Компьютерная графика
Современные технологии компьютерной графики и рендеринга

Основы работы с Vulkan. Архитектура и первый контакт

Лекция 10

02.04.02 ФИИТ

Разработка мобильных приложений и компьютерных игр

2025-2026

Почему Vulkan? Философия API

Vulkan — **НЕ замена** OpenGL, это **параллельная** ветка для **высоконагруженных** систем.

Его создавали, оглядываясь на современных гигантов вроде Unreal Engine 5 и Doom Eternal.

Vulkan имеет одно неоспоримое преимущество — **полную кросс-платформенность**, что позволяет разрабатывать приложение одновременно под Linux, Windows и Android

Взамен, вам придется работать с менее абстрактным и более сложным API.

Каждая мелочь, связанная с API, будет настраиваться вами с нуля.

Драйвер будет меньше вас ограничивать, а это означает, что вам придется проделать больше работы, чтобы обеспечить правильное поведение вашего приложения.

Рождение Vulkan. Историческая справка

2013: AMD анонсирует Mantle (низкоуровневый API для архитектуры GCN (Graphics Core Next))

2015: Khronos Group принимает Mantle как основу для Next Generation OpenGL

16 февраля 2016: Релиз Vulkan 1.0

2018: Vulkan 1.1 (многоустройственность)

2019: Vulkan 1.2 (интеграция расширений)

2022: Vulkan 1.3 (упрощение, 8 ключевых расширений включены в ядро)

Плати за то, что используешь — ничего за спиной

Сравнение производительности

Draw Calls per frame	OpenGL FPS	Vulkan FPS
1,000	280	275
10,000	95	245
50,000	18	190
100,000	4	98

Вывод: Vulkan масштабируется на сложные сцены, OpenGL упирается в CPU overhead

Многопоточность как главная фича

Схема OpenGL:

[Main Thread] → glDraw (ждать) → [GPU]
(остальные ядра CPU простаивают)

Схема Vulkan:

[Thread 1] → Record CB1 ↘
[Thread 2] → Record CB2 → [Main Thread] → QueueSubmit → [GPU]
[Thread 3] → Record CB3 ↗
(все ядра заняты работой)

Ключевой API: vkBeginCommandBuffer из любого потока

Мифы и реальность Vulkan

Миф	Реальность
Vulkan всегда быстрее	Только если сложная сцена (>10k объектов) или есть время оптимизировать
OpenGL умер	WebGL, OpenGL ES, старые проекты — никуда не делись
Vulkan нужен только для игр	Вычисления (GPGPU), VR/AR, CAD/CAM, профессиональная визуализация
На Vulkan нельзя быстро прототипировать	Можно, но с помощью фреймворков (bgfx, Diligent Engine, The Forge)

Когда выбирать Vulkan?

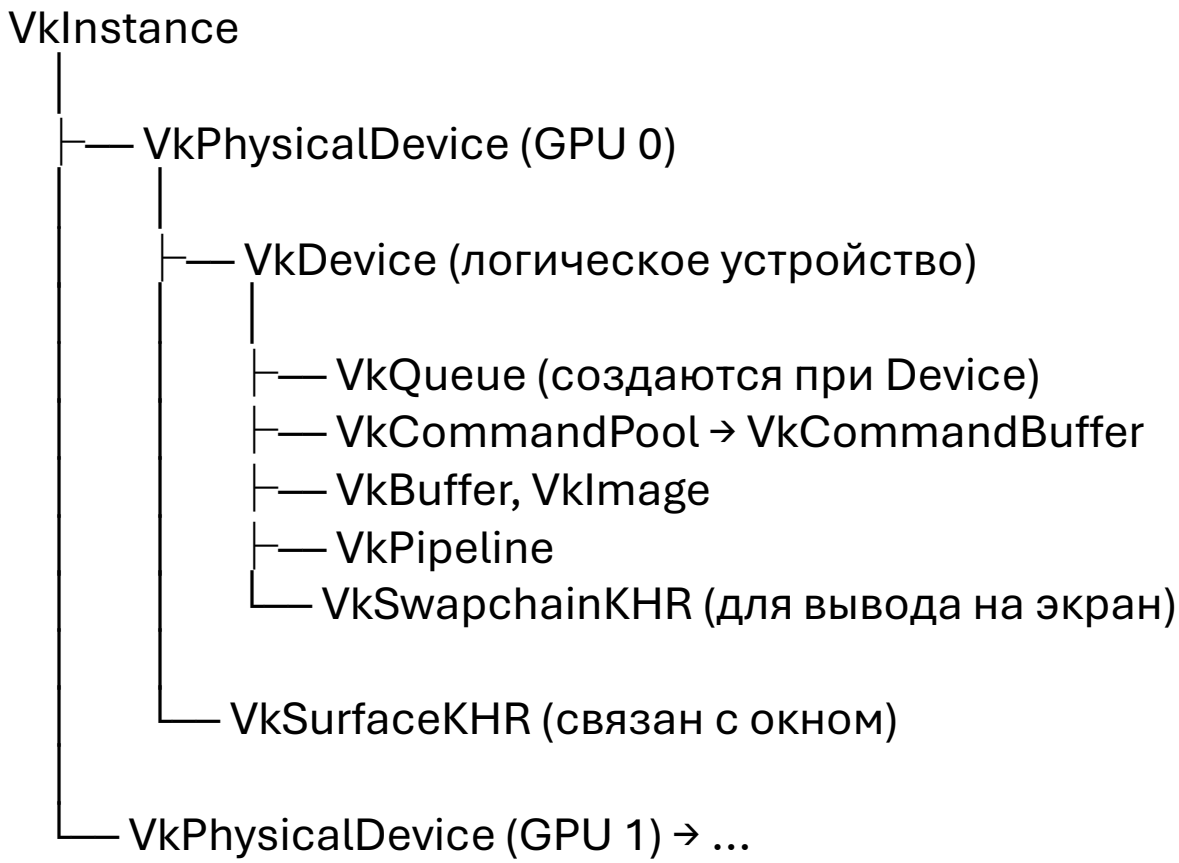
Зеленый свет (да):
- AAA игры и движки (Unreal, id Tech, Godot)
- Симуляторы с миллионами объектов
- AR/VR (каждый миллисекунда на счету)
- Высокочастотный рендеринг (400+ FPS)

