Instant-NGP中的NeRF渲染核心代码解析

Instant-NGP代码很多，但是大部分都是给交互功能写的代码，真正的训练和渲染的代码只占所有代码的一小部分。

Instant-NGP中的NeRF渲染主函数是`render_nerf`，调用栈位于`frame->train_and_render->render_frame->render_frame_main->render_nerf`。
其结构如下：

```
void Testbed::render_nerf(
    cudaStream_t stream,
    CudaDevice& device,
    const CudaRenderBufferView& render_buffer,
    const std::shared_ptr<NerfNetwork<network_precision_t>>& nerf_network,
    const uint8_t* density_grid_bitfield,
    const vec2& focal_length,
    const mat4x3& camera_matrix0,
    const mat4x3& camera_matrix1,
    const vec4& rolling_shutter,
    const vec2& screen_center,
    const Foveation& foveation,
    int visualized_dimension
) {
    float plane_z = m_slice_plane_z + m_scale;
    if (m_render_mode == ERenderMode::Slice) {
        plane_z = -plane_z;
    }

    ERenderMode render_mode = visualized_dimension > -1 ?
ERenderMode::EncodingVis : m_render_mode;

    const float* extra_dims_gpu = m_nerf.get_rendering_extra_dims(stream);

    NerfTracer tracer;

    // Our motion vector code can't undo grid distortions -- so don't render grid
distortion if DLSS is enabled.
    // (Unless we're in distortion visualization mode, in which case the
distortion grid is fine to visualize.)
    auto grid_distortion =
        m_nerf.render_with_lens_distortion && (!m_dlss || m_render_mode ==
ERenderMode::Distortion) ?
        m_distortion.inference_view() :
        Buffer2DView<const vec2>{};

    Lens lens = m_nerf.render_with_lens_distortion ? m_nerf.render_lens : Lens{};

    auto resolution = render_buffer.resolution;

    tracer.init_rays_from_camera(
        render_buffer.spp,
        nerf_network->padded_output_width(),
        nerf_network->n_extra_dims(),
        render_buffer.resolution,
        focal_length,
        camera_matrix0,
        camera_matrix1,
        rolling_shutter,
        screen_center,
        m_parallax_shift,
        m_snap_to_pixel_centers,
        m_render_aabb,
        m_render_aabb_to_local,
        m_render_near_distance,
        plane_z,
        m_aperture_size,
        foveation,
        lens,
        m_envmap.inference_view(),
        grid_distortion,
        render_buffer.frame_buffer,
        render_buffer.depth_buffer,
        render_buffer.hidden_area_mask ? render_buffer.hidden_area_mask-
>const_view() : Buffer2DView<const uint8_t>{},
        density_grid_bitfield,
        m_nerf.show_accel,
        m_nerf.max_cascade,
        m_nerf.cone_angle_constant,
        render_mode,
        stream
    );

    float depth_scale = 1.0f / m_nerf.training.dataset.scale;
    bool render_2d = m_render_mode == ERenderMode::Slice || m_render_mode ==
ERenderMode::Distortion;

    uint32_t n_hit;
    if (render_2d) {
        n_hit = tracer.n_rays_initialized();
    } else {
        n_hit = tracer.trace(
            nerf_network,
            m_render_aabb,
            m_render_aabb_to_local,
            m_aabb,
            focal_length,
            m_nerf.cone_angle_constant,
            density_grid_bitfield,
            render_mode,
            camera_matrix1,
            depth_scale,
            m_visualized_layer,
            visualized_dimension,
            m_nerf.rgb_activation,
            m_nerf.density_activation,
            m_nerf.show_accel,
            m_nerf.max_cascade,
            m_nerf.render_min_transmittance,
            m_nerf.glow_y_cutoff,
            m_nerf.glow_mode,
            extra_dims_gpu,
            stream
        );
    }
    RaysNerfSoa& rays_hit = render_2d ? tracer.rays_init() : tracer.rays_hit();

    if (render_2d) {
        // Store colors in the normal buffer
        uint32_t n_elements = next_multiple(n_hit, BATCH_SIZE_GRANULARITY);
        const uint32_t floats_per_coord = sizeof(NerfCoordinate) / sizeof(float)
+ nerf_network->n_extra_dims();
        const uint32_t extra_stride = nerf_network->n_extra_dims() *
sizeof(float); // extra stride on top of base NerfCoordinate struct

        GPUMatrix<float> positions_matrix{floats_per_coord, n_elements, stream};
        GPUMatrix<float> rgbsigma_matrix{4, n_elements, stream};

        linear_kernel(generate_nerf_network_inputs_at_current_position, 0,
stream, n_hit, m_aabb, rays_hit.payload, PitchedPtr<NerfCoordinate>
((NerfCoordinate*)positions_matrix.data(), 1, 0, extra_stride), extra_dims_gpu);

        if (visualized_dimension == -1) {
            nerf_network->inference(stream, positions_matrix, rgbsigma_matrix);
            linear_kernel(compute_nerf_rgba_kernel, 0, stream, n_hit,
(vec4*)rgbsigma_matrix.data(), m_nerf.rgb_activation, m_nerf.density_activation,
0.01f, false);
        } else {
            nerf_network->visualize_activation(stream, m_visualized_layer,
visualized_dimension, positions_matrix, rgbsigma_matrix);
        }

        linear_kernel(shade_kernel_nerf, 0, stream,
            n_hit,
            m_nerf.render_gbuffer_hard_edges,
            camera_matrix1,
            depth_scale,
            (vec4*)rgbsigma_matrix.data(),
            nullptr,
            rays_hit.payload,
            m_render_mode,
            m_nerf.training.linear_colors,
            render_buffer.frame_buffer,
            render_buffer.depth_buffer
        );
        return;
    }

    linear_kernel(shade_kernel_nerf, 0, stream,
        n_hit,
        m_nerf.render_gbuffer_hard_edges,
        camera_matrix1,
        depth_scale,
        rays_hit.rgba,
        rays_hit.depth,
        rays_hit.payload,
        m_render_mode,
        m_nerf.training.linear_colors,
        render_buffer.frame_buffer,
        render_buffer.depth_buffer
    );

    if (render_mode == ERenderMode::Cost) {
        std::vector<NerfPayload> payloads_final_cpu(n_hit);
        CUDA_CHECK_THROW(cudaMemcpyAsync(payloads_final_cpu.data(),
rays_hit.payload, n_hit * sizeof(NerfPayload), cudaMemcpyDeviceToHost, stream));
        CUDA_CHECK_THROW(cudaStreamSynchronize(stream));

        size_t total_n_steps = 0;
        for (uint32_t i = 0; i < n_hit; ++i) {
            total_n_steps += payloads_final_cpu[i].n_steps;
        }
        tlog::info() << "Total steps per hit= " << total_n_steps << "/" << n_hit
<< " = " << ((float)total_n_steps/(float)n_hit);
    }
}
```

