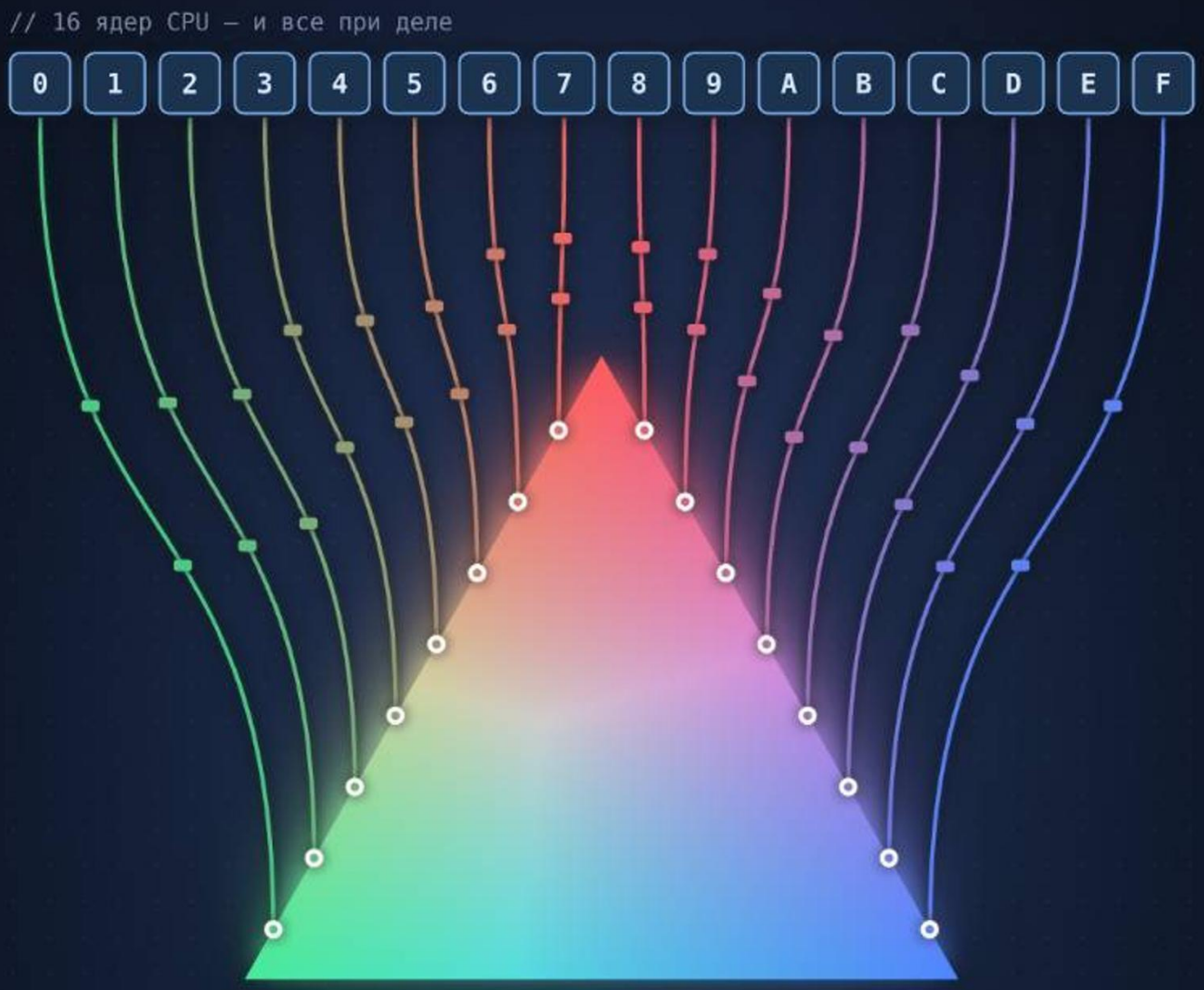
АЛЕКСЕЙ ПРИМАК

VULKAN
и многопоточность для смертных

Как заставить работать видеокарту
и все 16 ядер, а не себя




```
$ ./pepelac    # 16 потоков · 0 гонок · 0 утечек
```

Алексей Примак

«Vulkan и многопоточность для смертных» или: как заставить работать видеокарту и все 16 ядер, а не себя

«Автор»
2026

Примак А.

«Vulkan и многопоточность для смертных» или: как заставить работать видеокарту и все 16 ядер, а не себя / А. Примак — «Автор», 2026

Практическое руководство по Vulkan 1.3/1.4 и многопоточности на современном C++: от пустого окна до многопоточного рендерера. 31 глава, построчный разбор кода, 63 задачи с подробными решениями, шпаргалки по синхронизации и потокобезопасности. С юмором, но без потери сути.

© Примак А., 2026
© Автор, 2026

Содержание

Предисловие 7
Для кого эта книга 8
Технологический стек 9
Сквозной проект «Пепелац» 10
Условные обозначения 11
Как читать 12
Часть 0. Перед погружением 13
Глава 0. Зачем вам Vulkan, или Что от вас скрывал OpenGL 14
0.1. Видеокарта: тысяча художников и один занятой директор 14
0.2. Что OpenGL делал у тебя за спиной 14
0.3. Философия Vulkan: «я сам» 15
0.4. Карта объектов Vulkan 16
0.5. Разбор кода №1: порядок жизни и смерти 17
0.6. Путь одного кадра 22
0.7. Разбор кода №2: drawFrame построчно 24
0.8. А где тут многопоточность? 26
Шпаргалка главы 0 27
Задачи для разминки мозгов 27
Глава 1. Окружение, или Четыре часа CMake ради пустого окна 29
1.1. Vulkan SDK 29
1.2. Структура проекта 29
1.3. CMakeLists.txt построчно 30
1.4. Окно GLFW в RAII-обёртке 33
1.5. Разминка по C++: как устроен vk::raii 36
Шпаргалка главы 1 38
Задачи для разминки мозгов 39
Часть I. Первый треугольник 40
Глава 2. Instance и слои валидации: заводим лучшего друга 41
2.1. Расширения и слои 41
2.2. Код: создаём Instance 41
2.3. Разбор по шагам 46
2.4. То же самое на голом C 47
2.5. Как читать сообщения валидации 48
Шпаргалка главы 2 48
Задачи для разминки мозгов 48
Глава 3. Физическое и логическое устройство: собеседование для видеокарты 49
3.1. Физическое и логическое 49
3.2. Surface заранее 49
3.3. Код: выбор видеокарты 50
3.4. Разбор 56
Шпаргалка главы 3 57
Задачи для разминки мозгов 57
Глава 4. Surface и swapchain: карусель кадров 59
4.1. Как устроен swapchain 59

4.2. Режимы показа 59
4.3. Код: создание swapchain 60
4.4. Разбор 63
Шпаргалка главы 4 64
Задачи для разминки мозгов 65
Глава 5. Шейдеры: программы, которые живут в видеокарте 66
5.1. Графический конвейер 66
5.2. Первый шейдер на Slang 67
5.3. Компиляция в SPIR-V из CMake 68
5.4. Загрузка SPIR-V и shader module 70
Шпаргалка главы 5 71
Задачи для разминки мозгов 72
Глава 6. Graphics pipeline: запекаем всё заранее 73
6.1. Из чего состоит пайплайн 73
6.2. Код 73
6.3. Разбор 76
Шпаргалка главы 6 78
Задачи для разминки мозгов 78
Глава 7. Командные буферы: список дел для GPU 79
7.1. Пулы, буферы и их состояния 79
7.2. Layout картинки за пять минут 79
7.3. Код 80
7.4. Разбор 84
Шпаргалка главы 7 85
Задачи для разминки мозгов 85
Глава 8. Синхронизация I: первый живой кадр 86
8.1. Два инструмента синхронизации 86
8.2. Код: объекты синхронизации 86
8.3. Собираем всё вместе: pepelac.hpp 87
8.4. Конструктор и главный цикл 90
8.5. Запуск 92
Шпаргалка главы 8 93
Задачи для разминки мозгов 93
Глава 9. Пересоздание swapchain: окно, которое можно трогать 94
9.1. Как Vulkan сообщает о проблеме 94
9.2. Код: пересоздание 94
9.3. Разбор 97
Шпаргалка главы 9 99
Задачи для разминки мозгов 99
Часть II. Ресурсы и данные 100
Глава 10. Память GPU: куча, но не та, о которой вы думаете 101
10.1. Кучи и типы памяти 101
10.2. Как выделить буфер вручную (и почему не надо) 101
10.3. VMA в RAII-обёртке 102
10.4. Staging: как данные попадают в видеопамять 107
Шпаргалка главы 10 110
Задачи для разминки мозгов 110
Глава 11. Вершинные и индексные буферы 111
11.1. Как пайплайн читает вершины 111

11.2. Код: вершина и её описание 111
Конец ознакомительного фрагмента. 114

Алексей Примак «Vulkan и многопоточность для смертных» или: как заставить работать видеокарту и все 16 ядер, а не себя

Предисловие

Эта книга за 31 главу проводит тебя от пустого окна до многопоточного рендерера на Vulkan и современном C++. Опыт в графике и многопоточности не нужен. Нужны C++ на базовом уровне и терпение.

Итак, ты решил выучить Vulkan. Смело. Обычно к этому решению приходят в три этапа: «OpenGL устарел», «DirectX 12 есть только на Windows», «да насколько сложно нарисовать треугольник?». Ответ на последний вопрос: около тысячи строк кода. Добро пожаловать.

Хорошая новость в том, что Vulkan не сложный, а многословный. Каждая строка делает одну понятную вещь, просто строк много. OpenGL угадывал твои намерения и иногда ошибался. Vulkan ничего не угадывает: делает ровно то, что ты сказал. Глупости тоже.

Теперь о том, при чём тут многопоточность. Vulkan с самого начала проектировали под многоядерные процессоры. В OpenGL весь рендеринг жил в одном потоке, а остальные 15 ядер смотрели на это, как коллеги на джуна, который в пятницу вечером в одиночку чинит прод. В Vulkan команды для GPU можно записывать из всех потоков сразу, если знать правила. Про эти правила книга и написана.

Для кого эта книга

• Ты знаешь C++ на уровне классов, ссылок, конструкторов и деструкторов, std::vector, std::unique_ptr и лямбд.

• Ты помнишь из школы, что такое вектор и матрица. Остальную математику объясним.

• Ты никогда не писал на Vulkan, OpenGL или многопоточный код. Именно под тебя книга и рассчитана.

Железо: видеокарта с драйвером Vulkan 1.3 или новее. Это почти любая дискретная или встроенная карта примерно с 2016 года при свежих драйверах. Проверка через vulkaninfo разобрана в главе 1.

Технологический стек

Что	Чем	Почему
API	Vulkan 1.3+, фичи 1.4 с пометкой	dynamic rendering, synchronization2 и timeline semaphores уже в ядре
Язык	C++20, местами C++23	`std::jthread`, `std::latch`, designated initializers, `std::expected`
Обёртка	Vulkan-Hpp (`vk::raii`)	объекты уничтожаются сами в деструкторах
Окно	GLFW	просто, стабильно, кроссплатформенно
Шейдеры	Slang → SPIR-V	современный язык, компилятор `slangc` уже в Vulkan SDK
Память	Vulkan Memory Allocator (VMA)	писать свой аллокатор никто не будет, мы не мазохисты
Математика	GLM	стандарт де-факто
Сборка	CMake + vcpkg	боль, но привычная
Отладка	Validation layers, RenderDoc, ThreadSanitizer	три лучших друга новичка
Профилирование	Tracy	CPU и GPU на одной временной шкале

Стек совпадает с официальным Khronos Vulkan Tutorial (https://docs.vulkan.org/tutorial/latest/00_Introduction.html): Vulkan 1.4 как база, dynamic rendering вместо render passes, timeline semaphores, Slang, C++20 и Vulkan-Hpp с RAII. Значит, по ходу книги можно сверяться с первоисточником. Старые туториалы 2016–2020 годов учат через VkRenderPass и VkFramebuffer. Это как учиться водить на «Запорожце»: опыт ценный, но зачем.

Сквозной проект «Пепелац»

Вся книга строится вокруг одного проекта, который растёт от главы к главе: из пустого окна получается многопоточный рендерер. Назвали его «Пепелац», потому что сначала он тоже не летает.

Условные обозначения

• **ЗАПОМНИ** — правила, которые надо знать даже разбуженным в три часа ночи.

• **ШПАРГАЛКА** — сводка в конце главы для повторения.

• **ЦИТАТА** — реальные цитаты с указанием автора или мои собственные с пометкой «— автор книги».

• **ГРАБЛИ** — типичные ошибки новичков и их последствия.

• **ПОТОКИ** — можно ли трогать объекты главы из нескольких потоков. Такая заметка есть в каждой главе со второй, так что многопоточность идёт через всю книгу.

• **ЗАДАЧКИ** — задачи для разминки мозгов. Решения в приложении А.

Как читать

Каждая глава устроена одинаково: вступление, теория, код с построчным разбором, шпаргалка, задачи. Код в разборах пронумерован шагами (// —— Шаг 1 ——), и текст разбора идёт по тем же шагам. Читай по порядку: части опираются друг на друга.

Книга разбита на пять частей. Часть 0 даёт общую картину и окружение. Часть I доводит до первого треугольника. Часть II добавляет ресурсы: буферы, текстуры, 3D, барьеры. Часть III учит многопоточности в C++ без всякой графики. Часть IV соединяет Vulkan и потоки, ради этого всё и затевалось. Часть V про отладку, профилирование и финальный проект.

Почему потоки не с первой главы? Если учить Vulkan и потоки одновременно, баги прилетают с двух сторон, и непонятно, откуда именно. Сначала учимся ходить, потом жонглировать, и только потом жонглировать на ходу.

ЗАПОМНИ ДО СТАРТА. *В debug-сборке validation layers включены всегда, без исключений: без них Vulkan не выдаёт ошибок, а молча делает UB. Главу 15 не пропускать, даже если очень хочется. Особенно если очень хочется.*

ЦИТАТА. *«Vulkan не сложный. Он просто честный: всё, что раньше скрывал драйвер, теперь скрываешь ты, причём от начальства». — автор книги*

Часть 0. Перед погружением

Две главы без треугольников: сначала модель в голове, потом окружение на диске.

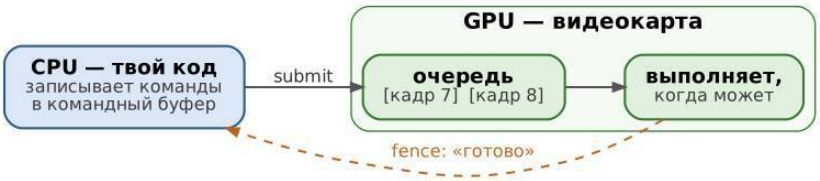
Глава 0. Зачем вам Vulkan, или Что от вас скрывал OpenGL

В этой главе мы не нарисуем ни пикселя. Зато ты узнаешь, из чего состоит Vulkan, как живёт один кадр и почему порядок полей в классе может уронить программу. Код Vulkan без модели в голове — это карго-культ на тысячу строк. Всё работает, пока не перестанет, и тогда никто не знает почему.

0.1. Видеокарта: тысяча художников и один занятой директор

CPU — это несколько очень умных профессоров. Каждый решает любую задачу, ветвится, предсказывает переходы. GPU — это тысячи простых рабочих, которые делают одну и ту же операцию над разными данными. Покрасить два миллиона пикселей — работа для тысячи рабочих. Разобрать JSON с конфигом — работа для профессора.

Главное: **CPU не рисует.** CPU пишет список инструкций (командный буфер), отправляет его в очередь и сразу занимается другими делами. GPU выполнит его позже — через микросекунды или через целый кадр. Это как письмо по почте: бросил в ящик и пошёл дальше.



Из асинхронности следует вся боль синхронизации. Пока GPU читает буфер, его нельзя перезаписывать. Пока GPU рисует в картинку, её нельзя показывать. Пока GPU использует объект, его нельзя удалять. В OpenGL за всем этим следил драйвер. В Vulkan — ты.

ЗАПОМНИ. *CPU и GPU работают параллельно и асинхронно. Вызов «нарисуй» на CPU ничего не рисует, а добавляет команду в список. Всё, что ты отдал GPU, нельзя трогать, пока GPU не скажет «готово».*

0.2. Что OpenGL делал у тебя за спиной

Вот типичный код отрисовки на OpenGL. Выглядит невинно.

фрагмент.cpp — C++

```
glUseProgram(program);              // «используй эти шейдеры»
glBindVertexArray(vao);             // «вот твои вершины»
glBindTexture(GL_TEXTURE_2D, tex);  // «вот твоя текстура»
glDrawArrays(GL_TRIANGLES, 0, 3);   // «рисуй!» ← и тут драйвер
  начинает думать
```

Внутри glDrawArrays драйвер делает кучу работы, о которой ты не просил:

1. **Проверяет ошибки.** Слинкована ли программа? Полная ли текстура? Совпадают ли форматы? На каждом вызове, в релизе тоже.
2. **Собирает состояние.** OpenGL — гигантский автомат состояний. Драйвер смотрит на всё, что ты включил за всю жизнь программы, и собирает из этого конфигурацию для железа.
3. **Может перекомпилировать шейдер.** На части железа некоторые состояния вшиваются прямо в машинный код шейдера. Поменял смешивание — драйвер тихо перекомпилирует шейдер посреди кадра. Отсюда загадочные подтормаживания.
4. **Раскладывает память.** Где лежит текстура, в видеопамяти или в оперативке? Не пора ли её перенести? Решает драйвер.
5. **Ставит синхронизацию.** Ты записал данные в буфер через glBufferSubData, а GPU ещё читает старую версию? Драйвер либо подождёт, либо тайно скопирует буфер.
6. **Решает, когда отправить работу.** Команды копятся во внутреннем буфере драйвера и уходят, когда он сочтёт нужным.

И всё это в одном потоке, к которому привязан контекст OpenGL. Результат: непредсказуемые фризы, разное поведение у разных производителей и потолок производительности одного ядра.

0.3. Философия Vulkan: «я сам»

Vulkan вышел в 2016 году. Его сделала Khronos Group на основе AMD Mantle. Идея простая: драйвер должен быть тонким и предсказуемым, а всё дорогое должно делаться явно и заранее.

Работа	Кто делает в OpenGL	Кто делает в Vulkan
Проверка ошибок	драйвер, всегда, даже в релизе	validation layers, только в debug
Компиляция шейдеров и состояний	драйвер, когда захочет	ты, при создании пайплайна
Распределение памяти	драйвер	ты (или VMA)
Синхронизация CPU и GPU	драйвер, неявно	ты: fences, semaphores, barriers
Запись команд	драйвер, в потоке контекста	ты, в любых потоках
Когда отправить работу на GPU	драйвер	ты (`submit`)

Что ты получаешь: предсказуемое время кадра, низкие накладные расходы на draw call и настоящую многопоточность. Чем платишь: объёмом кода и ответственностью. Ошибка в Vulkan — это не код ошибки, а UB: чёрный экран, мусор на картинке или «у меня работает, а на AMD падает».

Честность прежде всего: Vulkan нужен не всем. Для простой 2D-игры бери SDL, raylib или готовый движок. Vulkan нужен, когда важны производительность и контроль, когда ты

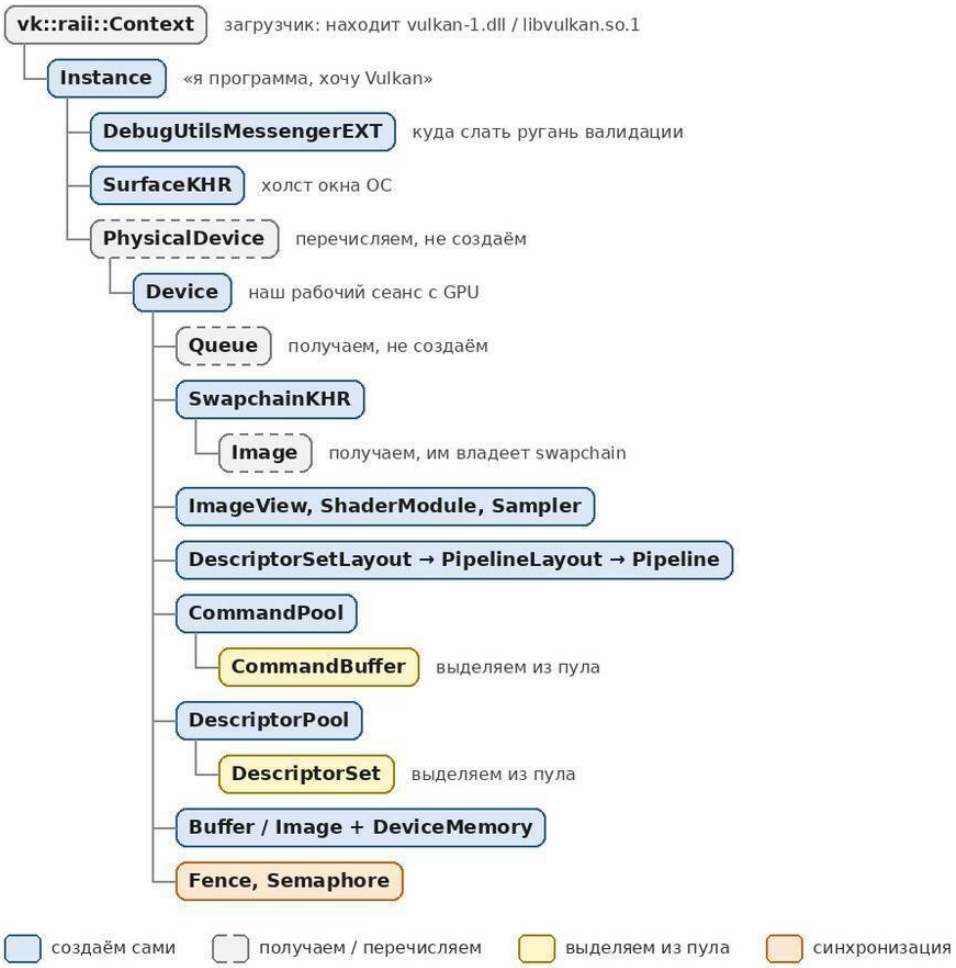
пишешь движок или хочешь понять, как на самом деле работает видеокарта. Последнее окупается, даже если потом ты будешь работать в Unreal.

ЦИТАТА. *«Явное лучше, чем неявное». — Тим Питерс, «Дзен Python» (PEP 20). Фраза про Python, но лучшего девиза для Vulkan не придумаешь.*

ГРАБЛИ. *Раз проверка ошибок живёт в слоях валидации, то без них программа с ошибкой может «работать». На твоей карте. Сегодня. Завтра обновится драйвер — и всё. Пустой лог валидации — единственное доказательство корректности.*

0.4. Карта объектов Vulkan

Всё в Vulkan — объекты, и они строго иерархичны. Каждый рождается от родителя и должен умереть раньше него.



Способ получения объекта определяет, кто и как его уничтожает:

Способ	Функции C API	Примеры	Как умирает
Создать	`vkCreate*` / `vkDestroy*`	Instance, Device, Buffer, Pipeline, Fence	`vkDestroy*` (у нас — деструктор `vk::raii`)
Выделить из пула	`vkAllocate*` / `vkFree*`	CommandBuffer, DescriptorSet, DeviceMemory	возвращается в пул или умирает вместе с ним
Перечислить / получить	`vkEnumerate*` / `vkGet*`	PhysicalDevice, Queue, картинки swapchain	никогда не уничтожается тобой

Buffer и Image — только «описание» ресурса: размер, формат, назначение. Память под них (DeviceMemory) выделяется отдельно и привязывается вручную. Это как договор аренды без квартиры: бумага есть, жить негде. В главе 10 эту работу возьмёт на себя VMA.