Красный свет (нет, используйте OpenGL):
- Прототипирование
- Образовательные проекты
- Простые 2D/3D демки
- Web-приложения (там WebGPU постепенно, но пока WebGL)

Экосистема Vulkan

Ваше приложение
Vulkan Loader (libvulkan.so) — поиск драйверов, выбор GPU
Валидационные слои (отладка) — можно подставить любые слои
Драйвер GPU (NVIDIA/AMD/Intel) — реализует VK_KHR_driver_properties
GPU

Архитектурный обзор: иерархия объектов



Отсутствие глобального состояния

OpenGL (проблемный код):

```
glUseProgram(progA);
glBindTexture(GL_TEXTURE_2D, tex5);
// ... другой поток случайно вызывает glBindTexture ...
glDrawArrays(); // Ошибка: использована tex5 или другая? Неизвестно
```

Vulkan (явное управление):

```
vkCmdBindPipeline(cmdBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, pipelineA);
vkCmdBindDescriptorSets(cmdBuffer, ... descriptorSetWithTex5);
vkCmdDraw(cmdBuffer, ...); // Точно использует tex5
```

Каждый CommandBuffer самодостаточен

Архитектура. Модель: **Экземпляр** -> Устройство -> Очередь

В Vulkan **нет глобального состояния**.
Есть иерархия объектов, которую нужно собрать вручную.

VkInstance (Экземпляр)

Это ваше приложение в мире Vulkan.

При создании вы указываете:

- **Слои (Layers)**: Механизм отладки.
  В дебаг-версии вы включаете VK_LAYER_KHRONOS_validation — и библиотека начнет орать на вас за каждую неправильную передачу указателя или нарушение спецификации.
  В релизе слои отключаются.
- **Расширения (Extensions)**: Vulkan — модульный. Базовое ядро минимально.
  Хотите рисовать в окно Windows? Запросите VK_KHR_win32_surface.
  Хотите трассировку лучей? Запросите VK_NV_ray_tracing.

Архитектура. Модель: Экземпляр -> **Устройство** -> Очередь

VkPhysicalDevice vs VkDevice

Это важнейшее различие с OpenGL.

- **PhysicalDevice (Физическое устройство)**: Это ваша видеокарта.
  У нее есть характеристики: имя, тип (дискретная/встроенная), лимиты (макс. размер текстуры), поддержка геометрических шейдеров.
  - Задача: Вы пишете код, который перечисляет все GPU в системе (например, встроенная Intel и дискретная NVIDIA). Затем вы выбираете самый быстрый или тот, у которого есть нужные вам очереди.

- **LogicalDevice (Логическое устройство)**: Это ваш дескриптор для работы с GPU. Вы создаете его для конкретного физического устройства, явно указывая:
  - Какие очереди вы будете использовать (графика, вычисления, копирование).
  - Какие расширения для устройства (например, VK_KHR_swapchain).

Архитектура. Модель: Экземпляр -> Устройство -> **Очередь**

Очереди (Queues)

В OpenGL вы вызываете glDraw… — и команда уходит … куда-то.
В Vulkan вы отправляете работу в конкретную **очередь**.

- **Семейства очередей (Queue Families)**: У GPU есть разные блоки.
  Один умеет только считать (Compute), другой — копировать память (Transfer), третий — рисовать (Graphics).
- Vulkan заставляет вас спросить у GPU: Уважаемый PhysicalDevice, у какого семейства очередей есть флаг VK_QUEUE_GRAPHICS_BIT?
- Вы получаете индекс (например, familyIndex = 0), и при создании VkDevice говорите: Мне нужна одна очередь из семейства 0.

SPIR-V: бинарный шейдерный формат

Проблема OpenGL: GLSL текст → Драйвер парсит и компилирует при старте → stuttering!

Решение Vulkan:
GLSL/HLSL исходник
↓ (glslc, glslang, dxc) - Инструменты для компиляции шейдеров в формат SPIR-V
SPIR-V (бинарный байткод, платформонезависимый)
↓ (vkCreateShaderModule)
Драйвер GPU (быстрое преобразование в машинный код)

Плюсы:
- Компиляция оффлайн (не на пользовательской машине)
- Язык-независимость (можно писать шейдеры на C++ через SPIRV-Cross)
- Обфускация кода шейдеров

GLSLC, glslang и DXC

Инструменты для компиляции шейдеров в универсальный промежуточный формат SPIR-V, который используется в графических API, таких как Vulkan.

