GAMES 106

现代图形绘制流水线原理与实践

绘制流水线原理（第三课时）

主讲：高涛

课程安排

教学目标


Vulkan的基本架构，绘制流程
Vulkan绘制对象创建，内存管理，以及调试方法和工具
Vulkan的多线程同步，基于移动端，一些常见的优化和实践
作业和反馈

目录

CONTENTS

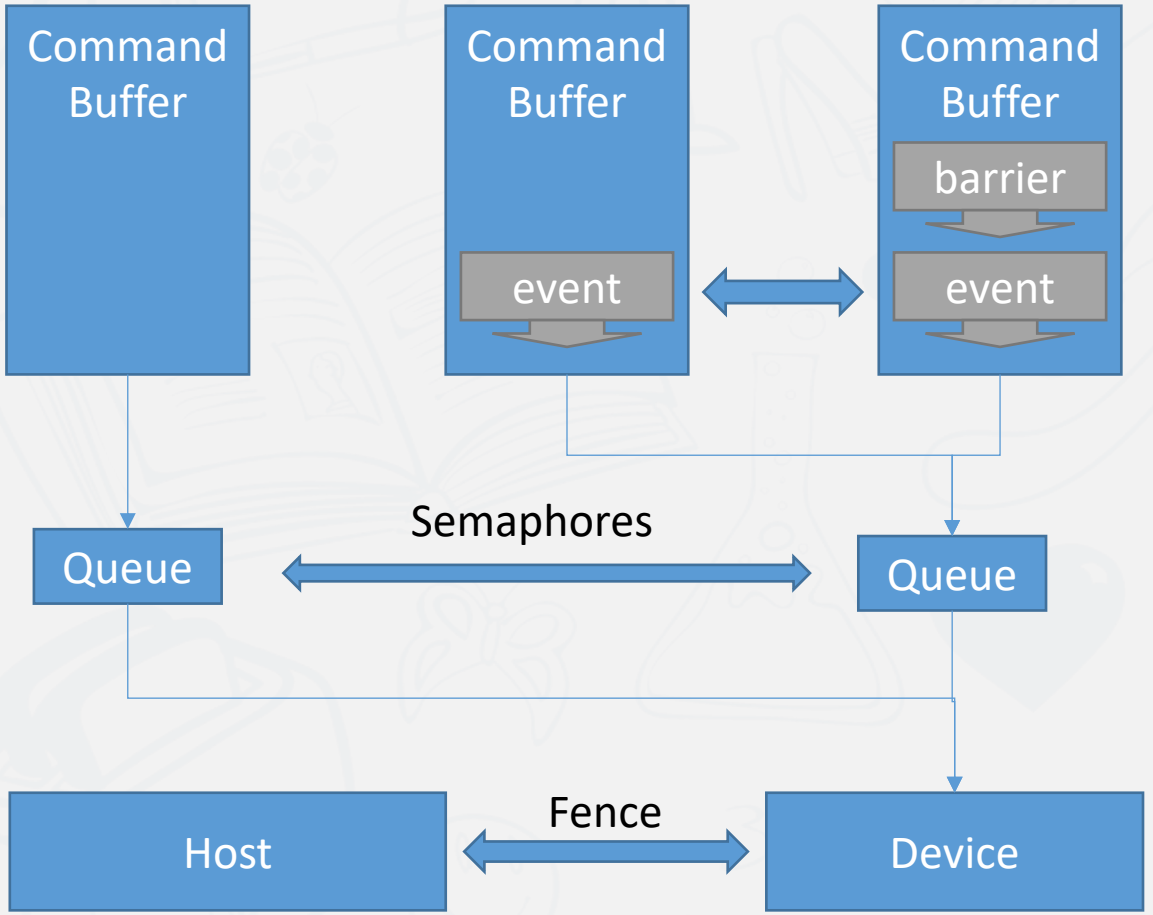
Vulkan同步原语

同步概览

栅栏（fence）
- 中等同步原语
- GPU Command Buffer完成，CPU 获取状态
- CPU可以比较高效地等待

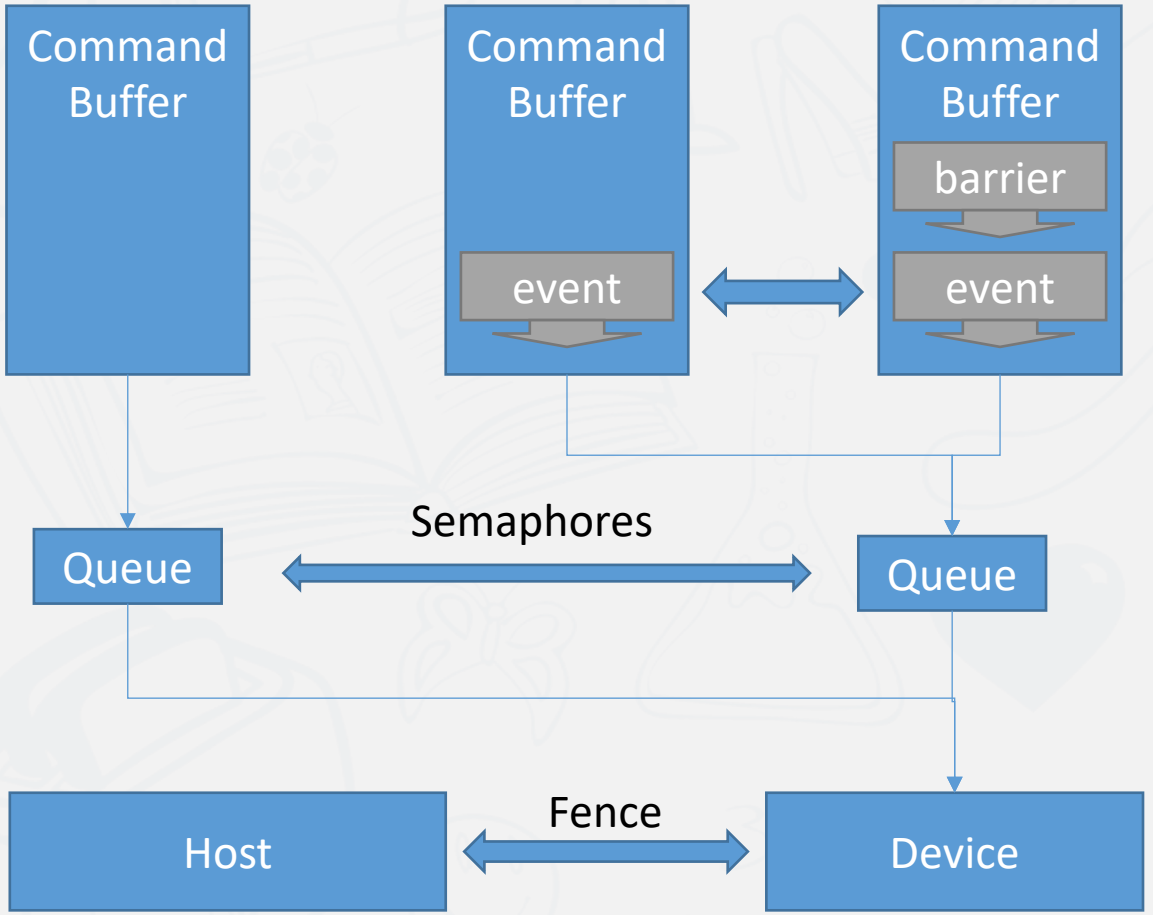
事件（event）
- 细粒度同步原语
- CPU和GPU都可以设置事件的状态
- GPU只可以等待事件
- CPU只可以获取事件的状态