0.5. Разбор кода №1: порядок жизни и смерти

Правило «ребёнок умирает раньше родителя» в C++ реализуется бесплатно, через порядок полей класса. Проверим это на программе без Vulkan. Её можно запустить прямо сейчас на godbolt.org (https://godbolt.org): свежий GCC или Clang, галочка Execute.

chapter00_raii_order.cpp C++

```cpp
// chapter00_raii_order.cpp
// Сборка: g++ -std=c++23 chapter00_raii_order.cpp -o raii_order
// Нужен свежий компилятор (GCC 14+, Clang 18+, свежая Visual Studio 2022) ради <print>.
// На MSVC добавь флаг /utf-8, иначе русский текст превратится в иероглифы.
#include <print>
#include <string_view>

// «Шумный» объект: громко сообщает о своём рождении и смерти.
// Идеальный подопытный — как стажёр в первую неделю.
class Noisy {
public:
    explicit Noisy(std::string_view name) : name_{name} {
        std::println("  создан:    {}", name_);
    }
    ~Noisy() {
        std::println("  уничтожен: {}", name_);
    }

    // Копировать запрещаем — ровно как у vk::raii-объектов.
    // Две копии одного VkDevice = два vkDestroyDevice = двойное удаление = UB.
    Noisy(const Noisy&)            = delete;
    Noisy& operator=(const Noisy&) = delete;

private:
    std::string_view name_;  // безопасно: строковые литералы живут всю программу
};

class App {
public:
    App()  { std::println("App: конструктор отработал, все поля уже живы"); }
    ~App() { std::println("App: деструктор начался, все поля ещё живы"); }

```

chapter00_raii_order.cpp (продолжение) C++

```cpp
private:
    // Порядок объявления = порядок создания.
    Noisy instance_{"Instance"};
    Noisy device_  {"Device (зависит от Instance)"};
    Noisy buffer_  {"Buffer (зависит от Device)"};
};

int main() {
    std::println("--- старт ---");
    {
        App app;
        std::println("--- работаем ---");
    }  // ← здесь app выходит из области видимости
    std::println("--- финиш ---");
}
```

```
$ ./raii_order
--- старт ---
  создан:    Instance
  создан:    Device (зависит от Instance)
  создан:    Buffer (зависит от Device)
App: конструктор отработал, все поля уже живы
--- работаем ---
App: деструктор начался, все поля ещё живы
  уничтожен: Buffer (зависит от Device)
  уничтожен: Device (зависит от Instance)
  уничтожен: Instance
--- финиш ---
```

Разбор построчно.

• explicit Noisy(std::string_view name) — explicit запрещает неявное преобразование. Без него компилятор молча превратил бы любую строку в Noisy там, где ты этого не хотел. Конструктор с одним аргументом делай explicit, если нет особой причины.

• : name_{name} — список инициализации. Поле создаётся сразу с нужным значением, а не пустым с последующим присваиванием.

• = delete для копирования — то же, что делают vk::raii-объекты. У ресурса может быть только один владелец. Перемещать (std::move) vk::raii-объекты можно: владение переходит, а источник становится пустым.

• std::string_view name_ — вид на чужую строку, а не копия. Здесь это безопасно: литералы живут до конца программы. Передай сюда временную std::string — и получишь висячую ссылку.

• Noisy instance_{"Instance"}; — инициализатор поля прямо в объявлении (default member initializer). Не надо перечислять поля в конструкторе.

• Порядок создания полей определяется **только порядком объявления в классе**. Порядок в списке инициализации конструктора ничего не меняет, компилятор лишь предупредит через -Wreorder. Уничтожение идёт строго в обратном порядке.

• Тело конструктора App() выполняется **после** создания всех полей. Тело деструктора ~App() выполняется **до** их уничтожения. Именно в теле деструктора мы будем ждать GPU.

• Фигурные скобки в main создают блок. На закрывающей скобке app умирает, и мы видим всю цепочку деструкторов до строки «финиш».

Теперь то же самое с настоящими объектами Vulkan. Это скелет нашего «Пепелаца». Он скомпилируется после настройки окружения в главе 1, но пока ничего не делает. Создавать объекты мы научимся в главах 2–8.

pepelac_skeleton.hpp C++

```cpp
// pepelac_skeleton.hpp — скелет, «мясо» нарастим к главе 8
#pragma once
#include <vulkan/vulkan_raii.hpp>

#include <cstdint>
#include <vector>

class Pepelac {
public:
    Pepelac() = default;  // создание объектов — главы 2–8
    ~Pepelac();

    Pepelac(const Pepelac&)            = delete;
    Pepelac& operator=(const Pepelac&) = delete;

    void drawFrame();

private:
    static constexpr uint32_t MAX_FRAMES_IN_FLIGHT = 2;

    void recordCommandBuffer(vk::raii::CommandBuffer& cmd, uint32_t
      imageIndex);  // глава 7

    // ВНИМАНИЕ: порядок полей — это порядок создания.
    // Уничтожаться они будут в ОБРАТНОМ порядке. Не переставляй без
      понимания!
    // (Окно GLFW живёт ещё «выше» — разберём в главе 1.)

    vk::raii::Context                  context;
        // загрузчик Vulkan
    vk::raii::Instance                 instance         = nullptr;
        // подключение к Vulkan
    vk::raii::DebugUtilsMessengerEXT   debugMessenger   = nullptr;
        // канал жалоб валидации
    vk::raii::SurfaceKHR               surface          = nullptr;
        // холст окна ОС
    vk::raii::PhysicalDevice           physicalDevice   = nullptr;
        // выбранная видеокарта
```

pepelac_skeleton.hpp (продолжение) C++

```
    vk::raii::Device                    device           = nullptr;
        // «рабочий сеанс» с GPU
    vk::raii::Queue                     graphicsQueue    = nullptr;
        // почтовый ящик GPU

    vk::raii::SwapchainKHR              swapchain        = nullptr;
        // карусель кадров
    std::vector<vk::Image>              swapchainImages;
        // НЕ raii: ими владеет swapchain
    std::vector<vk::raii::ImageView>    swapchainImageViews;
        // «взгляд» на картинку

    vk::raii::PipelineLayout            pipelineLayout   = nullptr;
        // интерфейс пайплайна
    vk::raii::Pipeline                  graphicsPipeline = nullptr;
        // рецепт отрисовки

    vk::raii::CommandPool               commandPool      = nullptr;
        // блокнот
    std::vector<vk::raii::CommandBuffer> commandBuffers;
        // листы, по одному на кадр в полёте

    std::vector<vk::raii::Semaphore>    imageAvailable;  // размер:
      MAX_FRAMES_IN_FLIGHT
    std::vector<vk::raii::Semaphore>    renderFinished;  // размер:
      число картинок swapchain
    std::vector<vk::raii::Fence>        inFlight;        // размер:
      MAX_FRAMES_IN_FLIGHT

    uint32_t currentFrame = 0;
};

inline Pepelac::~Pepelac() {
    // Тело деструктора выполняется ДО уничтожения полей — успеваем
      дождаться GPU.
    if (*device) {
        try {
            device.waitIdle();
```

pepelac_skeleton.hpp (продолжение) C++

```
57          } catch (...) {
58              // Деструктор не имеет права бросать исключения (иначе
 ↪                std::terminate).
59              // Если GPU уже «умер» (DeviceLost), ждать нечего — идём
 ↪                дальше.
60          }
61      }
62  }
```

Разбор.

• vk::raii::Context context; — единственное поле без = nullptr. Конструктор Context находит и загружает системную библиотеку Vulkan и достаёт из неё адреса первых функций. Без него нельзя создать ни один объект.

• = nullptr — у всех vk::raii-объектов есть «пустое» состояние: поле существует, но ничем не владеет. Настоящий объект присвоим позже, когда соберём параметры. Деструктор пустого объекта ничего не делает.

• Порядок полей повторяет карту из раздела 0.4 сверху вниз. Уничтожаться они будут снизу вверх: семафоры, командные буферы, пул… и в самом конце Instance. Ровно то, чего требует Vulkan.

• std::vector<vk::Image> swapchainImages — обычные хэндлы, не raii. Картинки swapchain мы получаем, а не создаём, и уничтожает их сам swapchain.

• commandBuffers объявлены после commandPool. Деструктор vk::raii::CommandBuffer возвращает буфер в пул, и пул в этот момент обязан быть жив.

• Размеры массивов синхронизации разные, и это не опечатка. Почему так, разберём в разделе 0.7.

• if (*device) — оператор * у vk::raii-объекта возвращает «голый» хэндл vk::Device, а тот умеет превращаться в bool. Если устройство не создалось (исключение при инициализации), ждать нечего.

• device.waitIdle() блокирует CPU, пока GPU не закончит всю работу. Иначе мы уничтожим буферы и пайплайны, которыми GPU ещё рисует последний кадр.

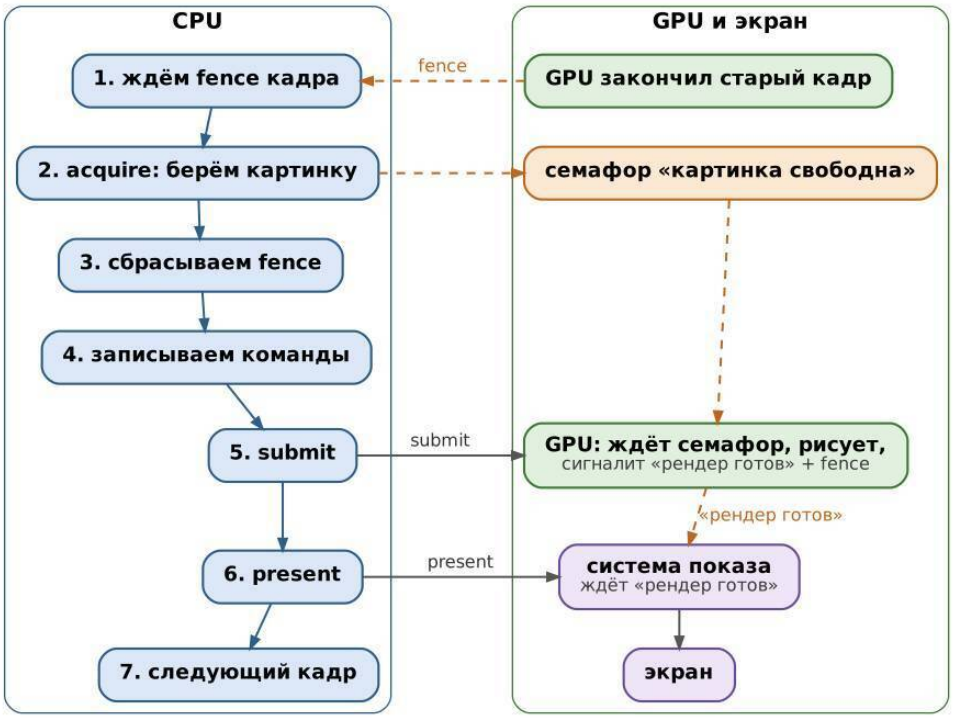
• try { … } catch (...) {} — деструкторы в C++ неявно noexcept. Исключение, вылетевшее из деструктора, вызовет std::terminate. Глотать исключения обычно плохо, но здесь это осознанный выбор: если GPU потерян, сделать уже ничего нельзя.

• inline перед определением деструктора нужен потому, что оно лежит в заголовочном файле. Без него два .cpp, включивших заголовок, дадут ошибку компоновщика «множественное определение».

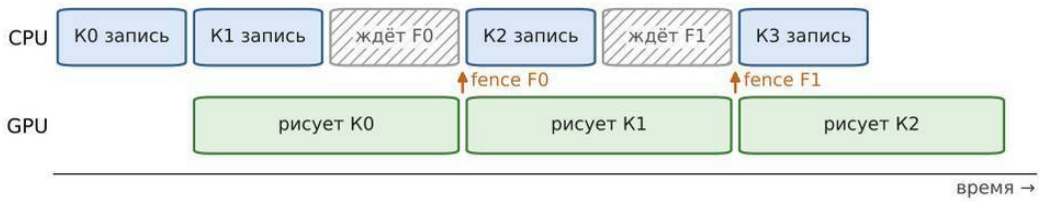
ГРАБЛИ. *Переставил device ниже commandPool «для красоты»? Устройство умрёт раньше пула, и деструктор пула вызовет vkDestroyCommandPool для уже мёртвого VkDevice. Это UB. В лучшем случае валидация завопит, в худшем программа упадёт на выходе, и только у твоих пользователей.*

0.6. Путь одного кадра

Каждый кадр в Vulkan проходит семь шагов. CPU делает их подряд, а GPU и система показа кадров отрабатывают свою часть асинхронно.



А вот как это выглядит во времени при двух кадрах в полёте (frames in flight), когда CPU быстрее GPU:



GPU не простаивает: пока он рисует кадр 0, CPU уже готовит кадр 1. Начать третий кадр CPU не может: его ресурсы (командный буфер, uniform-данные) ещё заняты кадром 0. Поэтому CPU ждёт fence F0. Такая программа упирается в GPU (GPU-bound), и это хорошо: карта загружена на 100%.

Почему два кадра, а не десять? Каждый лишний кадр в полёте — лишний кадр задержки между нажатием кнопки и реакцией на экране. Два — разумный баланс: GPU загружен, задержка маленькая.

0.7. Разбор кода №2: drawFrame построчно

Вот центральная функция любого Vulkan-приложения. Запускать её пока рано: объектам ещё неоткуда взяться. В главе 8 она заработает ровно в таком виде. Но понять её надо уже сейчас: всё остальное в книге нужно, чтобы эта функция работала.

фрагмент.cpp — C++

```cpp
void Pepelac::drawFrame() {
    // ── Шаг 1. Ждём, пока GPU дорисует кадр, который занимал эти ресурсы ──
    if (device.waitForFences(*inFlight[currentFrame], vk::True, UINT64_MAX)
        != vk::Result::eSuccess) {
        throw std::runtime_error("GPU не ответил на fence — что-то сильно не так");
    }

    // ── Шаг 2. Просим у swapchain номер свободной картинки ──
    auto [acquireResult, imageIndex] =
        swapchain.acquireNextImage(UINT64_MAX,
            *imageAvailable[currentFrame]);
    // acquireResult: eSuccess или eSuboptimalKHR. Ресайз окна — в главе 9.

    // ── Шаг 3. Сбрасываем fence — ТОЛЬКО после удачного acquire ──
    device.resetFences(*inFlight[currentFrame]);

    // ── Шаг 4. Записываем команды ──
    auto& cmd = commandBuffers[currentFrame];
    cmd.reset();
    recordCommandBuffer(cmd, imageIndex);  // барьеры + draw, глава 7

    // ── Шаг 5. Отправляем на GPU ──
    const vk::SemaphoreSubmitInfo waitInfo{
        .semaphore = *imageAvailable[currentFrame],
        .stageMask = vk::PipelineStageFlagBits2::eColorAttachmentOutput,
    };
    const vk::CommandBufferSubmitInfo cmdInfo{ .commandBuffer = *cmd };
    const vk::SemaphoreSubmitInfo signalInfo{
        .semaphore = *renderFinished[imageIndex],
        .stageMask = vk::PipelineStageFlagBits2::eAllCommands,
    };
    const vk::SubmitInfo2 submitInfo{
```

фрагмент.cpp (продолжение) C++

```
        .waitSemaphoreInfoCount   = 1, .pWaitSemaphoreInfos   =
          &waitInfo,
        .commandBufferInfoCount   = 1, .pCommandBufferInfos   =
          &cmdInfo,
        .signalSemaphoreInfoCount = 1, .pSignalSemaphoreInfos =
          &signalInfo,
    };
    graphicsQueue.submit2(submitInfo, *inFlight[currentFrame]);

    // ── Шаг 6. Просим показать картинку, когда рендер закончится ──
    const vk::PresentInfoKHR presentInfo{
        .waitSemaphoreCount = 1, .pWaitSemaphores =
          &*renderFinished[imageIndex],
        .swapchainCount     = 1, .pSwapchains     = &*swapchain,
        .pImageIndices      = &imageIndex,
    };
    [[maybe_unused]] const vk::Result presentResult =
      graphicsQueue.presentKHR(presentInfo);
    // eSuboptimalKHR обработаем в главе 9.

    // ── Шаг 7. Переходим к следующему набору ресурсов ──
    currentFrame = (currentFrame + 1) % MAX_FRAMES_IN_FLIGHT;
}
```

Шаг 1 — waitForFences**.** Блокирует CPU, пока fence не станет сигнальным. Сигналит его GPU, когда заканчивает командный буфер, отправленный с этим fence два кадра назад. commandBuffers[currentFrame] может всё ещё выполняться на GPU. Перезаписать его сейчас — всё равно что стирать доску, пока студенты переписывают.

• *inFlight[currentFrame] — * достаёт из vk::raii::Fence «голый» vk::Fence. raii-объект владеет ресурсом, а в функции передаётся именно хэндл. Этот приём будет встречаться постоянно.

• vk::True — «ждать все fence из списка». У нас он один, разницы нет.

• UINT64_MAX — тайм-аут в наносекундах, примерно 584 года. То есть «бесконечно».

• Результат помечен [[nodiscard]], поэтому мы его проверяем. Настоящие ошибки (например, потерю устройства) Vulkan-Hpp и так выбросит исключением.

• Почему не зависает самый первый кадр? Потому что fence создают сразу в сигнальном состоянии (флаг vk::FenceCreateFlagBits::eSignaled). Иначе мы вечно ждали бы кадр, которого не было.

Шаг 2 — acquireNextImage**.** Просим у swapchain номер картинки для рисования. Тонкий момент: **номер ты получаешь сразу, а саму картинку — позже.** Она ещё может висеть на мониторе. Когда она реально освободится, система сигналит семафор imageAvailable[currentFrame]. Ждать его будет GPU, а не CPU.

• auto [acquireResult, imageIndex] — структурное связывание (C++17). Функция возвращает пару «результат + значение», мы раскладываем её по двум именам.

• imageIndex не обязан идти по порядку 0, 1, 2: система вправе отдать 2, 0, 2, 1. И imageIndex — это **не** currentFrame! Это два независимых счётчика.

• Код eErrorOutOfDateKHR (окно изменило размер) Vulkan-Hpp выбрасывает как исключение vk::OutOfDateKHRError. Займёмся им в главе 9.

Шаг 3 — resetFences**.** Возвращает fence в несигнальное состояние, чтобы GPU смог снова его засигналить. Почему после acquire, а не сразу после ожидания? Если acquire выбросит исключение, мы выйдем из функции, не отправив работу. Сброшенный fence тогда никто не засигналит, и следующий кадр зависнет на шаге 1 навсегда.

Шаг 4 — запись. cmd.reset() стирает старые команды. Работает, только если пул создан с флагом eResetCommandBuffer (глава 7). auto& cmd — именно ссылка: vk::raii::CommandBuffer нельзя копировать.

Шаг 5 — submit2**.** Три структуры говорят GPU, чего ждать, что делать и кого оповестить.

• waitInfo — ждать семафор imageAvailable, но только перед стадией eColorAttachmentOutput (запись цвета в картинку). Вершинные шейдеры могут начать работу заранее, ждёт только запись пикселей.

• cmdInfo — сам командный буфер.

• signalInfo — когда все команды (eAllCommands) завершатся, засигналить renderFinished[imageIndex]. В конце кадра мелочиться со стадиями незачем.

• Второй аргумент submit2 — fence. Когда весь submit выполнится, GPU засигналит inFlight[currentFrame], и через два кадра шаг 1 дождётся именно его.

• Почему submit2, а не старый submit? Это часть synchronization2 из ядра Vulkan 1.3. У каждого семафора своя стадия, и те же типы стадий используются в барьерах. Меньше загадок.

• Имена полей в фигурных скобках (.semaphore = …) — designated initializers из C++20. Поля перечисляются в порядке объявления в структуре, пропускать можно. Поле sType Vulkan-Hpp заполняет сам. Для этого нужен дефайн VULKAN_HPP_NO_STRUCT_CONSTRUCTORS (глава 1).

Шаг 6 — presentKHR**.** Просим систему показа вывести картинку imageIndex, когда засигналится renderFinished[imageIndex]. Запись &*renderFinished[imageIndex] читается справа налево: * даёт ссылку на хэндл vk::Semaphore внутри raii-объекта, & берёт его адрес. Vulkan ждёт указатель на массив, а указатель на один элемент — это массив из одного. Очередь для показа мы используем ту же, что и для графики. На десктопах это почти всегда работает, проверять это научимся в главе 3.

Теперь главная загадка: почему imageAvailable индексируется currentFrame, а renderFinished — imageIndex? У present нет fence. Мы не знаем, когда система показа «съела» семафор и когда его можно использовать снова. Зато знаем другое: если картинку N нам снова отдали через acquire, её прошлый present точно завершён. Значит, семафор, привязанный к картинке N, свободен. Привяжи renderFinished к currentFrame — и валидация пожалуется на повторное использование семафора, который ещё может быть в работе.

Шаг 7. Переходим к следующему набору ресурсов по кругу: 0, 1, 0, 1…

ЗАПОМНИ. *Fence — сигнал от GPU к CPU: «я закончил, можешь трогать». Semaphore — сигнал между операциями на GPU: «сначала это, потом то», CPU его не видит. imageAvailable живёт по кадрам в полёте, renderFinished — по картинкам swapchain.*

ГРАБЛИ. *Путать imageIndex и currentFrame — самая популярная ошибка новичка. Командные буферы, fences и uniform-данные индексируются currentFrame. Картинки swapchain, их image views и renderFinished — imageIndex.*

0.8. А где тут многопоточность?

В большой сцене дороже всего на CPU шаг 4: запись десятков тысяч команд. Командные буферы — независимые объекты, и их можно записывать параллельно:



Кроме записи команд, в отдельные потоки уходят загрузка ресурсов (глава 23) и компиляция пайплайнов (глава 25). A submit и present остаются последовательными: очередь одна, и толкаться у неё из разных потоков нельзя.

ПОТОКИ (первое знакомство). *Функции создания объектов на одном VkDevice можно вызывать из разных потоков одновременно. Отправлять работу в одну очередь из двух потоков одновременно нельзя. Командный пул и все его буферы трогает только один поток за раз. Отсюда главное правило части IV: свой пул на каждый поток.*

Шпаргалка главы 0

- CPU записывает команды, GPU выполняет их асинхронно.
- Vulkan — явный API: компиляция, память и синхронизация — твоя работа.
- Иерархия: Instance → PhysicalDevice → Device → всё остальное.
- Три способа получить объект: create, allocate, enumerate/get.
- Ребёнок умирает раньше родителя. В C++ это обеспечивает порядок полей класса.
- Перед уничтожением всего — device.waitIdle().
- Кадр: wait fence → acquire → reset fence → record → submit → present.
- Fence: GPU → CPU. Semaphore: GPU → GPU.
- Два кадра в полёте — баланс между загрузкой GPU и задержкой.
- Запись команд параллелится, submit и present — нет.

Задачи для разминки мозгов

Задача 0.1. «Похороны в неправильном порядке». Джун собрал класс. Найди все проблемы с порядком уничтожения, объясни, что произойдёт при выходе из программы, и перепиши правильно. Бонус: почему компилятор не сказал ни слова?

фрагмент.cpp C++

```cpp
class BrokenApp {
    vk::raii::Context                    context;
    std::vector<vk::raii::CommandBuffer> commandBuffers;
    vk::raii::Device                     device         = nullptr;
    vk::raii::Instance                   instance       = nullptr;
    vk::raii::CommandPool                commandPool    = nullptr;
    vk::raii::PhysicalDevice             physicalDevice = nullptr;
};
```

Задача 0.2. «Бухгалтерия конвейера». CPU записывает кадр за 4 мс, GPU рисует его за 10 мс. Vsync выключен, present мгновенный, MAX_FRAMES_IN_FLIGHT = 2. (а) Какой будет FPS в установившемся режиме? (б) Сколько миллисекунд за кадр CPU ждёт fence? (в) Что станет с FPS и задержкой ввода при MAX_FRAMES_IN_FLIGHT = 1 и = 3?

Задача 0.3. «На каком кадре всё зависнет?» Джун «упростил» шаг 5 и убрал fence: graphicsQueue.submit2(submitInfo);. Всё остальное как в главе, fences созданы с eSignaled. (а) На каком кадре (считая с нуля) и на какой строке программа зависнет? (б) А если fences созданы без eSignaled? (в) А при MAX_FRAMES_IN_FLIGHT = 3?

Решения и разбор ваших неизбежных ошибок ищите в конце книги, в приложении А.

Глава 1. Окружение, или Четыре часа CMake ради пустого окна

К концу главы у тебя будет собирающийся проект «Пепелац», окно на GLFW и программа, которая сообщает версию Vulkan. Звучит скромно. На практике эта глава отнимает больше нервов, чем первый треугольник, потому что C++ до сих пор не договорился сам с собой, как подключать библиотеки.

