. Distance can be around 2vx to have some outline to ensure correct tapping.



img credit: Analyzing Deferred Rendering Techniques

Combining with Static Lighting

- Static lighting uses a hit cache (and must be traversed before dynamic if not cached).
- Dynamic light must re-traverse only to that hit cache by combining DDA and SDF as described before.
- If hit cache position reached, then the lighting is read from the static cascades.
- If intercepted, lighting is read from the dynamic cascades.



```
// Pseudocode!
vec3 static_hit_pos;

if (!has_hit_cache) {
        static_hit_pos = traverse_static_cascades();
} else {
        static_hit_pos = hit_cache;
}

vec3 dynamic_hit_pos = traverse_dynamic_cascades(static_hit_pos);

vec3 light;

if (dynamic_hit_pos == static_hit_pos) {

        light = compute_static_light();
} else if (using_dynamic_light) {
        // May be disabled for performance on low-end
        light = compute_dynamic_light();
} else {
        light = vec3(0.0);
}
```