信号量（semaphore）
- 被硬件以原子方式隐式设置
- 不能显式的设置和等待
- 在Queue中设置通知或者等待

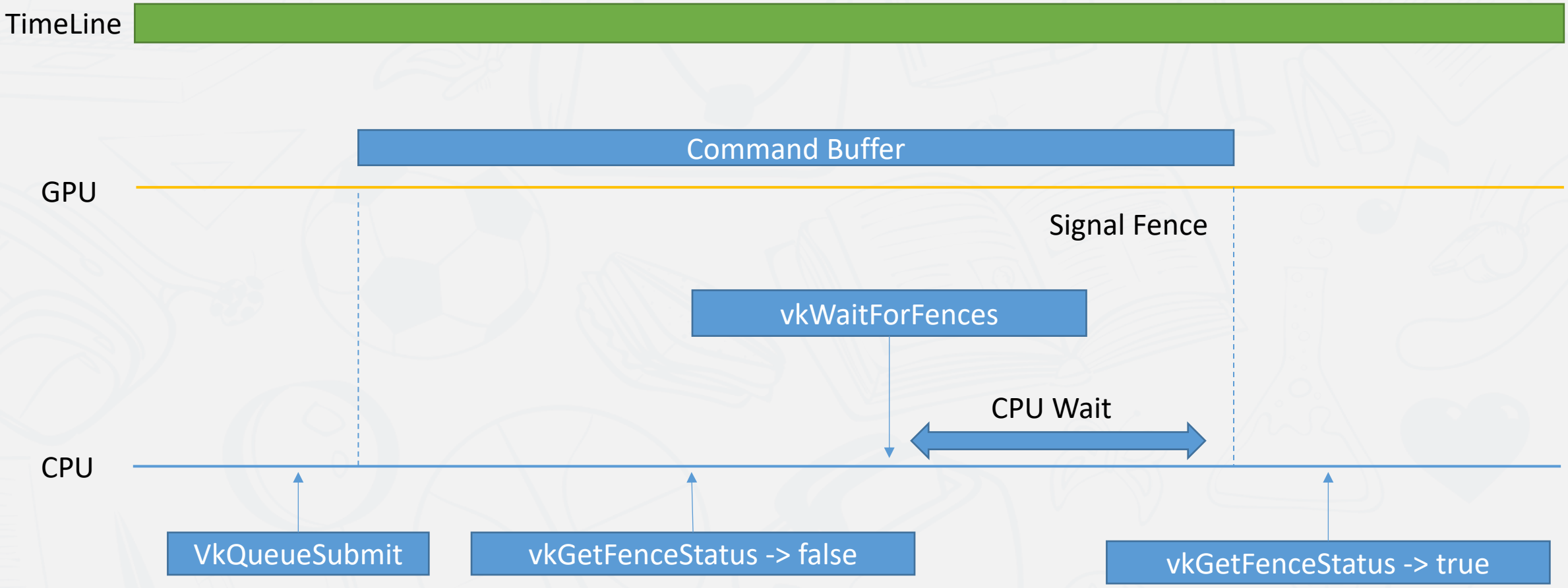


同步概览

屏障（barrier）
- 管理内存访问
- 管线各个状态的资源变化
  - 全局内存屏障（MemoryBarrier）
    - 全局内存的读写控制
  - 缓冲区内存屏障（BufferMemoryBarrier）
    - Buffer的读写控制
  - 图形内存屏障（ImageMemoryBarrier）
    - Image的读写控制