1.1. Vulkan SDK

Vulkan SDK от LunarG — это заголовки (включая Vulkan-Hpp), слои валидации, компилятор шейдеров slangc, утилиты vulkaninfo и vkconfig. Драйвер видеокарты SDK не содержит: он приходит от NVIDIA, AMD, Intel или Mesa.

• **Windows:** установщик с vulkan.lunarg.com (https://vulkan.lunarg.com). Он сам пропишет переменную VULKAN_SDK.

• **Linux:** архив SDK с того же сайта, распаковать и выполнить source setup-env.sh (лучше дописать в ~/.bashrc). Драйверы Mesa и пакет загрузчика libvulkan1 ставятся из репозитория дистрибутива.

• **macOS:** SDK содержит MoltenVK — прослойку Vulkan поверх Metal. Работает, но с нюансами (portability, глава 2).

Проверка в терминале:

терминал Shell

```
$ vulkaninfo --summary   # список GPU, версии API и драйверов
$ vkcube                 # крутящийся куб: если он есть, Vulkan жив
$ slangc -v              # компилятор шейдеров на месте
```

В выводе vulkaninfo ищи строку apiVersion у своей видеокарты. Нужно 1.3 или выше. Если там 1.2 — обнови драйвер, это лечит в 90% случаев. Заодно поставь RenderDoc (https://renderdoc.org): он понадобится, когда на экране окажется чёрный прямоугольник. А он окажется.

1.2. Структура проекта



Зависимости тянем через vcpkg (https://vcpkg.io) в режиме манифеста. Клонируешь vcpkg, запускаешь bootstrap-скрипт и прописываешь переменную VCPKG_ROOT. Дальше всё делает CMake.

vcpkg.json JSON

```json
{
  "name": "pepelac",
  "version": "0.1.0",
  "dependencies": [
    "glfw3",
    "glm",
    "vulkan-memory-allocator",
    "stb",
    "fastgltf",
    "tracy"
  ]
}
```

Пока нам нужны только GLFW и GLM. Остальное пригодится в главах 10 (VMA), 13 (stb_image), 14 (fastgltf) и 29 (Tracy). Vulkan-Hpp берём из SDK, а не из vcpkg: так версии заголовков и слоёв валидации точно совпадут.

1.3. CMakeLists.txt построчно

CMakeLists.txt — CMake

```cmake
cmake_minimum_required(VERSION 3.28)
project(Pepelac LANGUAGES CXX)

# ── Шаг 1. Стандарт языка ──
set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

# ── Шаг 2. Зависимости ──
find_package(Vulkan REQUIRED)            # из Vulkan SDK
find_package(glfw3 CONFIG REQUIRED)      # из vcpkg
find_package(glm CONFIG REQUIRED)

# ── Шаг 3. Исполняемый файл ──
add_executable(pepelac src/main.cpp)
target_link_libraries(pepelac PRIVATE Vulkan::Vulkan glfw glm::glm)

# ── Шаг 4. Дефайны ──
target_compile_definitions(pepelac PRIVATE
    VULKAN_HPP_NO_STRUCT_CONSTRUCTORS  # designated initializers для
      vk::-структур
    GLFW_INCLUDE_NONE                  # GLFW, не тащи заголовки
      OpenGL
    GLM_FORCE_DEPTH_ZERO_TO_ONE        # глубина 0..1, как в Vulkan
      (а не -1..1, как в OpenGL)
)

# ── Шаг 5. Предупреждения ──
if(MSVC)
    target_compile_options(pepelac PRIVATE /W4 /permissive- /utf-8)
else()
    target_compile_options(pepelac PRIVATE -Wall -Wextra -Wpedantic)
endif()

```

CMakeLists.txt (продолжение) — CMake

```cmake
# ── Шаг 6. Санитайзеры (пригодятся в части III) ──
option(PEPELAC_ASAN "AddressSanitizer + UBSan" OFF)
option(PEPELAC_TSAN "ThreadSanitizer" OFF)
if(NOT MSVC)
    if(PEPELAC_ASAN)
        target_compile_options(pepelac PRIVATE
          -fsanitize=address,undefined -fno-omit-frame-pointer)
        target_link_options(pepelac PRIVATE
          -fsanitize=address,undefined)
    elseif(PEPELAC_TSAN)
        target_compile_options(pepelac PRIVATE -fsanitize=thread -g)
        target_link_options(pepelac PRIVATE -fsanitize=thread)
    endif()
endif()
```

Сборка:

терминал — Shell

```
$ cmake -B build -S . -DCMAKE_BUILD_TYPE=Debug \
  -DCMAKE_TOOLCHAIN_FILE=$VCPKG_ROOT/scripts/buildsystems/vcpkg.cmake
$ cmake --build build
```

Разбор.

• **Шаг 1.** CMAKE_CXX_STANDARD 23 — включаем C++23 ради std::print и std::expected. Основная часть кода книги — чистый C++20. EXTENSIONS OFF убирает расширения GNU, чтобы код не сломался на MSVC. EXPORT_COMPILE_COMMANDS создаёт compile_commands.json — его читают clangd и другие IDE.

• **Шаг 2.** find_package(Vulkan) — встроенный модуль CMake. Он находит SDK по переменной VULKAN_SDK. CONFIG у других пакетов означает «искать конфиг, который положил vcpkg».

• **Шаг 3.** Vulkan::Vulkan — загрузчик Vulkan. vk::raii::Context подгружает его и сам, но линковка нужна VMA в главе 10. PRIVATE — зависимости не протекают наружу.

• **Шаг 4.** Три дефайна, без которых всё пойдёт не так. VULKAN_HPP_NO_STRUCT_CONSTRUCTORS убирает конструкторы из vk::-структур и делает их агрегатами. Только тогда работает запись { .semaphore = … }. GLM_FORCE_DEPTH_ZERO_TO_ONE заставляет glm::perspective строить матрицу под диапазон глубины Vulkan. Забудешь — половина сцены исчезнет по непонятной причине.

• **Шаг 5.** Предупреждения на максимум. /utf-8 для MSVC обязателен, иначе русские комментарии и строки превратятся в мусор. /permissive- включает строгое соответствие стандарту.

• **Шаг 6.** Санитайзеры — встроенные в компилятор ловцы ошибок. ASan ловит выход за границы и use-after-free, TSan — гонки данных. Вместе они не работают, поэтому elseif. TSan станет твоим главным инструментом в части III.

1.4. Окно GLFW в RAII-обёртке

GLFW — сишная библиотека с парами init/terminate и create/destroy. Оборачиваем их так, чтобы забыть вызвать очистку было невозможно.

src/window.hpp C++

```cpp
// src/window.hpp
#pragma once
#include <GLFW/glfw3.h>

#include <cstdio>
#include <memory>
#include <stdexcept>
#include <string>

// ── Шаг 1. Вся библиотека GLFW как один RAII-объект ──
class GlfwLibrary {
public:
    GlfwLibrary() {
        // Лямбда без захвата превращается в обычный указатель на функцию — то, что ждёт C API.
        glfwSetErrorCallback([](int code, const char* text) {
            std::fprintf(stderr, "GLFW %d: %s\n", code, text);
        });
        if (glfwInit() != GLFW_TRUE) {
            throw std::runtime_error("glfwInit не удался");
        }
        if (glfwVulkanSupported() != GLFW_TRUE) {
            glfwTerminate();  // деструктор не вызовется: объект не достроен
            throw std::runtime_error("GLFW не нашёл загрузчик Vulkan");
        }
    }
    ~GlfwLibrary() { glfwTerminate(); }

    GlfwLibrary(const GlfwLibrary&)            = delete;
    GlfwLibrary& operator=(const GlfwLibrary&) = delete;
};

```

src/window.hpp (продолжение) C++

```cpp
// ── Шаг 2. Окно — unique_ptr со своим удалителем ──
struct GlfwWindowDeleter {
    void operator()(GLFWwindow* window) const noexcept {
      glfwDestroyWindow(window); }
};
using WindowPtr = std::unique_ptr<GLFWwindow, GlfwWindowDeleter>;

// ── Шаг 3. Фабрика окон ──
inline WindowPtr createWindow(int width, int height, const
  std::string& title) {
    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);  // НЕ создавать
      контекст OpenGL!
    glfwWindowHint(GLFW_RESIZABLE, GLFW_TRUE);
    GLFWwindow* raw = glfwCreateWindow(width, height, title.c_str(),
      nullptr, nullptr);
    if (raw == nullptr) {
        throw std::runtime_error("glfwCreateWindow не удался");
    }
    return WindowPtr{raw};
}
```

src/main.cpp C++

```cpp
// src/main.cpp
#include <vulkan/vulkan_raii.hpp>  // важно: ДО glfw3.h, тогда GLFW объявит свои Vulkan-функции
#include "window.hpp"

#include <cstdlib>
#include <exception>
#include <print>

int main() {
    try {
        // ── Шаг 4. Порядок объявления = порядок жизни ──
        GlfwLibrary glfw;                                   // умрёт последним
        WindowPtr window = createWindow(1280, 720, "Пепелац");

        // ── Шаг 5. Первое соприкосновение с Vulkan ──
        vk::raii::Context context;
        const uint32_t version = context.enumerateInstanceVersion();
        std::println("Загрузчик Vulkan: {}.{}.{}",
          VK_API_VERSION_MAJOR(version),
                    VK_API_VERSION_MINOR(version),
                  VK_API_VERSION_PATCH(version));

        // ── Шаг 6. Главный цикл ──
        while (glfwWindowShouldClose(window.get()) != GLFW_TRUE) {
            glfwPollEvents();
        }
    } catch (const std::exception& e) {
        std::println(stderr, "Фатальная ошибка: {}", e.what());
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

Разбор.

• **Шаг 1.** GlfwLibrary — объект без данных, вся его ценность в конструкторе и деструкторе. Обрати внимание на ветку glfwVulkanSupported: если конструктор бросает исключение, деструктор этого объекта не вызывается. Всё, что конструктор успел захватить, он освобождает сам. Следующая строка кода после throw уже не выполнится.

• **Шаг 2.** std::unique_ptr умеет владеть чем угодно, если дать ему удалитель. Удалитель — пустая структура с operator(), поэтому размер WindowPtr равен размеру обычного указателя. Бесплатная абстракция, за что мы и любим C++.

• **Шаг 3.** GLFW_CLIENT_API = GLFW_NO_API — самая важная строка. По умолчанию GLFW создаёт контекст OpenGL, а с ним создать Vulkan surface нельзя.

• **Шаг 4.** glfw объявлен раньше window, значит окно уничтожится до glfwTerminate. Та же иерархия, что и в главе 0.

• **Шаг 5.** enumerateInstanceVersion спрашивает загрузчик, какую версию API он поддерживает. Версия упакована в одно 32-битное число, макросы VK_API_VERSION_* достают её части. Это версия загрузчика, а не видеокарты: про карту узнаем в главе 3.

• **Шаг 6.** glfwPollEvents забирает события ОС: нажатия, движение мыши, крестик. Без него ОС решит, что программа зависла.

• catch (const std::exception&) в main — последний рубеж. Vulkan-Hpp сообщает об ошибках исключениями (vk::SystemError и наследники), и они прилетят сюда с понятным текстом. При раскрутке стека все RAII-объекты корректно уничтожатся.

ГРАБЛИ. *Порядок #include имеет значение. glfw3.h объявляет glfwCreateWindowSurface, только если до него уже подключён vulkan.h. Переставишь — в главе 4 получишь «функция не объявлена» и час весёлого дебага.*

1.5. Разминка по C++: как устроен vk::raii

Чтобы vk::raii не был магией, напишем его игрушечную версию: универсальный владелец хэндла, который можно перемещать, но нельзя копировать.

фрагмент.cpp C++

```cpp
#include <cstdio>
#include <utility>

template <class Handle, class Deleter>
class Unique {
public:
    Unique() = default;
    explicit Unique(Handle h) noexcept : handle_{h} {}
    ~Unique() { reset(); }

    // ── Перемещение: забираем хэндл, источник становится пустым ──
    Unique(Unique&& other) noexcept :
      handle_{std::exchange(other.handle_, Handle{})} {}
    Unique& operator=(Unique&& other) noexcept {
        if (this != &other) {
            reset();                                        //
              сначала отдаём своё
            handle_ = std::exchange(other.handle_, Handle{});   //
              потом забираем чужое
        }
        return *this;
    }

    // ── Копирование запрещено: владелец всегда один ──
    Unique(const Unique&)            = delete;
    Unique& operator=(const Unique&) = delete;

    void reset() noexcept {
        if (handle_ != Handle{}) {
            Deleter{}(handle_);
            handle_ = Handle{};
        }
    }
    [[nodiscard]] Handle get() const noexcept { return handle_; }
    explicit operator bool() const noexcept { return handle_ !=
      Handle{}; }

```

фрагмент.cpp (продолжение) C++

```cpp
private:
    Handle handle_{};
};

struct CloseFile {
    void operator()(std::FILE* f) const noexcept { std::fclose(f); }
};
using UniqueFile = Unique<std::FILE*, CloseFile>;

int main() {
    UniqueFile a{std::fopen("log.txt", "w")};
    UniqueFile b = std::move(a);  // a пуст, b владеет файлом
    // UniqueFile c = b;          // ошибка компиляции — и слава богу
}   // b закроет файл, а ничего не сделает
```

Разбор.

• std::exchange(other.handle_, Handle{}) возвращает старое значение и кладёт на его место новое. Одна строка вместо трёх, и забыть обнулить источник невозможно.

• noexcept на перемещении обязателен. std::vector при росте перемещает элементы только если перемещение не бросает исключений. Иначе он попытается копировать, а копирование запрещено.

• if (this != &other) защищает от x = std::move(x). Без проверки reset() уничтожил бы хэндл, который мы тут же пытаемся забрать.

• vk::raii::Buffer устроен так же, только внутри лежат ещё указатель на родительское устройство и таблица функций. Отсюда важное следствие: vk::raii**-объект помнит своего родителя по указателю**. Если родитель умрёт первым, ребёнок вызовет функцию через висячий указатель.

ЗАПОМНИ. *Объект, владеющий ресурсом: копирование запрещено, перемещение noexcept, деструктор безопасен для пустого состояния. Это правило пяти в самой полезной форме.*

ПОТОКИ. *Большинство функций GLFW можно вызывать только из главного потока: glfwInit, glfwCreateWindow, glfwPollEvents. Это ограничение самих ОС, особенно macOS. Поэтому в нашей архитектуре главный поток всегда владеет окном и событиями, а работу мы раздаём остальным.*

Шпаргалка главы 1

• SDK = заголовки + валидация + slangc. Драйвер ставится отдельно.

• vulkaninfo --summary — первая команда при любых странностях.

• Дефайны: VULKAN_HPP_NO_STRUCT_CONSTRUCTORS, GLFW_INCLUDE_NONE, GLM_FORCE_DEPTH_ZERO_TO_ONE.

• GLFW_CLIENT_API = GLFW_NO_API, иначе surface не создать.

• vulkan_raii.hpp подключаем до glfw3.h.

• RAII-владелец: без копирования, с noexcept-перемещением.

Задачи для разминки мозгов

Задача 1.1. «Дескриптор с подвохом». В POSIX файловый дескриптор — это int, и ноль там валидное значение (stdin). Невалидное — -1. Доработай шаблон Unique так, чтобы «пустое» значение задавалось снаружи, и покажи UniqueFd.

Задача 1.2. «Инвентаризация». Напиши программу, которая через vk::raii::Context печатает все доступные instance-расширения и слои, а в конце сообщает, есть ли VK_LAYER_KHRONOS_validation. Без ручных циклов поиска — только алгоритмы std::ranges.

Решения — в приложении А.

Часть I. Первый треугольник

Восемь глав от пустого окна до цветного треугольника, который переживает изменение размера окна. Каждая глава добавляет в класс Pepelac один-два метода create…().

Глава 2. Instance и слои валидации: заводим лучшего друга

Instance — это твоё подключение к Vulkan. При создании ты говоришь, какую версию API хочешь, какие расширения и какие слои. Слой валидации — единственное, что стоит между тобой и UB. Поэтому в этой главе мы не только создадим Instance, но и научим программу ругаться на саму себя.

2.1. Расширения и слои

Расширение — дополнительная функциональность, не входящая в ядро. Расширения бывают уровня instance (работа с окнами ОС, отладка) и уровня device (swapchain, новые фичи GPU). Префикс в имени подсказывает, кто его придумал: KHR — Khronos, EXT — несколько компаний, NV/AMD/ARM — один производитель.

Слой — библиотека, которая встаёт между твоим кодом и драйвером и перехватывает все вызовы. Слой VK_LAYER_KHRONOS_validation проверяет каждый вызов на соответствие спецификации и сообщает о нарушениях.



Слои валидации замедляют программу в разы, поэтому включаются только в debug-сборке.

2.2. Код: создаём Instance

src/pepelac_instance.cpp C++

```cpp
// src/pepelac_instance.cpp
#include "pepelac.hpp"

#include <algorithm>
#include <format>
#include <print>
#include <span>
#include <stdexcept>
#include <string_view>
#include <vector>

namespace {

// ── Шаг 1. Валидация только в debug ──
#ifdef NDEBUG
constexpr bool kEnableValidation = false;
#else
constexpr bool kEnableValidation = true;
#endif
constexpr const char* kValidationLayer =
  "VK_LAYER_KHRONOS_validation";

// ── Шаг 2. Callback, который вызовет слой валидации ──
VKAPI_ATTR vk::Bool32 VKAPI_CALL debugCallback(
    vk::DebugUtilsMessageSeverityFlagBitsEXT      severity,
    vk::DebugUtilsMessageTypeFlagsEXT             type,
    const vk::DebugUtilsMessengerCallbackDataEXT* data,
    void* /*userData*/) {
    std::string_view level = "INFO";
    switch (severity) {
        case vk::DebugUtilsMessageSeverityFlagBitsEXT::eError:
          level = "ОШИБКА"; break;
        case vk::DebugUtilsMessageSeverityFlagBitsEXT::eWarning:
          level = "ВНИМАНИЕ"; break;
        default: break;
    }
    std::println(stderr, "[{}] {} {}", level, vk::to_string(type),
      data->pMessage);
```

src/pepelac_instance.cpp (продолжение) C++

```cpp
    return vk::False;  // «не прерывай вызов» — всегда False
}

vk::DebugUtilsMessengerCreateInfoEXT makeDebugMessengerInfo() {
    return vk::DebugUtilsMessengerCreateInfoEXT{
        .messageSeverity = vk::DebugUtilsMessageSeverityFlagBitsEXT:
          :eWarning |
                           vk::DebugUtilsMessageSeverityFlagBitsEXT:
                  :eError,
        .messageType     = vk::DebugUtilsMessageTypeFlagBitsEXT::eGe
          neral |
                           vk::DebugUtilsMessageTypeFlagBitsEXT::eVa
                  lidation |
                           vk::DebugUtilsMessageTypeFlagBitsEXT::ePe
                  rformance,
        .pfnUserCallback = &debugCallback,
    };
}

// ── Шаг 3. Какие расширения нужны ──
std::vector<const char*> requiredInstanceExtensions() {
    uint32_t count = 0;
    const char** glfwExtensions = glfwGetRequiredInstanceExtensions(
      &count);
    if (glfwExtensions == nullptr) {
        throw std::runtime_error("GLFW не знает, как создавать
          surface на этой системе");
    }
    std::vector<const char*> extensions(glfwExtensions,
      glfwExtensions + count);
    if constexpr (kEnableValidation) {
        extensions.push_back(vk::EXTDebugUtilsExtensionName);
    }
#ifdef __APPLE__
    extensions.push_back(vk::KHRPortabilityEnumerationExtensionName)
      ;  // MoltenVK
#endif
    return extensions;
```

src/pepelac_instance.cpp (продолжение) C++

```cpp
}

// ── Шаг 4. Проверяем, что всё запрошенное существует ──
void ensureSupported(const vk::raii::Context& context,
                     std::span<const char* const> extensions,
                     std::span<const char* const> layers) {
    const auto availableExt = context.enumerateInstanceExtensionProperties();
    for (const char* name : extensions) {
        const bool found = std::ranges::any_of(availableExt,
          [name](const vk::ExtensionProperties& p) {
            return std::string_view{p.extensionName.data()} == name;
        });
        if (!found) {
            throw std::runtime_error(std::format("Нет расширения instance: {}", name));
        }
    }
    const auto availableLayers = context.enumerateInstanceLayerProperties();
    for (const char* name : layers) {
        const bool found = std::ranges::any_of(availableLayers,
          [name](const vk::LayerProperties& p) {
            return std::string_view{p.layerName.data()} == name;
        });
        if (!found) {
            throw std::runtime_error(std::format("Нет слоя {} — установлен ли Vulkan SDK?", name));
        }
    }
}

}  // namespace

```

src/pepelac_instance.cpp (продолжение) C++

```cpp
// ── Шаг 5. Собственно создание ──
void Pepelac::createInstance() {
    const vk::ApplicationInfo appInfo{
        .pApplicationName   = "Pepelac",
        .applicationVersion = VK_MAKE_API_VERSION(0, 0, 1, 0),
        .pEngineName        = "Pepelac Engine",
        .engineVersion      = VK_MAKE_API_VERSION(0, 0, 1, 0),
        .apiVersion         = vk::ApiVersion13,
    };

    const std::vector<const char*> extensions =
      requiredInstanceExtensions();
    std::vector<const char*> layers;
    if constexpr (kEnableValidation) {
        layers.push_back(kValidationLayer);
    }
    ensureSupported(context, extensions, layers);

    const vk::DebugUtilsMessengerCreateInfoEXT debugInfo =
      makeDebugMessengerInfo();
    vk::InstanceCreateInfo createInfo{
        .pNext                   = kEnableValidation ? &debugInfo :
          nullptr,
        .pApplicationInfo        = &appInfo,
        .enabledLayerCount       =
          static_cast<uint32_t>(layers.size()),
        .ppEnabledLayerNames     = layers.data(),
        .enabledExtensionCount   =
          static_cast<uint32_t>(extensions.size()),
        .ppEnabledExtensionNames = extensions.data(),
    };
#ifdef __APPLE__
    createInfo.flags |= vk::InstanceCreateFlagBits::eEnumeratePortab
      ilityKHR;
#endif

    instance = vk::raii::Instance(context, createInfo);

```

src/pepelac_instance.cpp (продолжение) C++