GLSLC — это командная строка компилятор, который преобразует шейдеры на языках GLSL и HLSL в SPIR-V. Он использует аргументы, совместимые с Clang, и включает некоторые дополнительные функции, например, поддержку директив #line и #include.

Особенности GLSLC:

- принимает как исходные файлы на GLSL/HLSL, так и файлы ассемблера SPIR-V в качестве входных данных;
- позволяет указывать стадию шейдера (например, вершинный, фрагментный) через опцию -fshader-stage, директиву #pragma shader_stage или расширение файла;
- поддерживает расширение для поддержки синтаксиса включения (GL_GOOGLE_include_directive).

GLSLC, **glslang** и DXC

Glslang — это официальный справочный фронтенд для языков шейдеров OpenGL и OpenGL ES. Он реализует строгую интерпретацию спецификаций этих языков и предназначен для выявления проблем переносимости шейдеров.

Функции glslang:

- компиляция шейдеров GLSL и HLSL в SPIR-V;
- валидация шейдеров для выявления ошибок перед выполнением;
- преобразование шейдеров между различными форматами для совместимости с разными графическими API.

В пакет glslang могут входить инструменты для работы с SPIR-V, например, библиотека SPIRV-Tools.

GLSLC, glslang и **DXC**

DXC (DirectXShaderCompiler) — это компилятор для языка High-Level Shader Language (HLSL) от Microsoft. Он преобразует HLSL-код в DirectX Intermediate Language (DXIL) для платформ DirectX 12, а также может генерировать SPIR-V для использования с Vulkan.

Компоненты DXC:

- dxc.exe — командная строка компилятора для HLSL-шейдеров;
- dxcompiler.dll — компонентный компилятор, ассемблер, дизассемблер и валидатор;
- dxilconv.dll — конвертер из старого формата байт-кода DXBC в DXIL;
- dxv.exe — инструмент для валидации DXIL IR.

Важно: DXC имеет наиболее полную и актуальную поддержку для генерации SPIR-V из HLSL, поэтому часто рекомендуется как предпочтительный способ работы с этим языком в Vulkan.

Классический рендер-цикл для Vulkan

vkAcquireNextImageKHR() (ждет, когда освободится image из swapchain) — Дай мне чистый лист бумаги
Вы просите у системы следующий пустой кадровый буфер (место в памяти, где вы будете рисовать).
Если все буферы заняты (монитор только что показал старую картинку), функция засыпает (ждет)

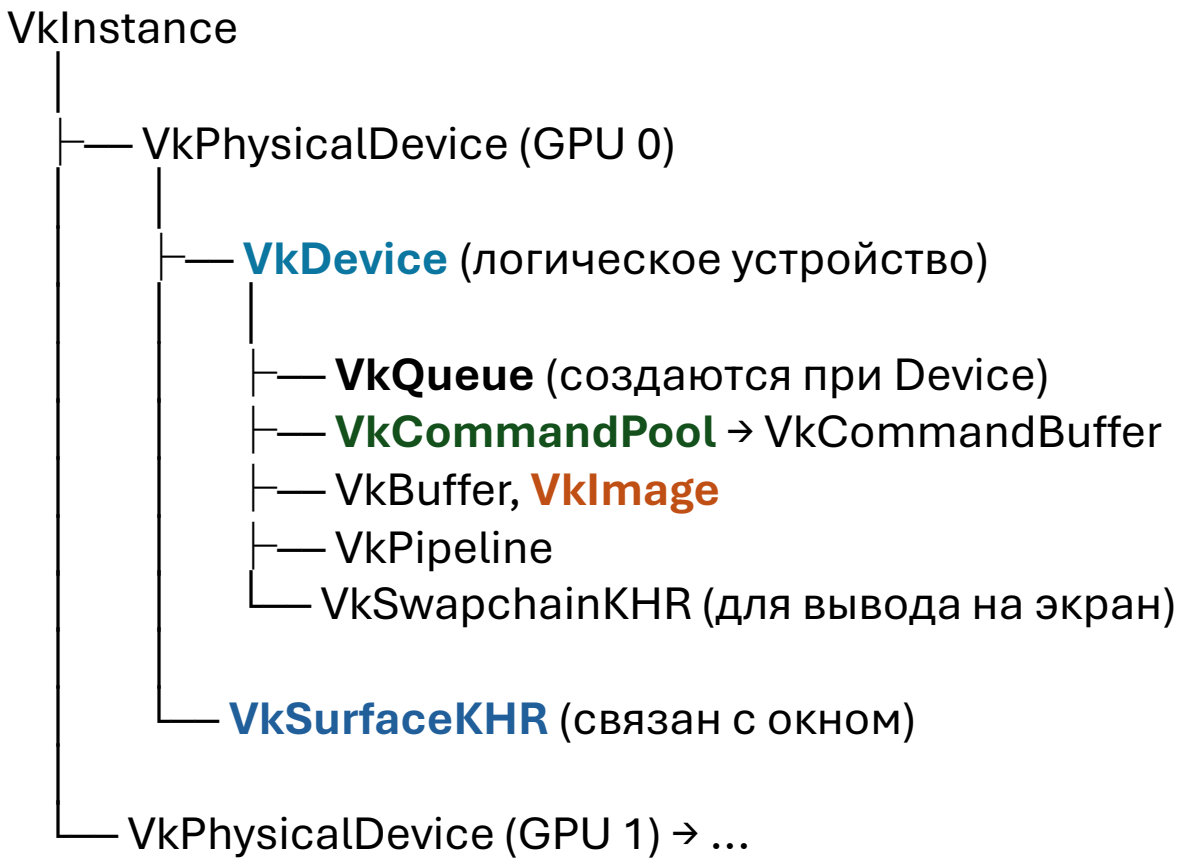
Запись команд — Рисуем на листе
Вы пишете **в CommandBuffer** команды (vkCmdBeginRenderPass, vkCmdDraw…) : очистить экран, нарисовать лес, нарисовать врага, нарисовать прицел. (На самом деле реальной отрисовки на GPU еще нет, только подготовка инструкций)

