3D ГРАФИКА

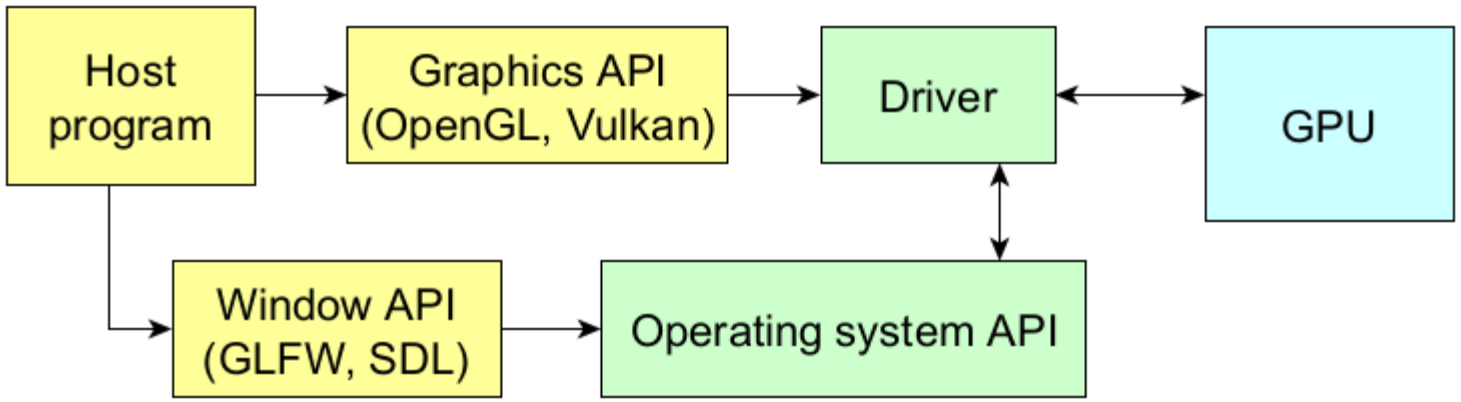
Программирование GPU, основы Vulkan API и подход к трёхмерной графике как к объектно-ориентированной системе

К. Владимиров, Intel, 2022
mail-to: konstantin.vladimirov@gmail.com

- GPU и OpenGL
- Логическая модель
- Vulkan API
- Физическая и объектная модель

GPU как вычислительная система

- Видеокарта решает задачу **рендеринга** т.е. двумерного представления трёхмерной сцены.
- Эта задача сложна и специфична. Графические процессоры всегда отличались от CPU и с ними традиционно работают через разные API.



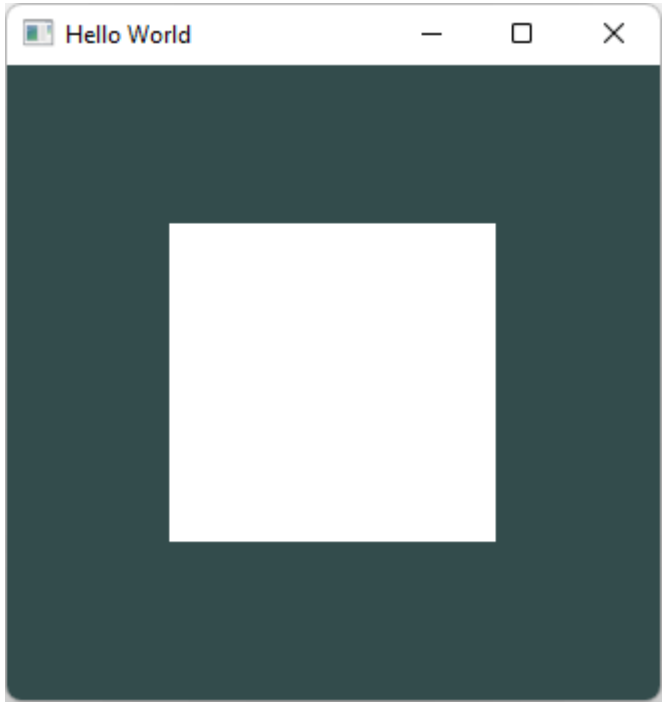
Давайте отрендерим квадрат

- OpenGL для рендеринга.

```
glClear(GL_COLOR_BUFFER_BIT);
glBegin(GL_QUADS);
glColor3f(1.0, 1.0, 1.0);
for (auto Coord : Vertices)
  glVertex3fv(Coord);
glEnd();
```

- GLFW для окон и управления.

```
glfwSwapBuffers(Wnd.get());
glfwPollEvents();
```



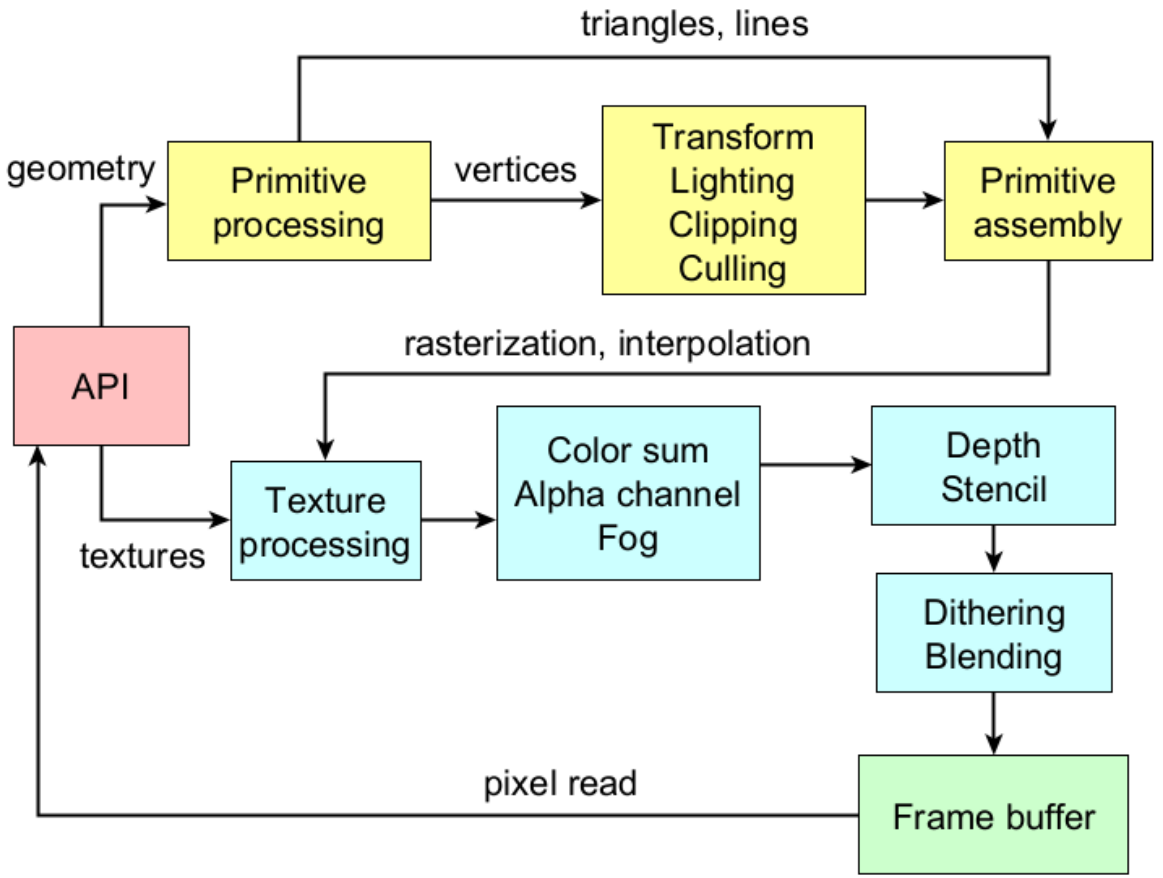
ogl-simplest.cc

Фиксированный конвейер

- Фиксированные блоки.
- Управляются отдельными функциями

```
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
glEnable(GL_DEPTH_CLAMP);
glEnable(GL_CULL_FACE);
glClear(GL_COLOR_BUFFER_BIT);
```

- Это тонна API функций и enums.



Общение с рантаймом

- Каждый раз когда вы дёргаете API функцию, вы дёргаете рантайм, который должен в какой-то момент послать информацию драйверу.

```
for (auto Coord : Vertices)
  glVertex3fv(Coord); // это вызов OpenGL runtime
```

- Проблема в том, что каждый такой API вызов предполагает накладные расходы, на которые вы идёте каждый фрейм. И которые сложно кешировать.
- Нам наоборот хочется максимум отдать в память GPU и минимально с ней взаимодействовать.
- Во многом это компенсируется тем, что в OpenGL возможны **расширения**.

Расширения OpenGL: буферы вершин

- Отрендерим тот же квадрат по другому: подготовим буферы вершин.

```
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);
glBindVertexArray(VAO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(Vertices),
  Vertices, GL_STATIC_DRAW);
```

- В цикле рендеринга теперь всё стало куда приятней.

```
glClear(GL_COLOR_BUFFER_BIT);
glBindVertexArray(VAO);
glDrawArrays(GL_QUADS, 0, 4);
```

Расширения и версии

- Буфера вершин были введены расширением `ARB_vertex_array_object` в OpenGL 2.1 и закреплены в стандарте OpenGL 3.0
- Расширения предлагаются участниками консорциума и их реально десятки.