可以看到，主要流程有三：

- `tracer.init_rays_from_camera`
- `tracer.trace`
- `shade_kernel_nerf`

生成光线`tracer.init_rays_from_camera`

顾名思义，对要渲染的图像上的每个像素生成一道光线，用于ray marching。

```
void Testbed::NerfTracer::init_rays_from_camera(
    uint32_t sample_index,
    uint32_t padded_output_width,
    uint32_t n_extra_dims,
    const ivec2& resolution,
    const vec2& focal_length,
    const mat4x3& camera_matrix0,
    const mat4x3& camera_matrix1,
    const vec4& rolling_shutter,
    const vec2& screen_center,
    const vec3& parallax_shift,
    bool snap_to_pixel_centers,
    const BoundingBox& render_aabb,
    const mat3& render_aabb_to_local,
    float near_distance,
    float plane_z,
    float aperture_size,
    const Foveation& foveation,
    const Lens& lens,
    const Buffer2DView<const vec4>& envmap,
    const Buffer2DView<const vec2>& distortion,
    vec4* frame_buffer,
    float* depth_buffer,
    const Buffer2DView<const uint8_t>& hidden_area_mask,
    const uint8_t* grid,
    int show_accel,
    uint32_t max_mip,
    float cone_angle_constant,
    ERenderMode render_mode,
    cudaStream_t stream
) {
    // Make sure we have enough memory reserved to render at the requested
resolution
    size_t n_pixels = (size_t)resolution.x * resolution.y;
    enlarge(n_pixels, padded_output_width, n_extra_dims, stream);

    const dim3 threads = { 16, 8, 1 };
    const dim3 blocks = { div_round_up((uint32_t)resolution.x, threads.x),
div_round_up((uint32_t)resolution.y, threads.y), 1 };
    init_rays_with_payload_kernel_nerf<<<blocks, threads, 0, stream>>>(
        sample_index,
        m_rays[0].payload,
        resolution,
        focal_length,
        camera_matrix0,
        camera_matrix1,
        rolling_shutter,
        screen_center,
        parallax_shift,
        snap_to_pixel_centers,
        render_aabb,
        render_aabb_to_local,
        near_distance,
        plane_z,
        aperture_size,
        foveation,
        lens,
        envmap,
        frame_buffer,
        depth_buffer,
        hidden_area_mask,
        distortion,
        render_mode
    );

    m_n_rays_initialized = resolution.x * resolution.y;

    CUDA_CHECK_THROW(cudaMemsetAsync(m_rays[0].rgba, 0, m_n_rays_initialized *
sizeof(vec4), stream));
    CUDA_CHECK_THROW(cudaMemsetAsync(m_rays[0].depth, 0, m_n_rays_initialized *
sizeof(float), stream));

    linear_kernel(advance_pos_nerf_kernel, 0, stream,
        m_n_rays_initialized,
        render_aabb,
        render_aabb_to_local,
        camera_matrix1[2],
        focal_length,
        sample_index,
        m_rays[0].payload,
        grid,
        (show_accel >= 0) ? show_accel : 0,
        max_mip,
        cone_angle_constant
    );
}
```

可以看到，主要流程有二：

- `init_rays_with_payload_kernel_nerf`：初始化`payload`，计算光线的起点和方向，保存在`payload`中
- `advance_pos_nerf_kernel`：根据`payload`中光线的起点和方向和`density_grid`计算采样的范围

初始化光线：`init_rays_with_payload_kernel_nerf`

```
__global__ void init_rays_with_payload_kernel_nerf(
    uint32_t sample_index,
    NerfPayload* __restrict__ payloads,
    ivec2 resolution,
    vec2 focal_length,
    mat4x3 camera_matrix0,
    mat4x3 camera_matrix1,
    vec4 rolling_shutter,
    vec2 screen_center,
    vec3 parallax_shift,
    bool snap_to_pixel_centers,
    BoundingBox render_aabb,
    mat3 render_aabb_to_local,
    float near_distance,
    float plane_z,
    float aperture_size,
    Foveation foveation,
    Lens lens,
    Buffer2DView<const vec4> envmap,
    vec4* __restrict__ frame_buffer,
    float* __restrict__ depth_buffer,
    Buffer2DView<const uint8_t> hidden_area_mask,
    Buffer2DView<const vec2> distortion,
    ERenderMode render_mode
) {
    uint32_t x = threadIdx.x + blockDim.x * blockIdx.x;
    uint32_t y = threadIdx.y + blockDim.y * blockIdx.y;

    if (x >= resolution.x || y >= resolution.y) {
        return;
    }

    uint32_t idx = x + resolution.x * y;

    if (plane_z < 0) {
        aperture_size = 0.0;
    }
    vec2 pixel_offset = ld_random_pixel_offset(snap_to_pixel_centers ? 0 :
sample_index);
    vec2 uv = vec2{(float)x + pixel_offset.x, (float)y + pixel_offset.y} /
vec2(resolution);
    mat4x3 camera = get_xform_given_rolling_shutter({camera_matrix0,
camera_matrix1}, rolling_shutter, uv, ld_random_val(sample_index, idx *
72239731));

    Ray ray = uv_to_ray(
        sample_index,
        uv,
        resolution,
        focal_length,
        camera,
        screen_center,
        parallax_shift,
        near_distance,
        plane_z,
        aperture_size,
        foveation,
        hidden_area_mask,
        lens,
        distortion
    );

    NerfPayload& payload = payloads[idx];
    payload.max_weight = 0.0f;

    depth_buffer[idx] = MAX_DEPTH();

    if (!ray.is_valid()) {
        payload.origin = ray.o;
        payload.alive = false;
        return;
    }

    if (plane_z < 0) {
        float n = length(ray.d);
        payload.origin = ray.o;
        payload.dir = (1.0f/n) * ray.d;
        payload.t = -plane_z*n;
        payload.idx = idx;
        payload.n_steps = 0;
        payload.alive = false;
        depth_buffer[idx] = -plane_z;
        return;
    }

    if (render_mode == ERenderMode::Distortion) {
        vec2 uv_after_distortion = pos_to_uv(ray(1.0f), resolution, focal_length,
camera, screen_center, parallax_shift, foveation);

        frame_buffer[idx].rgb() = to_rgb((uv_after_distortion - uv) * 64.0f);
        frame_buffer[idx].a = 1.0f;
        depth_buffer[idx] = 1.0f;
        payload.origin = ray(MAX_DEPTH());
        payload.alive = false;
        return;
    }

    ray.d = normalize(ray.d);

    if (envmap) {
        frame_buffer[idx] = read_envmap(envmap, ray.d);
    }

    float t = fmaxf(render_aabb.ray_intersect(render_aabb_to_local * ray.o,
render_aabb_to_local * ray.d).x, 0.0f) + 1e-6f;

    if (!render_aabb.contains(render_aabb_to_local * ray(t))) {
        payload.origin = ray.o;
        payload.alive = false;
        return;
    }

    payload.origin = ray.o;
    payload.dir = ray.d;
    payload.t = t;
    payload.idx = idx;
    payload.n_steps = 0;
    payload.alive = true;
}
```