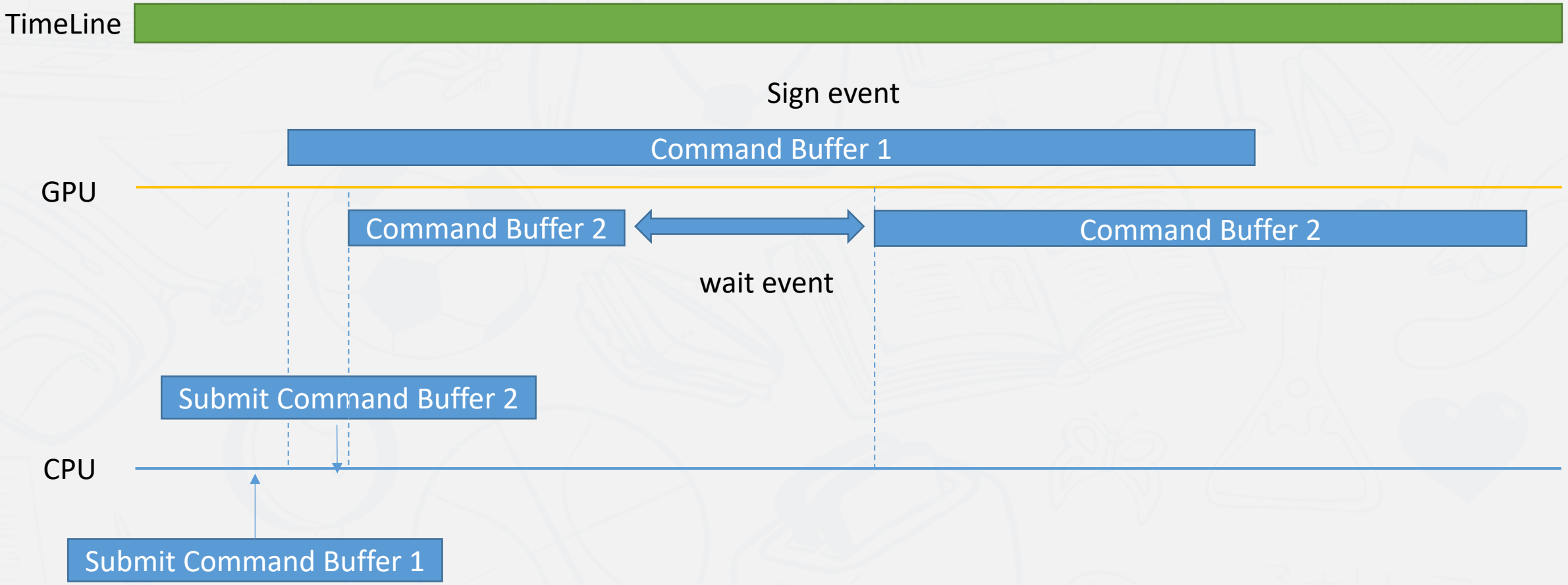


fence



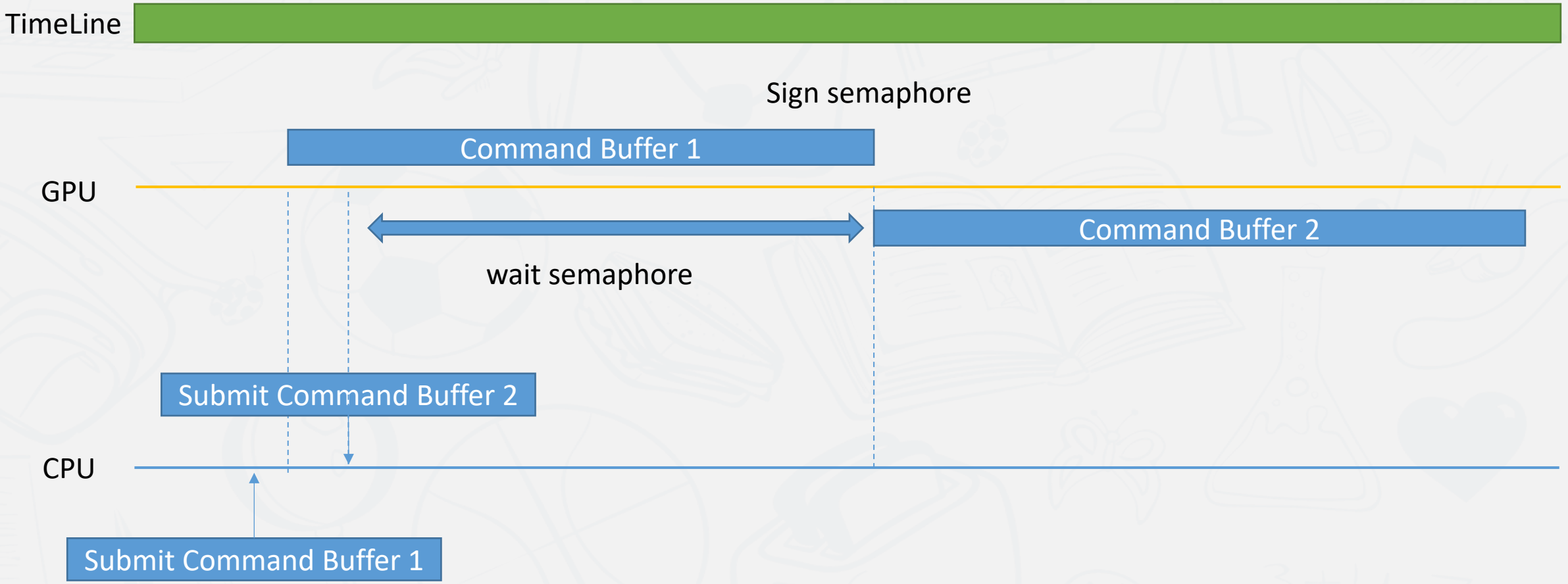
CPU 通过fence等待GPU Command Buffer 运行结束

event



Command Buffer2 等待 Command Buffer1 的Event执行结束

semaphore



Command Buffer2 等待 Command Buffer1 执行结束

barrier

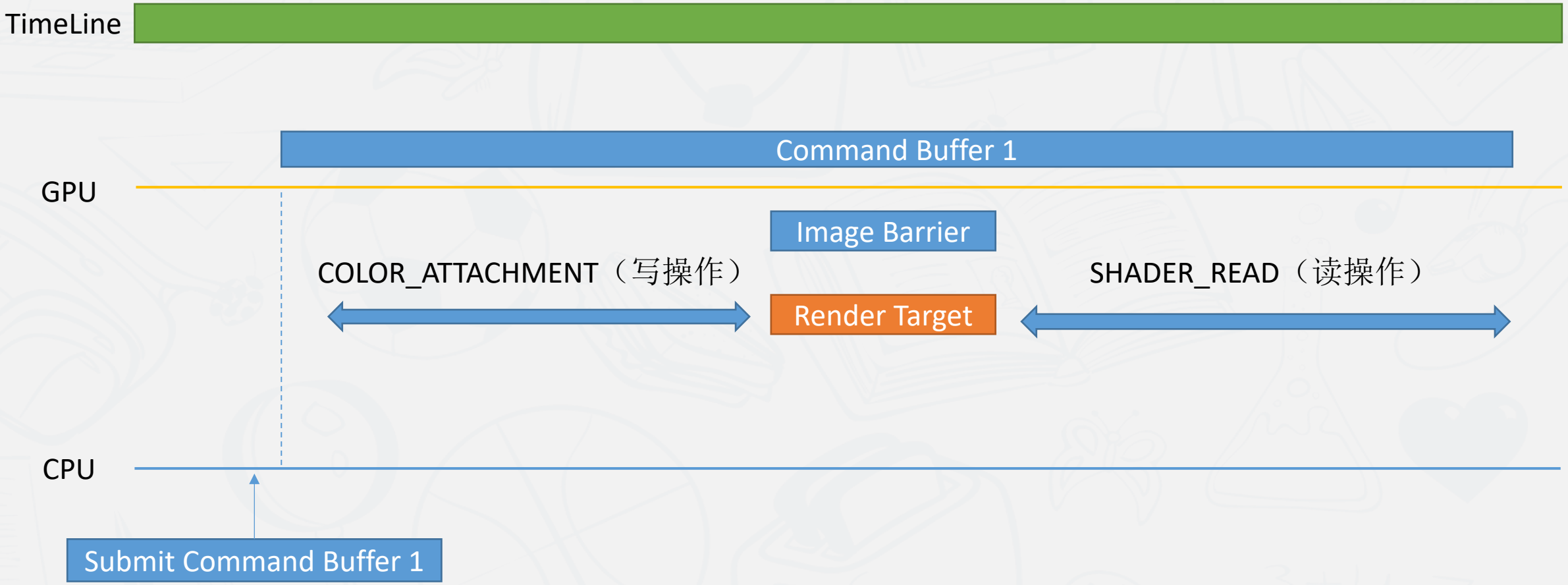
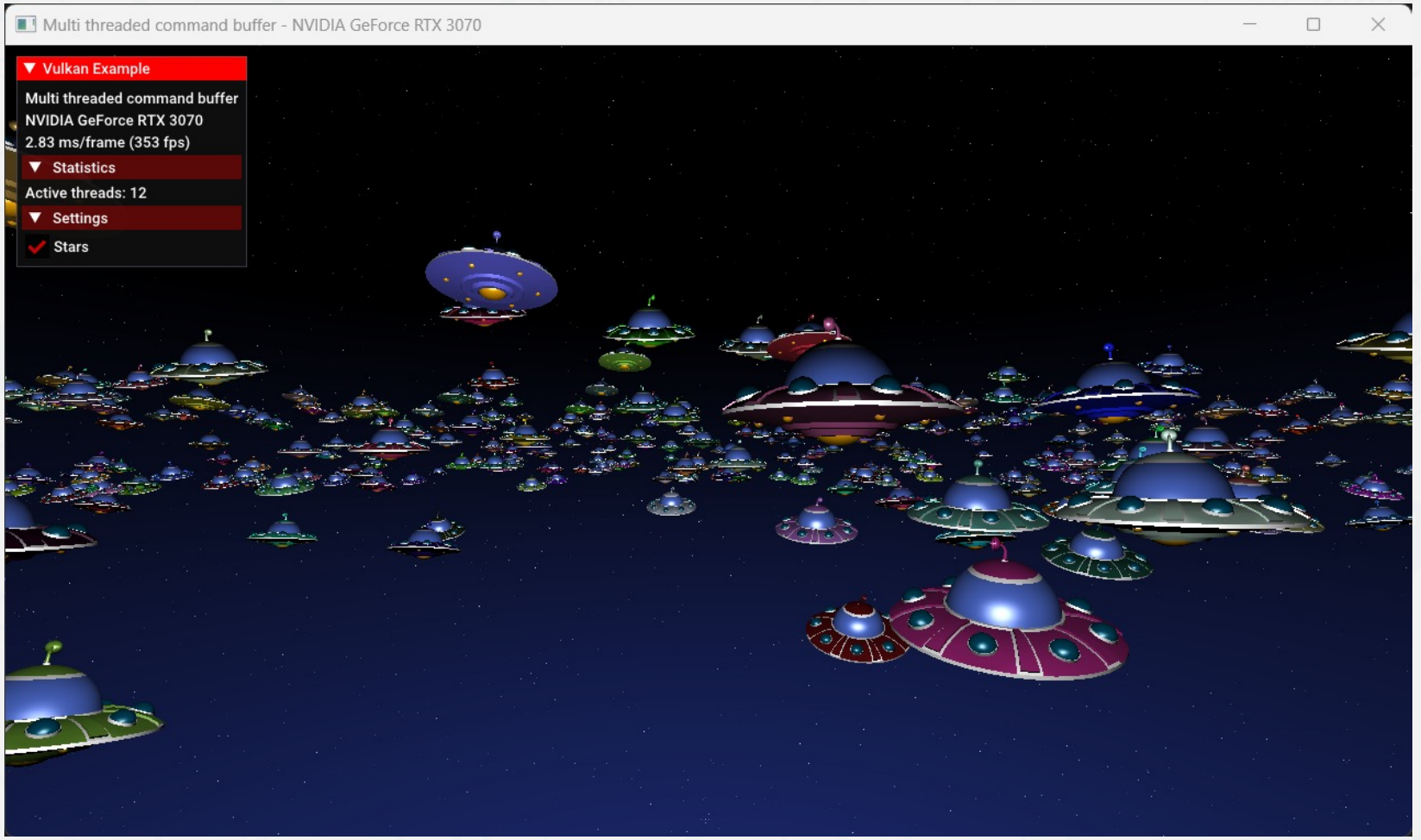