vkQueueSubmit() (отправляем CommandBuffer в графическую очередь) — Отправить чертеж в печать
Вы отправляете подготовленный список команд в очередь графического процессора (GPU). GPU начинает его обрабатывать

vkQueuePresentKHR() (просим показать результат на экране) — Повесить готовую картину на стену.
Вы говорите: GPU, закончил рисовать? А теперь покажи этот лист на мониторе вместо старого

Переход к шагу 1 — Цикл повторяется снова для следующего кадра (например, 60 раз в секунду)

Связь между иерархией объектов и рендер-циклом



Swapchain создается на основе **VkSurfaceKHR**.
Шаг 1 и 4 работают напрямую с ним

VkImage Это те самые листы бумаги (кадровые буферы), которые скрыты внутри **Swapchain**. Acquire дает доступ к конкретному **VkImage**

VkCommandBuffer создан из **VkCommandPool**. Шаг 2 заполняет его командами

VkQueue получена из **VkDevice**. Шаг 3 сабмитит CB в **VkQueue** (обычно GRAPHICS_QUEUE)

Семафоры (VkSemaphore) Это примитивы синхронизации между GPU и GPU (или GPU и Swapchain).
В схеме они связывают Шаг1 -> Шаг3 -> Шаг4

Детальный разбор. Шаг 1

vkAcquireNextImageKHR — это не просто дай индекс

Почему это важно: Vulkan работает **на опережение**.
Пока GPU рисует один кадр, CPU уже готовит следующий.

Семафор: Эта функция возвращает не просто номер картинки (imageIndex), а также сигналит семафор (semaphore).

Смысл: Я, Swapchain, гарантирую, что этот imageIndex сейчас никто не использует.
Когда буфер будет готов к рисованию — дерни вон тот семафор.

Если буферов нет: Функция может заблокировать поток (если вы указали таймаут) или вернуть ошибку VK_ERROR_OUT_OF_DATE_KHR (если окно изменило размер).

Детальный разбор. Шаг 3

vkQueueSubmit — привязка семафоров
Это самый хитрый момент в схеме.
На самом деле в vkQueueSubmit передается два семафора (через VkSubmitInfo):

1. Wait semaphore (Из шага 1): GPU, НЕ НАЧИНАЙ рисовать, пока Swapchain не скажет, что картинка imageIndex свободна.

2. Signal semaphore: GPU, когда ЗАКОНЧИШЬ рисовать, дерни этот семафор (чтобы Present знал, что картинка готова).

Детальный разбор. Шаг 4

vkQueuePresentKHR — финальное ожидание
Здесь в параметрах тоже указывается семафор (тот, который GPU дернул после окончания рисования).

Смысл: "Покажи картинку, ТОЛЬКО КОГДА GPU закончил ее рисовать".

Схема рендер-цикла (с синхронизацией) для одного кадра

1. vkAcquireNextImageKHR
(ждет, записывает в семафор A)

2. Запись команд в CommandBuffer
(vkCmdBeginRenderPass, vkCmdDraw...)

3. vkQueueSubmit()
[ОЖИДАЕТ: семафор A]
[СИГНАЛИТ: семафор B]

GPU начинает рисовать кадр
(ждет A → рисует)

4. vkQueuePresentKHR()
[ОЖИДАЕТ: семафор B]
(показать кадр на экране)

Swapchain показывает кадр на мониторе

Переход к шагу 1 (следующий кадр)

Код vs Графика: объем инициализации

Статистика:
- OpenGL: ~150-200 строк до первого треугольника
- Vulkan: ~1000-1500 строк до первого треугольника

Но! Это **одноразовые затраты**. **В рантайме** Vulkan **быстрее**.

Совет: Используйте **Vulkan-HPP** (C++ binding с RAII).
Vulkan-HPP — это заголовочные файлы (header-only) C++ привязки для API Vulkan.
Его цель — улучшить работу разработчиков с Vulkan, добавив возможности C++, при этом не внося дополнительных затрат на выполнение кода во время работы программы.

```
// Было (C API):
vkCreateInstance(&createInfo, nullptr, &instance);

// Стало (C++):
auto instance = vk::createInstance(createInfo);
```
Сокращает код на ~30%

Ключевые преимущества для будущих разработчиков движков

1. **Предсказуемая производительность**:
   Vulkan не тратит время на проверки, которые делал за вас OpenGL в рантайме.
   Вы все проверяете на этапе инициализации или через валидационные слои.
   Драйвер становится тупым и быстрым — он просто выполняет то, что вы сказали.