可以看到，最主要的计算是`uv_to_ray`这个函数，其输入各种相机参数和像素位置输出一个`Ray`类型的变量：

```
struct Ray {
    vec3 o;
    vec3 d;

    NGP_HOST_DEVICE vec3 operator()(float t) const {
        return o + t * d;
    }

    NGP_HOST_DEVICE void advance(float t) {
        o += d * t;
    }

    NGP_HOST_DEVICE float distance_to(const vec3& p) const {
        vec3 nearest = p - o;
        nearest -= d * dot(nearest, d) / length2(d);
        return length(nearest);
    }

    NGP_HOST_DEVICE bool is_valid() const {
        return d != vec3(0.0f);
    }

    static NGP_HOST_DEVICE Ray invalid() {
        return {{0.0f, 0.0f, 0.0f}, {0.0f, 0.0f, 0.0f}};
    }
};
```

`o`、`d`、`t`都是很常见的表示光线起点、方向、前进距离的变量名组合，很显然`uv_to_ray`就是在计算光线的起点和方向。

`init_rays_with_payload_kernel_nerf`中`uv_to_ray`之后就是各种赋值了，可以看到是把`o`、`d`、`t`赋值给`payload`，每个`payload`对应图像上的一个像素，其值的含义也可以猜出来：

```
    payload.origin = ray.o; // 光线的起点
    payload.dir = ray.d; // 光线的方向
    payload.t = t; // 光线上可以采样的最远距离
    payload.idx = idx; // 像素的ID，用于表示这个payload是图像上上的哪个像素
    payload.n_steps = 0; // 之后ray marching会用到，用于记录采样点数量
    payload.alive = true; // 之后ray marching会用到，光线不一定都会hit到场景中的点，hit
不到点的光线就是payload.alive = false
```

计算采样范围：`advance_pos_nerf_kernel`

```
__global__ void advance_pos_nerf_kernel(
    const uint32_t n_elements,
    BoundingBox render_aabb,
    mat3 render_aabb_to_local,
    vec3 camera_fwd,
    vec2 focal_length,
    uint32_t sample_index,
    NerfPayload* __restrict__ payloads,
    const uint8_t* __restrict__ density_grid,
    uint32_t min_mip,
    uint32_t max_mip,
    float cone_angle_constant
) {
    const uint32_t i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i >= n_elements) return;

    advance_pos_nerf(payloads[i], render_aabb, render_aabb_to_local, camera_fwd,
focal_length, sample_index, density_grid, min_mip, max_mip, cone_angle_constant);
}
```

没什么好说的，调用了`advance_pos_nerf`，主要看这个：

```
__device__ void advance_pos_nerf(
    NerfPayload& payload,
    const BoundingBox& render_aabb,
    const mat3& render_aabb_to_local,
    const vec3& camera_fwd,
    const vec2& focal_length,
    uint32_t sample_index,
    const uint8_t* __restrict__ density_grid,
    uint32_t min_mip,
    uint32_t max_mip,
    float cone_angle_constant
) {
    if (!payload.alive) {
        return;
    }

    vec3 origin = payload.origin;
    vec3 dir = payload.dir;
    vec3 idir = vec3(1.0f) / dir;

    float cone_angle = calc_cone_angle(dot(dir, camera_fwd), focal_length,
cone_angle_constant);

    float t = advance_n_steps(payload.t, cone_angle, ld_random_val(sample_index,
payload.idx * 786433));
    t = if_unoccupied_advance_to_next_occupied_voxel(t, cone_angle, {origin,
dir}, idir, density_grid, min_mip, max_mip, render_aabb, render_aabb_to_local);
    if (t >= MAX_DEPTH()) {
        payload.alive = false;
    } else {
        payload.t = t;
    }
}
```

上面这段是ray marching的代码，其主要是对`payload`和`density_grid`进行操作。
从逻辑上看是根据`payload`中的光线起点和方向和`density_grid`计算`if_unoccupied_advance_to_next_occupied_voxel`，计算出来`payload.t`。
顾名思义，这就是在计算每条光线能行进多长。