```
124     // — Шаг 6. Постоянный канал для сообщений —
125     if constexpr (kEnableValidation) {
126         debugMessenger = vk::raii::DebugUtilsMessengerEXT(instance,
  ↪           debugInfo);
127     }
128 }
```

2.3. Разбор по шагам

Шаг 1. Макрос NDEBUG стандартно определён в релизных сборках (именно он отключает assert). Мы превращаем его в constexpr bool, чтобы дальше писать if constexpr вместо #ifdef. Код внутри if constexpr компилируется и проверяется всегда, но в релизе выбрасывается. Опечатка в debug-ветке не доживёт до пятницы. Анонимный namespace { } делает всё внутри видимым только в этом .cpp.

Шаг 2. Сигнатура callback строго задана спецификацией. VKAPI_ATTR и VKAPI_CALL — макросы соглашения о вызове. На десктопах они обычно пустые, но на 32-битном Windows и Android без них стек разъедется. severity — ровно один бит, поэтому подходит switch. type — набор флагов, vk::to_string превратит его в читаемое { Validation }. Возвращаемое vk::False означает «продолжай вызов». vk::True резервирован для разработчиков самих слоёв.

Структура makeDebugMessengerInfo выбирает, что слушать: только предупреждения и ошибки всех типов. eVerbose и eInfo вывалят сотни строк на старте, включай их, только когда расследуешь что-то конкретное.

Шаг 3. GLFW сам знает, какие расширения нужны для окон на твоей ОС: VK_KHR_surface плюс платформенное (VK_KHR_win32_surface, VK_KHR_xcb_surface, VK_KHR_wayland_surface…). Он отдаёт массив строк, которым владеет сам, поэтому мы копируем указатели в свой std::vector. Конструктор vector(first, last) принимает пару указателей как итераторы. Константы вроде vk::EXTDebugUtilsExtensionName — те же строки, что макросы VK_EXT_DEBUG_UTILS_EXTENSION_NAME, только в пространстве имён vk.

Шаг 4. Если запросить несуществующее расширение, vkCreateInstance вернёт VK_ERROR_EXTENSION_NOT_PRESENT, и ты не узнаешь, какое именно. Поэтому проверяем заранее и с человеческим сообщением. std::span<const char* const> — невладеющий вид на массив: функция примет и vector, и array, и сишный массив без копирования. extensionName в Vulkan-Hpp — обёртка над char[256]. .data() даёт указатель на строку с нулём в конце, из него строим string_view для честного сравнения. Сравнение двух const char* через == сравнивает адреса, а не текст, и это классический баг новичка.

Шаг 5. ApplicationInfo представляет тебя драйверу. Имя и версия приложения не просто украшение: драйверы иногда включают по ним специальные оптимизации для известных движков. Главное поле — apiVersion: **максимальная версия Vulkan, которую ты собираешься использовать**. Это не гарантия, что видеокарта её поддерживает: это проверим в главе 3.

В InstanceCreateInfo самое интересное — pNext. Через него структуры сцепляются в цепочки, так Vulkan расширяет структуры без ломки ABI. Здесь мы прицепляем DebugUtilsMessengerCreateInfoEXT. Для чего? Постоянный messenger создаётся после instance и уничтожается до него. Ошибки в самих vkCreateInstance и vkDestroyInstance он не увидит. Структура в pNext создаёт временный messenger ровно на эти два вызова.

instance = vk::raii::Instance(context, createInfo); — создаём временный объект и перемещаем его в поле, которое было nullptr. При ошибке конструктор бросит исключение, и поле останется пустым. Строки расширений и слоёв Vulkan копирует внутрь себя, так что векторы можно спокойно уничтожить после вызова. Это общее правило: **всё, что ты передаёшь в** vkCreate***, нужно только на время вызова.**

Шаг 6. В чистом C функцию vkCreateDebugUtilsMessengerEXT пришлось бы доставать вручную через vkGetInstanceProcAddr: функции расширений не экспортируются загрузчиком. vk::raii::Instance делает это сам в конструкторе для всех функций сразу.

2.4. То же самое на голом C

Спецификация и половина примеров в интернете написаны на C API. Посмотри один раз на соответствие, и дальше читать спеку будет легко.

фрагмент.c

```c
VkApplicationInfo appInfo = {
    .sType      = VK_STRUCTURE_TYPE_APPLICATION_INFO,   // в
      Vulkan-Hpp заполняется само
    .pApplicationName = "Pepelac",
    .apiVersion = VK_API_VERSION_1_3,
};
VkInstanceCreateInfo createInfo = {
    .sType            = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO,
    .pApplicationInfo = &appInfo,
    /* ... слои и расширения ... */
};
VkInstance instance = VK_NULL_HANDLE;
VkResult result = vkCreateInstance(&createInfo, NULL, &instance);
if (result != VK_SUCCESS) { /* в Vulkan-Hpp — исключение */ }
/* ... */
vkDestroyInstance(instance, NULL);  /* в vk::raii — деструктор */
```

C API	Vulkan-Hpp
`VkInstanceCreateInfo`	`vk::InstanceCreateInfo`
`vkCreateInstance(...)` + `vkDestroyInstance`	`vk::raii::Instance(context, info)`
`VK_STRUCTURE_TYPE_...` в `sType`	заполняется автоматически
`VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT`	`vk::PipelineStageFlagBits2::eColorAttachmentOutput`
`VkResult` ≠ `VK_SUCCESS`	исключение `vk::SystemError`
`VK_TRUE` / `VK_FALSE`	`vk::True` / `vk::False`

2.5. Как читать сообщения валидации

У каждой ошибки валидации есть **VUID** (Valid Usage ID), например VUID-vkDestroyDevice-device-05137. Это точный адрес правила в спецификации. Сообщение состоит из трёх частей: VUID, список объектов (тип и хэндл) и текст правила. Алгоритм работы с ним такой:

1. Читаешь текст правила целиком. Он почти всегда точно говорит, что нарушено.

2. Ищешь VUID в спецификации на docs.vulkan.org (https://docs.vulkan.org) и смотришь контекст.

3. По хэндлам понимаешь, какой объект виноват. В главе 28 мы дадим объектам имена, и вместо 0x7f3a… в сообщении будет «Вершинный буфер корабля».

Полезный приём: поставь точку останова в debugCallback на ветку ошибки. Стек вызовов в отладчике покажет строку твоего кода, которая нарушила правило.

ЗАПОМНИ. *Правило нулевого лога: при включённой валидации программа не должна печатать ни одной ошибки и предупреждения. Никаких «это старое, оно всегда там было». Сегодняшнее предупреждение — завтрашний краш на чужой видеокарте.*

ГРАБЛИ. *Старые SDK (до 2025 года) объявляли pfnUserCallback с C-типами (VkBool32, VkDebugUtilsMessageSeverityFlagBitsEXT и т.д.). Если компилятор ругается на тип callback, обнови SDK или перепиши сигнатуру на C-типы.*

ПОТОКИ. *Callback валидации вызывается* ***в том потоке, который сделал ошибочный вызов Vulkan****. Когда в части IV команды начнут записываться из восьми потоков, callback станет вызываться параллельно. Один вызов std::println пишет строку целиком. А вот если callback пишет в общий std::vector или счётчик, нужна синхронизация.*

Шпаргалка главы 2

- Instance = версия API + расширения instance + слои.
- Расширения для окна берём у GLFW, VK_EXT_debug_utils добавляем в debug.
- Перед созданием проверяем наличие всего запрошенного.
- DebugUtilsMessengerCreateInfoEXT в pNext instance ловит ошибки создания и уничтожения instance.
- apiVersion — максимум, который ты собираешься использовать, а не гарантия поддержки.
- Данные для vkCreate* нужны только на время вызова.
- Нулевой лог валидации — непреложное правило.

Задачи для разминки мозгов

Задача 2.1. «Бухгалтер ошибок». Передай в callback через pUserData структуру со счётчиками ошибок и предупреждений. При выходе из программы напечатай итоги, а в debug-сборке завершайся с кодом ошибки, если была хотя бы одна. Учти, что callback может вызываться из разных потоков.

Задача 2.2. «Все претензии сразу». Версия ensureSupported из главы падает на первом ненайденном расширении. Перепиши её так, чтобы она собирала все отсутствующие расширения и слои и бросала одно исключение со списком. Используй std::views::filter.

Решения — в приложении А.

Глава 3. Физическое и логическое устройство: собеседование для видеокарты

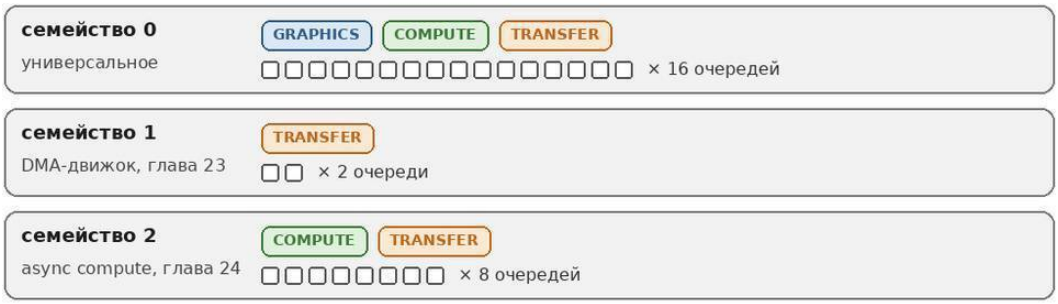
В компьютере может быть несколько GPU: дискретная карта, встроенная графика, программный рендерер llvmpipe. Vulkan показывает всех и предлагает тебе выбрать. В этой главе мы проведём собеседование, наймём лучшего кандидата и выдадим ему почтовые ящики — очереди.

3.1. Физическое и логическое

• **PhysicalDevice** — конкретная видеокарта как она есть. У неё можно только спрашивать: как зовут, что умеешь, сколько памяти. Мы её не создаём и не уничтожаем.

• **Device** (логическое устройство) — наш рабочий сеанс с этой картой. При создании мы перечисляем, какие фичи и очереди нам нужны. Всё не включённое запрещено, даже если железо умеет.

• **Queue family** — группа очередей с одинаковыми возможностями. Характерная картина для дискретной карты:



Разные семейства — это разные аппаратные блоки, которые могут работать одновременно. Это многопоточность уже на стороне GPU, и мы воспользуемся ею в части IV. А пока запомним номера всех трёх.

3.2. Surface заранее

Чтобы выбрать карту, надо знать, умеет ли она выводить картинку в наше окно. Для этого нужен surface. Подробно о нём в главе 4, а создаётся он одним вызовом GLFW:

фрагмент.cpp C++

```cpp
void Pepelac::createSurface() {
    VkSurfaceKHR raw = VK_NULL_HANDLE;
    if (glfwCreateWindowSurface(static_cast<VkInstance>(*instance),
      window, nullptr, &raw) != VK_SUCCESS) {
        throw std::runtime_error("Не удалось создать surface для
          окна");
    }
    surface = vk::raii::SurfaceKHR(instance, raw);  // теперь сырым
      хэндлом владеет raii
}
```

GLFW — сишная библиотека, она принимает и возвращает C-хэндлы. static_cast<VkInstance>(*instance) превращает vk::Instance в VkInstance. Конструктор vk::raii::SurfaceKHR(instance, raw) «усыновляет» чужой хэндл: теперь его уничтожит деструктор. window — невладеющий GLFWwindow*, который Pepelac получает в конструкторе. Окном владеет main, и оно живёт дольше рендерера.

3.3. Код: выбор видеокарты

src/pepelac_device.cpp C++

```cpp
// src/pepelac_device.cpp
#include "pepelac.hpp"

#include <algorithm>
#include <array>
#include <optional>
#include <print>
#include <string_view>

// ── Шаг 1. Что мы ищем ──
struct QueueFamilies {                      // объявлена в pepelac.hpp
    uint32_t                graphics = 0;  // graphics + present
    std::optional<uint32_t> asyncCompute;  // compute без graphics (глава 24)
    std::optional<uint32_t> transfer;      // только transfer (глава 23)
};

namespace {

constexpr std::array kDeviceExtensions = { vk::KHRSwapchainExtensionName };

using FeatureChain = vk::StructureChain<vk::PhysicalDeviceFeatures2,

                  vk::PhysicalDeviceVulkan11Features,

                  vk::PhysicalDeviceVulkan12Features,

                  vk::PhysicalDeviceVulkan13Features>;

// ── Шаг 2. Ищем нужные семейства очередей ──
std::optional<QueueFamilies> findQueueFamilies(const vk::raii::PhysicalDevice& gpu,
                                               const vk::raii::SurfaceKHR& surface) {
    const auto families = gpu.getQueueFamilyProperties();
```

src/pepelac_device.cpp (продолжение) C++

```cpp
    std::optional<uint32_t> graphics;
    QueueFamilies result;
    for (uint32_t i = 0; i < static_cast<uint32_t>(families.size());
       ++i) {
        const vk::QueueFlags flags = families[i].queueFlags;
        const bool hasGraphics = static_cast<bool>(flags &
          vk::QueueFlagBits::eGraphics);
        const bool hasCompute  = static_cast<bool>(flags &
          vk::QueueFlagBits::eCompute);
        const bool hasTransfer = static_cast<bool>(flags &
          vk::QueueFlagBits::eTransfer);

        if (!graphics && hasGraphics && gpu.getSurfaceSupportKHR(i,
          *surface) == vk::True) {
            graphics = i;
        }
        if (!result.asyncCompute && hasCompute && !hasGraphics) {
            result.asyncCompute = i;
        }
        if (!result.transfer && hasTransfer && !hasGraphics &&
          !hasCompute) {
            result.transfer = i;
        }
    }
    if (!graphics) {
        return std::nullopt;  // без графики с выводом на экран нам
          не по пути
    }
    result.graphics = *graphics;
    return result;
}

// ── Шаг 3. Собеседование: -1 = не подходит, иначе баллы ──
int rateDevice(const vk::raii::PhysicalDevice& gpu, const
  std::optional<QueueFamilies>& families) {
    const vk::PhysicalDeviceProperties props = gpu.getProperties();
    if (props.apiVersion < vk::ApiVersion13 || !families) {
        return -1;
```

src/pepelac_device.cpp (продолжение) C++

```cpp
    }

    const auto available = gpu.enumerateDeviceExtensionProperties();
    for (const char* name : kDeviceExtensions) {
        const bool found = std::ranges::any_of(available,
          [name](const vk::ExtensionProperties& p) {
            return std::string_view{p.extensionName.data()} == name;
        });
        if (!found) {
            return -1;
        }
    }

    const auto features = gpu.getFeatures2<vk::PhysicalDeviceFeature
      s2,

                vk::PhysicalDeviceVulkan11Features,

                vk::PhysicalDeviceVulkan12Features,

                vk::PhysicalDeviceVulkan13Features>();
    const auto& f10 = features.get<vk::PhysicalDeviceFeatures2>().fe
      atures;
    const auto& f11 = features.get<vk::PhysicalDeviceVulkan11Feature
      s>();
    const auto& f12 = features.get<vk::PhysicalDeviceVulkan12Feature
      s>();
    const auto& f13 = features.get<vk::PhysicalDeviceVulkan13Feature
      s>();
    if (!f10.samplerAnisotropy || !f11.shaderDrawParameters ||
      !f12.timelineSemaphore ||
        !f13.synchronization2 || !f13.dynamicRendering) {
        return -1;
    }

```

src/pepelac_device.cpp (продолжение) C++

```cpp
    int score = 0;
    if (props.deviceType == vk::PhysicalDeviceType::eDiscreteGpu)
      score += 1000;
    if (props.deviceType == vk::PhysicalDeviceType::eIntegratedGpu)
      score += 100;
    if (families->asyncCompute) score += 10;  // приятный бонус
    if (families->transfer)     score += 10;
    return score;
}

}  // namespace

// ── Шаг 4. Перебираем кандидатов ──
void Pepelac::pickPhysicalDevice() {
    int bestScore = -1;
    for (const vk::raii::PhysicalDevice& gpu :
      instance.enumeratePhysicalDevices()) {
        const auto candidateFamilies = findQueueFamilies(gpu,
          surface);
        const int score = rateDevice(gpu, candidateFamilies);
        std::println("GPU «{}»: {}",
          gpu.getProperties().deviceName.data(),
                    score < 0 ? std::string("не подходит") :
                  std::to_string(score));
        if (score > bestScore) {
            bestScore      = score;
            physicalDevice = gpu;              // PhysicalDevice
              можно копировать
            families       = *candidateFamilies;
        }
    }
    if (bestScore < 0) {
        throw std::runtime_error("Нет GPU с Vulkan 1.3, swapchain и
          нужными фичами");
    }
}

```

src/pepelac_device.cpp (продолжение) C++

```cpp
// ── Шаг 5. Нанимаем: создаём логическое устройство ──
void Pepelac::createLogicalDevice() {
    std::vector<uint32_t> familyIndices{ families.graphics };
    if (families.asyncCompute) familyIndices.push_back(*families.asyncCompute);
    if (families.transfer)     familyIndices.push_back(*families.transfer);

    const float priority = 1.0f;
    std::vector<vk::DeviceQueueCreateInfo> queueInfos;
    for (const uint32_t family : familyIndices) {
        queueInfos.push_back({ .queueFamilyIndex = family,
          .queueCount = 1, .pQueuePriorities = &priority });
    }

    const vk::StructureChain<vk::DeviceCreateInfo,
                             vk::PhysicalDeviceFeatures2,
                             vk::PhysicalDeviceVulkan11Features,
                             vk::PhysicalDeviceVulkan12Features,
                             vk::PhysicalDeviceVulkan13Features>
                  chain{
        vk::DeviceCreateInfo{
            .queueCreateInfoCount    =
              static_cast<uint32_t>(queueInfos.size()),
            .pQueueCreateInfos       = queueInfos.data(),
            .enabledExtensionCount   =
              static_cast<uint32_t>(kDeviceExtensions.size()),
            .ppEnabledExtensionNames = kDeviceExtensions.data(),
        },
        vk::PhysicalDeviceFeatures2{ .features = {
          .samplerAnisotropy = vk::True } },
        vk::PhysicalDeviceVulkan11Features{ .shaderDrawParameters =
          vk::True },
        vk::PhysicalDeviceVulkan12Features{ .timelineSemaphore =
          vk::True },
        vk::PhysicalDeviceVulkan13Features{ .synchronization2 =
          vk::True, .dynamicRendering = vk::True },
    };
```

src/pepelac_device.cpp (продолжение) C++

```
142
143     device         = vk::raii::Device(physicalDevice,
        ↪   chain.get<vk::DeviceCreateInfo>());
144     graphicsQueue = vk::raii::Queue(device, families.graphics, 0);
145     if (families.asyncCompute) computeQueue  =
        ↪   vk::raii::Queue(device, *families.asyncCompute, 0);
146     if (families.transfer)     transferQueue =
        ↪   vk::raii::Queue(device, *families.transfer, 0);
147 }
```

3.4. Разбор

Шаг 1. std::optional<uint32_t> — честный способ сказать «может и не быть». Магическое значение вроде UINT32_MAX работает до первой забытой проверки. optional же не даст достать значение без явного *. std::array kDeviceExtensions = {…} — CTAD: тип и размер компилятор выведет сам (std::array<const char*, 1>).

Шаг 2. queueFlags — набор флагов. Выражение flags & vk::QueueFlagBits::eGraphics даёт набор, а static_cast<bool> превращает его в ответ «есть хотя бы один бит». В Vulkan-Hpp флаги типобезопасны: случайно смешать QueueFlags с BufferUsageFlags не выйдет. getSurfaceSupportKHR(i, *surface) спрашивает, умеет ли семейство i выводить в наше окно. На десктопах графическое семейство почти всегда умеет, но спецификация этого не обещает (задача 3.2).

Логика поиска «выделенных» семейств строгая. Async compute — это compute **без** graphics: такая очередь с большой вероятностью работает на отдельном аппаратном блоке. Transfer — только transfer: это DMA-движок, копирующий данные параллельно с рендером. На встроенных картах их часто нет, и это нормально: тогда оптионалы останутся пустыми, и мы откатимся на графическую очередь.

Шаг 3. Собеседование из двух этапов. Сначала жёсткие требования, провал любого — отказ. Потом баллы за приятное. На жёсткие требования смотри внимательно:

• props.apiVersion — версия Vulkan, которую поддерживает **драйвер этой карты**. Версии упакованы так, что их можно сравнивать обычным <.

• getFeatures2<…>() — шаблонная функция Vulkan-Hpp. Она создаёт цепочку структур через pNext, вызывает vkGetPhysicalDeviceFeatures2 и отдаёт заполненную цепочку. Достаём звенья через .get<T>(). На C это десять строк ручной связки указателей, в которых легко ошибиться.

• Каждая фича проверяется не просто так. samplerAnisotropy нужен для текстур (глава 13), shaderDrawParameters — Slang-шейдерам, timelineSemaphore — части IV, synchronization2 и dynamicRendering — всей книге. В Vulkan 1.3 последние две обязательны, но проверка ничего не стоит, а от багов в драйверах спасает.

Шаг 4. instance.enumeratePhysicalDevices() возвращает вектор vk::raii::PhysicalDevice. В отличие от остальных raii-классов, PhysicalDevice копируется: он ничем не владеет и ничего не уничтожает. Выражение score < 0 ? std::string(…) : std::to_string(score) обязано иметь один тип в обеих ветках, поэтому литерал обёрнут в std::string.

Шаг 5. Самое интересное:

• DeviceQueueCreateInfo — по одной структуре на каждое семейство. Семейства в списке не должны повторяться, иначе валидация покраснеет. У нас они разные по построению, потому что у них разные флаги.

• pQueuePriorities — приоритет очереди от 0 до 1. Это подсказка драйверу, а не гарантия. Указатель должен жить до вызова vkCreateDevice, поэтому priority — локальная переменная функции, а не временный объект.

• vk::StructureChain<…> — жемчужина Vulkan-Hpp. Конструктор цепочки сам связывает pNext: DeviceCreateInfo → Features2 → Vulkan11 → Vulkan12 → Vulkan13. А ещё на этапе компиляции проверяет, что каждая структура имеет право расширять голову. Попробуй запихнуть туда vk::ImageCreateInfo — не скомпилируется.

• Фичи версии 1.0 (samplerAnisotropy) идут через PhysicalDeviceFeatures2 в цепочке. При этом поле pEnabledFeatures в DeviceCreateInfo обязано остаться nullptr. Одно из двух, не оба.

• vk::raii::Device(physicalDevice, …) загружает указатели на device-функции напрямую из драйвера, минуя промежуточные функции загрузчика. Это бесплатно экономит пару переходов на каждый вызов.

• vk::raii::Queue(device, family, 0) — получаем очередь номер 0 из семейства. Очереди создаются вместе с устройством и умирают с ним, деструктор raii::Queue ничего не делает.

В заголовок Pepelac добавляются поля GLFWwindow* window, QueueFamilies families, vk::raii::Queue computeQueue = nullptr и vk::raii::Queue transferQueue = nullptr — сразу после graphicsQueue.

ЗАПОМНИ. *Фича, которую ты не включил при создании Device, запрещена, даже если железо её умеет. Валидация поймает, а без неё получишь UB. Добавил в код новую фичу — добавь её и в проверку (rateDevice), и в создание устройства.*

ГРАБЛИ. *Ноутбуки с двумя GPU любят по умолчанию запускать приложения на встроенной графике. А в виртуалках и CI иногда есть только llvmpipe — рендерер на CPU. Печать имени выбранной карты в лог — не украшение, а первое, что ты попросишь у тестировщика с багом «у меня 3 FPS».*

ПОТОКИ. *Запросы к PhysicalDevice безопасны из любых потоков. А вот каждая VkQueue требует* ***внешней синхронизации****: два потока не могут одновременно вызвать submit в одну очередь. Отдельные очереди transfer и compute решают это изящно: у каждой свой поток-владелец, и мьютексы не нужны. Именно так устроены главы 23 и 24.*

Шпаргалка главы 3

• PhysicalDevice — справочное бюро, Device — рабочий сеанс.

• Жёсткие требования: apiVersion ≥ 1.3, VK_KHR_swapchain, нужные фичи, семейство graphics + present.

• Фичи включаются через StructureChain из Features2/11/12/13.

• Выделенные семейства: compute без graphics, transfer без всего. Опциональны.

• Одна VkQueue = один поток за раз.

Задачи для разминки мозгов

Задача 3.1. «Выбор по заказу». Добавь переменную окружения PEPELAC_GPU. Если она задана, выбирается первая подходящая карта, в имени которой есть эта подстрока. Если такой нет или она не прошла собеседование — предупреждение и обычный выбор по баллам.

Задача 3.2. «Карта с характером». Спецификация допускает GPU, у которого графическое семейство не умеет выводить на экран, а другое умеет. Перепиши findQueueFamilies с

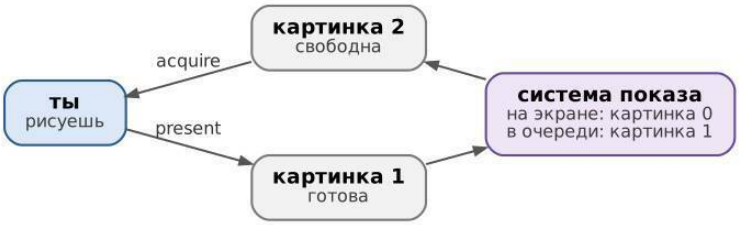
отдельным present-семейством, предпочитая совпадение с графическим. Покажи, как поменяется createLogicalDevice, чтобы семейства не повторялись.

Решения — в приложении А.

Глава 4. Surface и swapchain: карусель кадров

Vulkan не умеет рисовать «в окно». Он рисует в картинки, а показом картинок занимается отдельная система показа — presentation engine, часть ОС и драйвера. Swapchain — это договор между тобой и этой системой: набор картинок, которые вы передаёте друг другу по кругу.

4.1. Как устроен swapchain



У каждой картинки в каждый момент есть хозяин: либо ты (после acquire), либо система показа (после present). Рисовать в картинку, которая сейчас на экране, нельзя: получишь разрывы изображения. Поэтому картинок нужно минимум две, а лучше три.

4.2. Режимы показа

Режим	Что делает	Разрывы	Задержка	Когда брать
`eFifo`	очередь, показ строго по vsync	нет	выше	по умолчанию, гарантирован спецификацией
`eFifoRelaxed`	как FIFO, но опоздавший кадр показывает сразу	иногда	средняя	если FPS чуть ниже частоты монитора
`eMailbox`	новый кадр заменяет ожидающий, показ по vsync	нет	низкая	игры на десктопе
`eImmediate`	показывает немедленно	да	минимальная	бенчмарки, киберспорт

FIFO — единственный режим, который обязан поддерживать любой драйвер. Остальные надо проверять.

4.3. Код: создание swapchain

src/pepelac_swapchain.cpp C++