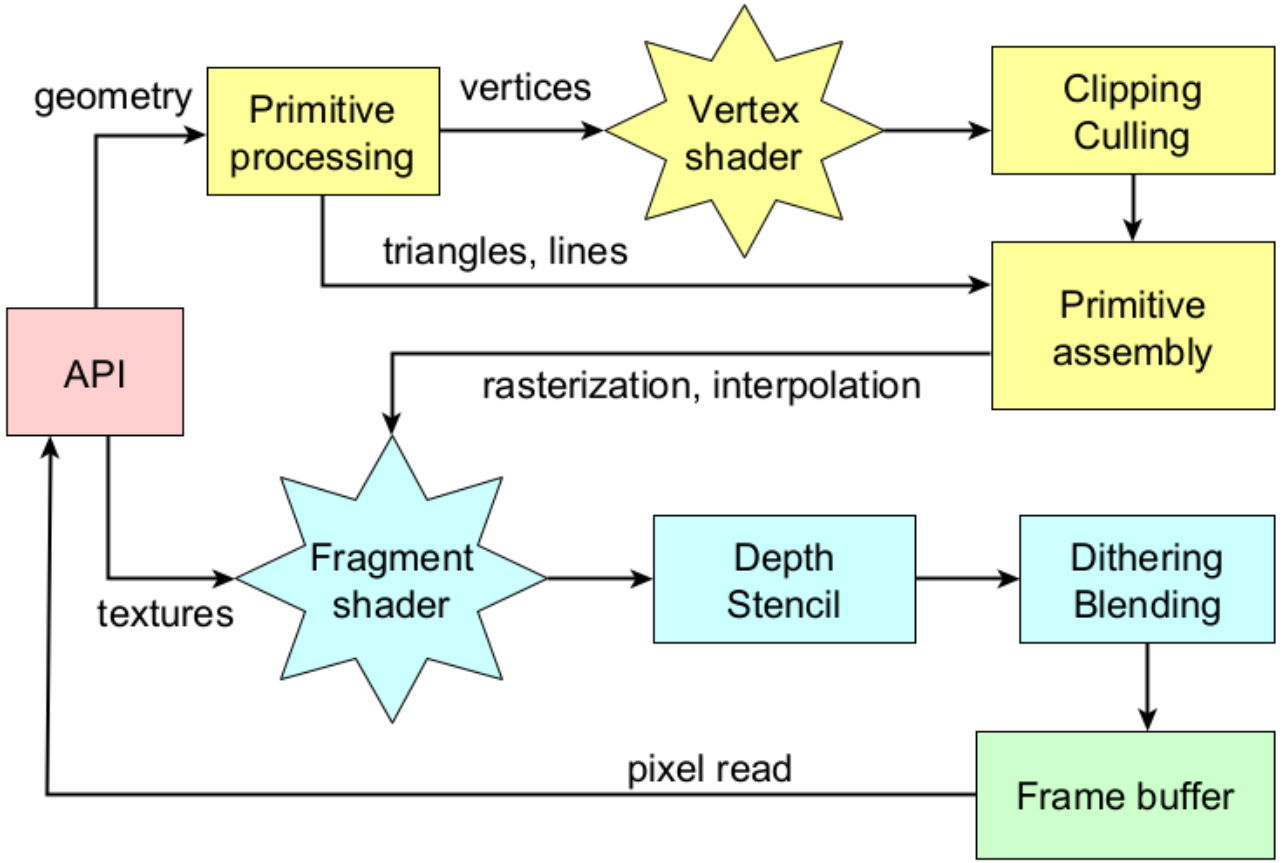
https://www.khronos.org/registry/OpenGL/extensions

- Существуют автоматизированные системы такие как glad, запрашивающие вам расширения и генерирующие хедер с доступными функциями.
- Для более тонкого контроля есть библиотека GLEW, позволяющая проверять доступность расширений и многое другое.

The OpenGL Extension Wrangler Library

Нефиксированный конвейер

- Первой идеей, появившейся достаточно рано была идея **шейдера** т.е. небольшой программы для видеокарты, которая позволяла бы гибко управлять светом и тенью на каждой вершине.
- Так в 2001 году в OpenGL 2.0 появился язык GLSL.
- В программировании GPU есть своя специфика.



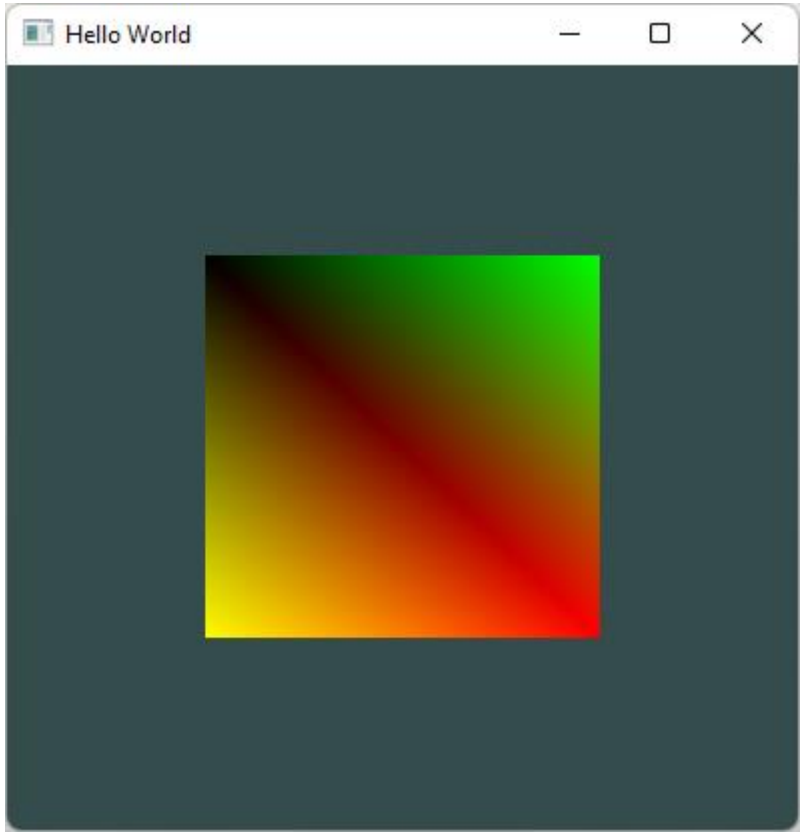
Вершинные шейдеры

- В примере с каждой вершиной связан цвет

```
// positions          // colors
0.5f,  0.5f,  0.0f, 0.0f, 1.0f, 0.0f,
```

- Этот цвет как атрибут вершины передаётся в вершинный шейдер

```
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;

out vec3 vColor;

...

vColor = aColor; // выход во фрагменты
```



simplest.vert

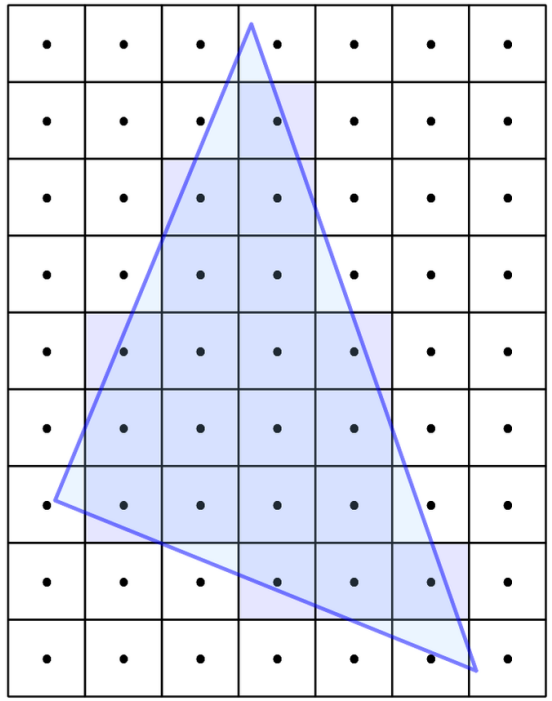
Binding points: glBindBuffer

```
GLfloat Vertices[] = {
  0.5f,  0.5f,  0.0f, 0.0f, 1.0f, 0.0f,
 -0.5f,  0.5f,  0.0f, 0.0f, 0.0f, 0.0f,

glBindBuffer(GL_ARRAY_BUFFER, VBO);

glVertexAttribPointer(0, 3, GL_FLOAT,
  GL_FALSE, 6 * sizeof(GLfloat),
  0 * sizeof(GLfloat));

glVertexAttribPointer(1, 3, GL_FLOAT,
  GL_FALSE, 6 * sizeof(GLfloat),
  3 * sizeof(GLfloat));
```

```
layout (location = 0)
in vec3 aPos;

layout (location = 1)
in vec3 aColor;

out vec3 vColor;

void main() {
  gl_Position =
    vec4(aPos, 1.0);
  vColor = aColor;
}
```

Что такое "фрагмент"?

- Фрагмент это выход растеризатора.
- Также можно сказать, что фрагмент это потенциальный пиксель.
- Когда каждый элемент геометрии растеризуется, мы получаем фрагменты на экране с двумерной позицией и цветом.
- Фрагментный шейдер это программа, индивидуально работающая для каждого фрагмента и трансформирующая его.



https://thebookofshaders.com

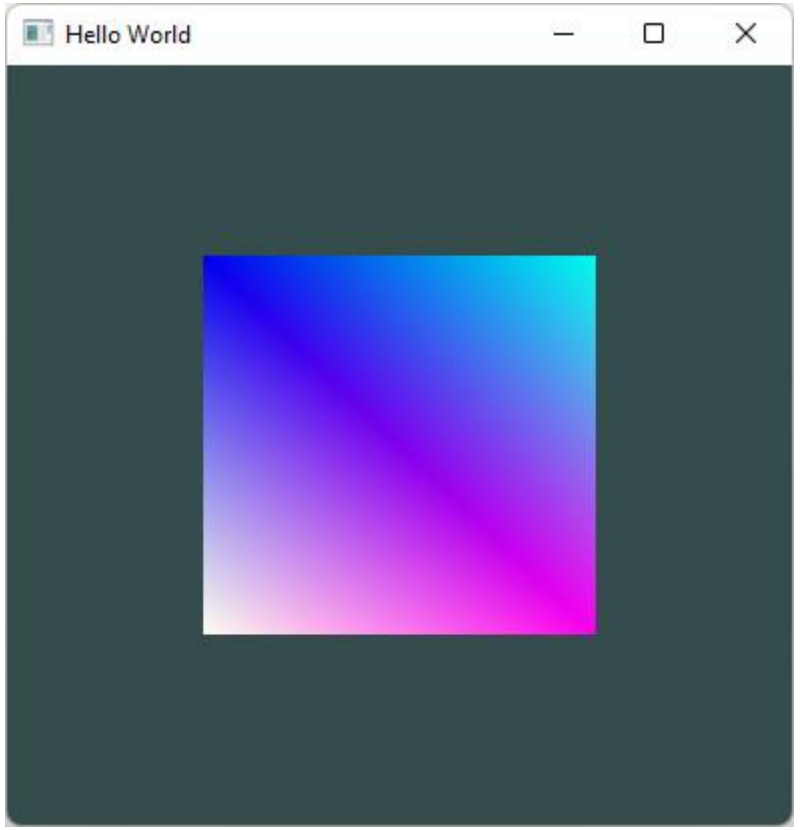
Фрагментные шейдеры

- Вершинный шейдер сообщает цвет в out-переменную.

```
vColor = aColor; // выход во фрагменты
```