具体看`if_unoccupied_advance_to_next_occupied_voxel`，就是一个while循环，在`density_grid`里面不断前进直到找到一个不透明的voxel挡住了这条光线：

```
template <bool MIP_FROM_DT=false>
NGP_HOST_DEVICE float if_unoccupied_advance_to_next_occupied_voxel(
    float t,
    float cone_angle,
    const Ray& ray,
    const vec3& idir,
    const uint8_t* __restrict__ density_grid,
    uint32_t min_mip,
    uint32_t max_mip,
    BoundingBox aabb,
    mat3 aabb_to_local = mat3::identity()
) {
    while (true) {
        vec3 pos = ray(t);
        if (t >= MAX_DEPTH() || !aabb.contains(aabb_to_local * pos)) {
            return MAX_DEPTH();
        }

        uint32_t mip = clamp(MIP_FROM_DT ? mip_from_dt(calc_dt(t, cone_angle),
pos) : mip_from_pos(pos), min_mip, max_mip);

        if (!density_grid || density_grid_occupied_at(pos, density_grid, mip)) {
            return t;
        }
        // Find largest empty voxel surrounding us, such that we can advance as far as possible in the next step.
        // Other places that do voxel stepping don't need this, because they don't rely on thread coherence as
        // much as this one here.
        while (mip < max_mip && !density_grid_occupied_at(pos, density_grid, mip+1)) {
            ++mip;
        }

        t = advance_to_next_voxel(t, cone_angle, pos, ray.d, idir, mip);
    }
}
```

其中的第二层while循环和图形学中的Mipmap概念有关，简单来说就是这个`density_grid`是分层的，mip越大`density_grid`中的voxel覆盖范围越大。
找到了`density_grid`中当前位置的最大的空voxel后，就可以调用`advance_to_next_voxel`前进一步。
所以才有注释里写的"Find largest empty voxel surrounding us, such that we can advance as far as possible in the next step."。

这里用于判断遮挡的函数`density_grid_occupied_at`长这样：

```
inline NGP_HOST_DEVICE bool density_grid_occupied_at(const vec3& pos, const uint8_t* density_grid_bitfield, uint32_t mip) {
    uint32_t idx = cascaded_grid_idx_at(pos, mip);
    if (idx == 0xFFFFFFFF) {
        return false;
    }
    return density_grid_bitfield[idx/8+grid_mip_offset(mip)/8] & (1<<(idx%8));
}
```

其中的两个核心函数`cascaded_grid_idx_at`用于获取`pos`所表示的点在mip level内的哪个位置、`grid_mip_offset`用于获取`mip`所表示的mip level从哪里开始：

```
inline NGP_HOST_DEVICE uint32_t cascaded_grid_idx_at(vec3 pos, uint32_t mip) {
    float mip_scale = scalbnf(1.0f, -mip); // 2^-mip
    pos -= vec3(0.5f); // 以0.5, 0.5, 0.5为中心缩放
    pos *= mip_scale; // mip越大每个voxel覆盖范围越大所以是缩小
    pos += vec3(0.5f); // 所以上述操作是以0.5, 0.5, 0.5为中心缩小坐标
    // 综上，坐标的取值范围是0到1，而不同的mip是以0.5, 0.5, 0.5为中心缩小坐标后再查询

    ivec3 i = pos * (float)NERF_GRIDSIZE(); // 放大到0到128，这里的ivec3里的xyz是整数
    if (i.x < 0 || i.x >= NERF_GRIDSIZE() || i.y < 0 || i.y >= NERF_GRIDSIZE() || i.z < 0 || i.z >= NERF_GRIDSIZE()) {
        return 0xFFFFFFFF;
    }

    return morton3D(i.x, i.y, i.z); // 获取该坐标在morton3D曲线上的位置
    // 官方注释：Calculates a 30-bit Morton code for the given 3D point located within the unit cube [0,1].
}

inline NGP_HOST_DEVICE uint32_t grid_mip_offset(uint32_t mip) {
    return NERF_GRID_N_CELLS() * mip; // 每个mip虽然尺度不一样，但是对应的数组大小都是一样的
    // 看上面那个函数就能明白，不同的mip是以0.5, 0.5, 0.5为中心缩小坐标后再查询，所以八个mip里面其实分别是靠近中心区域的半径为2,4,6,8,16,32,64,128的方块里是有用值，其他地方用不上
}
```

于是，对于每个输入坐标，`advance_pos_nerf_kernel`从`density_grid_bitfield`里面找到一个最近的遮挡它的区域，其返回的`payload.t`标志了这条光线最近的被遮挡点在何处。

附加知识：`density_grid`和`density_grid_bitfield`

注意前面浮点型`density_grid`的参数传到这里变成了`density_grid_bitfield`里面存的是二值😂，回去前面找找发现`tracer.trace`的输入确实是`density_grid_bitfield`而不是浮点型`density_grid`。
所以继续找找浮点型`density_grid`是怎么变成`density_grid_bitfield`的，发现在这里：

```
void Testbed::update_density_grid_mean_and_bitfield(cudaStream_t stream) {
    const uint32_t n_elements = NERF_GRID_N_CELLS();

    size_t size_including_mips = grid_mip_offset(NERF_CASCADES())/8;
    m_nerf.density_grid_bitfield.enlarge(size_including_mips);
    m_nerf.density_grid_mean.enlarge(reduce_sum_workspace_size(n_elements));

    CUDA_CHECK_THROW(cudaMemsetAsync(m_nerf.density_grid_mean.data(), 0,
sizeof(float), stream));
    reduce_sum(m_nerf.density_grid.data(), [n_elements] __device__ (float val) {
return fmaxf(val, 0.f) / (n_elements); }, m_nerf.density_grid_mean.data(),
n_elements, stream);

    linear_kernel(grid_to_bitfield, 0, stream, n_elements/8 * NERF_CASCADES(),
n_elements/8 * (m_nerf.max_cascade + 1), m_nerf.density_grid.data(),
m_nerf.density_grid_bitfield.data(), m_nerf.density_grid_mean.data());

    for (uint32_t level = 1; level < NERF_CASCADES(); ++level) {
        linear_kernel(bitfield_max_pool, 0, stream, n_elements/64,
m_nerf.get_density_grid_bitfield_mip(level-1),
m_nerf.get_density_grid_bitfield_mip(level));
    }

    set_all_devices_dirty();
}

uint8_t* Testbed::Nerf::get_density_grid_bitfield_mip(uint32_t mip) {
    return density_grid_bitfield.data() + grid_mip_offset(mip)/8;
}
```