```cpp
// src/pepelac_swapchain.cpp
#include "pepelac.hpp"

#include <algorithm>
#include <limits>
#include <span>

namespace {

// ── Шаг 1. Формат пикселей и цветовое пространство ──
vk::SurfaceFormatKHR chooseSurfaceFormat(std::span<const
  vk::SurfaceFormatKHR> formats) {
    const auto it = std::ranges::find_if(formats, [](const
      vk::SurfaceFormatKHR& f) {
        return f.format == vk::Format::eB8G8R8A8Srgb &&
               f.colorSpace == vk::ColorSpaceKHR::eSrgbNonlinear;
    });
    return it != formats.end() ? *it : formats.front();  // нет
      идеала — берём что дают
}

// ── Шаг 2. Режим показа ──
vk::PresentModeKHR choosePresentMode(std::span<const
  vk::PresentModeKHR> modes) {
    if (std::ranges::find(modes, vk::PresentModeKHR::eMailbox) !=
      modes.end()) {
        return vk::PresentModeKHR::eMailbox;
    }
    return vk::PresentModeKHR::eFifo;  // гарантирован спецификацией
}

// ── Шаг 3. Размер картинок в ПИКСЕЛЯХ ──
vk::Extent2D chooseExtent(const vk::SurfaceCapabilitiesKHR& caps,
  GLFWwindow* window) {
    if (caps.currentExtent.width !=
      std::numeric_limits<uint32_t>::max()) {
        return caps.currentExtent;  // система сама сказала, какой
          нужен размер
```

src/pepelac_swapchain.cpp (продолжение) C++

```cpp
    }
    int width = 0;
    int height = 0;
    glfwGetFramebufferSize(window, &width, &height);  // пиксели, а НЕ экранные координаты
    return vk::Extent2D{
        std::clamp(static_cast<uint32_t>(width),  caps.minImageExtent.width,  caps.maxImageExtent.width),
        std::clamp(static_cast<uint32_t>(height), caps.minImageExtent.height, caps.maxImageExtent.height),
    };
}

}  // namespace

void Pepelac::createSwapchain() {
    // ── Шаг 4. Спрашиваем, что умеет пара «видеокарта + окно» ──
    const vk::SurfaceCapabilitiesKHR caps = physicalDevice.getSurfaceCapabilitiesKHR(*surface);
    const auto formats = physicalDevice.getSurfaceFormatsKHR(*surface);
    const auto modes   = physicalDevice.getSurfacePresentModesKHR(*surface);

    const vk::SurfaceFormatKHR format      = chooseSurfaceFormat(formats);
    const vk::PresentModeKHR   presentMode = choosePresentMode(modes);
    const vk::Extent2D         extent      = chooseExtent(caps, window);

    // ── Шаг 5. Сколько картинок ──
    uint32_t imageCount = std::max(3u, caps.minImageCount);
    if (caps.maxImageCount > 0) {                       // 0 означает «без верхнего предела»
        imageCount = std::min(imageCount, caps.maxImageCount);
    }

```

src/pepelac_swapchain.cpp (продолжение) C++

```cpp
    // ── Шаг 6. Прозрачность окна нам не нужна ──
    const vk::CompositeAlphaFlagBitsKHR alpha =
        (caps.supportedCompositeAlpha &
          vk::CompositeAlphaFlagBitsKHR::eOpaque)
            ? vk::CompositeAlphaFlagBitsKHR::eOpaque
            : vk::CompositeAlphaFlagBitsKHR::eInherit;

    // ── Шаг 7. Создаём ──
    const vk::SwapchainCreateInfoKHR info{
        .surface          = *surface,
        .minImageCount    = imageCount,
        .imageFormat      = format.format,
        .imageColorSpace  = format.colorSpace,
        .imageExtent      = extent,
        .imageArrayLayers = 1,                                  //
          не стерео, не VR
        .imageUsage       = vk::ImageUsageFlagBits::eColorAttachment
          ,
        .imageSharingMode = vk::SharingMode::eExclusive,        //
          одна очередь и рисует, и показывает
        .preTransform     = caps.currentTransform,              //
          не вращаем (актуально для телефонов)
        .compositeAlpha   = alpha,
        .presentMode      = presentMode,
        .clipped          = vk::True,                           //
          пиксели под чужими окнами не важны
        .oldSwapchain     = nullptr,                            //
          пригодится в главе 9
    };
    swapchain       = vk::raii::SwapchainKHR(device, info);
    swapchainImages = swapchain.getImages();  // сколько их на
      самом деле — решает драйвер
    swapchainFormat = format.format;
    swapchainExtent = extent;
}

```

src/pepelac_swapchain.cpp (продолжение) C++

```cpp
// ── Шаг 8. «Окна» в картинки ──
void Pepelac::createImageViews() {
    swapchainImageViews.clear();
    swapchainImageViews.reserve(swapchainImages.size());
    for (const vk::Image image : swapchainImages) {
        swapchainImageViews.emplace_back(device,
          vk::ImageViewCreateInfo{
            .image            = image,
            .viewType         = vk::ImageViewType::e2D,
            .format           = swapchainFormat,
            .subresourceRange = {
                .aspectMask     = vk::ImageAspectFlagBits::eColor,
                .baseMipLevel   = 0,
                .levelCount     = 1,
                .baseArrayLayer = 0,
                .layerCount     = 1,
            },
        });
    }
}
```

4.4. Разбор

Шаг 1 — формат. B8G8R8A8 — четыре канала по 8 бит. Порядок BGR исторически родной для дисплеев, для шейдера разницы нет. Важнее суффикс Srgb. Шейдеры считают цвет в линейном пространстве (вдвое больше света = вдвое больше число), а монитор ждёт гамма-сжатые значения. При записи в sRGB-картинку железо само делает преобразование, бесплатно. Возьмёшь Unorm — средние тона станут темнее, а ты неделю будешь крутить освещение в шейдерах.

std::ranges::find_if(formats, …) возвращает итератор. Сравнение с formats.end() — проверка «нашли ли». formats.front() безопасен: спецификация гарантирует хотя бы один формат для поддерживаемого surface.

Шаг 2 — режим. Предпочитаем MAILBOX: низкая задержка без разрывов. Нет его — FIFO, он есть всегда. std::span<const T> принимает вектор, вернувшийся из Vulkan-Hpp, без копирования.

Шаг 3 — размер. currentExtent обычно равен размеру окна. Специальное значение 0xFFFFFFFF означает «размер решаешь ты», так делает, например, Wayland. Тогда берём размер **фреймбуфера в пикселях** и зажимаем в разрешённые границы. std::clamp(x, lo, hi) — короткая запись min(max(x, lo), hi). На экранах с масштабированием 200% окно «1280×720» занимает 2560×1440 пикселей, отсюда задача 4.2.

Шаг 4. Три вопроса к паре «видеокарта + surface». Ответы зависят и от драйвера, и от оконной системы, поэтому захардкодить их нельзя.

Шаг 5 — число картинок. Для MAILBOX нужно три: одна на экране, одна ждёт очереди, в одну рисуешь. std::max(3u, …) — 3u именно unsigned, иначе std::max не выведет тип из int и uint32_t и не скомпилируется. maxImageCount == 0 — ещё одно спецзначение, на нём спотыкается каждый второй. И важно: мы просим **минимум**, драйвер вправе создать больше. Реальное число узнаём из getImages(), и именно его используем для размера renderFinished.

Шаг 6 — прозрачность. compositeAlpha решает, смешивать ли окно с рабочим столом по альфе. Большинство систем поддерживают eOpaque, но не все, поэтому выбираем с проверкой. Битовое & на флагах даёт набор, а в условии ? : он явно и безопасно приводится к bool.

Шаг 7 — создание. Поля, в которых новички ошибаются чаще всего:

• imageUsage = eColorAttachment — мы будем рисовать в эти картинки как в цветовой буфер. Захочешь копировать в них (blit) — добавь eTransferDst, но сначала проверь caps.supportedUsageFlags.

• imageSharingMode = eExclusive — картинка принадлежит одному семейству очередей за раз. Это быстрее, чем eConcurrent. Передачу владения между семействами разберём в главе 23.

• preTransform = currentTransform — «ничего не вращать». На телефонах существует целая наука про поворот кадра без лишнего копирования, но на десктопе это всегда identity.

• clipped = vk::True — содержимое пикселей, перекрытых чужими окнами, нам неважно. Драйвер может их не рисовать.

• getImages() возвращает std::vector<vk::Image> — голые хэндлы без RAII. Ими владеет swapchain.

Шаг 8 — image views. Шейдеры и команды рендеринга работают не с картинкой, а с **видом** на неё. View говорит, как читать байты: какой формат, какие mip-уровни, какие слои массива, какой аспект (цвет, глубина, трафарет). Аналогия: std::string_view на std::string. Из одной картинки можно сделать несколько разных view.

• reserve заранее выделяет память в векторе, чтобы не было переездов при emplace_back.

• emplace_back(device, info) конструирует vk::raii::ImageView прямо в векторе. vk::ImageViewCreateInfo{…} приходится писать с именем типа: через аргументы шаблонной функции фигурные скобки без типа не проходят.

• components не задаём: по умолчанию там eIdentity, каналы не переставляются.

• В Pepelac появляются поля vk::Format swapchainFormat и vk::Extent2D swapchainExtent рядом со swapchain. Они нужны пайплайну (глава 6) и рендерингу (глава 7).

ЗАПОМНИ. *Image — память с пикселями, ImageView — инструкция, как их читать. Рисуют в view, барьеры ставят на image. И никогда не полагайся на то, что картинок столько, сколько ты просил.*

ГРАБЛИ. *MAILBOX рендерит кадры без остановки и выбрасывает лишние. На ноутбуке это 100% загрузки GPU, вентилятор в режиме «взлёт» и батарея на час. Для редактора или UI-приложения бери FIFO.*

ПОТОКИ. *Swapchain требует внешней синхронизации при acquireNextImage и при пересоздании. present требует её от очереди. Проще всего закрепить acquire и present за одним потоком — обычно главным или выделенным «потоком рендера». На Windows и macOS present ещё и любит поток, который обрабатывает сообщения окна. Не всегда, но лучше не проверять в пятницу.*

Шпаргалка главы 4

• Surface = окно для Vulkan. Swapchain = карусель картинок между тобой и системой показа.

• Формат: B8G8R8A8Srgb + SrgbNonlinear, иначе цвета поедут.

• Режим: MAILBOX если есть, иначе FIFO (есть всегда).

• Размер — в пикселях фреймбуфера: glfwGetFramebufferSize.

• Спецзначения: currentExtent = 0xFFFFFFFF, maxImageCount = 0.

• Реальное число картинок — из getImages().

• На каждую картинку — свой ImageView.

Задачи для разминки мозгов

Задача 4.1. «Настройка vsync». Сделай структуру настроек с полем bool vsync. С vsync режим всегда FIFO. Без vsync — первый поддержанный из списка MAILBOX → IMMEDIATE → FIFO. Реализуй выбор одной функцией по списку приоритетов, без лесенки if.

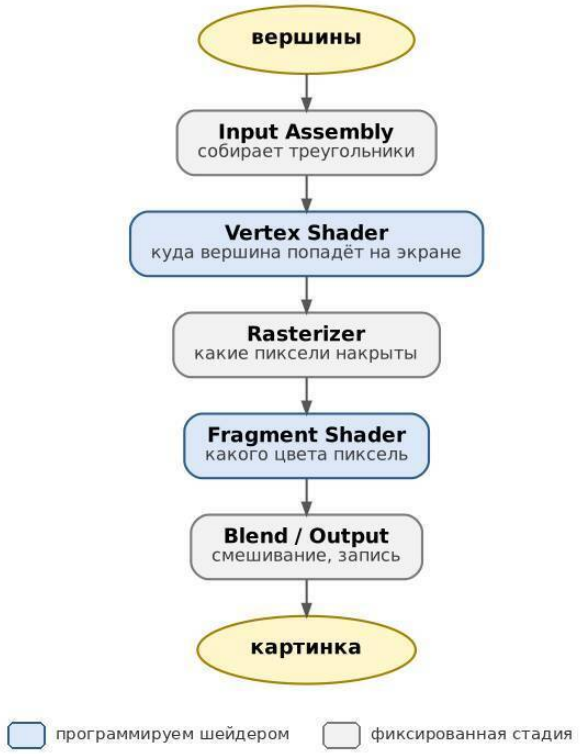
Задача 4.2. «Ноутбук с 4K-экраном». Объясни разницу между glfwGetWindowSize и glfwGetFramebufferSize на экране с масштабом 200%. Что именно сломается, если передать в swapchain размер окна? И нужен ли вообще glfwGetFramebufferSize, если currentExtent обычно заполнен?

Решения — в приложении А.

Глава 5. Шейдеры: программы, которые живут в видеокарте

Шейдер — это маленькая программа, которую GPU запускает тысячи раз параллельно: для каждой вершины, для каждого пикселя. Пишем мы его на Slang, компилируем в SPIR-V при сборке, а драйвер доводит SPIR-V до машинного кода своей видеокарты. Звучит как три компилятора подряд. Так и есть.

5.1. Графический конвейер



Два этапа мы пишем сами, остальные только настраиваем (глава 6).

• **Вершинный шейдер** запускается один раз на каждую вершину и возвращает её позицию в clip space. После деления на w получаются нормализованные координаты (NDC).

• **Растеризатор** решает, какие пиксели накрыты треугольником, и интерполирует выходы вершинного шейдера между вершинами.

• **Фрагментный шейдер** запускается на каждый накрытый пиксель и возвращает его цвет.

Координаты NDC в Vulkan: x от −1 (лево) до +1 (право), y от −1 (**верх**) до +1 (**низ**), z от 0 до 1. В OpenGL y смотрит вверх, а z идёт от −1. Половина багов переноса с OpenGL — перевёрнутая картинка и исчезнувшая глубина.

5.2. Первый шейдер на Slang

Slang — язык шейдеров с синтаксисом, близким к HLSL. В одном файле можно держать несколько точек входа, есть модули и даже дженерики. Вершины пока храним прямо в шейдере: буферы появятся в главе 11.

shaders/triangle.slang C++

```
// shaders/triangle.slang

// ── Шаг 1. Данные треугольника ──
static const float2 kPositions[3] = {
    float2( 0.0, -0.5),   // верх (в Vulkan ось Y смотрит ВНИЗ!)
    float2( 0.5,  0.5),   // правый нижний
    float2(-0.5,  0.5),   // левый нижний
};
static const float3 kColors[3] = {
    float3(1.0, 0.0, 0.0),  // красный
    float3(0.0, 1.0, 0.0),  // зелёный
    float3(0.0, 0.0, 1.0),  // синий
};

// ── Шаг 2. Что вершинный шейдер отдаёт дальше ──
struct VertexOutput {
    float4 position : SV_Position;  // обязательный выход: куда на экране
    float3 color;                   // произвольные данные для фрагментного шейдера
};

// ── Шаг 3. Вершинный шейдер ──
[shader("vertex")]
VertexOutput vertMain(uint vertexId : SV_VertexID) {
    VertexOutput output;
    output.position = float4(kPositions[vertexId], 0.0, 1.0);  // z = 0, w = 1
    output.color    = kColors[vertexId];
    return output;
}

// ── Шаг 4. Фрагментный шейдер ──
[shader("fragment")]
float4 fragMain(VertexOutput input) : SV_Target {
    return float4(input.color, 1.0);  // цвет уже проинтерполирован растеризатором
}
```

Разбор.

• **Шаг 1.** static const — константа уровня модуля, как в C++. float2, float3, float4 — вектора из 2, 3 и 4 чисел. Координаты уже в NDC, поэтому треугольник займёт половину окна по каждой оси. Порядок вершин на экране — по часовой стрелке. Запомни это, пригодится для отсечения граней в главе 6.

• **Шаг 2.** Семантика SV_Position (SV — system value) говорит компилятору: это не просто данные, а позиция для растеризатора. Поле color без семантики Slang сам привяжет к первому свободному location. Растеризатор интерполирует его между вершинами, отсюда градиент.

• **Шаг 3.** [shader("vertex")] помечает точку входа. SV_VertexID — номер вершины в вызове draw: 0, 1, 2. Именно так мы рисуем без вершинных буферов. В float4(xy, 0.0, 1.0) четвёртая компонента w = 1 означает «перспективы нет». В главе 12 её начнёт менять матрица проекции.

• **Шаг 4.** Фрагментный шейдер получает ту же структуру, уже проинтерполированную для конкретного пикселя. SV_Target — «записать в цветовой буфер номер 0». Альфа 1.0 — полностью непрозрачный.

5.3. Компиляция в SPIR-V из CMake

SPIR-V — бинарный промежуточный язык из 32-битных слов. Драйвер не парсит текст шейдера, как это было в OpenGL. Плюс двойной: нет багов разных парсеров GLSL у разных производителей, и ошибки в шейдерах ловятся при сборке, а не у пользователя.

CMakeLists.txt — CMake

```cmake
# ── Шаг 5. Находим компилятор из SDK ──
find_program(SLANGC slangc HINTS "$ENV{VULKAN_SDK}/bin" REQUIRED)

set(PEPELAC_SHADER_DIR "${CMAKE_CURRENT_BINARY_DIR}/shaders")

# ── Шаг 6. Функция: .slang → .spv при каждой сборке ──
function(add_slang_shader target source)
    get_filename_component(name "${source}" NAME_WE)
    set(output "${PEPELAC_SHADER_DIR}/${name}.spv")
    add_custom_command(
        OUTPUT  "${output}"
        COMMAND "${CMAKE_COMMAND}" -E make_directory
          "${PEPELAC_SHADER_DIR}"
        COMMAND "${SLANGC}" "${CMAKE_CURRENT_SOURCE_DIR}/${source}"
                -target spirv -profile spirv_1_4 -emit-spirv-directly
                -fvk-use-entrypoint-name -matrix-layout-column-major
                "$<$<CONFIG:Debug>:-g>"
                -o "${output}"
        DEPENDS "${CMAKE_CURRENT_SOURCE_DIR}/${source}"
        COMMENT "Slang: ${source}"
        VERBATIM)
    target_sources(${target} PRIVATE "${output}")
endfunction()

add_slang_shader(pepelac shaders/triangle.slang)
target_compile_definitions(pepelac PRIVATE
  PEPELAC_SHADER_DIR="${PEPELAC_SHADER_DIR}")
```

Разбор.

• **Шаг 5.** find_program ищет slangc в SDK и в PATH. REQUIRED остановит конфигурацию с понятной ошибкой, если SDK не найден.

• **Шаг 6.** add_custom_command описывает, как получить файл OUTPUT и от чего он зависит (DEPENDS). CMake пересоберёт шейдер только когда изменится .slang. target_sources с .spv нужен, чтобы сборка pepelac требовала свежего шейдера. Иначе команда никогда не запустится.

• Флаги slangc: -target spirv — целевой формат; -profile spirv_1_4 — версия SPIR-V (Vulkan 1.3 понимает до 1.6); -emit-spirv-directly — генерировать SPIR-V напрямую, без промежуточного GLSL; -fvk-use-entrypoint-name — сохранить имена vertMain и fragMain, иначе обе точки входа стали бы main и подрались; -matrix-layout-column-major — матрицы в памяти по столбцам, как в GLM (глава 12); -g в debug — отладочная информация, и RenderDoc покажет исходник шейдера.

• "$<$<CONFIG:Debug>:-g>" — generator expression: подставляет -g только в Debug. Выглядит как проклятие, работает как тернарный оператор.

• PEPELAC_SHADER_DIR пробрасываем в C++ как дефайн с абсолютным путём. Для разработки этого достаточно. В релизе шейдеры кладут рядом с исполняемым файлом или вшивают в него.

5.4. Загрузка SPIR-V и shader module

src/shader_loader.cpp C++

```cpp
// src/shader_loader.cpp
#include <filesystem>
#include <format>
#include <fstream>
#include <span>
#include <stdexcept>
#include <vector>

// ── Шаг 7. Читаем файл целиком в вектор 32-битных слов ──
std::vector<uint32_t> readSpirv(const std::filesystem::path& path) {
    std::ifstream file(path, std::ios::binary | std::ios::ate);  // ate: сразу в конец
    if (!file) {
        throw std::runtime_error(std::format("Не открыть шейдер: {}", path.string()));
    }
    const std::streamsize size = file.tellg();  // позиция в конце = размер файла
    if (size <= 0 || size % sizeof(uint32_t) != 0) {
        throw std::runtime_error(std::format("{}: размер не кратен 4, это не SPIR-V", path.string()));
    }
    std::vector<uint32_t> code(static_cast<size_t>(size) / sizeof(uint32_t));
    file.seekg(0);
    file.read(reinterpret_cast<char*>(code.data()), size);  // char* можно алиасить с чем угодно
    if (!file) {
        throw std::runtime_error(std::format("Не дочитать шейдер: {}", path.string()));
    }
    constexpr uint32_t kSpirvMagic = 0x07230203;
    if (code.front() != kSpirvMagic) {
        throw std::runtime_error(std::format("{}: нет магического числа SPIR-V", path.string()));
    }
    return code;
}
```

src/shader_loader.cpp (продолжение) C++