Image Barrier 屏蔽Render Target的读和写操作

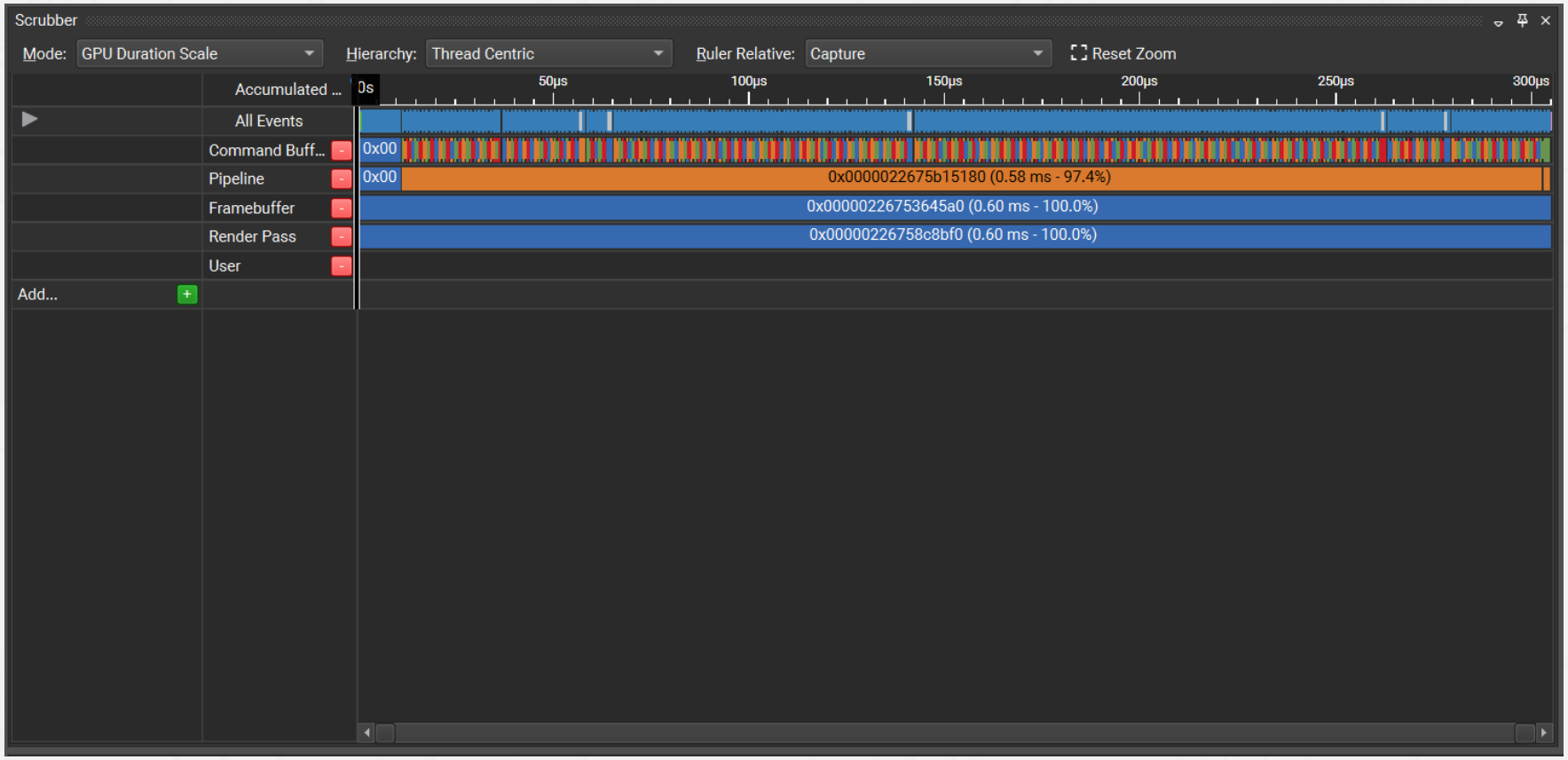
Multithreading Sample

- 简单的多线程，没有资源同步
- 每一个线程创建一个Command Buffer。
- 每一个Command Buffer绘制一个UFO
- 在同一个render pass中绘制