里面首先调用了一个`grid_to_bitfield`函数：

```
__global__ void grid_to_bitfield(
    const uint32_t n_elements,
    const uint32_t n_nonzero_elements,
    const float* __restrict__ grid,
    uint8_t* __restrict__ grid_bitfield,
    const float* __restrict__ mean_density_ptr
) {
    const uint32_t i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i >= n_elements) return;
    if (i >= n_nonzero_elements) {
        grid_bitfield[i] = 0;
        return;
    }

    uint8_t bits = 0;

    float thresh = std::min(NERF_MIN_OPTICAL_THICKNESS(), *mean_density_ptr);

    NGP_PRAGMA_UNROLL
    for (uint8_t j = 0; j < 8; ++j) {
        bits |= grid[i*8+j] > thresh ? ((uint8_t)1 << j) : 0;
    }

    grid_bitfield[i] = bits;
}
```

原来就是当浮点型`density_grid`中的某项超过一个阈值就给`density_grid_bitfield`对应的位置1。
这个阈值的函数`NERF_MIN_OPTICAL_THICKNESS`就这样😂，反正就是超过0.01就表示可见：

```
// Any alpha below this is considered "invisible" and is thus culled away.
inline constexpr __device__ float NERF_MIN_OPTICAL_THICKNESS() { return 0.01f; }
```

然后对每一层调用`bitfield_max_pool`函数，很明显，这for循环里又是max_pool又是level的，肯定就是根据细粒度的`density_grid_bitfield`生成粗粒度的`density_grid_bitfield`，就对应于前面提到的Mipmap分层density_grid。
看看这个`bitfield_max_pool`怎么个max_pool法：

```
__global__ void bitfield_max_pool(const uint32_t n_elements,
    const uint8_t* __restrict__ prev_level,
    uint8_t* __restrict__ next_level
) {
    const uint32_t i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i >= n_elements) return;

    uint8_t bits = 0;

    NGP_PRAGMA_UNROLL
    for (uint8_t j = 0; j < 8; ++j) {
        // If any bit is set in the previous level, set this
        // level's bit. (Max pooling.)
        bits |= prev_level[i*8+j] > 0 ? ((uint8_t)1 << j) : 0;
        // 3D Morton曲线每8个为一组最小单元（想象八叉树，类似），3D max pooling当前级的每一个方块又都对应上一级的8个方块，所以这里直接prev_level[i*8+j] > 0
        // 每个bits有8个bit对应8个当前级方块，所以for循环8次填满8个bit
    }

    uint32_t x = morton3D_invert(i>>0) + NERF_GRIDSIZE()/8;
    uint32_t y = morton3D_invert(i>>1) + NERF_GRIDSIZE()/8;
    uint32_t z = morton3D_invert(i>>2) + NERF_GRIDSIZE()/8;
    next_level[morton3D(x, y, z)] |= bits;
}
```

于是就达到了分层`density_grid_bitfield`的效果。

执行渲染`tracer.trace`

```
uint32_t Testbed::NerfTracer::trace(
    const std::shared_ptr<NerfNetwork<network_precision_t>>& network,
    const BoundingBox& render_aabb,
    const mat3& render_aabb_to_local,
    const BoundingBox& train_aabb,
    const vec2& focal_length,
    float cone_angle_constant,
    const uint8_t* grid,
    ERenderMode render_mode,
    const mat4x3 &camera_matrix,
    float depth_scale,
    int visualized_layer,
    int visualized_dim,
    ENerfActivation rgb_activation,
    ENerfActivation density_activation,
    int show_accel,
    uint32_t max_mip,
    float min_transmittance,
    float glow_y_cutoff,
    int glow_mode,
    const float* extra_dims_gpu,
    cudaStream_t stream
) {
    if (m_n_rays_initialized == 0) {
        return 0;
    }

    CUDA_CHECK_THROW(cudaMemsetAsync(m_hit_counter, 0, sizeof(uint32_t),
stream));

    uint32_t n_alive = m_n_rays_initialized;
    // m_n_rays_initialized = 0;

    uint32_t i = 1;
    uint32_t double_buffer_index = 0;
    while (i < MARCH_ITER) {
        RaysNerfSoa& rays_current = m_rays[(double_buffer_index + 1) % 2];
        RaysNerfSoa& rays_tmp = m_rays[double_buffer_index % 2];
        ++double_buffer_index;

        // Compact rays that did not diverge yet
        {
            CUDA_CHECK_THROW(cudaMemsetAsync(m_alive_counter, 0,
sizeof(uint32_t), stream));
            linear_kernel(compact_kernel_nerf, 0, stream,
                n_alive,
                rays_tmp.rgba, rays_tmp.depth, rays_tmp.payload,
                rays_current.rgba, rays_current.depth, rays_current.payload,
                m_rays_hit.rgba, m_rays_hit.depth, m_rays_hit.payload,
                m_alive_counter, m_hit_counter
            );
            CUDA_CHECK_THROW(cudaMemcpyAsync(&n_alive, m_alive_counter,
sizeof(uint32_t), cudaMemcpyDeviceToHost, stream));
            CUDA_CHECK_THROW(cudaStreamSynchronize(stream));
        }

        if (n_alive == 0) {
            break;
        }

        // Want a large number of queries to saturate the GPU and to ensure
compaction doesn't happen toooo frequently.
        uint32_t target_n_queries = 2 * 1024 * 1024;
        uint32_t n_steps_between_compaction = clamp(target_n_queries / n_alive,
(uint32_t)MIN_STEPS_INBETWEEN_COMPACTION,
(uint32_t)MAX_STEPS_INBETWEEN_COMPACTION);

        uint32_t extra_stride = network->n_extra_dims() * sizeof(float);
        PitchedPtr<NerfCoordinate> input_data((NerfCoordinate*)m_network_input,
1, 0, extra_stride);
        linear_kernel(generate_next_nerf_network_inputs, 0, stream,
            n_alive,
            render_aabb,
            render_aabb_to_local,
            train_aabb,
            focal_length,
            camera_matrix[2],
            rays_current.payload,
            input_data,
            n_steps_between_compaction,
            grid,
            (show_accel>=0) ? show_accel : 0,
            max_mip,
            cone_angle_constant,
            extra_dims_gpu
        );
        uint32_t n_elements = next_multiple(n_alive * n_steps_between_compaction,
BATCH_SIZE_GRANULARITY);
        GPUMatrix<float> positions_matrix((float*)m_network_input,
(sizeof(NerfCoordinate) + extra_stride) / sizeof(float), n_elements);
        GPUMatrix<network_precision_t, RM>
rgbsigma_matrix((network_precision_t*)m_network_output, network-
>padded_output_width(), n_elements);
        network->inference_mixed_precision(stream, positions_matrix,
rgbsigma_matrix);

        if (render_mode == ERenderMode::Normals) {
            network->input_gradient(stream, 3, positions_matrix,
positions_matrix);
        } else if (render_mode == ERenderMode::EncodingVis) {
            network->visualize_activation(stream, visualized_layer,
visualized_dim, positions_matrix, positions_matrix);
        }
        linear_kernel(composite_kernel_nerf, 0, stream,
            n_alive,
            n_elements,
            i,
            train_aabb,
            glow_y_cutoff,
            glow_mode,
            camera_matrix,
            focal_length,
            depth_scale,
            rays_current.rgba,
            rays_current.depth,
            rays_current.payload,
            input_data,
            m_network_output,
            network->padded_output_width(),
            n_steps_between_compaction,
            render_mode,
            grid,
            rgb_activation,
            density_activation,
            show_accel,
            min_transmittance
        );

        i += n_steps_between_compaction;
    }

    uint32_t n_hit;
    CUDA_CHECK_THROW(cudaMemcpyAsync(&n_hit, m_hit_counter, sizeof(uint32_t), 
cudaMemcpyDeviceToHost, stream));
    CUDA_CHECK_THROW(cudaStreamSynchronize(stream));
    return n_hit;
}
```

