📋 Vulkan GRRT 初始化核心清单

1. 物理设备与队列族查询 (`pickupPhysicalDevice`)

- **目标**：找到一个能干所有活的"全能"队列族。
- **关键点**：
  - 遍历所有 `QueueFamilyProperties` 。
  - **必须支持图形**：检查 `VK_QUEUE_GRAPHICS_BIT` 。
  - **必须支持计算**：检查 `VK_QUEUE_COMPUTE_BIT` （这是 GRRT 的核心）。
  - **必须支持显示**：调用 `vkGetPhysicalDeviceSurfaceSupportKHR` 返回 `true` 。
  - **结论**：找到一个索引 `graphicsComputeFamilyIndex` ，后续所有命令都提交给这个家族创建的队列。

2. 逻辑设备创建 (`createLogicDevice`)

- **目标**：创建设备接口。
- **关键点**：
  - 只需要创建一个队列（Queue），传入上面找到的 `graphicsComputeFamilyIndex` 。
  - 这个队列将负责：计算任务 to 同步 to 渲染任务。

3. 描述符布局 (`createDescriptorSetLayout`)

- **目标**：告诉 GPU 我们要用哪些资源。
- **关键点**：
  - **Binding 0 (UBO)**:
    - Type: `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER`
    - Stage: `VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_COMPUTE_BIT` (计算着色器需要读取摄像机参数)。
  - **Binding 1 (SSBO)**:
    - Type: `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER`
    - Stage: `VK_SHADER_STAGE_COMPUTE_BIT | VK_SHADER_STAGE_VERTEX_BIT` (计算写，顶点读)。

4. 图形管线 (`createGraphicsPipeline`)

- **目标**：配置渲染最终画面的管线。
- **关键点**：
  - **顶点输入 (Vertex Input)**:
    - `vertexBindingDescriptionCount = 0` (清空)。
    - `vertexAttributeDescriptionCount = 0` (清空)。
    - **原因**：我们不传顶点数据，而是在 Shader 里根据 `gl_VertexIndex` 直接生成全屏三角形/四边形，并去查 SSBO。
  - **着色器**：确保 Vert/Frag Shader 编译无误。

5. 计算管线 (`createComputePipeline` & Layout)

- **目标**：配置运行 GRRT 算法的管线。
- **关键点**：
  - **Layout**: 可以复用图形管线的 `DescriptorSetLayout` （如果绑定一致）。
  - **Shader**: 加载 `.comp` (Compute Shader) 模块。
  - **创建**: 使用 `vkCreateComputePipelines` 。这是一个独立的管线对象。

6. 缓冲区资源创建 (`createBuffers`)

- UBO (Uniform Buffer):
  - Size: `sizeof(UniformBufferObject)` 。
  - Usage: `VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT` 。
  - Memory: `HOST_VISIBLE | HOST_COHERENT` (CPU 每帧更新)。
- SSBO (Shader Storage Buffer):
  - Size: `Width * Height * sizeof(PixelResult)` (约 95MB)。
  - Usage: `VK_BUFFER_USAGE_STORAGE_BUFFER_BIT | VK_BUFFER_USAGE_VERTEX_BUFFER_BIT` (如果顶点着色器将其作为 buffer 读取)。
  - Memory: **`VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT`** (仅 GPU 可见，速度最快)。

7. 描述符池与集合 (`createDescriptorPool / Sets`)

- **目标**：分配资源绑定句柄。
- **关键点**：
  - **Pool Size**: 必须包含 `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER` ，数量至少为 `MAX_FRAMES_IN_FLIGHT` 。
  - **Update Sets**:
    - Binding 0: 写入 UBO 信息。
    - Binding 1: 写入 SSBO 信息 (Type: `STORAGE_BUFFER`)。

8. 命令池 (`createCommandPool`)

- **目标**：分配命令缓冲区的内存池。
- **关键点**：
  - `queueFamilyIndex` 必须是那个全能的 `graphicsComputeFamilyIndex` 。
  - Flags: `VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT` (允许每帧重录)。

9. 命令缓冲区记录 (`createCommandBuffers`) 🔥 (最关键步骤)

- **目标**：编排一帧的所有工作。
- **记录顺序**：
  1. **Bind Compute**: `vkCmdBindPipeline(..., VK_PIPELINE_BIND_POINT_COMPUTE, computePipeline)` 。
  2. **Bind Desc Sets**: `vkCmdBindDescriptorSets(..., computeLayout, ...)` 。
  3. **Dispatch (计算)**: `vkCmdDispatch(groupCountX, groupCountY, 1)` 。
     - *注意*： `groupCount = Resolution / local_size` (例如 1920/16, 1080/16)。
  4. 🔴 **Pipeline Barrier (同步)**: `vkCmdPipelineBarrier` 。
     - `srcStage`： `VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT` (等计算完)。
     - `dstStage`： `VK_PIPELINE_STAGE_VERTEX_SHADER_BIT` (顶点才能开始)。
     - `srcAccess`： `VK_ACCESS_SHADER_WRITE_BIT` (等写完)。
     - `dstAccess`： `VK_ACCESS_SHADER_READ_BIT` (顶点才能读)。
  5. **Begin RenderPass**: 开始渲染通道。
  6. **Bind Graphics**: `vkCmdBindPipeline(..., VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline)` 。
  7. **Bind Desc Sets**: 再次绑定描述符集（如果是同一套 Layout)。
  8. **Draw**: `vkCmdDraw(3, 1, 0, 0)` (绘制 3 个顶点覆盖全屏)。
  9. **End RenderPass**.

10. 同步对象（`createSyncObjects`）

- **目标**：控制 CPU 和 GPU 的帧节奏。
- **关键点**：
  - `imageAvailableSemaphore`：等待交换链图片就绪。
  - `renderFinishedSemaphore`：等待由于以上所有命令（Compute + Graphics）执行完毕。
  - `inFlightFence`：防止 CPU 跑得比 GPU 快太多。
  - *注*：因为是单一队列提交，Compute 和 Graphics 在同一个 `Submit` 里，所以不需要额外的 Compute Semaphore。

把这个清单放在手边，每写一个函数就对照一下，尤其是第 **9** 点的命令记录顺序和 Barrier 设置，那是程序能否正常运行的关键！祝你编码顺利！ 🚀