2. **Многопоточность** (Первое правило Vulkan):
   В OpenGL управление состоянием — монополия основного потока.
   Vulkan позволяет создавать командные буферы из разных потоков CPU параллельно.
   16 потоков одновременно пишут команды — на старт, внимание, марш!

3. **Явное управление ресурсами**:
   Вы сами выделяете память под буферы и текстуры, сами управляете синхронизацией.
   Ошибка? Да, легко.
   Но и оптимизация теперь не требует магии — вы просто делаете то, что нужно железу.

Различия в обработке ошибок

Вопрос
Почему в Vulkan нет встроенной функции Получить последнюю ошибку,
а вместо этого — система слоев?

Краткий ответ
В OpenGL ошибка — это исключительная ситуация, которую API проверяет на всякий случай (дорого).
В Vulkan ошибка — это всегда проблема разработчика, которую не нужно проверять в рантайме, а нужно выявить на этапе отладки.
Слои позволяют это сделать, а в релизе они отключаются, убирая проверки.

Обработка ошибок в OpenGL

В **OpenGL** графический **драйвер** — это огромный **конечный автомат**.
Любой вызов вроде glDrawArrays() или glBindTexture() может сгенерировать ошибку,
но проверять её нужно явно через glGetError().

Недостатки подхода OpenGL

- Драйвер всегда отслеживает состояние и проверяет корректность (например, привязан ли VAO?, скомпилирован ли шейдер?).
- Это создает ненулевые накладные расходы даже в релизной сборке.
- Разработчики часто игнорируют glGetError() или вызывают его только в отладке, что приводит к трудноуловимым багам.
- Непонятно, к какому именно вызову относится ошибка (нет контекста).

Vulkan: Никакой магии, только явный контроль

Vulkan спроектирован с двумя **главными целями**:

- Максимальная производительность в релизных сборках.
- Минимальная нагрузка на драйвер — драйвер должен быть тупым и быстрым, просто выполнять команды.

Следствие: Vulkan почти **не делает проверок в рантайме**.

- Если вы передали некорректный параметр (например, VK_NULL_HANDLE вместо реального буфера) — будет не ошибка, а неопределенное поведение (краш, артефакты, зависание системы).
- Если вы забыли привязать дескриптор перед рисованием — GPU нарисует мусор или ничего.

Почему так? Потому что **каждая проверка** — это:

- Ветвление в коде драйвера.
- Доступ к глобальным структурам (блокировки, кэш-промахи).
- Непредсказуемость времени выполнения.

В высоконагруженных системах (игры, симуляторы) даже 1-2% накладных расходов на проверки — это потерянные кадры.

Решение Vulkan: валидационные слои (Validation Layers)

Vulkan предлагает опциональную, подключаемую систему валидации.

Как это работает:

- Слои — это библиотеки, которые перехватывают все вызовы Vulkan ДО того, как они дойдут до драйвера.
  Слои — это **дополнительные программные компоненты**, которые **расширяют** функциональность API, **перехватывая** вызовы функций и модифицируя их поведение на пути от приложения к аппаратному обеспечению.
  Они упаковываются в виде разделяемых библиотек, которые динамически загружаются загрузчиком Vulkan

- Они проверяют корректность параметров, состояний, синхронизации.

- При ошибке выдают подробное сообщение: какой вызов, какой параметр, нарушение какой спецификации (с ссылкой на VUID — Vulkan Unique ID).

Пример

```
// VkResult — только фатальные ошибки (нет памяти, устройство потеряно)
VkResult result = vkCreateInstance(...);
if (result != VK_SUCCESS) {
   // Либо OOM, либо драйвер не поддерживает Vulkan
}
```

Но! Чтобы получить сообщение о том, что вы забыли привязать пайплайн, нужно:

- Включить слой VK_LAYER_KHRONOS_validation при создании VkInstance.
- Настроить callback через VK_EXT_debug_utils.

Тогда при ошибке вы увидите что-то вроде:

```
[ERROR] VUID-vkCmdDraw-None-02721: vkCmdDraw():
VkPipeline must be bound to the VK_PIPELINE_BIND_POINT_GRAPHICS bind point before this command.
```

Почему слои лучше?

Аспект	OpenGL (glGetError)	Vulkan (валидационные слои)
Время проверки	Всегда (даже в релизе)	Только при явном включении
Детали ошибки	Код (например, GL_INVALID_VALUE)	Человекочитаемое сообщение + VUID
Контекст ошибки	Неизвестно, какой вызов вызвал	Точное место в коде (если есть callback)
Производительность	Есть оверхед всегда	Нулевой оверхед в релизе
Расширяемость	Низкая (стандартные коды)	Высокая (можно писать свои слои)
Многопоточность	Проблемная (глобальное состояние)	Поддерживается (слои потокобезопасны)

Жизненный цикл: Debug vs Release

На этапе разработки:

- Включаем VK_LAYER_KHRONOS_validation (или несколько слоев).
- Настраиваем VK_EXT_debug_utils для вывода сообщений в консоль/файл.
- Получаем подробные отчёты о всех нарушениях спецификации.

В релизной сборке:

- Убираем все слои.
- Все вызовы идут напрямую в драйвер без лишних проверок.
- vkCreateInstance вызывается с enabledLayerCount = 0.

Результат:

Максимальная производительность без скрытых проверок.