最最核心的NeRF推断过程是`network->inference_mixed_precision(stream, positions_matrix, rgbsigma_matrix);`。
按照NeRF的运行逻辑，推断前的`compact_kernel_nerf`和`generate_next_nerf_network_inputs`就应该是采样过程；
推断后的`composite_kernel_nerf`就应该是体渲染过程。

再看外面这一个`while (i < MARCH_ITER)`，哦原来是ray marching，懂了懂了，一步一步执行"采样->推断->体渲染"过程呗。

`compact_kernel_nerf`这函数很简单，就是ray marching每一步开始时的初始化过程：

```
__global__ void compact_kernel_nerf(
    const uint32_t n_elements,
    vec4* src_rgba, float* src_depth, NerfPayload* src_payloads,
    vec4* dst_rgba, float* dst_depth, NerfPayload* dst_payloads,
    vec4* dst_final_rgba, float* dst_final_depth, NerfPayload* 
dst_final_payloads,
    uint32_t* counter, uint32_t* finalCounter
) {
    const uint32_t i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i >= n_elements) return;

    NerfPayload& src_payload = src_payloads[i];

    if (src_payload.alive) {
        uint32_t idx = atomicAdd(counter, 1);
        dst_payloads[idx] = src_payload;
        dst_rgba[idx] = src_rgba[i];
        dst_depth[idx] = src_depth[i];
    } else if (src_rgba[i].a > 0.001f) {
        uint32_t idx = atomicAdd(finalCounter, 1);
        dst_final_payloads[idx] = src_payload;
        dst_final_rgba[idx] = src_rgba[i];
        dst_final_depth[idx] = src_depth[i];
    }
}
```

观察这个函数调用的周围，可以发现`compact_kernel_nerf`的输入来自于`m_rays`，这个`m_rays`在`init_rays_from_camera`末尾被赋值，其定义为`RaysNerfSoa m_rays[2];`：

```
struct RaysNerfSoa {
#if defined(__CUDACC__) || (defined(__clang__) && defined(__CUDA__))
    void copy_from_other_async(const RaysNerfSoa& other, cudaStream_t stream) {
        CUDA_CHECK_THROW(cudaMemcpyAsync(rgba, other.rgba, size * sizeof(vec4),
cudaMemcpyDeviceToDevice, stream));
        CUDA_CHECK_THROW(cudaMemcpyAsync(depth, other.depth, size *
sizeof(float), cudaMemcpyDeviceToDevice, stream));
        CUDA_CHECK_THROW(cudaMemcpyAsync(payload, other.payload, size *
sizeof(NerfPayload), cudaMemcpyDeviceToDevice, stream));
    }
#endif

    void set(vec4* rgba, float* depth, NerfPayload* payload, size_t size) {
        this->rgba = rgba;
        this->depth = depth;
        this->payload = payload;
        this->size = size;
    }

    vec4* rgba;
    float* depth;
    NerfPayload* payload;
    size_t size;
};
```

很明显，`m_rays`用于在这个ray marching循环中交替使用，一项存储了前一次的计算结果，另一项用于当前计算。

采样`generate_next_nerf_network_inputs`

就是前进最多`n_steps`步记下每步对应的点位置在`network_input`里。

```
__global__ void generate_next_nerf_network_inputs(
    const uint32_t n_elements,
    BoundingBox render_aabb,
    mat3 render_aabb_to_local,
    BoundingBox train_aabb,
    vec2 focal_length,
    vec3 camera_fwd,
    NerfPayload* __restrict__ payloads,
    PitchedPtr<NerfCoordinate> network_input,
    uint32_t n_steps,
    const uint8_t* __restrict__ density_grid,
    uint32_t min_mip,
    uint32_t max_mip,
    float cone_angle_constant,
    const float* extra_dims
) {
    const uint32_t i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i >= n_elements) return;

    NerfPayload& payload = payloads[i]; // 获取该像素的payload开始计算

    if (!payload.alive) { // 该像素payload被标记无物则退出
        return;
    }

    vec3 origin = payload.origin;
    vec3 dir = payload.dir;
    vec3 idir = vec3(1.0f) / dir;

    float cone_angle = calc_cone_angle(dot(dir, camera_fwd), focal_length,
cone_angle_constant); // 看样子是计算相机的角度用于后续计算

    float t = payload.t; // 上一步的位置

    for (uint32_t j = 0; j < n_steps; ++j) { // 向前采样n_steps个
        t = if_unoccupied_advance_to_next_occupied_voxel(t, cone_angle, {origin,
dir}, idir, density_grid, min_mip, max_mip, render_aabb, render_aabb_to_local);
        if (t >= MAX_DEPTH()) { // 超范围了就记下当前采样多少步然后退出
            payload.n_steps = j;
            return;
        }

        float dt = calc_dt(t, cone_angle); // 计算当前步要前进多少，也就是确定下一个采样
点在光线上的位置
        network_input(i + j * n_elements)-
>set_with_optional_extra_dims(warp_position(origin + dir * t, train_aabb),
warp_direction(dir), warp_dt(dt), extra_dims, network_input.stride_in_bytes); //
XXXCONE
        // 在network_input里面记下当前的采样点和方向和步长，作为NeRF模型的输入
        t += dt;
    }

    payload.t = t;
    payload.n_steps = n_steps;
}
```