```cpp

// ── Шаг 8. Упаковываем в VkShaderModule ──
vk::raii::ShaderModule Pepelac::createShaderModule(std::span<const
  uint32_t> code) const {
    return vk::raii::ShaderModule(device, vk::ShaderModuleCreateInfo{
        .codeSize = code.size_bytes(),  // В БАЙТАХ, хотя pCode —
          указатель на uint32_t
        .pCode    = code.data(),
    });
}
```

Разбор.

• **Шаг 7.** Флаг std::ios::ate открывает файл с курсором в конце, и tellg() сразу даёт размер. Потом seekg(0) и чтение всего целиком. Читаем именно в vector<uint32_t>, а не в vector<char>. Vulkan требует, чтобы pCode был выровнен на 4 байта. Буфер char такой гарантии не даёт, а reinterpret_cast<const uint32_t*> поверх него — ещё и нарушение правил алиасинга. В обратную сторону (uint32_t* → char*) стандарт каст разрешает.

• Магическое число 0x07230203 — первое слово любого SPIR-V. Проверка стоит одну строку и экономит час, когда кто-то подсунет пустой файл или HTML-страницу «404», скачанную вместо ассета.

• **Шаг 8.** codeSize задаётся в байтах, а pCode — указатель на uint32_t. Нелогично, но так исторически сложилось. span::size_bytes() спасает от классической ошибки code.size() вместо code.size() * 4.

• Shader module — не скомпилированная программа, а просто контейнер с SPIR-V. Настоящая компиляция в машинный код GPU происходит при создании пайплайна в главе 6. После этого модуль можно уничтожить, пайплайн работать продолжит. Возврат vk::raii::ShaderModule по значению бесплатен благодаря гарантированному copy elision из C++17.

ЗАПОМНИ. *Цепочка шейдера: .slang → (slangc, при сборке) → .spv → (ShaderModule, в рантайме) → (драйвер, при создании пайплайна) → машинный код GPU. Ошибки синтаксиса ловит первый этап, а фризы из-за компиляции рождает последний (глава 25).*

ГРАБЛИ. *Имя точки входа в пайплайне (pName) должно точно совпадать с именем в SPIR-V. Забыл -fvk-use-entrypoint-name, а в коде пишешь "vertMain" — валидация напишет, что точки входа нет. Без валидации драйвер может просто упасть.*

ПОТОКИ. *vkCreateShaderModule можно вызывать из любых потоков одновременно, device внешней синхронизации не требует. Чтение файлов с диска — идеальная работа для фонового потока. К этому вернёмся в главах 23 и 25.*

Шпаргалка главы 5

• Программируемые этапы: вершинный и фрагментный шейдеры. Растеризатор между ними интерполирует.

• NDC в Vulkan: Y вниз, Z от 0 до 1.

• Slang: [shader("vertex")], SV_Position, SV_VertexID, SV_Target.

• slangc … -target spirv -fvk-use-entrypoint-name — несколько точек входа в одном .spv.

- SPIR-V читаем в vector<uint32_t>: выравнивание на 4 байта.
- codeSize — в байтах.
- Shader module нужен только до создания пайплайна.

Задачи для разминки мозгов

Задача 5.1. «Квадрат из воздуха». Измени шейдер так, чтобы вызов draw(6, 1, 0, 0) рисовал квадрат из двух треугольников с четырьмя цветными углами. Без буферов и без массива из шести позиций: четыре угла и массив индексов. Оба треугольника должны идти по часовой стрелке.

Задача 5.2. «Паспорт шейдера». Второе слово SPIR-V — версия формата: байт старшей версии в битах 16–23, младшей — в битах 8–15. Доработай readSpirv, чтобы она проверяла минимальный размер заголовка (пять слов) и версию не выше 1.6. Почему версия выше 1.6 для нас ошибка?

Решения — в приложении А.

Глава 6. Graphics pipeline: запекаем всё заранее

В OpenGL ты менял состояние вызовами glEnable(GL_BLEND) где угодно, а драйвер на каждом draw call собирал из этого конфигурацию железа. В Vulkan всё состояние отрисовки собирается один раз в объект **pipeline**. Он компилируется заранее и больше не меняется. Хочешь другое смешивание — делай другой пайплайн. Звучит неудобно, но именно так исчезают загадочные фризы из главы 0.

6.1. Из чего состоит пайплайн

Блок	Что настраивает	У нас
Shader stages	какие шейдеры и точки входа	`vertMain` + `fragMain`
Vertex input	формат вершинных буферов	пусто (до главы 11)
Input assembly	как вершины собираются в примитивы	список треугольников
Viewport / scissor	область вывода и обрезки	динамические
Rasterization	заливка, отсечение граней	fill, cull back, по часовой
Multisample	сглаживание MSAA	1 сэмпл (выключено)
Depth/stencil	тест глубины	нет (до главы 14)
Color blend	смешивание с тем, что уже нарисовано	выключено
Dynamic state	что задаётся не в пайплайне, а командами	viewport, scissor
Layout	какие ресурсы видят шейдеры	пустой (до главы 12)
Rendering info	форматы картинок, в которые рисуем	формат swapchain

Про dynamic rendering. Раньше пайплайн создавался под конкретный VkRenderPass — громоздкий объект с подпроходами и зависимостями. В Vulkan 1.3 достаточно сообщить форматы картинок через PipelineRenderingCreateInfo. Минус два типа объектов и минус сотня строк.

6.2. Код

src/pepelac_pipeline.cpp C++

```cpp
// src/pepelac_pipeline.cpp
#include "pepelac.hpp"

#include <array>
#include <filesystem>

void Pepelac::createGraphicsPipeline() {
    // ── Шаг 1. Шейдеры: один модуль, две точки входа ──
    const std::vector<uint32_t> spirv =
        readSpirv(std::filesystem::path{PEPELAC_SHADER_DIR} /
          "triangle.spv");
    const vk::raii::ShaderModule module = createShaderModule(spirv);
       // умрёт в конце функции

    const std::array stages{
        vk::PipelineShaderStageCreateInfo{
            .stage = vk::ShaderStageFlagBits::eVertex, .module =
              *module, .pName = "vertMain" },
        vk::PipelineShaderStageCreateInfo{
            .stage = vk::ShaderStageFlagBits::eFragment, .module =
              *module, .pName = "fragMain" },
    };

    // ── Шаг 2. Вершины и примитивы ──
    const vk::PipelineVertexInputStateCreateInfo   vertexInput{};
      // вершины пока в шейдере
    const vk::PipelineInputAssemblyStateCreateInfo inputAssembly{
        .topology = vk::PrimitiveTopology::eTriangleList,
          // каждые 3 вершины = треугольник
    };

    // ── Шаг 3. Viewport и scissor зададим командами — здесь только
      их количество ──
    const vk::PipelineViewportStateCreateInfo viewportState{
      .viewportCount = 1, .scissorCount = 1 };

```

src/pepelac_pipeline.cpp (продолжение) C++

```cpp
    // ── Шаг 4. Растеризация ──
    const vk::PipelineRasterizationStateCreateInfo rasterizer{
        .depthClampEnable        = vk::False,
        .rasterizerDiscardEnable = vk::False,              // True = ничего не рисовать вообще
        .polygonMode             = vk::PolygonMode::eFill,
        .cullMode                = vk::CullModeFlagBits::eBack,
        .frontFace               = vk::FrontFace::eClockwise, // наш треугольник идёт по часовой
        .depthBiasEnable         = vk::False,
        .lineWidth               = 1.0f,                   // другие значения требуют фичу wideLines
    };

    // ── Шаг 5. Сглаживание выключено ──
    const vk::PipelineMultisampleStateCreateInfo multisampling{
        .rasterizationSamples = vk::SampleCountFlagBits::e1,
    };

    // ── Шаг 6. Смешивание: просто записываем цвет поверх ──
    const vk::PipelineColorBlendAttachmentState blendAttachment{
        .blendEnable    = vk::False,
        .colorWriteMask = vk::ColorComponentFlagBits::eR |
          vk::ColorComponentFlagBits::eG |
                          vk::ColorComponentFlagBits::eB |
                  vk::ColorComponentFlagBits::eA,
    };
    const vk::PipelineColorBlendStateCreateInfo colorBlend{
        .attachmentCount = 1,
        .pAttachments    = &blendAttachment,  // по одной структуре на каждый цветовой буфер
    };

    // ── Шаг 7. Что остаётся динамическим ──
    constexpr std::array dynamicStates{ vk::DynamicState::eViewport,
      vk::DynamicState::eScissor };
    const vk::PipelineDynamicStateCreateInfo dynamicState{
```

src/pepelac_pipeline.cpp (продолжение) C++

```cpp
        .dynamicStateCount = static_cast<uint32_t>(dynamicStates.size
          ()),
        .pDynamicStates    = dynamicStates.data(),
    };

    // ── Шаг 8. Layout: какие внешние ресурсы видят шейдеры (пока
      никакие) ──
    pipelineLayout = vk::raii::PipelineLayout(device,
      vk::PipelineLayoutCreateInfo{});

    // ── Шаг 9. Собираем всё вместе + формат для dynamic rendering
      ──
    const vk::StructureChain<vk::GraphicsPipelineCreateInfo,
      vk::PipelineRenderingCreateInfo> chain{
        vk::GraphicsPipelineCreateInfo{
            .stageCount          =
              static_cast<uint32_t>(stages.size()),
            .pStages             = stages.data(),
            .pVertexInputState   = &vertexInput,
            .pInputAssemblyState = &inputAssembly,
            .pViewportState      = &viewportState,
            .pRasterizationState = &rasterizer,
            .pMultisampleState   = &multisampling,
            .pColorBlendState    = &colorBlend,
            .pDynamicState       = &dynamicState,
            .layout              = *pipelineLayout,
            .renderPass          = nullptr,  // да, без render pass.
              Добро пожаловать в 1.3
        },
        vk::PipelineRenderingCreateInfo{
            .colorAttachmentCount    = 1,
            .pColorAttachmentFormats = &swapchainFormat,
        },
    };

    // ── Шаг 10. Компиляция. Именно здесь драйвер превращает SPIR-V
      в машинный код ──
    graphicsPipeline = vk::raii::Pipeline(device, nullptr,
      chain.get<vk::GraphicsPipelineCreateInfo>());
}
```

6.3. Разбор

Шаг 1. Один shader module и две стадии, отличающиеся stage и pName. Модуль — локальная переменная: после создания пайплайна он больше не нужен, и RAII уберёт его на выходе из функции. std::array stages{…} выводит тип и размер сам (CTAD), но только если тип первого элемента написан явно.

Шаг 2. Пустой vertexInput — «вершинных буферов нет», всё берётся из SV_VertexID. Топология eTriangleList: вершины 0-1-2 — первый треугольник, 3-4-5 — второй. Есть ещё strip, fan, линии и точки, но list — самый предсказуемый вариант.

Шаг 3. Viewport — прямоугольник, в который растягивается NDC. Scissor — прямоугольник, за пределами которого пиксели отбрасываются. Оба зависят от размера окна. Запечёшь их в пайплайн — при каждом ресайзе придётся пересоздавать пайплайн. Поэтому они динамические (шаг 7), а здесь указывается только их количество.

Шаг 4. Растеризатор:

• depthClampEnable — прижимать глубину к границам вместо отсечения. Нужно для теней, требует отдельной фичи.

• polygonMode = eFill — заливать треугольники. eLine (каркас) требует фичу fillModeNonSolid, смотри задачу 6.1.

• cullMode = eBack + frontFace = eClockwise — треугольники, идущие на экране по часовой, считаются лицевыми. Остальные отбрасываются до фрагментного шейдера. У замкнутой модели задние грани всё равно не видны, а экономия — почти половина работы.

• lineWidth = 1.0f — обязательное поле, даже если ты рисуешь треугольники. Оставишь 0 — получишь ошибку валидации.

Шаг 5. Один сэмпл на пиксель, то есть без MSAA. Вернёмся к этому в главе 14.

Шаг 6. Смешивание выключено: новый цвет просто заменяет старый. colorWriteMask — какие каналы вообще записывать. Классическая диверсия для новичка: забыть маску. По умолчанию там ноль, и ты получишь идеально работающий пайплайн, который ничего не рисует. Валидация про это не скажет: формально всё законно.

Шаг 7. Динамическое состояние — компромисс между «всё запечено» и «всё как в OpenGL». Viewport и scissor делают динамическими всегда, это ничего не стоит. Vulkan 1.3 позволяет сделать динамическим почти всё (cull mode, тест глубины, топологию). Это уменьшает число пайплайнов, подробнее в главе 25.

Шаг 8. Pipeline layout описывает «интерфейс» шейдеров к внешнему миру: наборы дескрипторов и push constants. У нас пока пусто, но создать layout всё равно надо. Это поле класса, а не локальная переменная: он понадобится при записи команд.

Шаг 9. Все структуры передаются по указателям, поэтому они обязаны жить до вызова на шаге 10. Локальные переменные функции — идеально. А вот адрес временного объекта, написанного прямо в инициализаторе, взять нельзя: компилятор не даст, и правильно сделает. PipelineRenderingCreateInfo в цепочке сообщает: «рисовать будем в одну картинку формата swapchainFormat». Этот формат обязан совпасть с форматом картинки в beginRendering (глава 7).

Шаг 10. vk::raii::Pipeline(device, nullptr, info) — второй аргумент это pipeline cache. Пока его нет, появится в главе 25. Именно здесь происходит дорогая работа: драйвер компилирует SPIR-V с учётом всего запечённого состояния. На сложном шейдере это десятки и сотни миллисекунд. Тысяча материалов — и игра грузится минуту (спойлер к главе 25).

ЗАПОМНИ. *Пайплайн неизменяем. Дорого его создать, дёшево переключить. Всё, что меняется каждый кадр или зависит от окна, делай динамическим состоянием. Данные (матрицы, цвета) идут в буферы и push constants, а не в новые пайплайны.*

ГРАБЛИ. *Треугольник не виден, валидация молчит? Три подозреваемых: нулевой colorWriteMask, неверный frontFace (треугольник отсёкся как задний), нулевой размер viewport. Временно поставь cullMode = eNone. Появился — значит, дело в порядке вершин.*

ПОТОКИ. *Неизменяемость — лучший друг многопоточности. Готовый пайплайн можно одновременно использовать в командных буферах из любого числа потоков без блокировок. А vkCreateGraphicsPipelines можно вызывать параллельно из разных потоков. На этом построена фоновая компиляция в главе 25.*

Шпаргалка главы 6

• Пайплайн = шейдеры + фиксированные состояния + layout + форматы картинок. Неизменяем.
• Viewport и scissor — всегда динамические.
• cullMode + frontFace должны согласовываться с порядком вершин.
• colorWriteMask = RGBA, иначе ничего не нарисуется.
• Dynamic rendering: PipelineRenderingCreateInfo в pNext, renderPass = nullptr.
• Shader module можно уничтожить сразу после создания пайплайна.

Задачи для разминки мозгов

Задача 6.1. «Рентген». Добавь второй пайплайн, который рисует каркас (ePolygonMode::eLine), и переключение между пайплайнами по клавише F1. Какую фичу устройства нужно включить и в каких двух местах кода из главы 3? Как избежать копипаста всей функции?

Задача 6.2. «Полупрозрачность». Настрой смешивание «поверх» (alpha blending): итог = новый × α + старый × (1 − α). Выпиши все поля PipelineColorBlendAttachmentState. Что поменять в шейдере, чтобы треугольник стал полупрозрачным? И почему для нескольких прозрачных объектов важен порядок отрисовки?

Решения — в приложении А.

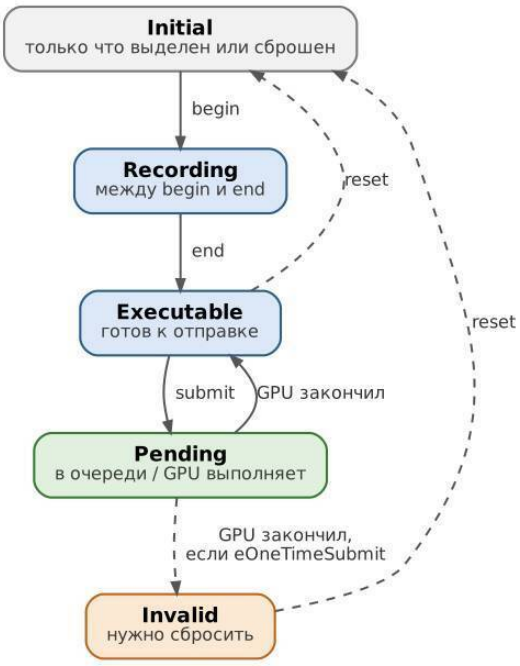
Глава 7. Командные буферы: список дел для GPU

Всё, что делает GPU, сначала записывается в командный буфер: очистка, отрисовка, копирование, барьеры. Запись — чистая работа CPU, GPU при этом ничего не делает. Именно поэтому запись можно разнести по потокам (глава 21). В этой главе мы запишем первый кадр целиком.

7.1. Пулы, буферы и их состояния

Командные буферы не создаются, а **выделяются из пула**. Пул — локальный аллокатор памяти для команд, привязанный к одному семейству очередей. Буфер, выделенный из graphics-пула, нельзя отправить в transfer-очередь.

У командного буфера есть жизненный цикл:

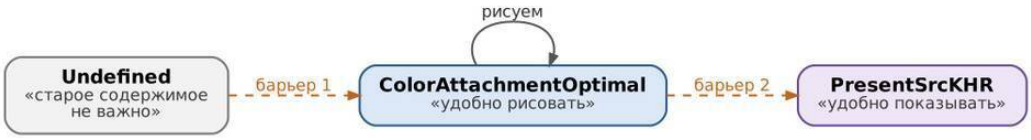


Самое опасное состояние — **Pending**: буфер отправлен и выполняется GPU. В это время его нельзя ни сбрасывать, ни перезаписывать, ни уничтожать. Ровно для этого в главе 0 стоял waitForFences.

7.2. Layout картинки за пять минут

GPU хранит пиксели картинки по-разному в зависимости от того, что с ней делают: сжатие для рендеринга, другая раскладка для чтения из шейдера, третья для показа на экране. Эта раскладка называется **layout**. Перед каждым способом использования картинку нужно перевести в подходящий layout. Перевод делает барьер — команда, которая заодно упорядочивает работу GPU.

Для нашего кадра нужны два перехода:



Всю теорию барьеров разберём в главе 15. Сейчас достаточно знать: барьер говорит «всё, что до меня, на стадии X должно закончиться, прежде чем всё, что после, начнётся на стадии Y».

7.3. Код

src/pepelac_commands.cpp C++

```cpp
// src/pepelac_commands.cpp
#include "pepelac.hpp"

namespace {

// ── Шаг 1. Параметры перехода в одной структуре — чтобы не путать 6 аргументов ──
struct ImageTransition {
    vk::ImageLayout         oldLayout;
    vk::ImageLayout         newLayout;
    vk::PipelineStageFlags2 srcStage;
    vk::AccessFlags2        srcAccess;
    vk::PipelineStageFlags2 dstStage;
    vk::AccessFlags2        dstAccess;
};

void transitionImage(const vk::raii::CommandBuffer& cmd, vk::Image image, const ImageTransition& t) {
    const vk::ImageMemoryBarrier2 barrier{
        .srcStageMask        = t.srcStage,
        .srcAccessMask       = t.srcAccess,
        .dstStageMask        = t.dstStage,
        .dstAccessMask       = t.dstAccess,
        .oldLayout           = t.oldLayout,
        .newLayout           = t.newLayout,
        .srcQueueFamilyIndex = vk::QueueFamilyIgnored,  // владелец не меняется (глава 23)
        .dstQueueFamilyIndex = vk::QueueFamilyIgnored,
        .image               = image,
        .subresourceRange    = {
            .aspectMask = vk::ImageAspectFlagBits::eColor,
            .baseMipLevel = 0, .levelCount = 1, .baseArrayLayer = 0, .layerCount = 1,
        },
    };
    cmd.pipelineBarrier2(vk::DependencyInfo{
      .imageMemoryBarrierCount = 1, .pImageMemoryBarriers = &barrier });
```

src/pepelac_commands.cpp (продолжение) C++

```cpp
}

}  // namespace

// ── Шаг 2. Пул и буферы ──
void Pepelac::createCommandPool() {
    commandPool = vk::raii::CommandPool(device,
      vk::CommandPoolCreateInfo{
        .flags            = vk::CommandPoolCreateFlagBits::eResetCommandBuffer,  // разрешить reset() поштучно
        .queueFamilyIndex = families.graphics,
    });
}

void Pepelac::createCommandBuffers() {
    commandBuffers = device.allocateCommandBuffers(vk::CommandBufferAllocateInfo{
        .commandPool        = *commandPool,
        .level              = vk::CommandBufferLevel::ePrimary,  // отправляется напрямую в очередь
        .commandBufferCount = MAX_FRAMES_IN_FLIGHT,
    });
}

// ── Шаг 3. Запись кадра ──
void Pepelac::recordCommandBuffer(vk::raii::CommandBuffer& cmd,
  uint32_t imageIndex) {
    cmd.begin(vk::CommandBufferBeginInfo{ .flags =
      vk::CommandBufferUsageFlagBits::eOneTimeSubmit });

    // ── Шаг 4. Барьер 1: готовим картинку к рисованию ──
    transitionImage(cmd, swapchainImages[imageIndex], {
        .oldLayout = vk::ImageLayout::eUndefined,                    // старое содержимое нам не нужно
        .newLayout = vk::ImageLayout::eColorAttachmentOptimal,
        .srcStage  = vk::PipelineStageFlagBits2::eColorAttachmentOutput,  // сцепка с ожиданием acquire
        .srcAccess = {},
```

src/pepelac_commands.cpp (продолжение) C++

```cpp
        .dstStage  = vk::PipelineStageFlagBits2::eColorAttachmentOut
          put,
        .dstAccess = vk::AccessFlagBits2::eColorAttachmentWrite,
    });

    // ── Шаг 5. Куда рисуем и что делать со старым содержимым ──
    const vk::ClearValue clearColor = vk::ClearColorValue(0.02f,
      0.02f, 0.05f, 1.0f);  // почти чёрный космос
    const vk::RenderingAttachmentInfo colorAttachment{
        .imageView   = *swapchainImageViews[imageIndex],
        .imageLayout = vk::ImageLayout::eColorAttachmentOptimal,
        .loadOp      = vk::AttachmentLoadOp::eClear,    // в начале
          залить clearColor
        .storeOp     = vk::AttachmentStoreOp::eStore,   // в конце
          сохранить результат
        .clearValue  = clearColor,
    };
    const vk::RenderingInfo renderingInfo{
        .renderArea           = { .offset = {0, 0}, .extent =
          swapchainExtent },
        .layerCount           = 1,
        .colorAttachmentCount = 1,
        .pColorAttachments    = &colorAttachment,
    };

    // ── Шаг 6. Рисуем ──
    cmd.beginRendering(renderingInfo);
    cmd.bindPipeline(vk::PipelineBindPoint::eGraphics,
      *graphicsPipeline);
    cmd.setViewport(0, vk::Viewport{
        .x = 0.0f, .y = 0.0f,
        .width  = static_cast<float>(swapchainExtent.width),
        .height = static_cast<float>(swapchainExtent.height),
        .minDepth = 0.0f, .maxDepth = 1.0f,
    });
    cmd.setScissor(0, vk::Rect2D{ .offset = {0, 0}, .extent =
      swapchainExtent });
```

src/pepelac_commands.cpp (продолжение) C++

```
    cmd.draw(3, 1, 0, 0);  // 3 вершины, 1 экземпляр, с вершины 0,
      с экземпляра 0
    cmd.endRendering();