Multithreading Sample

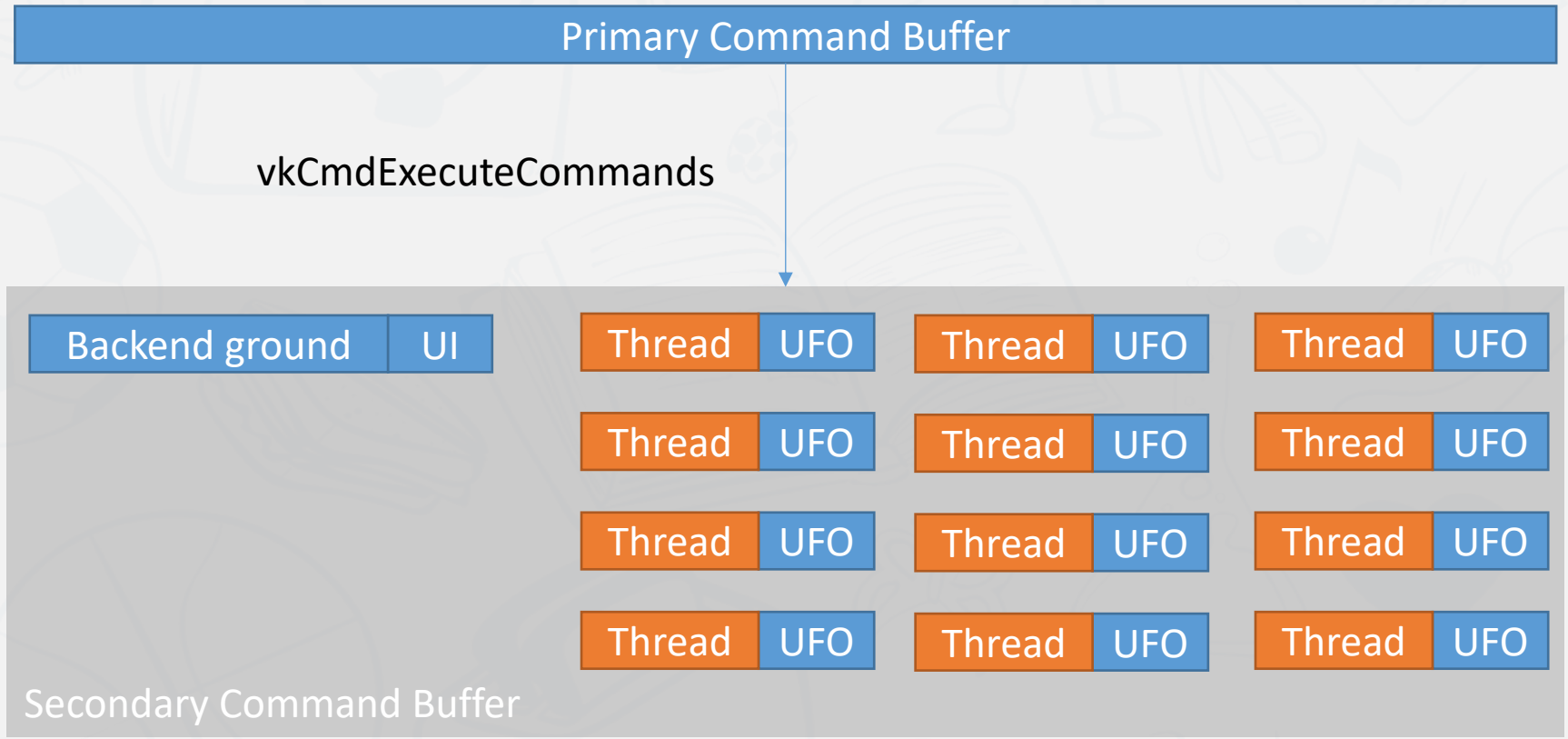
- 简单的多线程，没有资源同步
- 每一个线程创建一个Command Buffer。
- 每一个Command Buffer绘制一个UFO
- 在同一个render pass中绘制



Nsight 抓帧截图

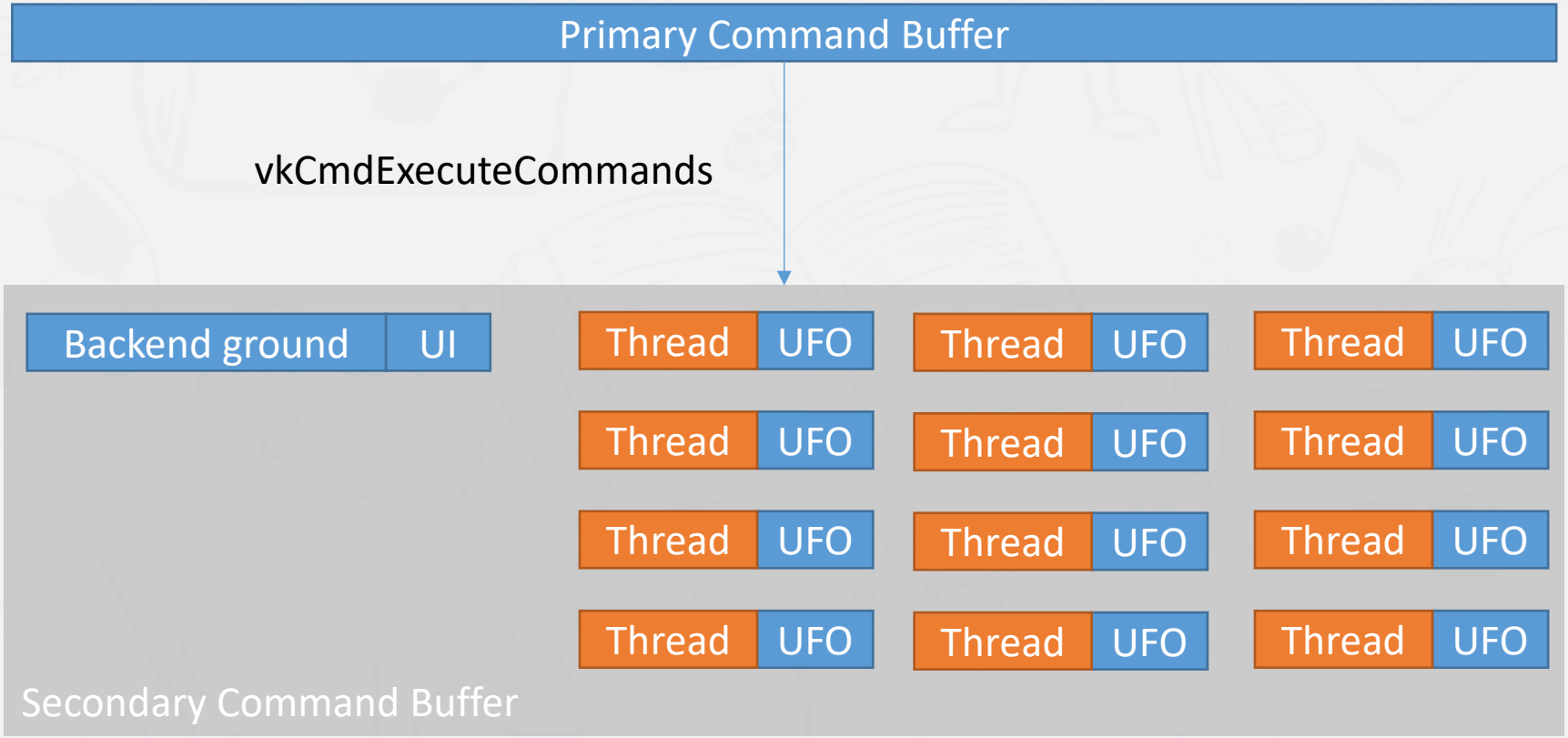
Multithreading Sample

- 简单的多线程，没有资源同步
- 每一个线程创建一个Command Buffer。
- 每一个Command Buffer绘制一个UFO
- 在同一个render pass中绘制