- Далее выходной цвет каждой вершины растеризуется интеполируется.
- Фрагментный шейдер добавляет синусоиду в синий канал каждого фрагмента:

```
gl_FragColor = vec4(vColor.xy,
  vColor.z + abs(sin(time)), 1.0);
```



sincolor.frag

Uniform и varying переменные

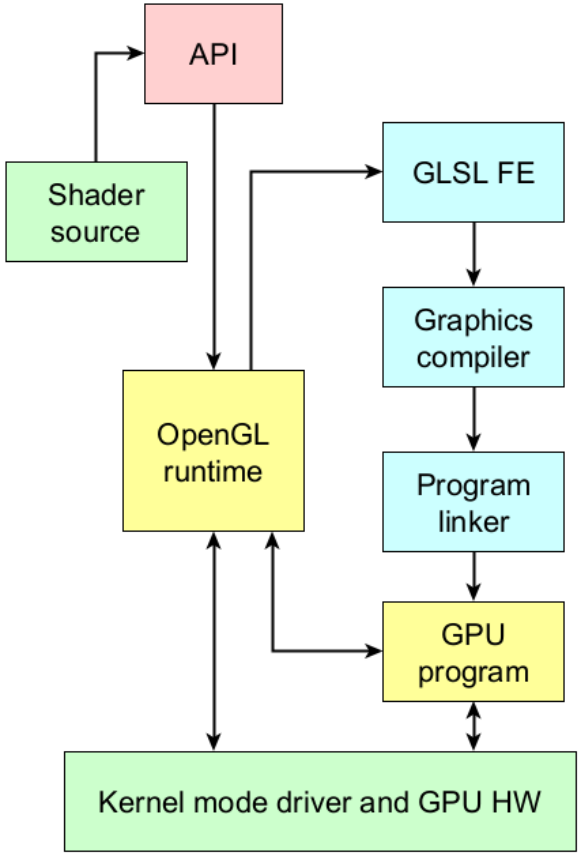
- Шейдер работает сверхпараллельно и независимо: для каждого объекта.
- Переменная, варьирующаяся от объекта называется varying. Общая на всех называется uniform (например время).

```
in vec3 vColor; // varying (приходит от растеризатора)

uniform float time; // uniform

void main() {
  for (f : all fragments)
    gl_FragColor[f] = vec4(vColor[f].xy,
                           vColor[f].z + abs(sin(time)), 1.0);
}
```


ogl-frag-shader.cc

Компиляция и исполнение шейдеров

- Необходимость компиляции делает графический драйвер гораздо сложнее: там появляется компилятор.
- Вызовы glCompileShader, glLinkProgram это вызовы возвращающие (возможно) ошибку и лог компиляции.
- Компилятор OpenGL для графики Intel является LLVM-based и содержит более 150 оптимизационных фаз.
- При исполнении, шейдер можно включить через glUseProgram и можно переключить на другой.



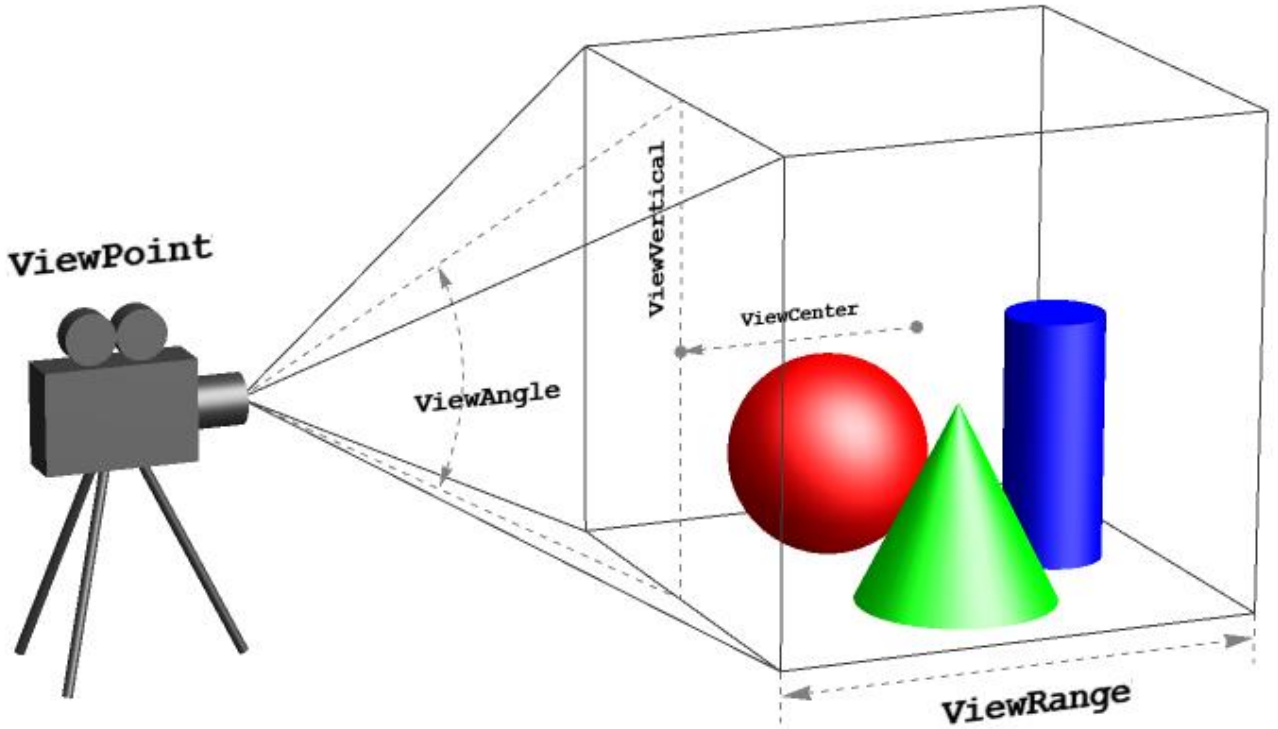
Обсуждение: а где же 3D?

- Пока что от обещанной трёхмерной графики мы видим только двумерный квадрат.

- GPU и OpenGL
  - *Логическая модель*
- Vulkan API
- Физическая и объектная модель

Обсуждение: сцена и отображение

- У нас есть мировые координаты сцены. Внутри сцены расположена модель.
- Как перейти от координат сцены к координатам модели?
- Каким образом перейти от координат модели к координатам вида?
- Можно ли дополнительно учесть проекцию?



OpenGL matrices tutorial

Шейдер для трансформации

```
glm::vec3 Position;
glm::vec3 Up;

// ....

glm::mat4 Model(1.0f);

glm::mat4 View = glm::lookAt(
  Position, LookTo, Up);

Projection = glm::perspective(
  glm::radians(FoV), Aspect,
  Near, Far);
```

```
in vec3 aPos;
in vec3 aColor;

out vec3 vColor;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
  gl_Position =
    projection * view * model *
    vec4(aPos, 1.0);
  vColor = aColor;
}
```

Давайте отрендерим куб

- Первый вариант: послать в режиме QUADS $6 * 4$ вершин.
- Второй вариант: $2 * 4$ вершин, $6 * 4$ индексов.

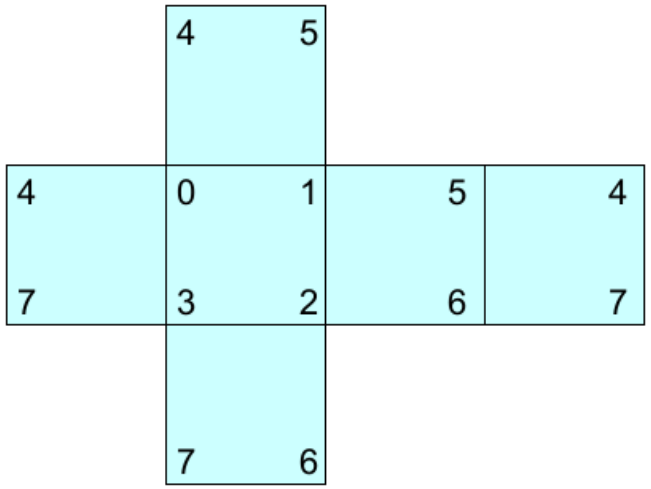
```
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(Vertices),
             Vertices, GL_STATIC_DRAW);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, IBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(Indices),
             Indices, GL_STATIC_DRAW);
```

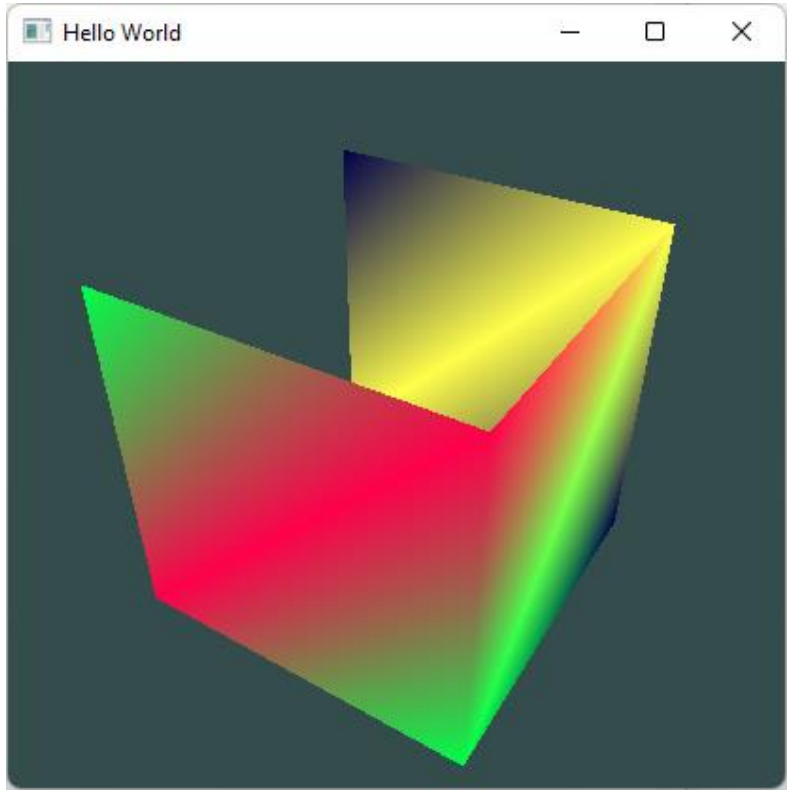
- Это несколько меньше данных для посылки на видеокарту (48 байт против 72) и это показывает ещё одну binding point.

