基于Vulkan的场景渲染

1. 简介

在场景中画出墙、放置门、桌椅等等东西，并在三维体系中沿着规定的路线浏览。场景渲染使用Vulkan图形API，使用语言为C++（17）。 模仿Vulkan提供的官方教程，并进行重构，采用模块化设计，基于面向对象编程思想。

Vulkan作为下一代图形API以及OpenGL的继承者，保留了GL跨平台和开发等特性，重要的是提供了高性能接口，强化了底层多线程使用。

1. 环境设置

Vulkan官方教程中用到了以下的环境：

* [GLFW3](https://github.com/glfw/glfw) 用于创建窗口
* [Vulkan SDK](https://vulkan.lunarg.com/sdk/home) Vulkan开发工具（1.1.126.0 ver.）
* [glm](https://github.com/g-truc/glm) 图像软件数学库
* [stb](https://github.com/nothings/stb) 简单易用的文件加载库
* [tinyobjloader](https://github.com/syoyo/tinyobjloader) obj模型加载库

而在实际应用中，本程序还依赖了以下环境

* [shaderc](https://github.com/google/shaderc) 着色器语言编译器api
* [boost](https://github.com/boostorg/boost) C++标准库扩展

同时，程序不直接使用vulkan api中的头文件，而是使用了vulkan sdk中提供的C++封装后的头文件vulkan.hpp。

1. 结构设计
2. utility

存放一些编程中用到的基本工具，如时间、常数、模板函数和模板元编程特性等。

1. vulkan\utility
2. Vulkan本身出现时间不长，许多api还比较原始，同样功能实现相比OpenGL需要写数倍代码，给使用带来了很多困难，所以在项目中提供了一些常用的特性封装。
3. constant中存放了vulkan一些常数以及类型。
4. info提供了info\_proxy类封装vulkan中含有指针的info类，方便内存管理。同时使用模板特化方式指定规定的vulkan handle对象的信息类型。
5. UniqueHandle提供了Vulkan对象自动化内存管理。object为vulkan.hpp的UniqueHandle进一步封装，包含info\_proxy类。 同时使用模板全特化方式简化对象创建。例如：

// Specializations implemented

        [[nodiscard]] static typename base::base create\_unique\_handle(

            const owner\_type&,

            const dispatch\_type&,

            const base\_info\_type&,

            const optional<AllocationCallbacks> & = nullopt

        );

template<>

    auto object<Fence>::create\_unique\_handle(

        const owner\_type& owner,

        const dispatch\_type& dispatch,

        const base\_info\_type& info,

        const optional<AllocationCallbacks>& allocator

    ) -> base::base

    {

        return owner.createFenceUnique(info, allocator ? Optional{\*allocator} : nullptr, dispatch);

    }

那么初始化时只需要简单调用：

template<typename HandleType>

void object<HandleType>::initialize(

    const owner\_type& owner,

    const dispatch\_type& dispatch,

    const optional<AllocationCallbacks>& allocator

)

{

    base::base::operator=(create\_unique\_handle(owner, dispatch, this->info(), allocator));

}

...

fence.initialize(device\_)

1. static\_memory提供类型安全的图形设备（即GPU）特定类型内存管理。第一层模板指定类型以及是否为缓存。存在三个个嵌套类，基类提供基本初始化操作，绑定缓冲区对应的物理内存。两个子类，一个是每种类型固定大小的数组，一个是每种类型运行时才确定大小数组。例如定义一个顶点与对应索引的内存对象：

static\_memory<true,vertex, uint32\_t>::vector\_values transfer\_memory\_{};

在初始化时确定大小以及buffer的种类：

transfer\_memory\_ = decltype(transfer\_memory\_){

    \*physical\_device\_,

    device\_,

    array<BufferUsageFlags, 2>{BufferUsageFlagBits::eVertexBuffer, BufferUsageFlagBits::eIndexBuffer},

    {vertices.size(), indices.size()}

};

通过模板反射使得以类型安全方式写入数据：

transfer\_memory\_.initialize(\*physical\_device\_);