Multithreading Sample

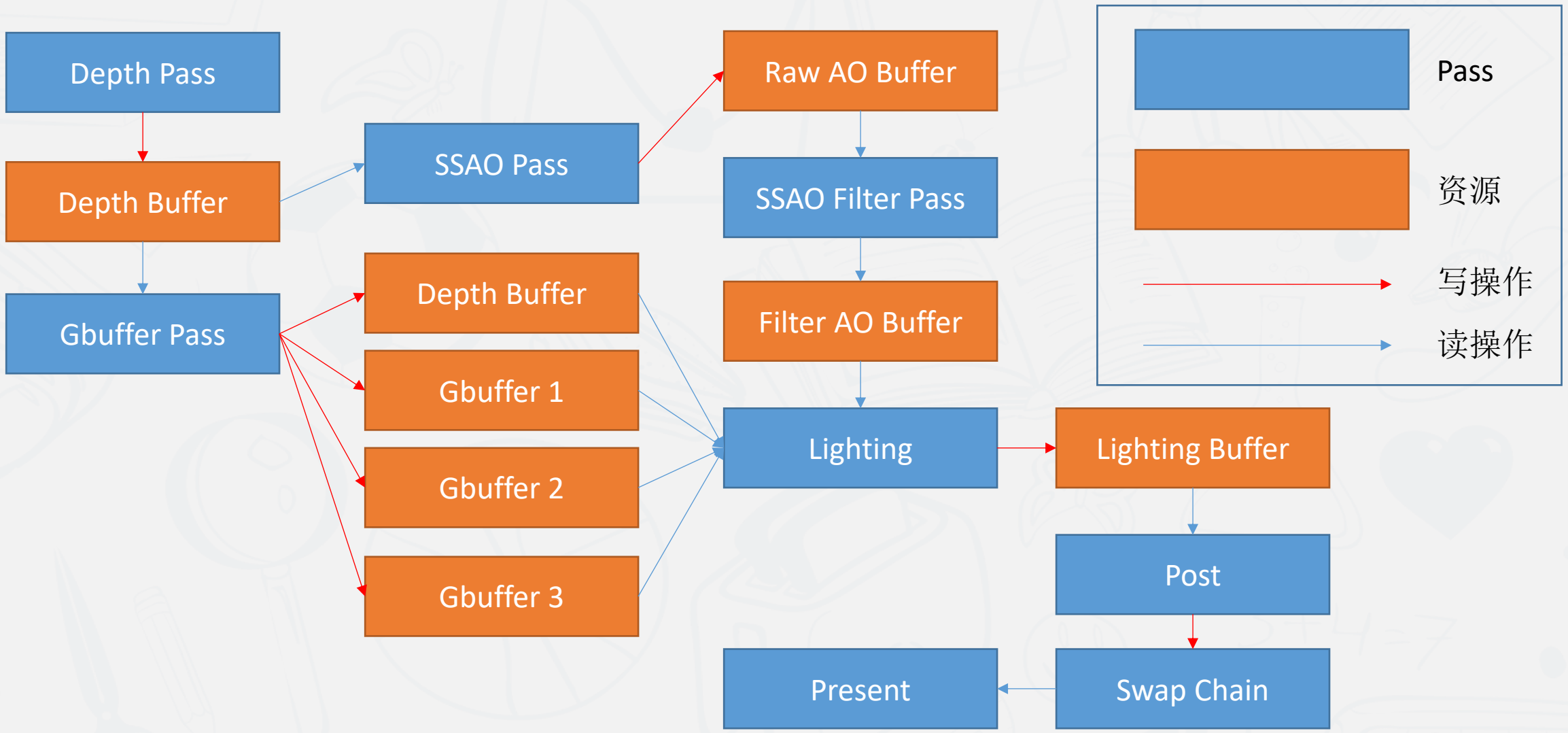
- 简单的多线程，没有资源同步
- 每一个线程创建一个Command Buffer。
- 每一个Command Buffer绘制一个UFO
- 在同一个render pass中绘制



如何处理资源竞争？

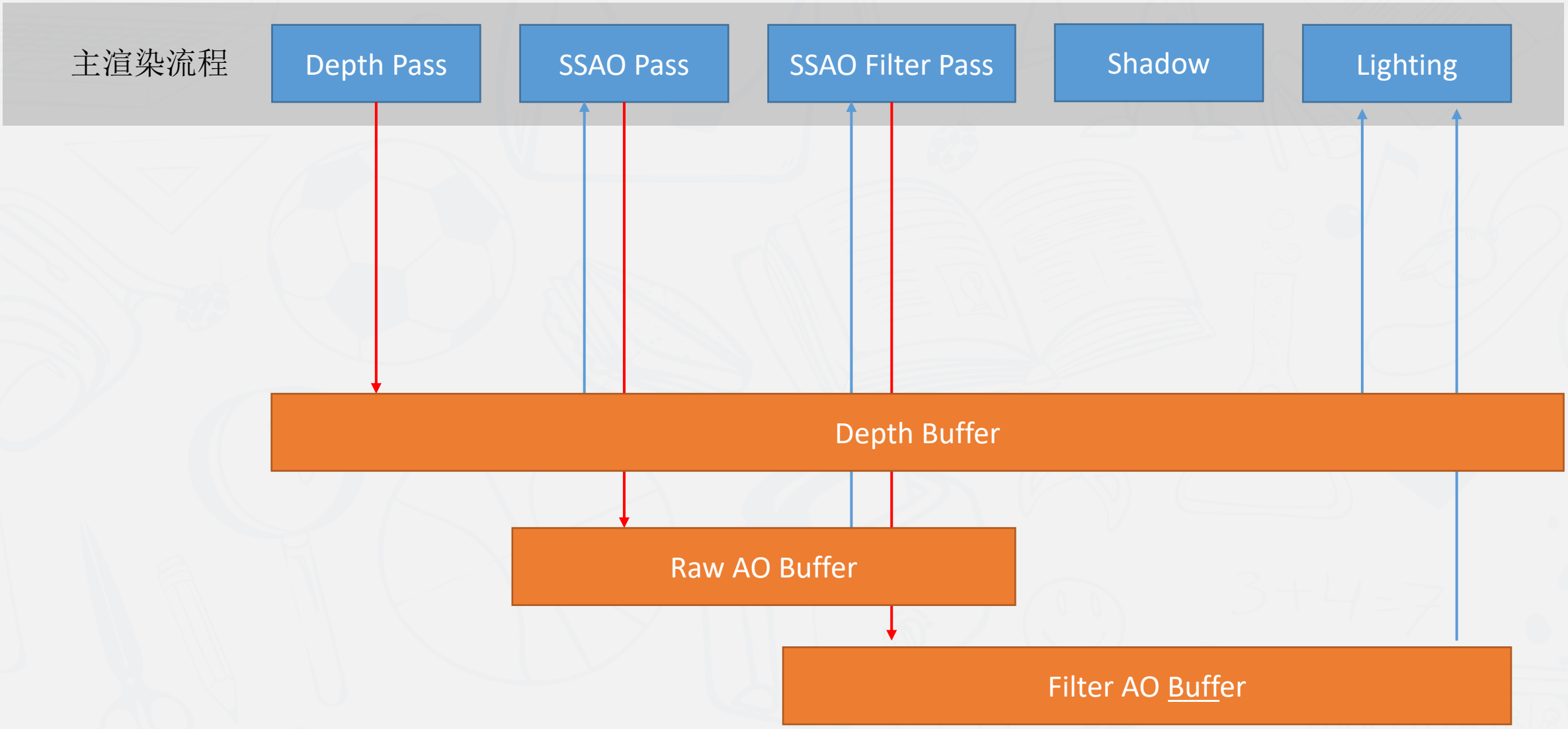
Frame Graph

简单的FG


Depth Pass
Depth Buffer
Gbuffer Pass
SSAO Pass
Depth Buffer
Gbuffer 1
Gbuffer 2
Gbuffer 3
Raw AO Buffer
SSAO Filter Pass
Filter AO Buffer
Lighting
Lighting Buffer
Post
Swap Chain
Present
Pass
资源
写操作
读操作