Первая проблема: culling

- Небольшая ошибка с буферами индексов

```
GLubyte Indices[] = {
 // quads
 0, 3, 2, 1,
 0, 3, 7, 4,
 1, 2, 6, 5,
 5, 1, 0, 4,
 3, 7, 6, 2,
 5, 6, 7, 4,
};
```





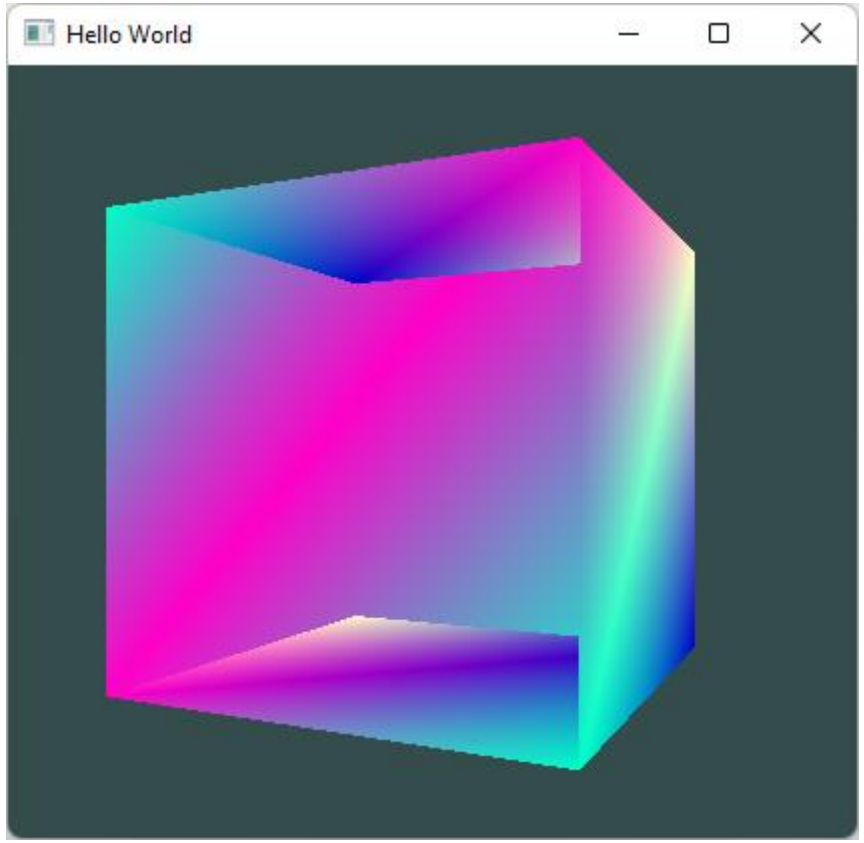
- Демонстрирует face culling

Вторая проблема: depth

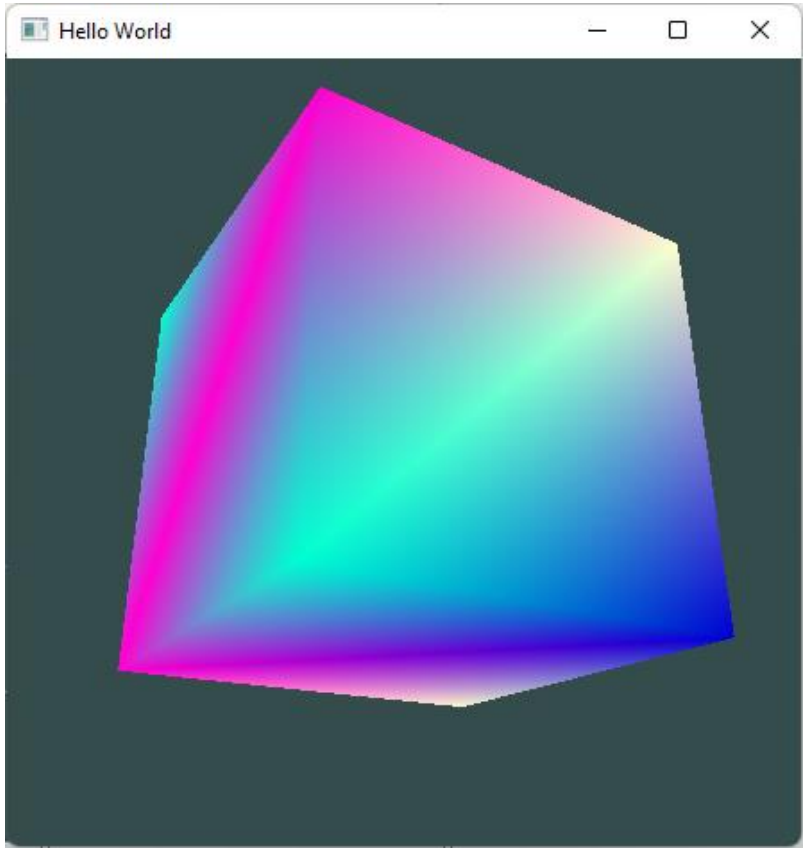
- Даже если правильно угадать с буферами, но забыть depth и culling checks, всё ещё могут быть артефакты

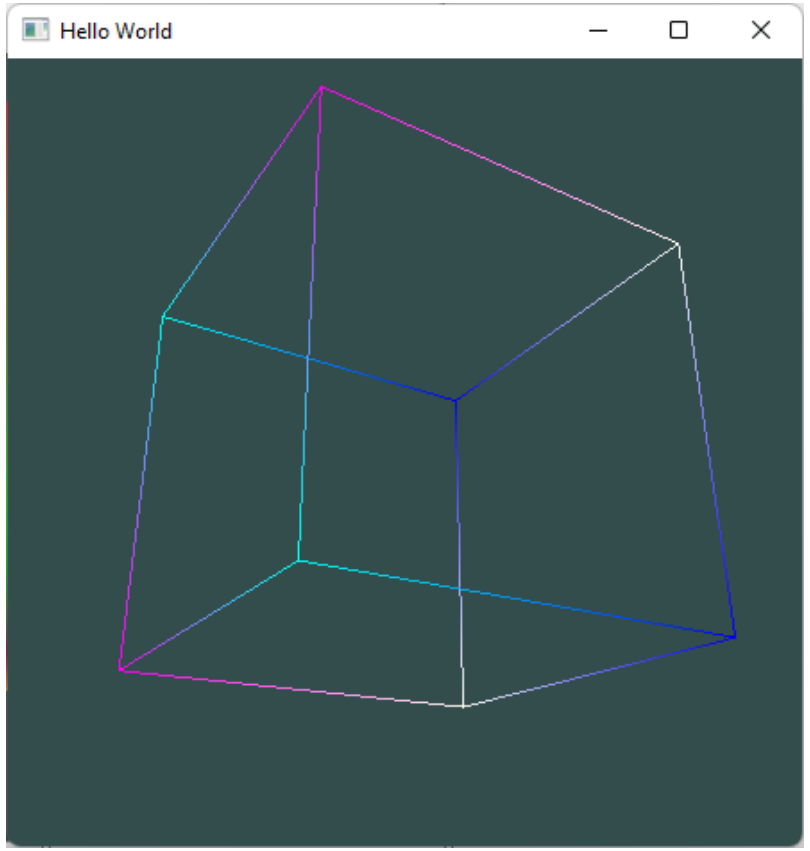
```
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);

glEnable(GL_DEPTH_CLAMP);

glEnable(GL_CULL_FACE);
```

- Конвейер OpenGL плохо понимает что человек имел в виду. Для него важен режим геометрии.



Режимы геометрии: QUADS vs LINES



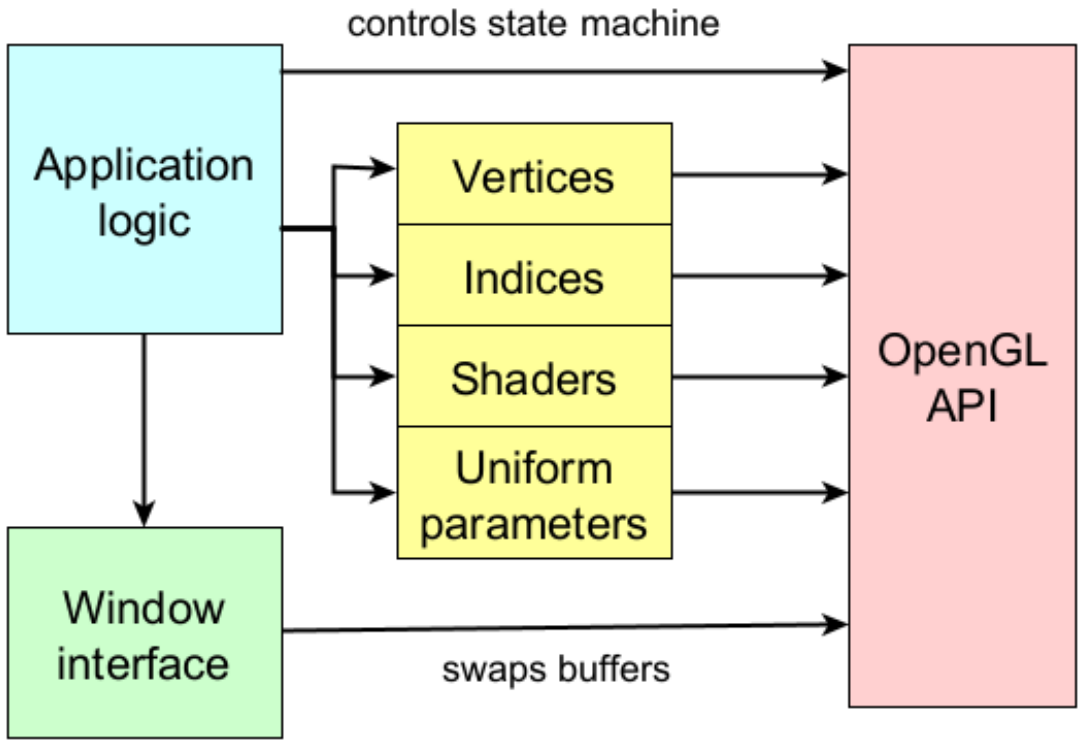


Обсуждение: архитектура

- Покритикуйте код поворота кубика, выложенный на гитхабе по ссылке
- С чего бы вы начали проектирование?

