Architectural Principles: In-Place, Zero-Overhead C++

🎯 Goal

Maximum performance and predictability for ray tracing by applying strict in-place and zero-overhead coding practices in C++:

- Zero-overhead abstraction (no hidden costs).

- In-place computation (no unnecessary temporaries).

- Manual memory and computation control.

📌 Core Rules

1. Minimize Temporary Objects

- Prefer `*this *= x` over `*this = *this * x`.

- Avoid return-by-value in hot paths.

- Use `set()` to reuse objects: `dst.set(src).normalize();`.

2. Everything In-Place

- All modifying methods work on `*this` and return `*this&`.

- Avoid new allocations in math functions.

3. Stack-Only Allocation

- Use `vec<T,3> tmp[3]` instead of dynamic memory.

- No heap allocations (`new`, `delete`, `malloc`) in tracing.

4. RAII and Lifetime Control

- Use automatic storage duration.

- No smart pointers or manual memory management.

5. Avoid Unnecessary Copies

- Avoid copying large structures like `onb<T>` or `vec<T,N>` unless required.

- Prefer `const T&` or `T&&` over `T` by value.

6. Stack > Heap

- Stack memory is fast, cache-friendly, and auto-managed.

- Heap is slower, fragmented, and risky in hot paths.

🧠 Style & Structure

- Modifying functions are in-place and return `*this&`.

- Use `[[nodiscard]]` for meaningful results:

`[[nodiscard]] constexpr T length_squared() const noexcept;`

Naming Conventions:

- `set()` — manual assignment.

- `normalize()`, `rotate_around_axis()` — in-place.

- `random_direction()` — modifies caller.

- `vec_to_local()`, `vec_from_local()` — external, non-copying, in-place.

🚫 Forbidden

- `new`, `delete`, `malloc`, `free`

- `std::shared_ptr`, `std::unique_ptr` in hot paths

- `std::function`, `std::bind` in compute kernels

- Unnecessary object copying: `*this = ...`

- Branching where math suffices: prefer `std::max()`, `std::clamp()`

📦 Design Notes

- Direction generation handles polar zone instability via `math::constants<T>::polar_zone`.

- Spherical coordinates only valid for normalized vectors.

- Code is written to ensure predictable CPU-level behavior.