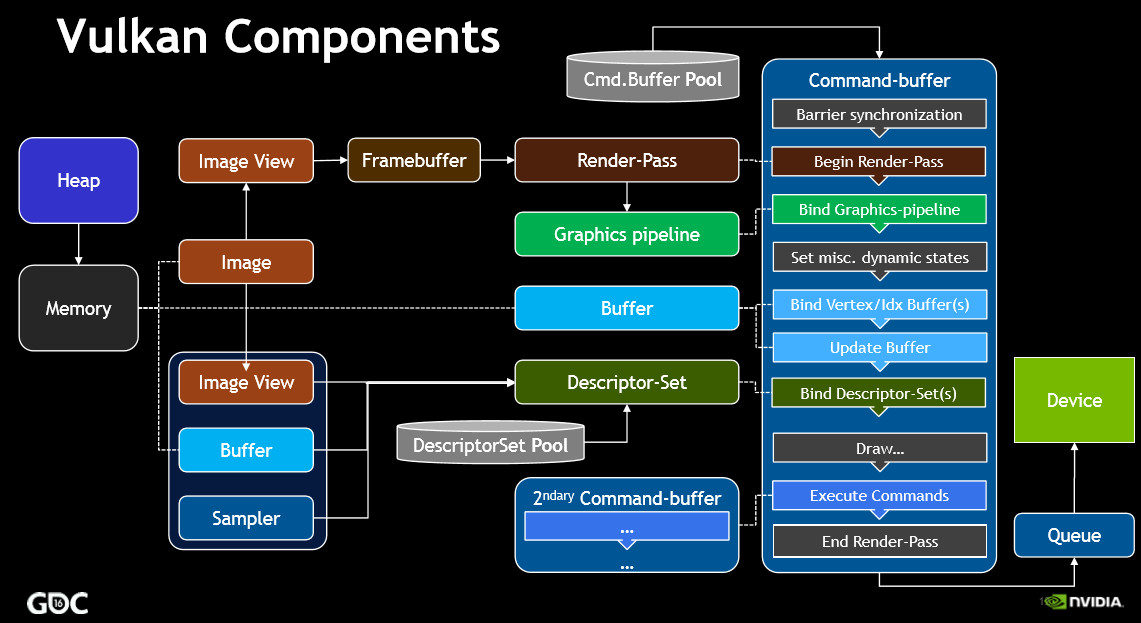
transfer\_memory\_.write(std::move(vertices));

transfer\_memory\_.write(std::move(indices));

1. image是vulkan image的封装。对image的创建和其他基本操作进行封装。
2. stb image提供外部图像操作。
3. shaderc用于编译着色器语言。
4. vulkan\_sample

依照Vulkan教程步骤，按顺序填充各个对象创建信息并初始化，并提供接口改变一些参数。

1. 实现



1. 基本组件初始化

* 底层api组件instance、physical device、device
* 窗口创建组件surface、window、
* 基本渲染组件render pass、pipeline、swap chain、image view、frame buffer、command buffer、queue
* CPU与GPU渲染同步组件 fence、semaphore

1. 模型绘制

使用obj格式的模型

* 1. 在generate\_texture\_image\_create\_info中初始化模型用到的图像。
  2. 在generate\_model中获取模型并初始化相关变量。
  3. 在generate\_buffer\_allocate\_info中对模型的每个mesh遍历获取顶点（vertex）、对应顶点索引（index）与材质（material）。为了程序简单，这里只用到了第一个材质中纹理图片。

for(const auto& shape : model\_.shapes)

{

    const auto& mesh = shape.mesh;

    meshes\_.push\_back(

        {

            static\_cast<uint32\_t>(indices.size()),

            static\_cast<uint32\_t>(mesh.indices.size()),

            mesh.material\_ids.size() ?

            &texture\_image\_map\_[path{model\_.materials[mesh.material\_ids.front()].diffuse\_texname}.stem().

                generic\_u8string()] :

            nullptr

        }

    );

    for(const auto& index : mesh.indices)

    {

        vertex vertex = {

            {

                model\_.attribute.vertices[3 \* index.vertex\_index + 0],

                model\_.attribute.vertices[3 \* index.vertex\_index + 1],

                model\_.attribute.vertices[3 \* index.vertex\_index + 2]

            },

            vec3{1},

            {

                model\_.attribute.texcoords[2 \* index.texcoord\_index],

                1 - model\_.attribute.texcoords[2 \* index.texcoord\_index + 1]

