Fluid Simulator - FFT ocean simulation pipeline

Perfect — let's go point by point and expand each stage of that **FFT-based ocean simulation pipeline** into a detailed explanation with terminology, Vulkan/C++ implementation hints, and why each step matters. I'll tie this to what's in Tessendorf's *Simulating Ocean Water* Simulating Ocean Water, Stam's stable fluid methods StamFluidforGames, and GPU optimization ideas GPU Optimization for High-Quali... .

---

1. Spectral update (compute shader)

- **Concept:** The ocean starts with a *precomputed random spectrum* $h_0(k)$, where each wave vector k has an amplitude and random phase (Gaussian-distributed). This encodes the statistical ocean model (Phillips spectrum or similar).
- **Formula:** At runtime you evolve it as

$$h(k,t) = h_0(k)e^{i\omega(k)t} + \overline{h_0(-k)}e^{-i\omega(k)t}$$

  where $\omega(k) = \sqrt{g \mid k \mid}$ (deep water dispersion relation).
- **Implementation:**
  - Store `h0` in a GPU buffer at initialization.
  - Each frame, dispatch a small compute shader that multiplies by $e^{i\omega(k)t}$ for each frequency bin.
  - Hermitian symmetry is required (FFT output must be real-valued), so you combine positive and negative k properly.

---

2. Inverse FFT (VkFFT)

- **Concept:** The frequency spectrum gives us waves in *Fourier space*, but we need *heights in real space* $h(x,t)$. That's what the inverse FFT does.
- **Implementation:**
  - Use **VkFFT**, a Vulkan-based FFT library, which lets you enqueue 2D FFTs into your Vulkan command buffer.
  - For memory efficiency: use **R2C/C2R** (real-to-complex / complex-to-real) since the data is Hermitian.
  - After this, you have a 2D grid of floating-point height values.

---

3. Optional slope IFFTs (for normals)

- **Concept:** Shading needs surface normals. Instead of using finite differences on the height field (which can be noisy), compute slopes in Fourier space:

$$\partial h/\partial x = \mathrm{F}^{-1}(ik_x h(k,t)), \quad \partial h/\partial y = \mathrm{F}^{-1}(ik_y h(k,t))$$

- **Implementation:**
  - Multiply `h(k,t)` by `i * kx` and `i * ky` in a compute shader.
  - Run two more inverse FFTs → slope_x and slope_y textures.
  - These provide smooth normals for lighting.

---

4. Convert/transfer FFT result → VkImage

- **Concept:** The FFT output lives in a GPU buffer. Your graphics shaders want to sample from an image/texture.
- **Implementation:**
  - Either use VkFFT's image interop (if supported) or do a `vkCmdCopyBufferToImage` into a 2D texture.
  - Format: usually `VK_FORMAT_R32_SFLOAT` (single channel, 32-bit float). For slopes, `RG32_SFLOAT`.

---

5. Graphics pass

- **Concept:** Now render a flat grid mesh, and in the vertex shader displace its vertices by the ocean height texture.
- **Implementation:**
  - Generate a tiled plane (e.g. 256×256 vertices).
  - In the vertex shader:

    ```glsl
    float h = texture(heightTex, uv).r;
    vec3 displaced = vec3(pos.x, h, pos.z);
    ```
  - Normals: reconstruct from slope textures, or compute with `dFdx/dFdy`.
  - Shading: add Fresnel term (view-dependent reflection), reflection/refraction sampling, and foam masks.

---

6. Local interactions (optional)

- **Concept:** FFT ocean = large-scale, statistical waves. But it doesn't support boats, splashes, or wakes. To fix this, overlay a **local simulation**.
- **Two approaches:**
  - **Stam's stable fluids** StamFluidforGames: a 2D velocity/density grid updated with semi-Lagrangian advection, diffusion, and projection.
  - **Shallow-water solver:** solves simplified height+velocity equations for ripples/wakes.
- **Implementation:**
  - Run a small grid simulation around the camera or boats.
  - Add its displacement onto the FFT ocean via blending.

✅ **Summary of the runtime frame:**

1. Update spectrum with dispersion relation → compute shader.
2. Inverse FFT → heights.
3. (Optional) IFFTs → slopes.
4. Copy results into VkImages.
5. Graphics pass displaces grid and shades.
6. (Optional) Local simulation overlay for interaction.

Do you want me to **walk you through the actual Vulkan setup code** for this pipeline (descriptor sets, compute dispatch, VkFFT integration, render pass), or should I first expand on the **math side (Tessendorf vs Stam)** so you see *why* each step is there?