    // — Шаг 7. Барьер 2: отдаём картинку системе показа —
    transitionImage(cmd, swapchainImages[imageIndex], {
        .oldLayout = vk::ImageLayout::eColorAttachmentOptimal,
        .newLayout = vk::ImageLayout::ePresentSrcKHR,
        .srcStage  = vk::PipelineStageFlagBits2::eColorAttachmentOut
          put,
        .srcAccess = vk::AccessFlagBits2::eColorAttachmentWrite,
        .dstStage  = vk::PipelineStageFlagBits2::eBottomOfPipe,  //
          дальше сторожит семафор
        .dstAccess = {},
    });

    cmd.end();
}
```

7.4. Разбор

Шаг 1. У барьера шесть параметров, и четыре из них попарно одного типа. Функция f(a, b, c, d, e, f) с такими аргументами — гарантированная путаница. Структура с designated initializers делает каждый вызов самодокументируемым. А перепутать .srcStage и .dstStage не даст компилятор: порядок полей обязан совпадать с объявлением.

ImageMemoryBarrier2 и pipelineBarrier2 — из synchronization2. В старом API стадии задавались аргументами функции сразу для всех барьеров. В новом они у каждого барьера свои, и это намного читабельнее. subresourceRange говорит, какую часть картинки переводим: у нас один mip и один слой. У текстур с mip-уровнями (глава 13) будет иначе.

Шаг 2. Флаг eResetCommandBuffer разрешает сбрасывать буферы по одному (cmd.reset() в drawFrame). Без него сбрасывается только весь пул целиком. В главе 22 мы перейдём именно к этому варианту, он быстрее. allocateCommandBuffers возвращает std::vector<vk::raii::CommandBuffer>. Каждый буфер помнит свой пул и вернётся в него в деструкторе. ePrimary — буфер, который отправляется напрямую в очередь. Про secondary поговорим в главе 21.

Шаг 3. eOneTimeSubmit обещает драйверу: буфер будет отправлен один раз и перезаписан. Драйвер может сделать из этого мелкие оптимизации. Мы и так перезаписываем буфер каждый кадр.

Шаг 4. Первый барьер. oldLayout = eUndefined говорит: старое содержимое можно выбросить, мы всё равно зальём картинку цветом. Это дешевле перевода из ePresentSrcKHR с сохранением данных. Тонкость со стадиями: srcStage = eColorAttachmentOutput совпадает со стадией, на которой submit ждёт семафор imageAvailable (глава 0, шаг 5). Ожидание семафора и барьер сцепляются в цепочку: переход layout случится после того, как система показа отдаст картинку. Поставь здесь eTopOfPipe — и переход может начаться раньше, чем картинка освободится.

Шаг 5. Dynamic rendering целиком умещается в две структуры:

• RenderingAttachmentInfo — одна картинка, в которую рисуем. loadOp решает, что делать со старым содержимым в начале: eClear (залить цветом), eLoad (сохранить) или eDontCare (неважно). storeOp решает, сохранять ли результат. На мобильных GPU с тайловой архитектурой правильные load/store экономят самое дорогое — пропускную способность памяти.

• vk::ClearColorValue(r, g, b, a) — объединение (union) для float/int/uint цветов. У объединений в Vulkan-Hpp конструкторы остаются, поэтому здесь круглые скобки. Цвет очистки задаётся в линейном пространстве: sRGB-формат сам сделает преобразование.

• RenderingInfo — весь «проход рендеринга»: область, число слоёв, список цветовых картинок, глубина (глава 14).

Шаг 6. Команды отрисовки. Порядок важен: bindPipeline до setViewport, если пайплайн заявил viewport динамическим. На практике динамическое состояние переживает смену пайплайна, но так надёжнее. setViewport(0, viewport) — первый аргумент задаёт номер первого viewport, второй — ArrayProxy, куда можно передать один объект или массив. draw(3, 1, 0, 0) запускает вершинный шейдер три раза с SV_VertexID = 0, 1, 2.

Шаг 7. Второй барьер: все записи цвета должны завершиться (srcAccess = eColorAttachmentWrite), а картинка — перейти в ePresentSrcKHR. В приёмнике никого нет: дальше порядок обеспечивает семафор renderFinished. Забудешь этот барьер — валидация скажет, что картинка показывается в неверном layout. На части железа получишь мусор, на части — всё будет «работать». До первого пользователя с другой видеокартой.

ЗАПОМНИ. *Запись команд ничего не выполняет. cmd.draw — это строчка в списке дел, а не отрисовка. Выполнение начнётся после submit, и до сигнала fence буфер нельзя трогать.*

ГРАБЛИ. *Формат картинки в beginRendering обязан совпадать с форматом в PipelineRenderingCreateInfo пайплайна. Поменялся формат swapchain при пересоздании (бывает при перетаскивании окна на HDR-монитор) — пайплайн тоже надо пересоздать.*

ПОТОКИ. *Командный пул* ***и все буферы, выделенные из него,*** *требуют внешней синхронизации. Два потока не могут одновременно записывать два разных буфера из одного пула: пул один, и аллокатор внутри него тоже один. Решение в главе 21 тривиально: по пулу на поток. А вот сама запись GPU не трогает, и в этом её сила.*

Шпаргалка главы 7

• Пул → буферы. Пул привязан к семейству очередей.

• Состояния: Initial → Recording → Executable → Pending. В Pending буфер не трогать.

• Кадр: барьер (Undefined → ColorAttachment) → beginRendering → pipeline, viewport, scissor, draw → endRendering → барьер (ColorAttachment → PresentSrc).

• loadOp: Clear / Load / DontCare. storeOp: Store / DontCare.

• srcStage первого барьера = стадия ожидания семафора acquire.

• Один пул = один поток за раз.

Задачи для разминки мозгов

Задача 7.1. «Дышащий космос». Сделай так, чтобы цвет очистки плавно пульсировал с периодом 3 секунды. Используй std::chrono::steady_clock. Почему для этого не нужно ни пересоздавать пайплайн, ни менять шейдер? И почему steady_clock, а не system_clock?

Задача 7.2. «Сплитскрин». Нарисуй треугольник дважды: в левой и в правой половинах окна, внутри одного beginRendering. Затем ответь: что увидишь, если viewport — левая половина, а scissor — левая четверть? А наоборот?

Решения — в приложении А.

Глава 8. Синхронизация I: первый живой кадр

Все детали на месте: instance, устройство, swapchain, пайплайн, команды. Не хватает объектов синхронизации и сборки всего вместе. В конце этой главы на экране появится треугольник. Около тысячи строк ради трёх вершин, да. Зато каждую из этих строк ты понимаешь.

8.1. Два инструмента синхронизации

	Fence	**Binary semaphore**
Кто сигналит	GPU, по завершении submit	GPU или система показа
Кто ждёт	CPU (`waitForFences`)	GPU (в `submit`/`present`)
Сброс	вручную, `resetFences`	автоматически при ожидании
Аналогия	SMS «посылка доставлена»	эстафетная палочка
Правило	нельзя отправлять с уже засигналенным fence	каждому сигналу ровно одно ожидание

Последняя строка таблицы важнее всего. Binary semaphore работает парами: один signal, один wait. Два signal подряд без wait между ними — ошибка. Wait без предшествующего signal — ошибка и вечное зависание. В части IV появятся timeline-семафоры без этих ограничений. Но для acquire и present нужны именно бинарные.

8.2. Код: объекты синхронизации

фрагмент.cpp C++

```cpp
void Pepelac::createSyncObjects() {
    imageAvailable.clear();
    renderFinished.clear();
    inFlight.clear();

    // ── Шаг 1. По комплекту на каждый кадр в полёте ──
    for (uint32_t i = 0; i < MAX_FRAMES_IN_FLIGHT; ++i) {
        imageAvailable.emplace_back(device,
          vk::SemaphoreCreateInfo{});
        inFlight.emplace_back(device, vk::FenceCreateInfo{
            .flags = vk::FenceCreateFlagBits::eSignaled,  // иначе
              первый кадр зависнет навсегда
        });
    }
    // ── Шаг 2. По семафору на каждую картинку swapchain (почему —
      глава 0, шаг 6) ──
    for (size_t i = 0; i < swapchainImages.size(); ++i) {
        renderFinished.emplace_back(device,
          vk::SemaphoreCreateInfo{});
    }
}
```

Разбор. clear() в начале пригодится в главе 9, когда функция будет вызываться повторно. emplace_back(device, info) создаёт RAII-объект прямо в векторе. У пустого SemaphoreCreateInfo{} заполнен только sType, а больше и не нужно: бинарный семафор — тип по умолчанию. Флаг eSignaled нужен, потому что первое, что делает drawFrame, — ждёт fence. Кадра до первого не было, и без флага ждать пришлось бы вечно.

8.3. Собираем всё вместе: pepelac.hpp

Вот заголовок рендерера целиком, со всем, что накопилось за главы 2–8. Дальше в книге будут показываться только новые поля.

src/pepelac.hpp C++

```cpp
// src/pepelac.hpp
#pragma once
#include <vulkan/vulkan_raii.hpp>  // ДО GLFW
#include <GLFW/glfw3.h>

#include <cstdint>
#include <filesystem>
#include <optional>
#include <span>
#include <vector>

struct QueueFamilies {
    uint32_t                graphics = 0;
    std::optional<uint32_t> asyncCompute;
    std::optional<uint32_t> transfer;
};

std::vector<uint32_t> readSpirv(const std::filesystem::path& path);

class Pepelac {
public:
    explicit Pepelac(GLFWwindow* window);
    ~Pepelac();
    Pepelac(const Pepelac&)            = delete;
    Pepelac& operator=(const Pepelac&) = delete;

    void drawFrame();

private:
    static constexpr uint32_t MAX_FRAMES_IN_FLIGHT = 2;

    void createInstance();
    void createSurface();
    void pickPhysicalDevice();
    void createLogicalDevice();
    void createSwapchain();
    void createImageViews();
    void createGraphicsPipeline();
```

src/pepelac.hpp (продолжение) C++

```cpp
    void createCommandPool();
    void createCommandBuffers();
    void createSyncObjects();
    void recordCommandBuffer(vk::raii::CommandBuffer& cmd, uint32_t
      imageIndex);
    [[nodiscard]] vk::raii::ShaderModule
      createShaderModule(std::span<const uint32_t> code) const;

    // ── Порядок полей = порядок создания. Уничтожение — в
      обратном. ──
    GLFWwindow*                          window = nullptr;  // не
      владеем: окно живёт в main

    vk::raii::Context                    context;
    vk::raii::Instance                   instance       = nullptr;
    vk::raii::DebugUtilsMessengerEXT     debugMessenger = nullptr;
    vk::raii::SurfaceKHR                 surface        = nullptr;
    vk::raii::PhysicalDevice             physicalDevice = nullptr;
    QueueFamilies                        families;
    vk::raii::Device                     device         = nullptr;
    vk::raii::Queue                      graphicsQueue  = nullptr;
    vk::raii::Queue                      computeQueue   = nullptr;
    vk::raii::Queue                      transferQueue  = nullptr;

    vk::raii::SwapchainKHR               swapchain      = nullptr;
    std::vector<vk::Image>               swapchainImages;
    vk::Format                           swapchainFormat =
      vk::Format::eUndefined;
    vk::Extent2D                         swapchainExtent;
    std::vector<vk::raii::ImageView>     swapchainImageViews;

    vk::raii::PipelineLayout             pipelineLayout   = nullptr;
    vk::raii::Pipeline                   graphicsPipeline = nullptr;

    vk::raii::CommandPool                commandPool      = nullptr;
    std::vector<vk::raii::CommandBuffer> commandBuffers;

```

src/pepelac.hpp (продолжение) C++

```
    std::vector<vk::raii::Semaphore>     imageAvailable;  //
      [MAX_FRAMES_IN_FLIGHT]
    std::vector<vk::raii::Semaphore>     renderFinished;  // [число
      картинок swapchain]
    std::vector<vk::raii::Fence>         inFlight;        //
      [MAX_FRAMES_IN_FLIGHT]
    uint32_t                             currentFrame = 0;
};
```

8.4. Конструктор и главный цикл

src/pepelac.cpp C++

```
// src/pepelac.cpp
#include "pepelac.hpp"

// ── Шаг 3. Инициализация — строго сверху вниз по карте объектов ──
Pepelac::Pepelac(GLFWwindow* window) : window{window} {
    createInstance();        // глава 2
    createSurface();         // глава 3
    pickPhysicalDevice();    // глава 3
    createLogicalDevice();   // глава 3
    createSwapchain();       // глава 4
    createImageViews();      // глава 4
    createGraphicsPipeline();// глава 6
    createCommandPool();     // глава 7
    createCommandBuffers();  // глава 7
    createSyncObjects();     // эта глава
}
// ~Pepelac() и drawFrame() — ровно как в главе 0.
```

src/main.cpp C++

```cpp
// src/main.cpp
#include "pepelac.hpp"
#include "window.hpp"

#include <chrono>
#include <cstdlib>
#include <format>
#include <print>

int main() {
    try {
        GlfwLibrary glfw;
        WindowPtr   window = createWindow(1280, 720, "Пепелац");
        Pepelac     renderer{window.get()};  // объявлен после окна
          → умрёт раньше него

        // ── Шаг 4. Счётчик FPS в заголовке окна ──
        using Clock = std::chrono::steady_clock;
        auto     lastReport = Clock::now();
        uint32_t frames     = 0;

        // ── Шаг 5. Главный цикл ──
        while (glfwWindowShouldClose(window.get()) != GLFW_TRUE) {
            glfwPollEvents();
            renderer.drawFrame();

            ++frames;
            const auto now = Clock::now();
            if (now - lastReport >= std::chrono::seconds{1}) {
                const double seconds =
                  std::chrono::duration<double>(now -
                  lastReport).count();
                const std::string title = std::format("Пепелац —
                  {:.0f} FPS", frames / seconds);
                glfwSetWindowTitle(window.get(), title.c_str());
                frames     = 0;
                lastReport = now;
            }
```

src/main.cpp (продолжение) C++

```
        }
    } catch (const std::exception& e) {
        std::println(stderr, "Фатальная ошибка: {}", e.what());
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

Разбор.

• **Шаг 3.** Порядок вызовов в конструкторе совпадает с порядком полей в классе. Это не обязано совпадать, но когда совпадает, сумасшедшего порядка уничтожения не будет. Что будет, если, скажем, createGraphicsPipeline бросит исключение? Тело конструктора прервётся, деструктор ~Pepelac **не вызовется**, а все уже созданные поля уничтожатся в обратном порядке. Никаких утечек, никаких if (initialized). Это и есть радость RAII.

• explicit Pepelac(GLFWwindow*) запрещает неявное превращение указателя на окно в рендерер. Иначе вызов функции, принимающей const Pepelac&, с аргументом-окном создал бы второй Vulkan-рендерер. Смешно, пока не случится.

• **Шаг 4.** steady_clock — монотонные часы, они не прыгают при синхронизации времени ОС. Для замеров только они. duration<double> даёт секунды с дробной частью. Заголовок обновляем раз в секунду, а не каждый кадр: glfwSetWindowTitle — системный вызов и стоит недёшево.

• **Шаг 5.** Цикл простой: события → кадр. Скорость цикла регулирует сам drawFrame: с FIFO он блокируется в acquireNextImage или на fence, когда CPU убегает вперёд. Никаких sleep не нужно.

• Деструктор renderer ждёт waitIdle и потом разбирает Vulkan снизу вверх. Затем умирает окно, затем GLFW. Порядок объявления в main делает всю работу.

8.5. Запуск

Собираем, запускаем и видим на тёмно-синем фоне треугольник с красной, зелёной и синей вершинами и плавным градиентом. Лог валидации пуст. В заголовке окна частота монитора (с FIFO) или тысячи кадров (с MAILBOX).

Если ничего не видно, пройди по списку:

1. Лог валидации пуст? Если нет — исправляй сверху вниз, первая ошибка часто порождает все остальные.

2. Виден фон, но нет треугольника? colorWriteMask, frontFace, имена точек входа.

3. Нет даже фона? Барьеры из главы 7 и семафоры в submit2.

4. Всё чёрное и программа висит? Fence создан без eSignaled или не передан в submit2.

5. Всё ещё непонятно? Запусти через RenderDoc и посмотри, что реально делает кадр (глава 28).

ЦИТАТА. *«Первый треугольник на Vulkan — как первый прыжок с парашютом: тысяча строк инструктажа ради трёх секунд восторга. Зато второй треугольник стоит одну строку». — автор книги*

ЗАПОМНИ. *Никогда не вызывай device.waitIdle() или queue.waitIdle() в каждом кадре «для надёжности». Это убивает параллельную работу CPU и GPU, ради которой придуманы*

кадры в полёте. waitIdle — инструмент для выхода из программы и пересоздания swapchain, а не для рабочего цикла.

ГРАБЛИ. *Попробуй сейчас изменить размер окна. В лучшем случае картинка растянется, в худшем программа упадёт с vk::OutOfDateKHRError. Swapchain был создан под старый размер, и чинить это мы будем в следующей главе.*

ПОТОКИ. *Это первая настоящая параллельная программа в книге, хотя поток у нас один. CPU и GPU — два независимых исполнителя, fence — точка их встречи, аналог join для потока. Правило «не трогай то, что читает другой исполнитель» для командного буфера — то же самое правило про data race, которое в главе 16 встретится для потоков CPU.*

Шпаргалка главы 8

• Fence: GPU → CPU, сброс вручную, создаём с eSignaled.

• Binary semaphore: GPU → GPU, строго пара signal/wait.

• imageAvailable и inFlight — по числу кадров в полёте; renderFinished — по числу картинок.

• Инициализация сверху вниз, исключение в середине безопасно благодаря RAII.

• waitIdle в каждом кадре — антипаттерн.

• Ничего не видно → валидация → пайплайн → барьеры → семафоры → RenderDoc.

Задачи для разминки мозгов

Задача 8.1. «Кто кого ждёт». Перед waitForFences проверь состояние fence без ожидания (getStatus()) и посчитай долю кадров, в которых CPU пришлось ждать GPU. Выводи её рядом с FPS. Что она говорит о том, во что упирается программа? Как на неё влияет FIFO?

Задача 8.2. «Для надёжности». Джун добавил graphicsQueue.waitIdle() в конец drawFrame, чтобы «точно ничего не сломалось». Нарисуй временную диаграмму CPU/GPU для CPU = 4 мс и GPU = 10 мс и посчитай FPS до и после «улучшения». Какие объекты синхронизации стали лишними и почему это всё равно плохая идея?

Решения — в приложении А.

Глава 9. Пересоздание swapchain: окно, которое можно трогать

Треугольник есть, но пользователь тянет за угол окна — и всё падает. Swapchain создавался под конкретный размер, и когда окно меняется, система показа считает его устаревшим. В этой главе научимся пересоздавать его аккуратно: без падений, без утечек и без вечно зависших fence.

9.1. Как Vulkan сообщает о проблеме

Сигнал	Откуда	Что значит	Что делать
`eErrorOutOfDateKHR` (в Vulkan-Hpp — исключение `vk::OutOfDateKHRError`)	acquire, present	swapchain больше нельзя использовать	пересоздать немедленно
`eSuboptimalKHR`	acquire, present	работает, но не идеально (например, растягивается)	дорисовать кадр, потом пересоздать
callback размера окна	GLFW	размер поменялся	пересоздать после present
размер 0×0	GLFW	окно свёрнуто	ждать, пока развернут

Последние две строки нужны потому, что не все драйверы и оконные системы надёжно сообщают out-of-date. И создать swapchain размером 0×0 нельзя в принципе.

9.2. Код: пересоздание

Сначала одно маленькое изменение в createSwapchain из главы 4:

фрагмент.cpp — C++

```
        .clipped          = vk::True,
        .oldSwapchain     = *swapchain,  // было nullptr; при первом
          создании там и так пустой хэндл
    };
    swapchain = vk::raii::SwapchainKHR(device, info);  // новый
      создаётся ДО уничтожения старого
```

Теперь само пересоздание:

src/pepelac_swapchain.cpp C++

```cpp
// src/pepelac_swapchain.cpp (продолжение)

// ── Шаг 1. Слушаем изменение размера окна (вызывается из конструктора Pepelac) ──
void Pepelac::installResizeCallback() {
    glfwSetWindowUserPointer(window, this);
    glfwSetFramebufferSizeCallback(window, [](GLFWwindow* w, int /*width*/, int /*height*/) {
        auto* self = static_cast<Pepelac*>(glfwGetWindowUserPointer(w));
        self->framebufferResized = true;  // только флажок! Никакого Vulkan внутри callback
    });
}

void Pepelac::recreateSwapchain() {
    // ── Шаг 2. Свёрнутое окно: спим, пока его не развернут ──
    int width = 0;
    int height = 0;
    glfwGetFramebufferSize(window, &width, &height);
    while (width == 0 || height == 0) {
        glfwWaitEvents();  // блокируется до следующего события, CPU не греется
        glfwGetFramebufferSize(window, &width, &height);
    }

    // ── Шаг 3. Ждём, пока GPU отпустит старые картинки ──
    device.waitIdle();

    // ── Шаг 4. Сносим всё, что смотрит на старые картинки ──
    swapchainImageViews.clear();
    const vk::Format oldFormat = swapchainFormat;

    // ── Шаг 5. Новый swapchain (старый передаётся как oldSwapchain) и новые view ──
    createSwapchain();
    createImageViews();

```

src/pepelac_swapchain.cpp (продолжение) C++

```
    // ── Шаг 6. Формат поменялся? Пайплайн запекал старый — пересобираем ──
    if (swapchainFormat != oldFormat) {
        createGraphicsPipeline();
    }

    // ── Шаг 7. Число картинок поменялось? Добиваем семафоры ──
    while (renderFinished.size() < swapchainImages.size()) {
        renderFinished.emplace_back(device, vk::SemaphoreCreateInfo{});
    }
}
```

И обновлённые шаги 2 и 6 из drawFrame:

фрагмент.cpp C++

```cpp
    // ── Шаг 2 (новая версия). acquire с обработкой устаревания ──
    uint32_t imageIndex = 0;
    try {
        auto [acquireResult, index] =
            swapchain.acquireNextImage(UINT64_MAX,
              *imageAvailable[currentFrame]);
        imageIndex = index;  // eSuboptimalKHR тоже сюда: кадр
          дорисуем, пересоздадим после present
    } catch (const vk::OutOfDateKHRError&) {
        recreateSwapchain();
        return;              // fence ещё НЕ сброшен — следующий
          кадр не зависнет
    }

    // ... шаги 3–5 без изменений ...

    // ── Шаг 6 (новая версия). present + решение о пересоздании ──
    bool needRecreate = framebufferResized;
    try {
        const vk::Result presentResult =
          graphicsQueue.presentKHR(presentInfo);
        needRecreate = needRecreate || presentResult ==
          vk::Result::eSuboptimalKHR;
    } catch (const vk::OutOfDateKHRError&) {
        needRecreate = true;
    }
    if (needRecreate) {
        framebufferResized = false;
        recreateSwapchain();
    }
```

В заголовок добавляются методы installResizeCallback(), recreateSwapchain() и поле bool framebufferResized = false;. installResizeCallback() вызывается в конце конструктора.

9.3. Разбор

Изменение в createSwapchain**.** Поле oldSwapchain подсказывает драйверу, что новый swapchain заменяет старый. Он может переиспользовать ресурсы и допоказать уже поставленные в очередь кадры. Самое тонкое место — строка swapchain = vk::raii::SwapchainKHR(device, info);. В C++ сначала вычисляется правая часть (новый объект), и только потом перемещающее присваивание уничтожает старый хэндл. В момент создания нового старый ещё жив, ровно как требует спецификация. RAII сделал правильный порядок бесплатно.

Шаг 1 — callback. GLFW — C-библиотека и не знает про наш класс. У неё есть версия «рюкзака»: glfwSetWindowUserPointer хранит произвольный указатель внутри окна. Callback — лямбда **без захвата**, только такая превращается в указатель на функцию. this она достаёт из «рюкзака» через static_cast. Хранить this во внешней библиотеке безопасно, потому что Pepelac

нельзя ни копировать, ни перемещать: адрес объекта не меняется за всю его жизнь. Внутри callback только ставим флажок. Callback вызывается изнутри glfwPollEvents, в середине чего угодно, и пересоздавать там swapchain — верный способ себе навредить.

Шаг 2 — свёрнутое окно. У свёрнутого окна фреймбуфер 0×0, а swapchain такого размера создать нельзя. glfwWaitEvents() усыпляет поток до первого события ОС, так что свёрнутая игра не ест батарею. В однопоточной программе это нормально. В части IV поток рендера будет просто пропускать кадры, потому что события окна живут в главном потоке.

Шаг 3 — waitIdle**.** Редкий случай, когда он уместен: ресайз случается не каждый кадр, а простота дороже пары миллисекунд. После него ни один командный буфер уже не ссылается на старые view.

Шаг 4 — сносим view. View смотрят на картинки, которыми владеет старый swapchain. Их нужно уничтожить до того, как старый swapchain умрёт и заберёт картинки с собой. vector::clear() вызовет деструктор каждого raii::ImageView. При этом иерархия из главы 0 соблюдена и внутри одной функции.

Шаги 5–6. Переиспользуем функции из главы 4 без изменений, кроме oldSwapchain. Пайплайн не зависит от размера окна: viewport и scissor мы сделали динамическими именно ради этого момента. А вот от формата картинки зависит, поэтому проверяем. Формат меняется редко, но перетаскивание окна на монитор с другим цветовым профилем — вполне реальный сценарий.

Шаг 7 — семафоры. Новый swapchain вправе дать другое число картинок, а renderFinished индексируется imageIndex. Мы только добавляем недостающие семафоры и не уничтожаем старые. waitIdle ждёт очереди GPU, но не систему показа. Семафор последнего present формально может быть ещё в работе, и уничтожать его опасно. Точное решение даёт расширение VK_KHR_swapchain_maintenance1 с fence для present. В Roadmap 2026 есть целый пункт про улучшения swapchain, так что на новом железе оно будет всё чаще.

Обновлённый шаг 2 drawFrame**.** acquireNextImage бросает vk::OutOfDateKHRError — наследника vk::SystemError. Ловим именно его, а не все исключения подряд: потерю устройства молча проглотить нельзя. И снова про порядок из главы 0: fence сбрасывается **после** успешного acquire. Благодаря этому ранний return безопасен: fence остался сигнальным, и следующий вызов drawFrame пройдёт шаг 1 без ожидания. Именно здесь порядок шагов из главы 0 оправдывает себя.

Обновлённый шаг 6. Suboptimal — не ошибка, кадр уже показан. Пересоздаём **после** present, чтобы не терять кадр и не оставлять полученную через acquire картинку непоказанной. Флаг framebufferResized обрабатывается в том же месте: сначала показать, потом перестраиваться.