            }

        };

        auto&& it = vertices\_map.find(vertex);

        if(it == vertices\_map.cend())

        {

            it = vertices\_map.insert(

                {vertex, static\_cast<decltype(vertices\_map)::mapped\_type>(vertices\_map.size())}

            ).first;

            vertices.push\_back(vertex);

        }

        indices.push\_back(it->second);

    }

}

* 1. 将顶点、索引写入到device memory中，包含两个步骤
     1. 写入到local buffer

transfer\_memory\_.write(std::move(vertices));

transfer\_memory\_.write(std::move(indices));

* + 1. 利用command buffer将local buffer传送到device buffer

transfer\_memory\_.write\_transfer\_command(front\_command\_buffer);

* 1. 为每个需要绘制的mesh绑定不同的descriptor set，descriptor set用于指定对应texture（mesh.texture->image\_view()）。

for(const auto& mesh : meshes\_)

{

    device\_->updateDescriptorSets(

        {

            info\_proxy<WriteDescriptorSet>{

                {},

                {{\*transform\_buffer\_, 0, whole\_size<decltype(DescriptorBufferInfo::range)>}},

                {},

                {\*\*mesh.descriptor\_set, 0, 0, 1, DescriptorType::eUniformBuffer}

            },

            info\_proxy<WriteDescriptorSet>{

                {{\*texture\_sampler\_, \*mesh.texture->image\_view(), ImageLayout::eShaderReadOnlyOptimal}},

                {},

                {},

                {\*\*mesh.descriptor\_set, 1, 0, 1, DescriptorType::eCombinedImageSampler}

            }

        },

        {},

        device\_.dispatch()

    );

}

* 1. 遍历mesh，命令缓冲区绑定对应的descriptor set并绘制mesh的顶点。

for(const auto& mesh : meshes\_)

{

    buffer->bindDescriptorSets(

        PipelineBindPoint::eGraphics,

        \*pipeline\_layout\_,

        0,

        \*\*mesh.descriptor\_set,

        {},

        device\_.dispatch()

    );

    buffer->drawIndexed(mesh.index\_count, 1, mesh.first\_index, 0, 0, device\_.dispatch());

}

1. 相机位置变换

同样是通过descriptor set，绑定的uniform buffer，下述为vertex shader中的代码。

layout(binding = 0) uniform transform { mat4 mat; }tf;

...

gl\_Position = tf.mat \* vec4(in\_position, 1.0);

在descriptor set创建之前要指定布局，绑定shader中的变量

DescriptorSetLayoutBinding{0, DescriptorType::eUniformBuffer, 1, ShaderStageFlagBits::eVertex}

在generate\_render\_info中将descriptor set与uniform buffer绑定，并提供修改uniform buffer数据的函数。

void vulkan\_sample::set\_transform(decltype(transform\_mat\_) mat)

{

    transform\_mat\_ = std::move(mat);

    write(transform\_buffer\_memory\_, device\_, transform\_mat\_);

}

最后在main.cpp中进行更新，具体的变换会涉及到数学知识与MVP模型，这里不多展开。

1. 总结

项目从暑假开始构思，图形api没有选择D3D或OpenGL，而是Vulkan，是因为想锻炼自己能力，事前已经知道它的学习难度，加上资源不足，只能自己一点点摸坑。Vulkan官方提供了教程（[vulkan-tutorial](https://vulkan-tutorial.com/Development_environment)），但是代码风格完全不是C++，IDE里数百个warning，结构混乱。在几个月时间内把内容读完并按照自己理解重构，一步步实现相同功能，从三角形到矩形，再到图片，最后是模型。Vulkan在各个组件初始化时需要显式指明各种繁杂信息，往往错一个参数就会抛出异常或者导致最后结果不正确。

最后选择模型加载时思考过使用gltf模型，同为Khronos组织开发，gltf格式的模型未来会有广泛使用趋势，但无奈学习资源少，只有SaschaWillems提供的[Vulkan-glTF-PBR](https://github.com/SaschaWillems/Vulkan-glTF-PBR)有完整例子。而代码和tutorial风格类似，并且更加复杂，在尝试几天后发现不可能重构抽取里面的加载模型代码，这个方案最终被废弃。（vulkan/gltf）