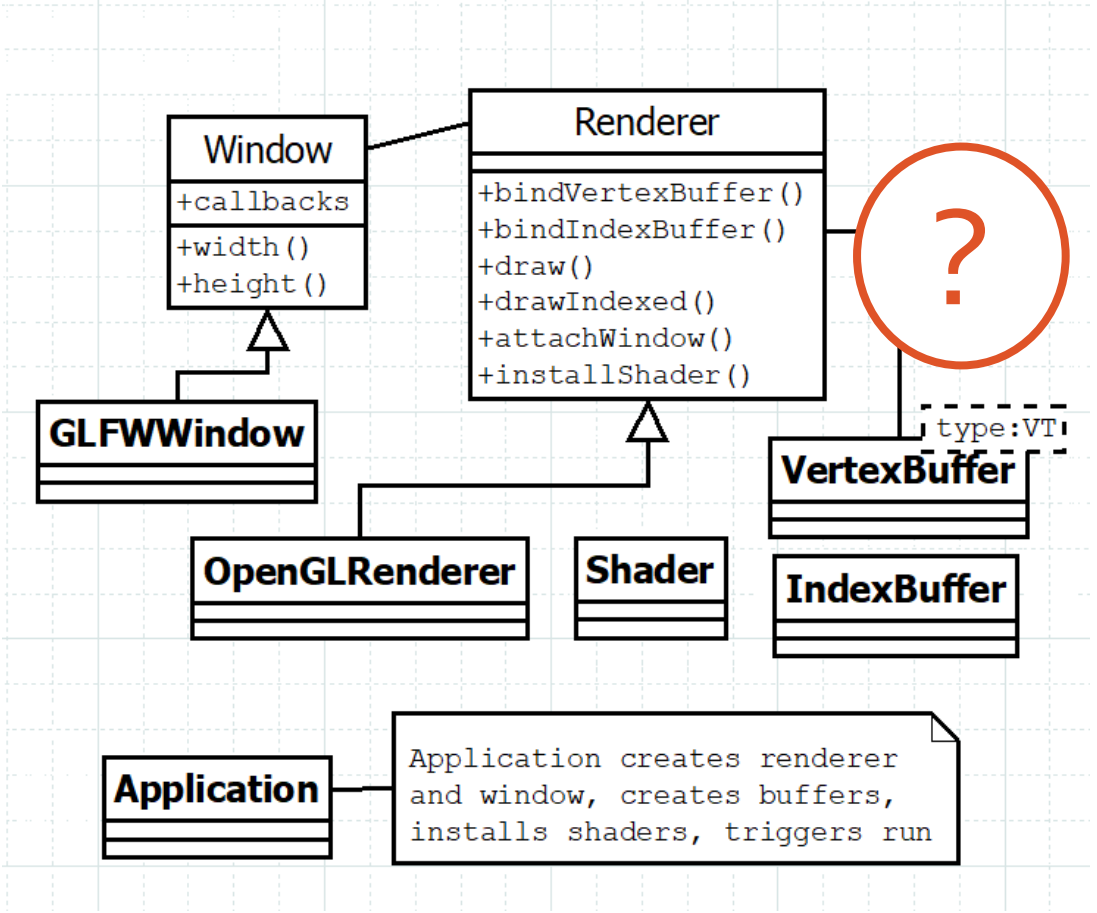
Что происходит в программе?

- Приложение формирует геометрию, шейдеры и прочее и кормит OpenGL API
- Кроме того приложение взаимодействует с оконным интерфейсом
- Который сам по себе может взаимодействовать с API, например для перерисовки
- Где тут место для рендерера?



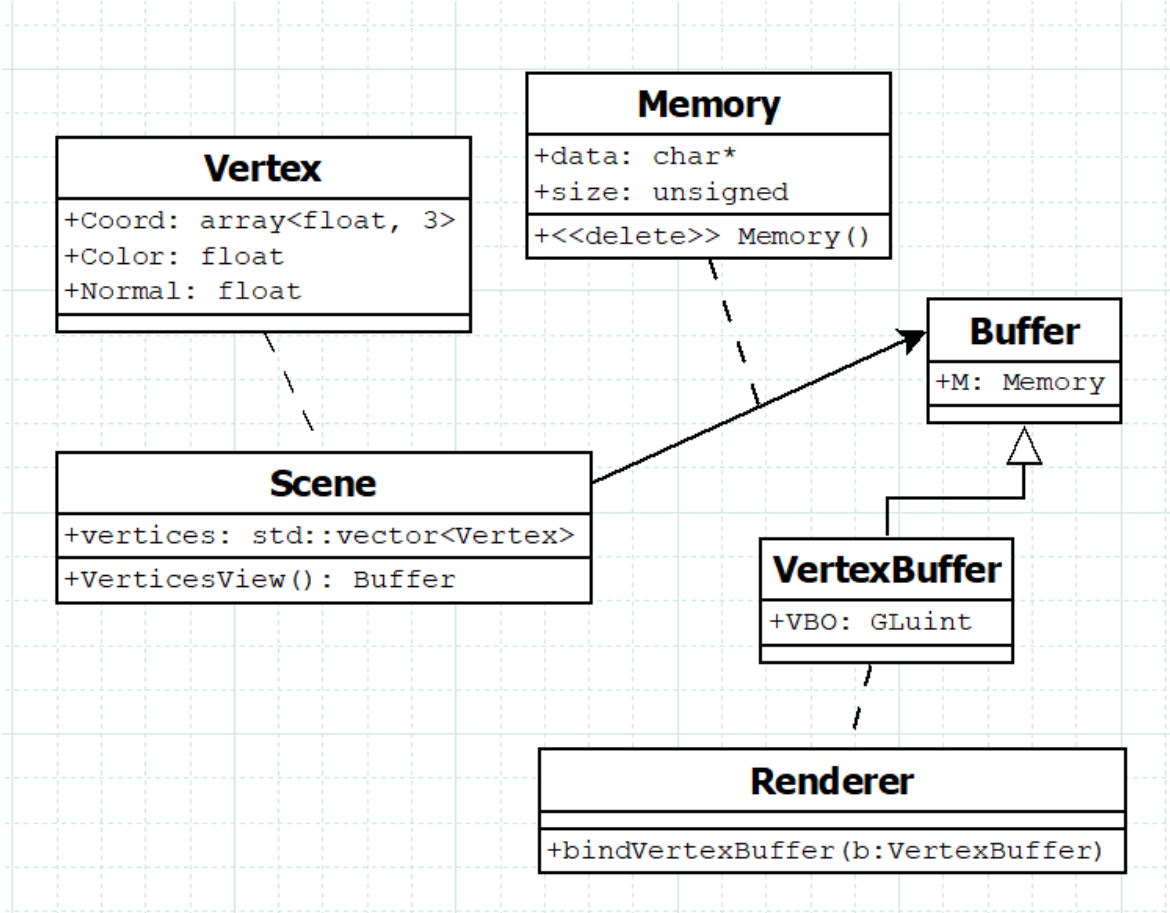
Обсуждение: что такое вершина?

- Атрибутами вершины могут быть
  - Координаты
  - Цвет
  - Нормали (для правильного освещения)
  - Что угодно ещё (у нас же программируемый конвейер)
- Значит ренедерер должен как-то принимать обобщённый буфер вершин.



Идея сцены и простые вершины

- Класс сцены хранит всю информацию о вершинах и геометрии.
- Его взаимодействие с рендерером можно сделать низкоуровневым.
- Рендерер предоставляет обёртку Index/Vertex Buffer Object над любыми данными
- Любые данные передаются туда как сырая память



Ассиметрия в параметрах шейдера

- Vertex/index buffers можно трактовать как наследники одной структуры.

```
struct Buffer{
  virtual size_t size() const = 0;
  virtual void push(Memory) = 0;
  virtual ~Buffer() = default;
};
```

- Но параметры шейдера могут потребовать установки uniform переменных в таком же зависимом от сцены ключе.
- Следует ли завести для них отдельный интерфейс и где?

Uniform buffer objects

```
in vec3 aPos;
in vec3 aColor;

out vec3 vColor;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
  // используем
}
```

```
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;

layout (location = 0) out vec3 vColor;

layout (std140) uniform Matrices {
  mat4 model;
  mat4 view;
  mat4 projection;
};

void main() {
  // тут всё это используем
}
```

Обсуждение

- Кто и в какой момент должен переключать автомат OpenGL?

glEnable(GL_DEPTH_CLAMP);

glEnable(GL_CULL_FACE);

и т. д.

- Разумеется довольно странно на каждый чих делать по методу в рендерере
- Мы видим, что сама модель OpenGL как гигантского конечного автомата делает его объектное представление неудобным.
- И, как мы дальше увидим, неэффективным.

- GPU и OpenGL
- Логическая модель
  - Vulkan API
- Физическая и объектная модель

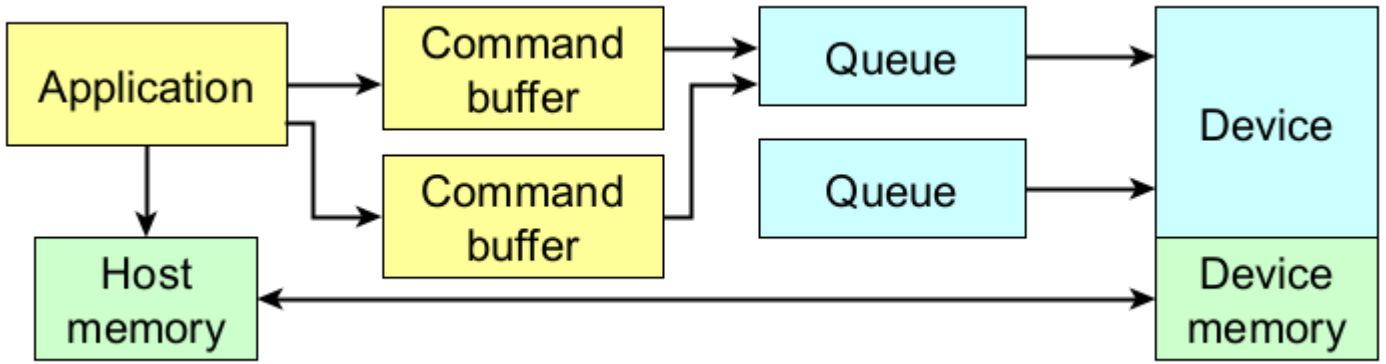
На пути к Vulkan

- OpenGL API слишком высокоуровневое. У него всегда был некоторый перекос в сторону усложнения графических драйверов.
- Из-за этого OpenGL мало пригоден как к проиводству игр, так и к мобильным приложениям.
- Необходимость нового графического API была осознана к 2016-му.
- В отличии от OpenGL, Vulkan пока не идёт в стандартной поставке OS и его SDK приходится скачивать отдельно.

https://www.lunarg.com/vulkan-sdk/

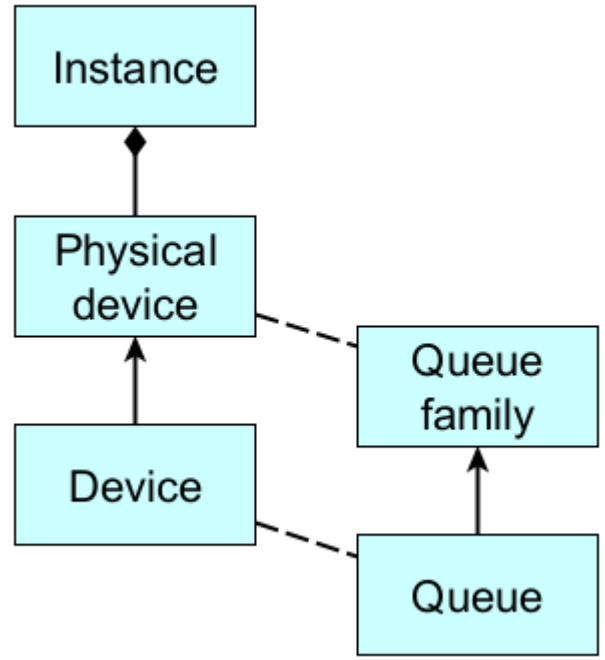
Концептуальная модель Вулкана

- Основные отличия от OpenGL: возможность записать несколько буферов команд и использовать несколько очередей устройства.
- Добавлено явное управление памятью и разные типы памяти.
- Кроме того API отвязано от условного "экрана", swap chain для Вулкана это расширение, рендеринг может идти куда угодно.