这里`network_input`用于记录NeRF模型的输入，其中的元素`NerfCoordinate`也就是点的位置方向采样步长：

```
struct NerfPosition {
    NGP_HOST_DEVICE NerfPosition(const vec3& pos, float dt)
    :
    p{pos}
#ifdef TRIPLANAR_COMPATIBLE_POSITIONS
    , x{pos.x}
#endif
    {}
    vec3 p;
#ifdef TRIPLANAR_COMPATIBLE_POSITIONS
    float x;
#endif
};

struct NerfDirection {
    NGP_HOST_DEVICE NerfDirection(const vec3& dir, float dt) : d{dir} {}
    vec3 d;
};

struct NerfCoordinate {
    NGP_HOST_DEVICE NerfCoordinate(const vec3& pos, const vec3& dir, float dt) :
pos{pos, dt}, dt{dt}, dir{dir, dt} {}
    NGP_HOST_DEVICE void set_with_optional_extra_dims(const vec3& pos, const
vec3& dir, float dt, const float* extra_dims, uint32_t stride_in_bytes) {
        this->dt = dt;
        this->pos = NerfPosition(pos, dt);
        this->dir = NerfDirection(dir, dt);
        copy_extra_dims(extra_dims, stride_in_bytes);
    }
    inline NGP_HOST_DEVICE const float* get_extra_dims() const { return (const
float*)(this + 1); }
    inline NGP_HOST_DEVICE float* get_extra_dims() { return (float*)(this + 1); }

    NGP_HOST_DEVICE void copy(const NerfCoordinate& inp, uint32_t
stride_in_bytes) {
        *this = inp;
        copy_extra_dims(inp.get_extra_dims(), stride_in_bytes);
    }
    NGP_HOST_DEVICE inline void copy_extra_dims(const float *extra_dims, uint32_t
stride_in_bytes) {
        if (stride_in_bytes >= sizeof(NerfCoordinate)) {
            float* dst = get_extra_dims();
            const uint32_t n_extra = (stride_in_bytes - sizeof(NerfCoordinate)) /
sizeof(float);
            for (uint32_t i = 0; i < n_extra; ++i) dst[i] = extra_dims[i];
        }
    }

    NerfPosition pos;
    float dt;
    NerfDirection dir;
};
```

体渲染`composite_kernel_nerf`

体渲染，一言以蔽之，就是沿着光线方向对NeRF的推断结果(RGBA)进行积分（求和）。
在Instant-NGP中，`composite_kernel_nerf`还兼具判定每条光线的ray marching是否结束的功能。
具体来说就是沿着光线方向积分透明度值，直到透明度值大于某个阈值。
如果经过`n_steps`步后透明度值仍然小于阈值，则判定该条光线ray marching未结束，`payload.alive = true`，则其在下一轮中继续进行ray marching。