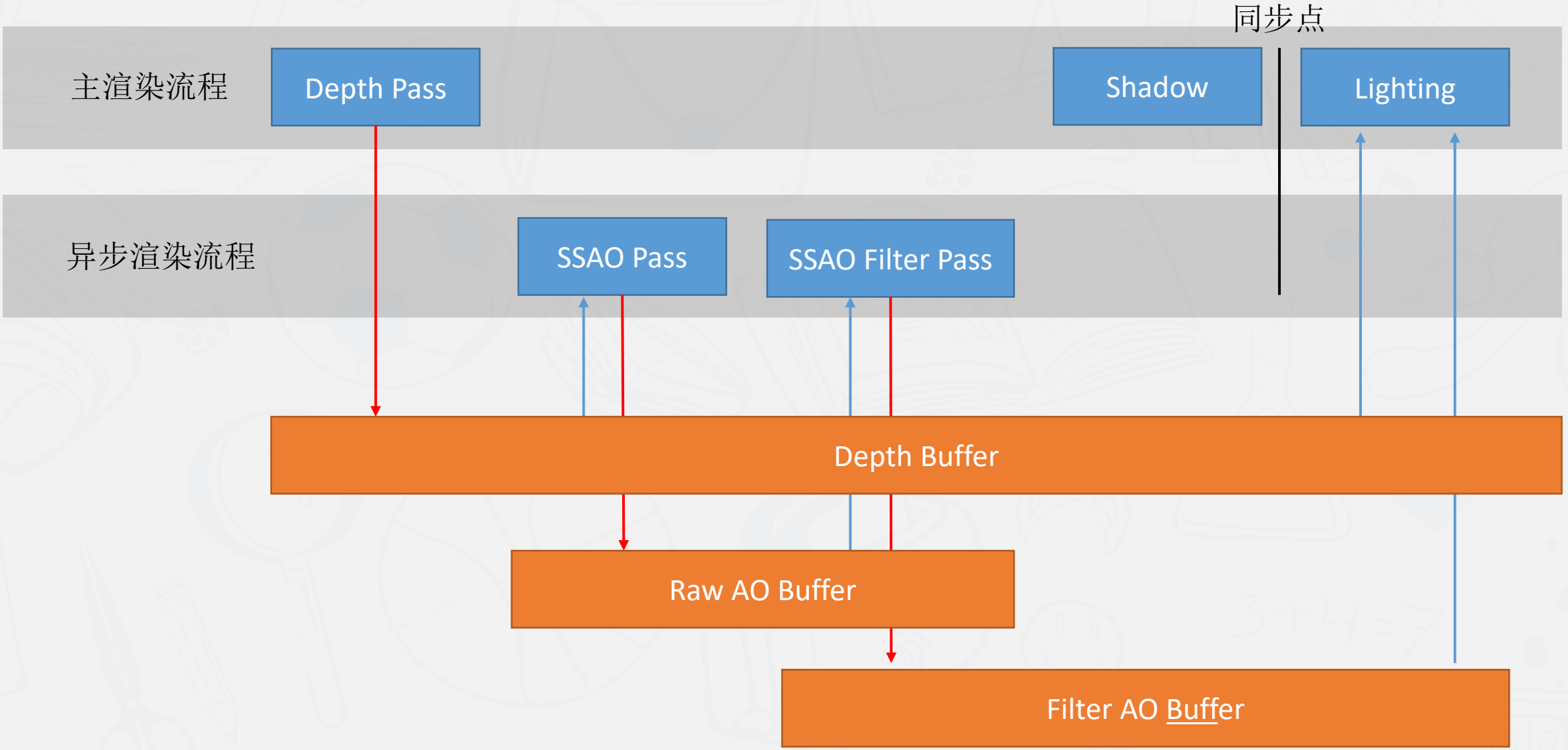
渲染流程

Frame Graph

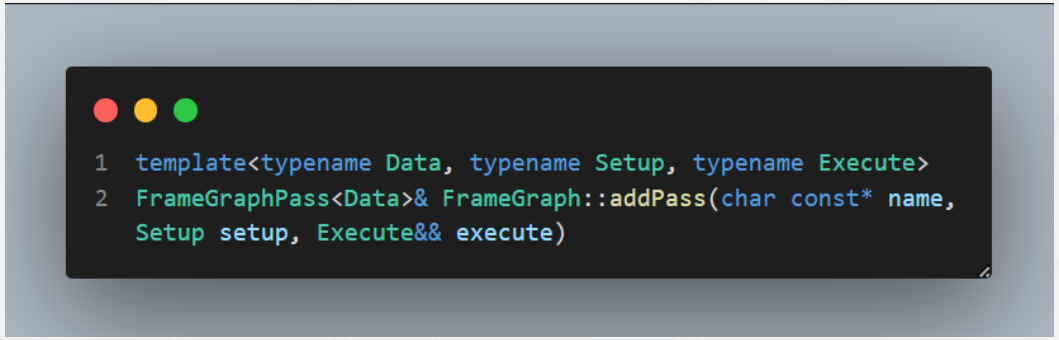


异步渲染流程

Frame Graph



Filament 的实现



```
1  template<typename Data, typename Setup, typename Execute>
2  FrameGraphPass<Data>& FrameGraph::addPass(char const* name,
   Setup setup, Execute&& execute)
```