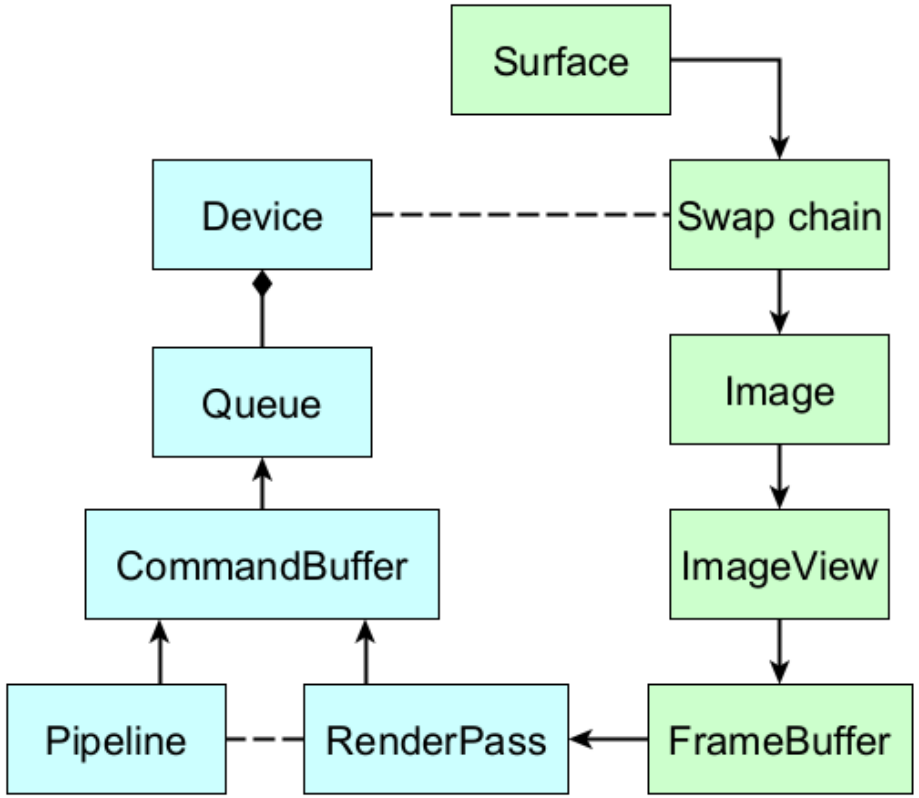
Основы: Instance, Device, Queue

- vkCreateInstance (в одном приложении м.б. несколько)
- vkEnumeratePhysicalDevices
  - API поддерживает работу с несколькими физическими устройствами для каждого Instance.
- vkGetPhysicalDeviceQueueFamilyProperties
  - Очереди могут быть разных типов.
- vkCreateDevice
  - Логическое устройство может содержать много очередей.
- vkGetDeviceQueue
  - Дескриптор очереди далее нужен для использования в API



Рендеринг: swap chain, images, pipeline

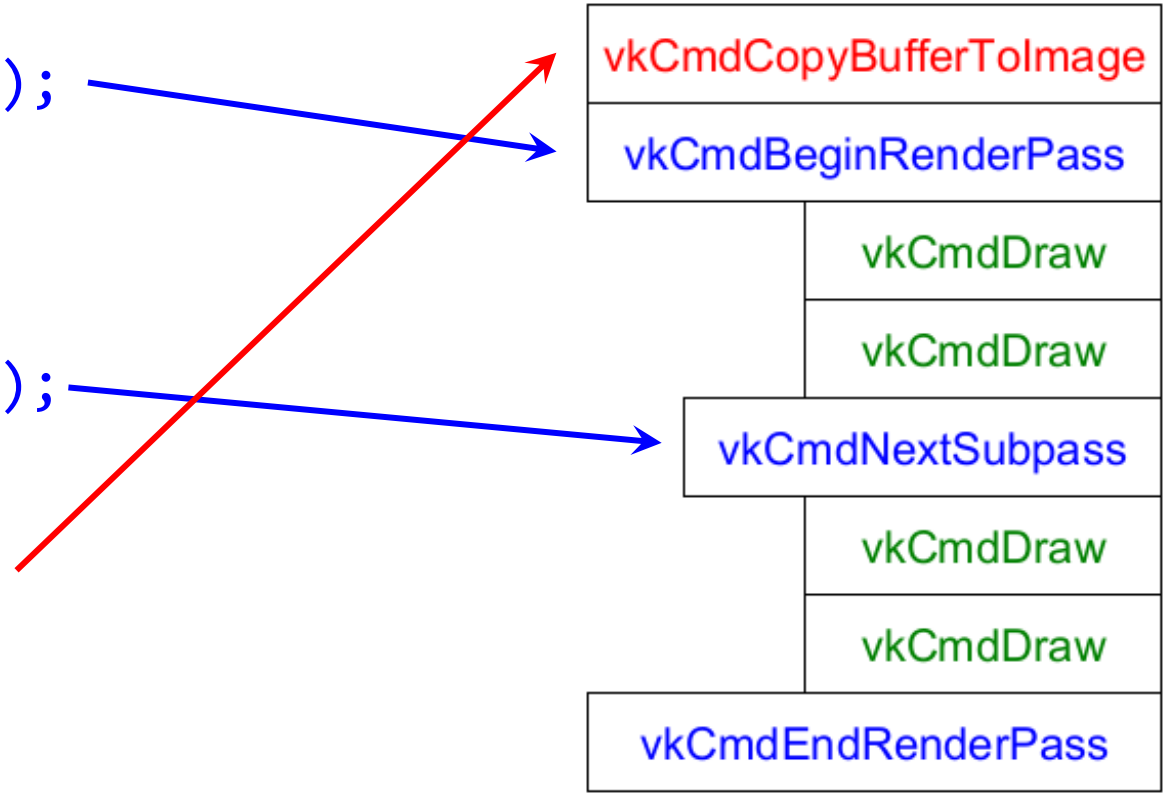
- Image здесь это то, что пойдёт на экран. Программа сама делает двойную (тройную и т.п.) буферизацию
- Три новых важных термина
  - Render pass
  - Pipeline
  - Command buffer
- Казалось бы хм… pipeline? Для OpenGL он один и глобальный.



Understanding vulkan objects

Пасс рендеринга

```
glBindFrameBuffer(...);
glDrawArrays(...);
glDrawArrays(...);
glBindFrameBuffer(...);
glDrawArrays(...);
glTexSubImage2D(...);
glDrawArrays(...);
```



Фиксированный конвейер

VI = vertex input
IA = input assembly
TS = tesselation
VP = viewport
RS = raster
MS = multisample
DS = depth / stencil
CB = color blend

- К конвейеру обязательно привязывается renderpass.
- Единожды созданный конвейер не изменяется. Его можно только пересоздать
- Программа может оперировать любым количеством конвейеров

VS = vertex shader
CS = tessellation control
ES = tessellation evaluation
GS = geometry shader
FS = fragment shader

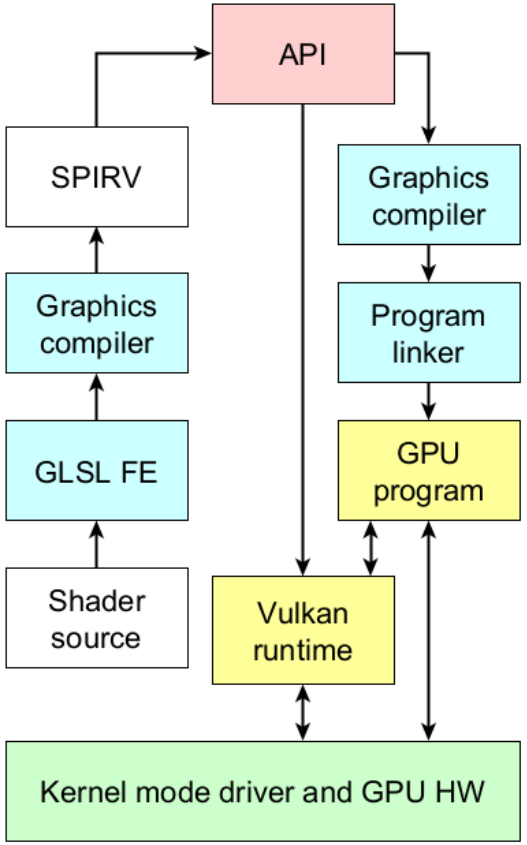
VkShaderStageFlagBits.html

Прекомпиляция шейдеров

- Тратить в рантайм время на запуск clang это расточительно.
- Шейдеры предварительно компилируются в SPIRV

```
glslc simplest-v.vert -o
      simplest-v.vert.spv
```

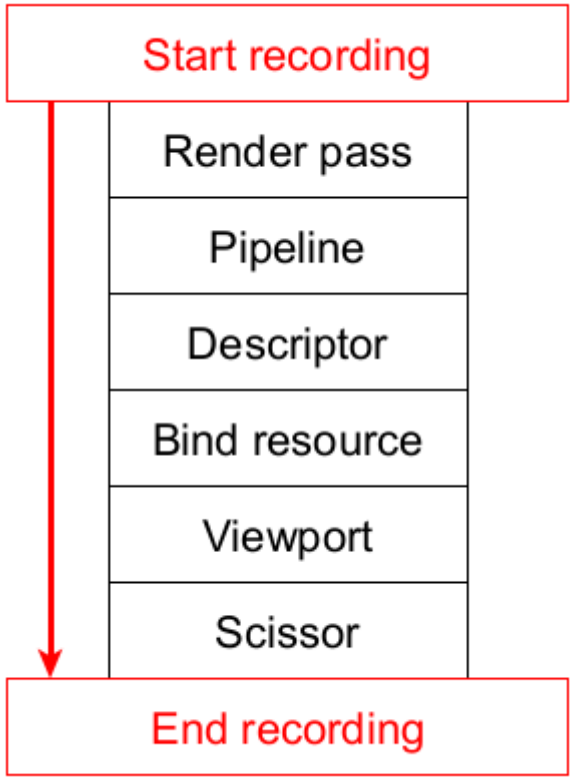
- SPIRV это единое представление для Vulkan, OpenGL и OpenCL (со своими расширениями)
- Бинарный формат можно дизассемблировать в нечто, напоминающее LLVM IR.