Практический паттерн: Выбор устройства и очереди. 1

// 1. **Перечисляем все GPU**
uint32_t deviceCount = 0;

vkEnumeratePhysicalDevices(instance, &deviceCount, nullptr);
// запрос количества доступных GPU с поддержкой Vulkan
// Передаём nullptr в третьем параметре — значит, мы **не хотим получать сами устройства**,
// только их количество

vector<VkPhysicalDevice> devices(deviceCount);
// Выделяем вектор нужного размера (ровно deviceCount элементов)

vkEnumeratePhysicalDevices(instance, &deviceCount, devices.data()); // заполнение массива
// Передаём devices.data() — указатель на внутренний буфер вектора.
// Функция **заполняет этот буфер** дескрипторами устройств.
// Снова передаём &deviceCount — функция может **скорректировать** это значение, если // количество устройств изменилось между вызовами (маловероятно, но технически возможно)

Ключевые концепции Vulkan, демонстрируемые этим кодом

1. **Двухфазная инициализация** (Two-phase initialization)

Это паттерн, который встречается в Vulkan повсеместно:

- Вызвать функцию с nullptr, чтобы получить размер.
- Выделить буфер.
- Вызвать функцию снова, чтобы заполнить буфер.

2. **Отсутствие исключений** и исключительно VkResult

- Vulkan (C API) **не выбрасывает исключения**.
- Все функции возвращают VkResult — перечисление кодов успеха/ошибки.
- **Важно:** Даже фатальные ошибки (например, instance невалиден) не приведут к крашу немедленно — вы должны проверять результат.

Практический паттерн: Выбор устройства и очереди. 2

```
// 2. Ищем подходящий device
for (const auto& device : devices) {
    // Проверяем наличие графической очереди
    // Проверяем поддержку swapchain
    if (graphicsQueueFound && swapchainSupported) {
        physicalDevice = device;
        break;
    }
}
```

Проверяем, есть ли очередь для графики

```
VkPhysicalDeviceProperties properties;
vkGetPhysicalDeviceProperties(device, &properties);

  // Проверяем, есть ли очередь для графики
  uint32_t queueFamilyCount = 0;
  vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, nullptr);
  vector<VkQueueFamilyProperties> queueFamilies(queueFamilyCount);
  vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, queueFamilies.data());

  int graphicsQueueIndex = -1;
  for (uint32_t i = 0; i < queueFamilyCount; i++) {
     if (queueFamilies[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) {
        graphicsQueueIndex = i;
        break;
     }
  }
```

Проверяем поддержку swapchain

```
/ / Проверяем поддержку swapchain
    uint32_t extensionCount = 0;
    vkEnumerateDeviceExtensionProperties(device, nullptr, &extensionCount, nullptr);
    vector<VkExtensionProperties> extensions(extensionCount);
    vkEnumerateDeviceExtensionProperties(device, nullptr, &extensionCount, extensions.data());

    bool swapchainSupported = false;
    for (const auto& ext : extensions) {
        if (strcmp(ext.extensionName, VK_KHR_SWAPCHAIN_EXTENSION_NAME) == 0) {
            swapchainSupported = true;
            break;
        }
    }
```

Практический паттерн: Выбор правильного GPU

```
VkPhysicalDeviceProperties props;
vkGetPhysicalDeviceProperties(device, &props);
```

Перевод: Расскажи мне всё о конкретном GPU: как его зовут, сколько у него памяти, что он умеет

Что лежит в props после вызова

Поле	Тип	Пример	Зачем нужно
deviceName	char[256]	"NVIDIA GeForce RTX 3060"	Показать пользователю или логировать
deviceType	VkPhysicalDeviceType	VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU	Нам нужен дискретный GPU, а не встроенный
apiVersion	uint32_t	VK_API_VERSION_1_3	Версия Vulkan, которую поддерживает драйвер
driverVersion	uint32_t	531.41	Версия драйвера (кодирована)
vendorID	uint32_t	0x10DE (NVIDIA)	ID производителя
deviceID	uint32_t_t	0x2504 (RTX 3060)	ID конкретной модели
limits	VkPhysicalDeviceLimits	огромная структура	maxImageDimension2D, maxPushConstantsSize и т.д.

Как это используется для выбора GPU

```
// Пример: выбираем дискретный GPU (игровой), игнорируем встроенный
for (const auto& device : devices) {
   vkGetPhysicalDeviceProperties(device, &props);

   if (props.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU) {
      physicalDevice = device;
      break;  // нашли хорошую видеокарту — берём её
   }
}
// А если не нашли дискретный? Тогда берём любой, который есть
```

Типы GPU (VkPhysicalDeviceType)

Значение	Что означает	Когда выбирать
VK_PHYSICAL_DEVICE_TYPE_OTHER	Неизвестный/нестандартный тип	Запасной вариант
VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU	Встроенная в CPU (Intel Iris, AMD Radeon Graphics, Apple M1/M2)	Ноутбуки, экономия энергии
VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU	Отдельная видеокарта (NVIDIA RTX, AMD Radeon RX)	✅ Игровой компьютер, рабочая станция
VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU	Виртуальный GPU (в облаке, VM)	Облачные серверы, удалённый рендеринг
VK_PHYSICAL_DEVICE_TYPE_CPU	CPU рендерит программно	Запасной вариант, отладка без GPU