接口原型

Filament 的实现

Frame Graph

```cpp
template<typename Data, typename Setup, typename Execute>
FrameGraphPass<Data>& FrameGraph::addPass(char const* name,
Setup setup, Execute&& execute)
```

```cpp
auto& SSAOPass = fg.addPass<SSAOPassData>("SSAO Pass",
[&](FrameGraph::Builder& builder, auto& data) {
    auto const& desc = builder.getDescriptor(depth);

    data.depth = builder.sample(depth);
    data.ssao = builder.createTexture("SSAO Buffer", {
            .width = desc.width,
            .height = desc.height,
            .depth = computeBentNormals ? 2u : 1u,
            .type = Texture::Sampler::SAMPLER_2D_ARRAY,
            .format = (lowPassFilterEnabled || highQualityUpsampling || computeBentNormals) ?
                    TextureFormat::RGB8 : TextureFormat::R8
    });

    data.ao = builder.createSubresource(data.ssao, "SSAO attachment", { .layer = 0 });
    data.bn = builder.createSubresource(data.ssao, "Bent Normals attachment", { .layer = 1 });
    data.ao = builder.write(data.ao, FrameGraphTexture::Usage::COLOR_ATTACHMENT);
    data.bn = builder.write(data.bn, FrameGraphTexture::Usage::COLOR_ATTACHMENT);

    auto depthAttachment = data.depth;
    depthAttachment = builder.read(depthAttachment,
            FrameGraphTexture::Usage::DEPTH_ATTACHMENT);
    builder.declareRenderPass("SSAO Target", {
            .attachments = { .color = { data.ao, data.bn }, .depth = depthAttachment },
            .clearColor = { 1.0f },
            .clearFlags = TargetBufferFlags::COLOR0 | TargetBufferFlags::COLOR1
    });
},
```

```cpp
[=](FrameGraphResources const& resources,
        auto const& data, DriverApi& driver) {
    auto depth = resources.getTexture(data.depth);
    auto ssao = resources.getRenderPassInfo();
    auto const& desc = resources.getDescriptor(data.depth);

    auto& material = getPostProcessMaterial("saoBentNormals");

    FMaterialInstance* const mi = material.getMaterialInstance(mEngine);
    mi->setParameter("depth", depth, {
            .filterMin = SamplerMinFilter::NEAREST_MIPMAP_NEAREST });

    mi->commit(driver);
    mi->use(driver);

    PipelineState pipeline(material.getPipelineState(mEngine));
    pipeline.rasterState.depthFunc = RasterState::DepthFunc::L;
    assert_invariant(ssao.params.readOnlyDepthStencil & RenderPassParams::READONLY_DEPTH);
    render(ssao, pipeline, driver);
});
```

Setup，构建pass的输入输出，资源占用

Execute 创建command buffer 执行渲染

Vulkan优化

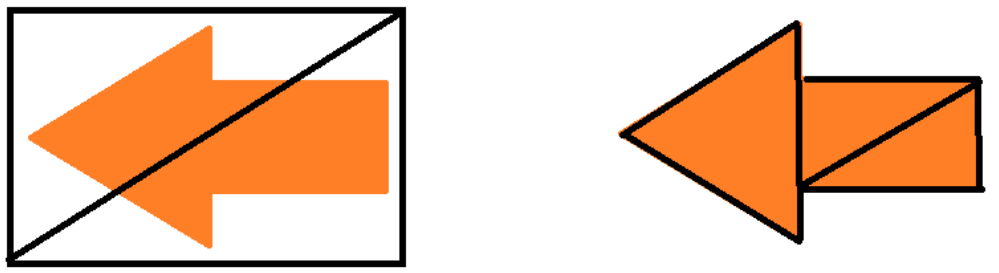
移动端常见优化

- Vertex Shader
  - 使用Index Draw Call 无论顶点是否被复用
  - 不使用32位的index buffer
- Fragment Shader
  - shader中使用半精度浮点数half进行计算
  - 不需要做pre depth
  - 避免使用alpha blend，discard 等会破坏early z的渲染方式
  - render pass 中的attachment 使用LOAD_OP_CLEAR 或者 LOAD_OP_DONT_CARE.
  - 不要使用uber-shader中
  - 减少一次draw call的代码量
  - 减少通用寄存器（general purpose register（GPR））的使用
    - 两个vec2 打包成一个vec4
    - 短的循环代码，展开来写

参考资料

- Arm GPU Best Practices Developer Guide
- Guides @ Snapdragon Game Toolkit (qualcomm.com)

移动端常见优化



生成UI对应的几何，替代用正方形

alpha混合UI贴图

- Vertex Shader
  - 使用Index Draw Call 无论顶点是否被复用
  - 不使用32位的index buffer
- Fragment Shader
  - shader中使用半精度浮点数half进行计算
  - 不需要做pre depth
  - 避免使用alpha blend，discard 等会破坏early z的渲染方式
  - render pass 中的attachment 使用LOAD_OP_CLEAR 或者 LOAD_OP_DONT_CARE.
  - 不要使用uber-shader中
  - 减少一次draw call的代码量
  - 减少通用寄存器（general purpose register（GPR））的使用
    - 两个vec2 打包成一个vec4
    - 短的循环代码，展开来写

参考资料

- Arm GPU Best Practices Developer Guide
- Guides @ Snapdragon Game Toolkit (qualcomm.com)

移动端常见优化

- Vertex Shader
  - 使用Index Draw Call 无论顶点是否被复用
  - 不使用32位的index buffer
- Fragment Shader
  - shader中使用半精度浮点数half进行计算
  - 不需要做pre depth
  - 避免使用alpha blend，discard 等会
  - render pass 中的attachment 使用LO
  - 不要使用uber-shader中
  - 减少一次draw call的代码量
  - 减少通用寄存器（general purpose
    - 两个vec2 打包成一个vec4
    - 短的循环代码，展开来写

```
1  展开如下所示的循环
2  for (i = 0; i < 4; ++i) {
3      diffuse += ComputeDiffuseContribution(normal, light[i]);
4  }
5
6  代码片段将替换为
7  diffuse += ComputeDiffuseContribution(normal, light[0]);
8  diffuse += ComputeDiffuseContribution(normal, light[1]);
9  diffuse += ComputeDiffuseContribution(normal, light[2]);
10 diffuse += ComputeDiffuseContribution(normal, light[3]);
```

参考资料

- Arm GPU Best Practices Developer Guide
- Guides @ Snapdragon Game Toolkit (qua

灯光渲染的例子

Deferred lighting

```
CPU:
for mesh in scene mesh:
    gbuffer(mesh)
for light in lights:
    deferred_light_pass(light, gbuffer)

light_pass shader:
shadow(light, gbuffer)
lighting(light, mesh attribute)
```

VS

Forward lighting

```
CPU:
for mesh in scene mesh:
    forward_lighting_pass(lights, mesh)

forward_lighting_pass Shader:
for light in lights:
    shadow(light, mesh attribute)
    lighting(light, mesh attribute)
```

- 每一个pass都很轻量
- 对灯光数量没有太大的限制
- Vulkan Subpass可以解决带宽的问题，实际的硬件实现并不好

- 更适配移动端Tile-based rendering
- Pass的代码量会很多
- 灯光数量有限制，适合只有1-3个灯光场景

灯光渲染的例子

实现方式

- 限制方向光的数量以及阴影的数量
- 依然使用Forward lighting
- 每一个带阴影的灯光用单独的pass，因为阴影的shader计算代价很大
- 在一个pass中计算不带阴影的灯光计算，灯光上限为6个
- 有更多的不带阴影的点光源和方向感使用cluster lighting

```
for mesh in scene mesh:
    for light in shadow lights:
        forward_shadow_lighting_pass(lights, mesh)
    forward_no_shadow_lighting_pass(Noshadow lights, mesh)
    if (many Noshadow point/spot lights)
        cluster_lighting_pass(many Noshadow point/spot lights, mesh)

```

优点

- 一定程度地放宽了灯光的限制。
- 没有Deferred lighting对带宽的要求那么大

缺点

- Draw Call数量变得更多
- 透明物体不能很好的计算正确的混合结果

谢谢