SPIRV.html

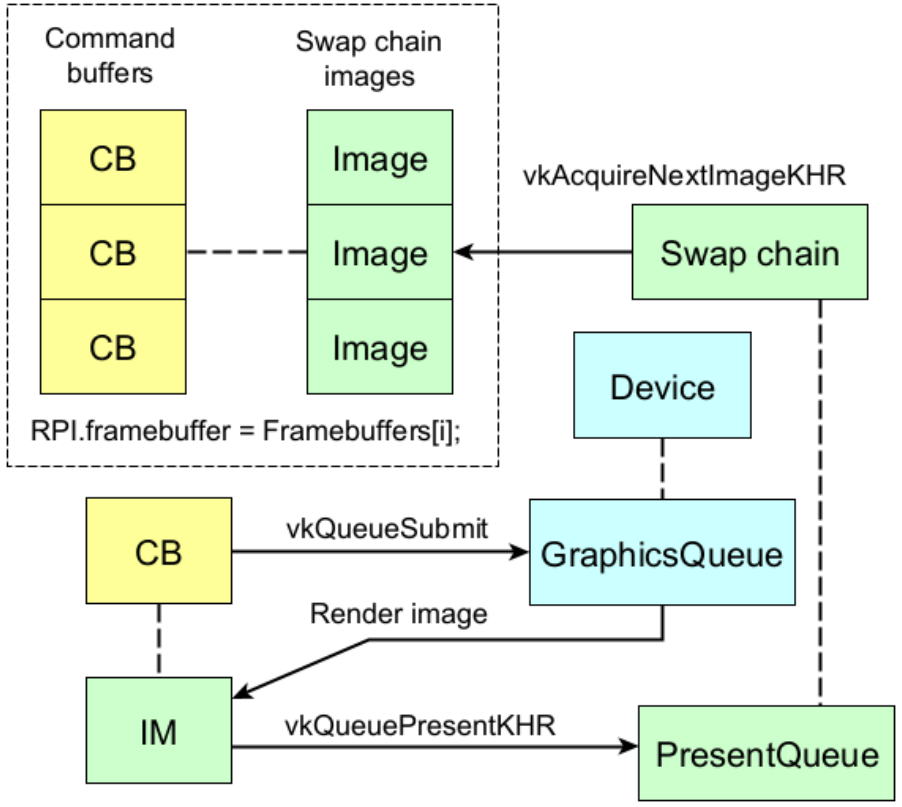
Буфер команд

- Буфер команд включает в себя описание конвейера, рендер пасс и всё над чем они будут работать.
- Именно сюда биндятся все ресурсы (буферы вершин, буферы индексов и т.п.)
- Также именно тут настраиваются viewport/scissors чтобы не приходилось пересоздавать pipeline ради их настройки.
- И далее буфера команд отправляются на очередь.



Цикл отображения

- Командный буфер связывается с картинкой для swapchain заранее в RenderPassInfo.
- Командный буфер уходит в графическую очередь, рендерит картинку.
- Далее эта картинка отправляется в перезентационную очередь на swapchain.
- Разумеется это не обязательная схема.



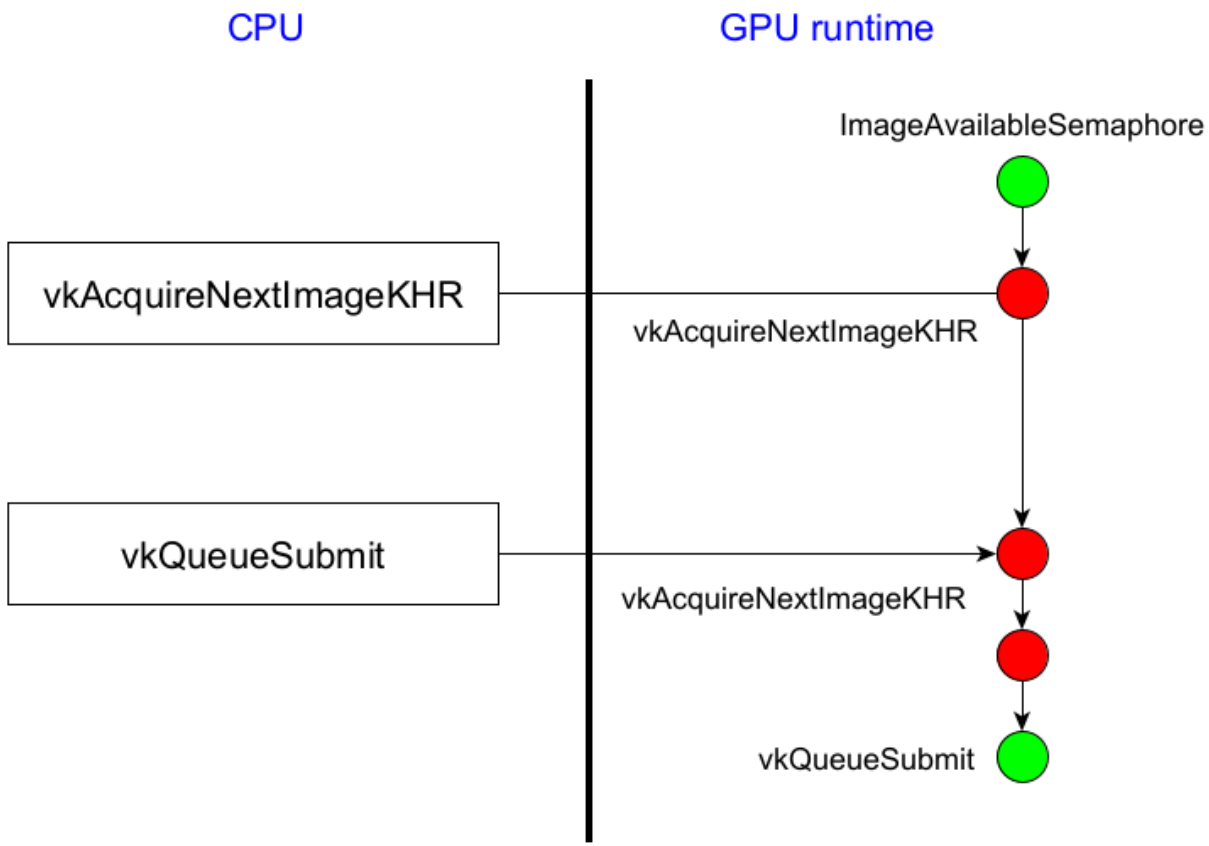
Проблема синхронизации

```
vkAcquireNextImageKHR(...., &imageIndex); // non-blocking

VkSubmitInfo submitInfo;
submitInfo.pCommandBuffers = &CommandBuffers[imageIndex];

vkQueueSubmit(GraphicsQueue, ...., submitInfo); // same

VkPresentInfoKHR presentInfo;
presentInfo.pImageIndices = &imageIndex;

vkQueuePresentKHR(PresentQueue, &presentInfo); // same
```

- Хорошая ли идея дождаться готовности кадра т.е. сделать все эти вызовы блокирующими?

Семафоры для синхронизации

- Некоторые API берут специальные объекты "семафоры".
- Эти объекты не допускают начала реального исполнения, пока не происгналены.
- vkQueueSubmit берет два семафора: один она ждёт, второй ставит для vkQueuePresentKHR



Page synchronization

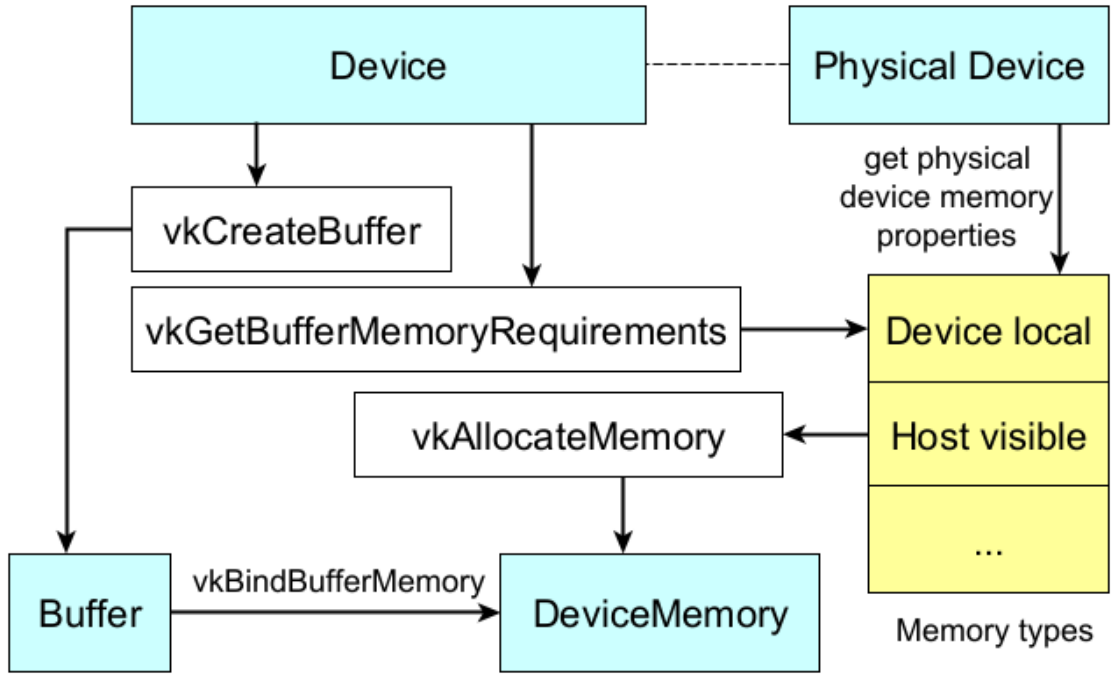
Фенсы для синхронизации

- Семафор используется внутри GPU runtime.
- В отличии от него фенс позволяет синхронизировать CPU и GPU.
- Со стороны CPU мы вызываем vkWaitForFences

```
vkWaitForFences(...., InFlight[CurrentFrame]); // wait
vkAcquireNextImageKHR(....)
vkResetFences(...., InFlight[CurrentFrame]); // reset
vkQueueSubmit(...., InFlight[CurrentFrame]); // set
```

Управление памятью

- Каждое физическое устройство возвращает массив VkMemoryType.
- Логическое устройство создаёт буфер с отдельными create/usage flags.
- Например USAGE_TRANSFER и HOST_COHERENT.
- Далее нужно связать логический тип буфера с физическим типом памяти для него и выделить память.