所以，整个NeRF渲染的过程实际上并没有用到`density_grid`而只在初始化光线的时候用到了`density_grid_bitmap`。
想想也挺合理，结合这里的ray marching过程看，采样点并不是一次生成好的，而是在ray marching过程中一步一步前进的，每前进一步就对所有光线生成一批采样点，让模型推断输出density进行积分，一直这样循环直到所有光线上的density都超过给定阈值。

```
__global__ void composite_kernel_nerf(
    const uint32_t n_elements,
    const uint32_t stride,
    const uint32_t current_step,
    BoundingBox aabb,
    float glow_y_cutoff,
    int glow_mode,
    mat4x3 camera_matrix,
    vec2 focal_length,
    float depth_scale,
    vec4* __restrict__ rgba,
    float* __restrict__ depth,
    NerfPayload* payloads,
    PitchedPtr<NerfCoordinate> network_input,
    const network_precision_t* __restrict__ network_output,
    uint32_t padded_output_width,
    uint32_t n_steps,
    ERenderMode render_mode,
    const uint8_t* __restrict__ density_grid,
    ENerfActivation rgb_activation,
    ENerfActivation density_activation,
    int show_accel,
    float min_transmittance
) {
    const uint32_t i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i >= n_elements) return;

    NerfPayload& payload = payloads[i];

    if (!payload.alive) {
        return;
    }

    vec4 local_rgba = rgba[i];
    float local_depth = depth[i];
    vec3 origin = payload.origin;
    vec3 cam_fwd = camera_matrix[2];
    // Composite in the last n steps
    uint32_t actual_n_steps = payload.n_steps;
    uint32_t j = 0;
    for (; j < actual_n_steps; ++j) {
        tvec<network_precision_t, 4> local_network_output;
        local_network_output[0] = network_output[i + j * n_elements + 0 *
stride];
        local_network_output[1] = network_output[i + j * n_elements + 1 *
stride];
        local_network_output[2] = network_output[i + j * n_elements + 2 *
stride];
        local_network_output[3] = network_output[i + j * n_elements + 3 *
stride];
        const NerfCoordinate* input = network_input(i + j * n_elements);
        vec3 warped_pos = input->pos.p;
        vec3 pos = unwarp_position(warped_pos, aabb);

        float T = 1.f - local_rgba.a;
        float dt = unwarp_dt(input->dt);
        float alpha = 1.f - __expf(-
network_to_density(float(local_network_output[3]), density_activation) * dt);
        if (show_accel >= 0) {
            alpha = 1.f;
        }
        float weight = alpha * T;

        vec3 rgb = network_to_rgb_vec(local_network_output, rgb_activation);

        if (glow_mode) { // random grid visualizations ftw!
#if 0
            if (0) {  // extremely startrek edition
                float glow_y = (pos.y - (glow_y_cutoff - 0.5f)) * 2.f;
                if (glow_y>1.f) glow_y=max(0.f,21.f-glow_y*20.f);
                if (glow_y>0.f) {
                    float line;
                    line =max(0.f,cosf(pos.y*2.f*3.141592653589793f *
16.f)-0.95f);
                    line+=max(0.f,cosf(pos.x*2.f*3.141592653589793f *
16.f)-0.95f);
                    line+=max(0.f,cosf(pos.z*2.f*3.141592653589793f *
16.f)-0.95f);
                    line+=max(0.f,cosf(pos.y*4.f*3.141592653589793f *
16.f)-0.975f);
                    line+=max(0.f,cosf(pos.x*4.f*3.141592653589793f *
16.f)-0.975f);
                    line+=max(0.f,cosf(pos.z*4.f*3.141592653589793f *
16.f)-0.975f);
                    glow_y=glow_y*glow_y*0.5f + glow_y*line*25.f;
                    rgb.y+=glow_y;
                    rgb.z+=glow_y*0.5f;
                    rgb.x+=glow_y*0.25f;
                }
            }
#endif
            float glow = 0.f;

            bool green_grid = glow_mode & 1;
            bool green_cutline = glow_mode & 2;
            bool mask_to_alpha = glow_mode & 4;

            // less used?
            bool radial_mode = glow_mode & 8;
            bool grid_mode = glow_mode & 16; // makes object rgb go black!

            {
                float dist;
                if (radial_mode) {
                    dist = distance(pos, camera_matrix[3]);
                    dist = min(dist, (4.5f - pos.y) * 0.333f);
                } else {
                    dist = pos.y;
                }

                if (grid_mode) {
                    glow = 1.f / max(1.f, dist);
                } else {
                    float y = glow_y_cutoff - dist; // - (ii*0.005f);
                    float mask = 0.f;
                    if (y > 0.f) {
                        y *= 80.f;
                        mask = min(1.f, y);
                        //if (mask_mode) {
                        //  rgb.x=rgb.y=rgb.z=mask; // mask mode
                        //} else
                        {
                            if (green_cutline) {
                                glow += max(0.f, 1.f - abs(1.f -y)) * 4.f;
                            }

                            if (y>1.f) {
                                y = 1.f - (y - 1.f) * 0.05f;
                            }

                            if (green_grid) {
                                glow += max(0.f, y / max(1.f, dist));
                            }
                        }
                    }
                    if (mask_to_alpha) {
                        weight *= mask;
                    }
                }
            }

            if (glow > 0.f) {
                float line;
                line  = max(0.f, cosf(pos.y * 2.f * 3.141592653589793f * 16.f) -
0.975f);
                line += max(0.f, cosf(pos.x * 2.f * 3.141592653589793f * 16.f) -
0.975f);
                line += max(0.f, cosf(pos.z * 2.f * 3.141592653589793f * 16.f) -
0.975f);
                line += max(0.f, cosf(pos.y * 4.f * 3.141592653589793f * 16.f) -
0.975f);
```

```
                line += max(0.f, cosf(pos.x * 4.f * 3.141592653589793f * 16.f) -
0.975f);
                line += max(0.f, cosf(pos.z * 4.f * 3.141592653589793f * 16.f) -
0.975f);
                line += max(0.f, cosf(pos.y * 8.f * 3.141592653589793f * 16.f) -
0.975f);
                line += max(0.f, cosf(pos.x * 8.f * 3.141592653589793f * 16.f) -
0.975f);
                line += max(0.f, cosf(pos.z * 8.f * 3.141592653589793f * 16.f) -
0.975f);
                line += max(0.f, cosf(pos.y * 16.f * 3.141592653589793f * 16.f) -
0.975f);
                line += max(0.f, cosf(pos.x * 16.f * 3.141592653589793f * 16.f) -
0.975f);
                line += max(0.f, cosf(pos.z * 16.f * 3.141592653589793f * 16.f) -
0.975f);
                if (grid_mode) {
                    glow = /*glow*glow*0.75f + */ glow * line * 15.f;
                    rgb.y = glow;
                    rgb.z = glow * 0.5f;
                    rgb.x = glow * 0.25f;
                } else {
                    glow = glow * glow * 0.25f + glow * line * 15.f;
                    rgb.y += glow;
                    rgb.z += glow * 0.5f;
                    rgb.x += glow * 0.25f;
                }
            }
        } // glow

        if (render_mode == ERenderMode::Normals) {
            // Network input contains the gradient of the network output w.r.t.
input.
            // So to compute density gradients, we need to apply the chain rule.
            // The normal is then in the opposite direction of the density
gradient (i.e. the direction of decreasing density)
            vec3 normal = -
network_to_density_derivative(float(local_network_output[3]), density_activation)
* warped_pos;
            rgb = normalize(normal);
        } else if (render_mode == ERenderMode::Positions) {
            rgb = (pos - 0.5f) / 2.0f + 0.5f;
        } else if (render_mode == ERenderMode::EncodingVis) {
            rgb = warped_pos;
        } else if (render_mode == ERenderMode::Depth) {
            rgb = vec3(dot(cam_fwd, pos - origin) * depth_scale);
        } else if (render_mode == ERenderMode::AO) {
            rgb = vec3(alpha);
        }

        if (show_accel >= 0) {
            uint32_t mip = max((uint32_t)show_accel, mip_from_pos(pos));
            uint32_t res = NERF_GRIDSIZE() >> mip;
            int ix = pos.x * res;
            int iy = pos.y * res;
            int iz = pos.z * res;
            default_rng_t rng(ix + iy * 232323 + iz * 727272);
            rgb.x = 1.f - mip * (1.f / (NERF_CASCADES() - 1));
            rgb.y = rng.next_float();
            rgb.z = rng.next_float();
        }

        local_rgba += vec4(rgb * weight, weight);
        if (weight > payload.max_weight) {
            payload.max_weight = weight;
            local_depth = dot(cam_fwd, pos - camera_matrix[3]);
        }

        if (local_rgba.a > (1.0f - min_transmittance)) {
            local_rgba /= local_rgba.a;
            break;
        }
    }

    if (j < n_steps) {
        payload.alive = false;
        payload.n_steps = j + current_step;
    }

    rgba[i] = local_rgba;
    depth[i] = local_depth;
}
```