ЗАПОМНИ. *Что зависит от swapchain, то и пересоздаём: image views всегда, семафоры при смене числа картинок, пайплайн при смене формата. В главе 14 в этот список добавится буфер глубины: он должен совпадать по размеру с окном.*

ГРАБЛИ. *Положиться только на eErrorOutOfDateKHR нельзя: на части драйверов он не приходит при ресайзе, и картинка просто растягивается. Положиться только на callback тоже нельзя: out-of-date может прийти и без ресайза, например при переходе в полноэкранный режим. Нужны оба механизма.*

ПОТОКИ. *Пересоздание swapchain — точка глобальной синхронизации. Во-первых, device.waitIdle() требует, чтобы* ***ни один другой поток*** *не отправлял работу в любую очередь в этот момент. Во-вторых, ни один поток не должен записывать команды со старыми view. В части IV это решается так: пересоздание делается на границе кадров, когда все рабочие потоки завершили запись.*

Шпаргалка главы 9

• Сигналы: OutOfDateKHRError (исключение), eSuboptimalKHR, callback размера, размер 0×0.

• Порядок: дождаться разворачивания → waitIdle → снести views → новый swapchain с oldSwapchain → views → (пайплайн, семафоры при необходимости).

• swapchain = vk::raii::SwapchainKHR(...) — новый рождается до смерти старого.

• Suboptimal — пересоздать после present.

• Callback окна только ставит флажок.

• Fence сбрасываем после acquire, иначе ранний return завешивает программу.

Задачи для разминки мозгов

Задача 9.1. «На весь экран». Добавь переключение полноэкранного режима по F11 через glfwSetWindowMonitor. Что именно запустит пересоздание swapchain? Почему нельзя вызвать recreateSwapchain() прямо из callback клавиатуры? Как запомнить и восстановить положение и размер окна?

Задача 9.2. «Бережливый пересоздатель». Составь таблицу: какие объекты «Пепелаца» (с учётом будущего буфера глубины из главы 14) зависят от размера swapchain, какие — от формата, какие — от числа картинок. Перепиши recreateSwapchain так, чтобы она сравнивала старые и новые параметры и пересоздавала ровно то, что нужно.

Решения — в приложении А.

Часть II. Ресурсы и данные

Шесть глав о том, как данные попадают в видеопамять и как шейдеры их находят. В конце — главная глава книги, про барьеры.

Глава 10. Память GPU: куча, но не та, о которой вы думаете

В C++ ты пишешь new, и память просто появляется. В Vulkan ты сначала выбираешь, **какая** память нужна: быстрая для GPU, доступная с CPU или какой-то гибрид. Потом выделяешь её большими кусками, потому что число выделений ограничено. Потом сам нарезаешь их под буферы с учётом выравнивания. Писать такой аллокатор самому — отличный способ потратить месяц. Поэтому разберёмся в механике и возьмём VMA.

10.1. Кучи и типы памяти

У видеокарты есть несколько **куч** (heaps) — физических пулов памяти, и несколько **типов** памяти — способов использовать эти кучи. Типичная дискретная карта:

Тип	Флаги	Где физически	Для чего
Device local	`DEVICE_LOCAL`	видеопамять	вершины, текстуры, всё, что GPU читает постоянно
Upload	`HOST_VISIBLE | HOST_COHERENT`	оперативная память, видна GPU через PCIe	staging-буферы, uniform-данные кадра
Readback	`HOST_VISIBLE | HOST_CACHED`	оперативная память с кэшем CPU	чтение результатов GPU на CPU
ReBAR	`DEVICE_LOCAL | HOST_VISIBLE`	видеопамять, доступная CPU	прямая запись с CPU без копирования

У встроенной графики память общая, и почти всё сразу DEVICE_LOCAL | HOST_VISIBLE. Вывод: захардкодить номер типа нельзя, его ищут по флагам на каждой машине.

Пара слов про флаги. HOST_COHERENT означает, что запись CPU становится видна GPU без ручного vkFlushMappedMemoryRanges. HOST_CACHED делает чтение с CPU быстрым. Без него чтение идёт мимо кэша и медленнее в десятки раз.

10.2. Как выделить буфер вручную (и почему не надо)

фрагмент.cpp C++

```cpp
// Только для понимания. В «Пепелаце» этого кода не будет.
uint32_t findMemoryType(const vk::raii::PhysicalDevice& gpu,
  uint32_t allowedTypes,
                        vk::MemoryPropertyFlags wanted) {
    const vk::PhysicalDeviceMemoryProperties props =
      gpu.getMemoryProperties();
    for (uint32_t i = 0; i < props.memoryTypeCount; ++i) {
        const bool allowed = (allowedTypes & (1u << i)) != 0;
            // бит i = тип i подходит ресурсу
        const bool hasAll  = (props.memoryTypes[i].propertyFlags &
          wanted) == wanted;
        if (allowed && hasAll) return i;
    }
    throw std::runtime_error("Нет подходящего типа памяти");
}

// 1) создать буфер (только описание, памяти нет)
vk::raii::Buffer buffer(device, vk::BufferCreateInfo{
    .size = 1024, .usage = vk::BufferUsageFlagBits::eVertexBuffer });
// 2) спросить, сколько и какой памяти ему нужно
const vk::MemoryRequirements req = buffer.getMemoryRequirements();
  // size, alignment, memoryTypeBits
// 3) выделить память подходящего типа
vk::raii::DeviceMemory memory(device, vk::MemoryAllocateInfo{
    .allocationSize  = req.size,
    .memoryTypeIndex = findMemoryType(physicalDevice,
      req.memoryTypeBits,

                  vk::MemoryPropertyFlagBits::eDeviceLocal) });
// 4) привязать
buffer.bindMemory(*memory, 0);
```

Четыре шага, и всё работает. Проблема в шаге 3: **лимит** maxMemoryAllocationCount на многих драйверах равен 4096. Выделишь память под каждую из десяти тысяч моделей — и упрёшься в лимит. Да и каждое выделение — дорогой системный вызов. Правильно выделять блоки по 64–256 МБ и нарезать их самому, учитывая alignment и совместимость типов. Это ровно то, что делает VMA.

10.3. VMA в RAII-обёртке

VMA — сишная библиотека в одном заголовке от AMD, стандарт де-факто. Подключение: find_package(VulkanMemoryAllocator CONFIG REQUIRED) и GPUOpen::VulkanMemoryAllocator в target_link_libraries. Реализация должна оказаться ровно в одном .cpp:

src/vma_impl.cpp C++

```cpp
// src/vma_impl.cpp — единственное место, где компилируется тело VMA
#define VMA_IMPLEMENTATION
#include <vk_mem_alloc.h>
```

src/gpu_memory.hpp C++

```cpp
// src/gpu_memory.hpp
#pragma once
#include <vulkan/vulkan_raii.hpp>
#include <vk_mem_alloc.h>

#include <cstddef>
#include <span>
#include <stdexcept>
#include <utility>

// ── Шаг 1. Аллокатор: один на устройство ──
class GpuAllocator {
public:
    GpuAllocator() = default;
    GpuAllocator(const vk::raii::Instance& instance, const
      vk::raii::PhysicalDevice& gpu,
                 const vk::raii::Device& device) {
        const VmaAllocatorCreateInfo info{
            .flags            = 0,  // по умолчанию VMA
              потокобезопасен (внутри мьютекс)
            .physicalDevice   = static_cast<VkPhysicalDevice>(*gpu),
            .device           = static_cast<VkDevice>(*device),
            .instance         = static_cast<VkInstance>(*instance),
            .vulkanApiVersion = VK_API_VERSION_1_3,
        };
        if (vmaCreateAllocator(&info, &handle_) != VK_SUCCESS) {
            throw std::runtime_error("vmaCreateAllocator не
              удался");
        }
    }
    ~GpuAllocator() { reset(); }

    GpuAllocator(GpuAllocator&& other) noexcept :
      handle_{std::exchange(other.handle_, nullptr)} {}
    GpuAllocator& operator=(GpuAllocator&& other) noexcept {
        if (this != &other) {
            reset();
            handle_ = std::exchange(other.handle_, nullptr);
```

src/gpu_memory.hpp (продолжение) C++

```
        }
        return *this;
    }
    GpuAllocator(const GpuAllocator&)            = delete;
    GpuAllocator& operator=(const GpuAllocator&) = delete;

    [[nodiscard]] VmaAllocator get() const noexcept { return
      handle_; }

private:
    void reset() noexcept {
        if (handle_ != nullptr) {
            vmaDestroyAllocator(handle_);
            handle_ = nullptr;
        }
    }
    VmaAllocator handle_ = nullptr;
};

// ── Шаг 2. Буфер вместе со своей памятью ──
class GpuBuffer {
public:
    GpuBuffer() = default;
    GpuBuffer(VmaAllocator allocator, vk::DeviceSize size,
      vk::BufferUsageFlags usage,
              VmaAllocationCreateFlags flags)
        : allocator_{allocator}, size_{size} {
        const VkBufferCreateInfo bufferInfo{
            .sType       = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO,
            .size        = size,
            .usage       = static_cast<VkBufferUsageFlags>(usage),
            .sharingMode = VK_SHARING_MODE_EXCLUSIVE,
        };
        const VmaAllocationCreateInfo allocInfo{
            .flags = flags,
            .usage = VMA_MEMORY_USAGE_AUTO,  // VMA сам выберет тип
              по usage и flags
        };
```

src/gpu_memory.hpp (продолжение) C++

```cpp
        VkBuffer          raw = VK_NULL_HANDLE;
        VmaAllocationInfo info{};
        if (vmaCreateBuffer(allocator, &bufferInfo, &allocInfo,
          &raw, &allocation_, &info) != VK_SUCCESS) {
            throw std::runtime_error("vmaCreateBuffer не удался");
        }
        buffer_ = vk::Buffer{raw};
        mapped_ = info.pMappedData;  // не nullptr, только если
          просили MAPPED
    }
    ~GpuBuffer() { reset(); }

    GpuBuffer(GpuBuffer&& o) noexcept
        : allocator_{std::exchange(o.allocator_, nullptr)},
          buffer_{std::exchange(o.buffer_, {})},
          allocation_{std::exchange(o.allocation_, nullptr)},
            mapped_{std::exchange(o.mapped_, nullptr)},
          size_{std::exchange(o.size_, 0)} {}
    GpuBuffer& operator=(GpuBuffer&& o) noexcept {
        if (this != &o) {
            reset();
            allocator_  = std::exchange(o.allocator_, nullptr);
            buffer_     = std::exchange(o.buffer_, {});
            allocation_ = std::exchange(o.allocation_, nullptr);
            mapped_     = std::exchange(o.mapped_, nullptr);
            size_       = std::exchange(o.size_, 0);
        }
        return *this;
    }
    GpuBuffer(const GpuBuffer&)            = delete;
    GpuBuffer& operator=(const GpuBuffer&) = delete;

    // ── Шаг 3. Запись данных с CPU (только для host-visible
      буферов) ──
    void upload(std::span<const std::byte> data, vk::DeviceSize
      offset = 0) {
        if (offset + data.size() > size_) {
```

src/gpu_memory.hpp (продолжение) C++

```
            throw std::out_of_range("GpuBuffer::upload: данные не
              лезут в буфер");
        }
        // Сам сделает map + memcpy + flush (для не-coherent
          памяти) + unmap.
        if (vmaCopyMemoryToAllocation(allocator_, data.data(),
          allocation_, offset, data.size()) != VK_SUCCESS) {
            throw std::runtime_error("vmaCopyMemoryToAllocation не
              удался");
        }
    }

    [[nodiscard]] vk::Buffer     get() const noexcept    { return
      buffer_; }
    [[nodiscard]] void*          mapped() const noexcept { return
      mapped_; }
    [[nodiscard]] vk::DeviceSize size() const noexcept   { return
      size_; }

private:
    void reset() noexcept {
        if (allocation_ != nullptr) {
            vmaDestroyBuffer(allocator_,
              static_cast<VkBuffer>(buffer_), allocation_);
            allocation_ = nullptr;
            buffer_     = vk::Buffer{};
            mapped_     = nullptr;
        }
    }
    VmaAllocator   allocator_  = nullptr;
    vk::Buffer     buffer_;
    VmaAllocation  allocation_ = nullptr;
    void*          mapped_     = nullptr;
    vk::DeviceSize size_       = 0;
};
```

Разбор.

• **Шаг 1.** VmaAllocatorCreateInfo — C-структура, но designated initializers работают и для неё. VMA живёт в мире C, поэтому хэндлы Vulkan-Hpp приводим к C-типам через static_cast. Функции Vulkan VMA берёт из загрузчика, с которым мы слинковались через Vulkan::Vulkan в главе 1. Класс перемещаемый, но не копируемый — ровно по шаблону Unique из главы 1.

• **Шаг 2.** vmaCreateBuffer делает все четыре шага из 10.2 сразу и берёт память из своих больших блоков. VMA_MEMORY_USAGE_AUTO — рекомендуемый способ: вместо того чтобы указывать флаги памяти, ты описываешь намерение. Флагов в аллокации нет — значит, CPU туда не пишет, и VMA выберет видеопамять. Флаг VMA_ALLOCATION_CREATE_HOST_ACCESS_SEQUENTIAL_WRITE_BIT — «CPU будет писать последовательно», VMA выберет память для загрузки.

VMA_ALLOCATION_CREATE_MAPPED_BIT — держать память постоянно отображённой, указатель будет в pMappedData.

• Перемещение через std::exchange обнуляет каждое поле источника. Иначе деструктор источника освободит память, которую теперь считает своей новый владелец. И добро пожаловать в double free.

• **Шаг 3.** std::span<const std::byte> — универсальный «кусок сырых байтов». Любой массив простых структур превращается в него через std::as_bytes(std::span{vec}). Проверка границ до копирования стоит одну строку. Без неё переполнение буфера испортит соседние данные в видеопамяти, и никакой ASan этого не увидит. vmaCopyMemoryToAllocation сам сделает flush, если VMA выбрал некогерентную память. Ручной memcpy в mapped() этого не делает, там нужен vmaFlushAllocation.

10.4. Staging: как данные попадают в видеопамять

CPU не может писать в обычную видеопамять. Поэтому данные едут в два прыжка:



src/pepelac_memory.cpp C++

```cpp
// src/pepelac_memory.cpp
#include "pepelac.hpp"

// ── Шаг 4. Синхронная отправка одноразовой работы (для загрузки) ──
void Pepelac::immediateSubmit(const std::function<void(const
  vk::raii::CommandBuffer&)>& record) {
    std::vector<vk::raii::CommandBuffer> buffers =
      device.allocateCommandBuffers({
        .commandPool = *commandPool, .level =
          vk::CommandBufferLevel::ePrimary, .commandBufferCount = 1
          });
    const vk::raii::CommandBuffer& cmd = buffers.front();

    cmd.begin({ .flags = vk::CommandBufferUsageFlagBits::eOneTimeSubm
      it });
    record(cmd);
    cmd.end();

    const vk::raii::Fence fence(device, vk::FenceCreateInfo{});  //
      без eSignaled — ждём настоящий сигнал
    const vk::CommandBufferSubmitInfo cmdInfo{ .commandBuffer = *cmd
      };
    graphicsQueue.submit2(vk::SubmitInfo2{ .commandBufferInfoCount =
      1, .pCommandBufferInfos = &cmdInfo }, *fence);
    if (device.waitForFences(*fence, vk::True, UINT64_MAX) !=
      vk::Result::eSuccess) {
        throw std::runtime_error("immediateSubmit: GPU не
          дождались");
    }
}   // здесь умирают fence и командный буфер — GPU их уже не
  использует

// ── Шаг 5. Загрузка в видеопамять через staging ──
GpuBuffer Pepelac::createDeviceLocalBuffer(std::span<const
  std::byte> data, vk::BufferUsageFlags usage) {
    GpuBuffer staging(allocator.get(), data.size(),
      vk::BufferUsageFlagBits::eTransferSrc,
```

src/pepelac_memory.cpp (продолжение) C++

```
                        VMA_ALLOCATION_CREATE_HOST_ACCESS_SEQUENTIAL_WR
                    ITE_BIT);
    staging.upload(data);

    GpuBuffer result(allocator.get(), data.size(), usage |
      vk::BufferUsageFlagBits::eTransferDst, 0);
    immediateSubmit([&](const vk::raii::CommandBuffer& cmd) {
        cmd.copyBuffer(staging.get(), result.get(),
                       vk::BufferCopy{ .srcOffset = 0, .dstOffset =
                  0, .size = data.size() });
    });
    return result;  // staging умирает здесь: копирование уже
      завершено
}
```

Разбор.

• **Шаг 4.** immediateSubmit — «сделай на GPU и дождись». Для загрузки при старте простота важнее скорости. В главе 23 мы заменим его асинхронным загрузчиком в отдельном потоке. std::function принимает любую лямбду с захватом. Его цена (возможная аллокация) тут ничтожна на фоне ожидания GPU. Fence создаётся **без** eSignaled: теперь нам нужен честный сигнал от GPU. Порядок уничтожения снова работает на нас: fence объявлен после буфера и умирает первым, но к этому моменту он уже засигнален.

• **Шаг 5.** Классический staging. Обрати внимание на флаги usage: у staging — eTransferSrc (отсюда копируют), у цели — eTransferDst (сюда копируют) плюс настоящее назначение. Забудешь — валидация скажет всё, что о тебе думает. Лямбда захватывает всё по ссылке ([&]). Это безопасно, потому что она выполняется синхронно, пока staging и result живы. return result; не копирует: сработает NRVO или перемещение.

• В Pepelac появляется поле GpuAllocator allocator; сразу после очередей. Его создаёт createLogicalDevice последней строкой: allocator = GpuAllocator(instance, physicalDevice, device);. Все GpuBuffer объявляются ниже и умрут раньше аллокатора, а аллокатор — раньше устройства.

ЗАПОМНИ. *Никогда не выделяй VkDeviceMemory на каждый ресурс: лимит всего около 4096 выделений. Намерения для VMA: «только GPU» = без флагов; «CPU пишет» = HOST_ACCESS_SEQUENTIAL_WRITE; «CPU читает» = HOST_ACCESS_RANDOM; «постоянно отображённая» = + MAPPED.*

ГРАБЛИ. *Деструктор GpuBuffer освобождает память сразу и не спрашивает, закончил ли GPU. В createDeviceLocalBuffer это безопасно благодаря waitForFences. Но стоит уничтожить буфер посреди работы, пока его читает кадр в полёте, — и получишь use-after-free на GPU. Решение (отложенное удаление) — в главе 22.*

ПОТОКИ. *VMA по умолчанию потокобезопасен: внутри него мьютекс, и создавать буферы можно из любых потоков. Флаг VMA_ALLOCATOR_CREATE_EXTERNALLY_SYNCHRONIZED_BIT убирает мьютекс для однопоточных приложений, но мы его не трогаем: в части IV аллокации пойдут из потока-загрузчика. А вот immediateSubmit пока однопоточный: он берёт командный пул и графическую очередь, а оба требуют внешней синхронизации.*

Шпаргалка главы 10

• Куча = физическая память, тип = способ её использовать. Тип ищем по флагам и memoryTypeBits.

• Ручной путь: buffer → requirements → allocate → bind. Лимит выделений ~4096.

• VMA: один аллокатор, vmaCreateBuffer, VMA_MEMORY_USAGE_AUTO + флаги-намерения.

• Staging: host-visible TransferSrc → copyBuffer → device-local TransferDst.

• vmaCopyMemoryToAllocation делает flush сам. Ручной memcpy — не делает.

• Порядок полей: device → allocator → буферы.

Задачи для разминки мозгов

Задача 10.1. «Типобезопасная загрузка». Напиши шаблон createDeviceLocalBuffer(std::span<const T> items, usage), который принимает массив любых структур и сам превращает его в байты. Ограничь шаблон концептом. Почему без ограничения вызов с std::vector<std::string> скомпилируется и что тогда окажется в видеопамяти?

Задача 10.2. «ReBAR — и без пересадок». На картах с Resizable BAR и на встроенной графике видеопамять видна CPU, и staging не нужен. Используй флаг VMA_ALLOCATION_CREATE_HOST_ACCESS_ALLOW_TRANSFER_INSTEAD_BIT и функцию vmaGetAllocationMemoryProperties, чтобы во время выполнения решать: копировать напрямую или через staging.

Решения — в приложении А.

Глава 11. Вершинные и индексные буферы

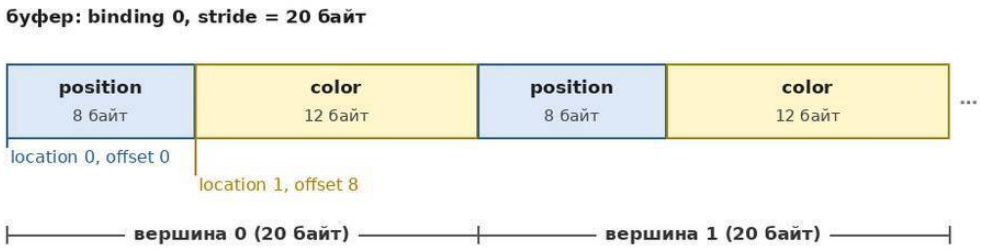
Хардкод вершин в шейдере годится для треугольника и не годится ни для чего больше. В этой главе геометрия переезжает в буферы, а наш треугольник превращается в квадрат из четырёх вершин и шести индексов. Скромно. Но именно так устроена любая модель из миллиона треугольников.

11.1. Как пайплайн читает вершины

Пайплайну нужно знать две вещи:

• **Binding** — откуда читать: номер буфера, шаг между вершинами (stride) и частота смены (на каждую вершину или на каждый экземпляр).

• **Attribute** — что читать: номер входа шейдера (location), формат данных и смещение внутри вершины.



Индексный буфер — массив номеров вершин. У квадрата четыре угла, но два треугольника требуют шесть вершин. Индексы позволяют хранить каждую вершину один раз. В настоящей модели одна вершина в среднем входит в шесть треугольников, и GPU кэширует результат вершинного шейдера по индексу. Экономия и памяти, и вычислений.

11.2. Код: вершина и её описание

src/vertex.hpp C++

```cpp
// src/vertex.hpp
#pragma once
#include <vulkan/vulkan.hpp>
#include <glm/glm.hpp>

#include <array>
#include <cstddef>
#include <type_traits>

// ── Шаг 1. Вершина как она лежит в памяти ──
struct Vertex {
    glm::vec2 position;
    glm::vec3 color;

    // ── Шаг 2. Откуда читать ──
    static constexpr vk::VertexInputBindingDescription binding() {
        return { .binding = 0, .stride = sizeof(Vertex), .inputRate
          = vk::VertexInputRate::eVertex };
    }

    // ── Шаг 3. Что читать ──
    static constexpr std::array<vk::VertexInputAttributeDescription,
      2> attributes() {
        return {{
            { .location = 0, .binding = 0, .format =
              vk::Format::eR32G32Sfloat,
              .offset = offsetof(Vertex, position) },
            { .location = 1, .binding = 0, .format =
              vk::Format::eR32G32B32Sfloat,
              .offset = offsetof(Vertex, color) },
        }};
    }
};

// ── Шаг 4. Страховка от «а давай добавим в Vertex виртуальный
  метод» ──
static_assert(std::is_trivially_copyable_v<Vertex>, "Vertex
  копируется в GPU побайтно");
static_assert(sizeof(Vertex) == 20, "Размер вершины должен совпадать
  с ожиданием шейдера");
```

Разбор.

• **Шаг 1.** Обычная структура из GLM-векторов. glm::vec2 — два float, 8 байт; glm::vec3 — три float, 12 байт. Выравнивание у обоих 4 байта, поэтому дыр нет и размер ровно 20.

• **Шаг 2.** stride = sizeof(Vertex) — на сколько байт сдвигаться к следующей вершине. eVertex означает «новая запись на каждую вершину», eInstance — «на каждый экзем-

пляр» (задача 11.1). Функция constexpr: структуры Vulkan-Hpp можно строить на этапе компиляции.

• **Шаг 3.** location должен совпасть с [[vk::location(N)]] в шейдере. Форматы для вершин называются как форматы картинок, и это сбивает с толку: eR32G32Sfloat — просто «два 32-битных float», никакого красного и зелёного. offsetof(Vertex, color) вычисляет смещение поля в байтах (8). Поменяешь порядок полей — смещение пересчитается само. Двойные фигурные скобки {{ … }} нужны потому, что std::array — структура со встроенным сишным массивом.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «Литрес».

Прочитайте эту книгу целиком, купив полную легальную версию на Литрес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.