VkMemoryPropertyFlagBits.html

VkBufferUsageFlagBits.html

Отображение памяти

- Допустим мы создали на устройстве staging buffer

```
VkBuffer stagingBuffer; // usage = transfer_src
VkDeviceMemory stagingBufferMemory; // property = host_visible
```

- Теперь хочется заполнить его с хоста (например вершинами).
- Для этого память (если она host visible) можно просто отобразить на устройство.

```
void *data;

vkMapMemory(Device, stagingBufferMemory, 0, bufferSize, 0, &data);
std::copy(Vertices.begin(), Vertices.end(), cast<Vertex *>(data));
vkUnmapMemory(Device, stagingBufferMemory);
```

Обсуждение

- Достаточно ли вы поняли идею Вулкана, чтобы догадаться как скопировать память из буфера в буфер **внутри** устройства?
- Или как скопировать в буфер, если он не host-visible?
- И вообще как сделать операцию с памятью в общем случае?

Command buffer спешит на помощь

- Команда которую можно положить в очередь, это в частности команда записи памяти.

```
vkBeginCommandBuffer(commandBuffer, &beginInfo);

vkCmdCopyBuffer(commandBuffer, srcBuffer, dstBuffer, ....);

vkEndCommandBuffer(commandBuffer);
```

- Далее можно сразу отправить её в очередь и заблокироваться, дожидаясь ответа.

```
vkQueueSubmit(GraphicsQueue, ....); // transfer queue?
vkQueueWaitIdle(GraphicsQueue);
```

Демо

- Покажем очевидное превосходство в FPS на примере.

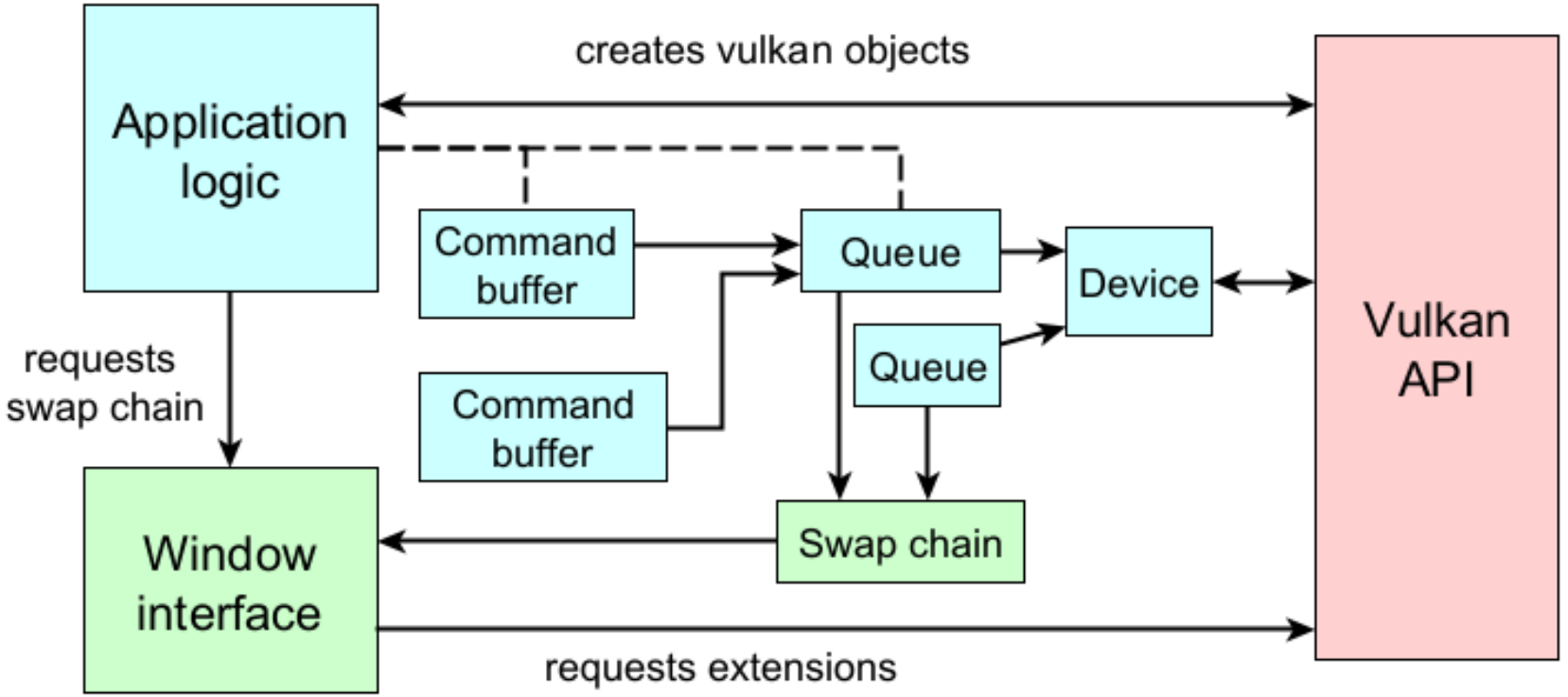
```
build2> Release\uniform_buffer.exe -ogl
```

Обсуждение

- Покритикуйте простейшую программу по ссылке.
- Что бы вы там улучшили или перепроектировали и как?

- GPU и OpenGL
- Логическая модель
- Vulkan API
- Физическая и объектная модель

Что происходит в программе на Vulkan?



Vulkan это объектная модель

```
// объект
VkDevice Device;

// конструктор
vkCreateDevice(PhysDevice, &createInfo, nullptr, &Device)

// методы
vkGetDeviceQueue(Device, GraphicsFamily, 0, &GraphicsQueue);
vkCreateImageView(Device, &createInfo, nullptr, &ImageView);
vkCreateRenderPass(Device, &renderPassInfo, nullptr, &Pass)

// деструктор
vkDestroyDevice(Device, nullptr);
```

Но она немного C style

```
VkBufferCreateInfo bufferInfo{};
bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
bufferInfo.size = size;
bufferInfo.usage = usage;
bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
```

- В таком подходе может быть такое, что
  - sType не совпал с настоящим типом
  - Инициализированы не все нужные поля
  - Константа записанная в sharingMode не имеет отношения к sharing mode
  - И т.д.
- Мы хотели бы сделать всё иначе.

Vulkan-Hpp: C++ API

```
struct BufferCreateInfo {
  BufferCreateInfo(BufferCreateFlags flags_ = {},
    DeviceSize         size_  = {},
    BufferUsageFlags   usage_ = {},
    SharingMode        sharingMode_ = SharingMode::eExclusive,
    uint32_t           queueFamilyIndexCount_ = {},
    const uint32_t *   pQueueFamilyIndices_   = {});

.....

BufferCreateInfo bufferInfo(size, usage);
```


Vulkan-Hpp

Безопасные флаги

- Мы хотели бы чтобы работало нечто вроде

```
enum class QueueFlagBits : VkQueueFlags  {
    eGraphics       = VK_QUEUE_GRAPHICS_BIT,
    eCompute        = VK_QUEUE_COMPUTE_BIT,
    eTransfer       = VK_QUEUE_TRANSFER_BIT,
    eSparseBinding = VK_QUEUE_SPARSE_BINDING_BIT,
    eProtected      = VK_QUEUE_PROTECTED_BIT,
....
vk::QueueFlags bits = vk::QueueFlagBits::eGraphics |
                      vk::QueueFlagBits::eCompute;
```

- Задача в том как удобней определить `vk::QueueFlags`?

Доступ к нижележащему типу

```
template <typename BitType> class Flags {
  using MaskType = typename std::underlying_type<BitType>::type;
  MaskType m_mask;

....

  Flags<BitType> operator|(Flags<BitType> const & rhs) const {
      return Flags<BitType>(m_mask | rhs.m_mask);
   }

using QueueFlags = Flags<QueueFlagBits>;
```

- Здесь мы предполагаем, что мы инстанцированы исключительно enum class

```
enum class QueueFlagBits : VkQueueFlags // <-- underlying
```

Размер кода существенно улучшается

```
auto mapping = vk::ComponentMapping{
  vk::ComponentSwizzle::eR, vk::ComponentSwizzle::eG,
  vk::ComponentSwizzle::eB, vk::ComponentSwizzle::eA };

auto subrange = vk::ImageSubresourceRange{
  vk::ImageAspectFlagBits::eColor, 0, 1, 0, 1};

for (auto image : swapchain_images_) {
  vk::ImageViewCreateInfo image_view_create_info(
    vk::ImageViewCreateFlags(), image, vk::ImageViewType::e2D,
    format_, mapping, subrange);
  swapchain_image_views_.push_back(
    device_->createImageViewUnique(image_view_create_info));
}
```

Unique pointers/resources

- Большая часть объектов ведёт себя unique-pointer-подобно

```
template <typename Type, typename Dispatch>
class UniqueHandle :
  public UniqueHandleTraits<Type, Dispatch>::deleter{
...
```

- Это позволяет как на прошлом слайде завернуть в самоумирающий по уничтожению `swapchain_image_views_` объект:

```
swapchain_image_views_.push_back(
  device_->createImageViewUnique(image_view_create_info));
```

Обсуждение

- Простая обёртка это хорошо, но давайте вернёмся к высокоуровневой архитектуре.
- Как мы расширим рендерер с учётом воможностей вулкана, что станет с логической моделью, что станет со сценой?

Литература

1. ISO/IEC, "Information technology – Programming languages – C++", ISO/IEC 14882:2017
2. The C++ Programming Language (4th Edition)
3. John M. Kessenich, Graham Seller, Dave Shreiner – OpenGL Programming Guide: The Official Guide to Learning OpenGL, 9-th edition, 2016
4. Parminder Singh – Learning Vulkan, Packt, 2016
5. Alexander Overvoorde – Vulkan Tutorial, self-published, 2020
6. Dustin Land – Getting explicit: How Hard is Vulkan really, GDC 2018
7. Jason Ekstrand – What Can Vulkan do for You?, The Linux Foundation, 2017
8. Michael Worcester – Getting Started with Vulkan, The Khronos Group, 2017
9. Karl Shultz – Vulkan Tutorial, DevU 2017