Пример более профессионального кода выбора

```
for (const auto& device : devices) {
    VkPhysicalDeviceProperties props;
    vkGetPhysicalDeviceProperties(device, &props);

    // Приоритет: дискретная > встроенная > остальные
    if (props.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU) {
        selectedDevice = device;
        break;  // лучший вариант, сразу берём
    }
    else if (props.deviceType == VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU) {
        selectedDevice = device;  // запоминаем, но продолжаем искать дискретную
    }
    // если дошли сюда и selectedDevice всё ещё VK_NULL_HANDLE — берём что есть
}
```

Пример реального вывода

Device name: NVIDIA GeForce RTX 3060
Device type: DISCRETE_GPU
API version: 1.3.242
Vendor ID: 0x10DE (NVIDIA)
Memory: 12 GB
Max texture size: 32768 x 32768
Max shader model: 6.6

Что ещё можно проверить через props

```
// Проверяем, достаточно ли текстурного размера для игры
if (props.limits.maxImageDimension2D < 16384) {
   // ГПУ слишком стар, отключаем 4K-текстуры
}

// Проверяем, поддерживает ли GPU геометрические шейдеры
if (props.limits.maxGeometryShaderInvocations == 0) {
   // Не поддерживает — используем альтернативный метод
}
```

Практический паттерн: Шаг 4

```
// 4. Ищем семейство очередей с поддержкой графики
  uint32_t queueFamilyCount = 0;
  vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, nullptr);
  vector<VkQueueFamilyProperties> families(queueFamilyCount);
  vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, families.data());

  uint32_t graphicsFamily = UINT32_MAX;
  for (uint32_t i = 0; i < queueFamilyCount; i++) {
    if (families[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) {
      graphicsFamily = i;
      break;
    }
  }

  if (graphicsFamily != UINT32_MAX) {
    // Успех! Запоминаем это устройство и индекс семейства
    physicalDevice = device;
    queueFamilyIndex = graphicsFamily;
    break;
  }
}
```

Шаг A: Узнаём, сколько семейств очередей у GPU

```
uint32_t queueFamilyCount = 0;
vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, nullptr);
```

Перевод: Сколько видов очередей поддерживает этот GPU? (не сколько очередей всего, а сколько разных типов)

Пример ответа: GPU говорит: У меня 3 семейства очередей

Семейство	Что умеет
Семейство 0	Графика + Compute + Transfer
Семейство 1	Transfer (только копирование данных)
Семейство 2	Compute (только математические расчёты)

Шаг B: Получаем свойства каждого семейства

```
vector<VkQueueFamilyProperties> families(queueFamilyCount);
vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, families.data());
```

Перевод: Теперь расскажи подробно о каждом семействе: что оно умеет, сколько в нём очередей

Шаг C: Ищем семейство с графикой

```
uint32_t graphicsFamily = UINT32_MAX;
for (uint32_t i = 0; i < queueFamilyCount; i++) {
    if (families[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) {
        graphicsFamily = i;
        break;
    }
}
```

Перевод: Перебираем все семейства. У того, который умеет в графику (флаг VK_QUEUE_GRAPHICS_BIT), запоминаем индекс и выходим из цикла

UINT32_MAX — это маркер ещё не нашли (т.к. индексы не могут быть такими большими).

Шаг D: Если нашли — сохраняем и выходим

```
if (graphicsFamily != UINT32_MAX) {
   physicalDevice = device;
   queueFamilyIndex = graphicsFamily;
   break;  // выходим из внешнего цикла перебора GPU
}
```

Перевод: Нашли подходящее устройство с графической очередью. Запоминаем его и прекращаем поиск

Визуализация: как это выглядит в памяти

Наш код ищет: у какого семейства queueFlags содержит VK_QUEUE_GRAPHICS_BIT?
Находит: семейство 0. Сохраняет graphicsFamily = 0.

Что такое queueFlags (возможные значения)

Флаг	Что умеют очереди в этом семействе
VK_QUEUE_GRAPHICS_BIT	Рисовать (рендеринг, треугольники, меши)
VK_QUEUE_COMPUTE_BIT	Выполнять compute-шейдеры (математика, физика, ИИ)
VK_QUEUE_TRANSFER_BIT	Копировать данные (буферы, текстуры)
VK_QUEUE_SPARSE_BINDING_BIT	Управлять разреженной памятью

Важно: Часто эти флаги комбинируются.
Графическое семейство **обычно** умеет **ВСЁ** (и compute, и transfer).

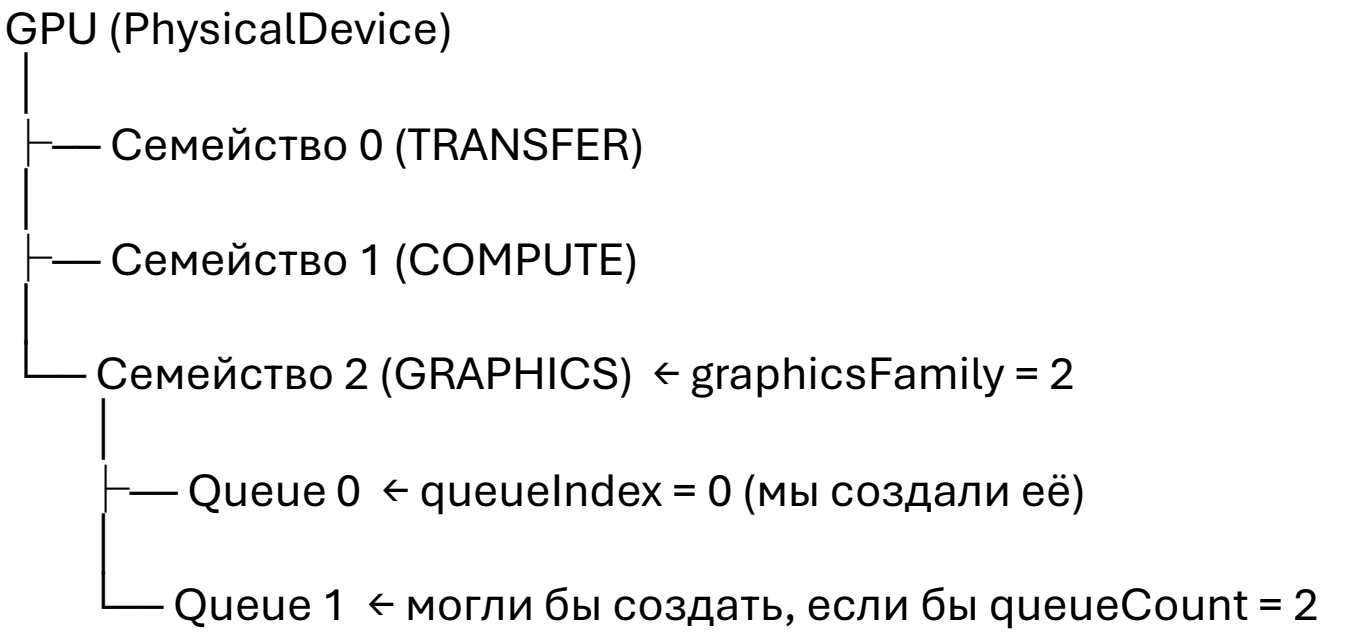
Практический пример (что дальше с этим индексом)

```
// После того как нашли graphicsFamily, создаём Device и получаем Queue
VkDeviceQueueCreateInfo queueCreateInfo{}; // Создаём структуру, описывающую какую очередь мы хотим
queueCreateInfo.queueFamilyIndex = graphicsFamily;  // ← тот самый индекс
queueCreateInfo.queueCount = 1;
float priority = 1.0f; // приоритет (1.0 = высокий)
queueCreateInfo.pQueuePriorities = &priority;

VkDeviceCreateInfo deviceInfo{}; // Создаём логическое устройство (VkDevice)
deviceInfo.queueCreateInfoCount = 1;
deviceInfo.pQueueCreateInfos = &queueCreateInfo;

vkCreateDevice(physicalDevice, &deviceInfo, nullptr, &logicalDevice);

VkQueue queue;  // сюда запишется хендл очереди
vkGetDeviceQueue(// А потом получаем саму очередь
   logicalDevice,     // из какого устройства
   graphicsFamily,    // из какого семейства (индекс 2)
   0,              // номер очереди в семействе (первая, т.к. queueCount=1)
   &queue          // сюда запишется результат
);
```

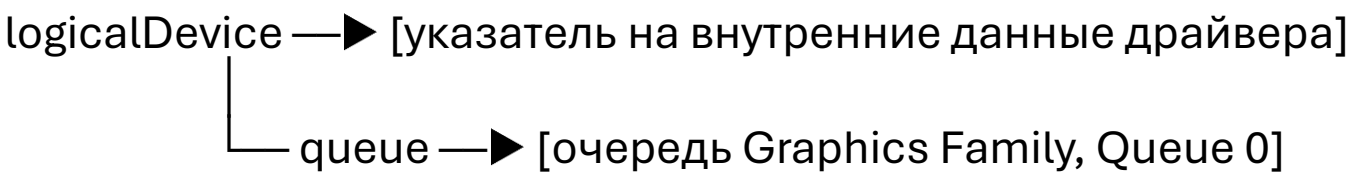
ИСПОЛЬЗУЕМ очередь для отправки команд

```
// ... заполнили command buffer ...
VkSubmitInfo submitInfo{};
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &commandBuffer;

// Отправляем в очередь на исполнение!
vkQueueSubmit(queue, 1, &submitInfo, VK_NULL_HANDLE);
```

Визуализация: что происходит в памяти



После создания Device и вызова vkGetDeviceQueue:



Почему два вызова: vkCreateDevice и vkGetDeviceQueue?

Вызов	Что делает	Почему нельзя объединить
vkCreateDevice	Создаёт **все** очереди, описанные в queueCreateInfo	Драйвер резервирует ресурсы
vkGetDeviceQueue	Возвращает **уже существующий** хендл очереди	Очередь уже создана внутри драйвера, мы просто доступ к ней

Что если нужно несколько очередей из одного семейства?

```
// Хотим 2 очереди из одного семейства
queueCreateInfo.queueCount = 2;
float priorities[] = {1.0f, 0.5f};  // разный приоритет
queueCreateInfo.pQueuePriorities = priorities;

vkCreateDevice(...);

// Получаем обе очереди
VkQueue queue0, queue1;
vkGetDeviceQueue(logicalDevice, graphicsFamily, 0, &queue0);  // первая очередь
vkGetDeviceQueue(logicalDevice, graphicsFamily, 1, &queue1);  // вторая очередь

// Теперь можно отправлять команды параллельно из разных потоков!
```

Итог одной схемой