Architektur und Programmierung von Grafik- und Koprozessoren

Programmieren mit dem Vulkan API

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2018

Auflisten der Physikalischen Devices

vkEnumeratePhysicalDevices

Parameter

1.) `instance (VkInstance)`
Gültige Vulkan Instanz

2.) `pPhysicalDeviceCount (uint32_t*)`
Anzahl physikalischer Devices

3.) `pPhysicalDevices (VkPhysicalDevice*)`
Liste der physikalischen Devices

Auflisten der Physikalischen Devices

vkEnumeratePhysicalDevices

```
// Enumerate physical devices
uint32_t pdcnt = 0;
vkEnumeratePhysicalDevices(&pdcnt, nullptr);
std::vector<VkPhysicalDevice> devices(pdcnt);
vkEnumeratePhysicalDevices(&pdcnt, devices.data());
// Print some device properties and features
for (auto& dev : devices) {
    VkPhysicalDeviceProperties props;
    vkGetPhysicalDeviceProperties(dev, &props);
    std::cout << props.deviceName << '\n';
    std::cout << std::boolalpha << bool(props.deviceType
        == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU) << '\n';
    //...
    VkPhysicalDeviceFeatures feats;
    vkGetPhysicalDeviceFeatures(dev, &feats);
    std::cout<<std::boolalpha<<(bool)feats.geometryShader
        <<' '<<(bool)feats.tessellationShader<<'\n';
    //...
}
```

Auflisten der Physikalischen Devices

VkPhysicalDeviceProperties

Eigenschaften wie z. B. Name der Grafikkarte, Vendor ID, API- und Treiberversionen etc.:

```
struct VkPhysicalDeviceProperties {
    uint32_t apiVersion;
    uint32_t driverVersion;
    uint32_t vendorID;
    uint32_t deviceID;
    VkPhysicalDeviceType deviceType;
    char deviceName[VK_MAX_PHYSICAL_DEVICE_NAME_SIZE];
    uint8_t pipelineCacheUUID[VK_UUID_SIZE];
    VkPhysicalDeviceLimits limits;
    VkPhysicalDeviceSparseProperties sparseProperties;
};
```

Auflisten der Physikalischen Devices

VkPhysicalDeviceFeatures

Features, die der auf der GPU implementierte Rasterisierungsalgorithmus unterstützt, z. B.:

```
struct VkPhysicalDeviceFeatures {
    bool imageCubeArray;
    bool tessellationShader;
    bool multiDrawIndirect;
    bool samplerAnisotropy;
    bool textureCompressionETC2;
    bool pipelineStatisticsQuery;
    bool shaderFloat64;
    //...
};
```

Queue Arten

Kommandos (z. B. Zeichenbefehle, vgl. Vorlesungsteil 3) werden mit Vulkan explizit in *Queues* submittiert. Es gibt unterschiedliche Queues für unterschiedliche Arten von Kommandos.

Das ist grundlegend anders als mit OpenGL. Mit OpenGL sieht der Entwickler keine Kommandos. Command Buffer und Queues werden vollständig durch den Treiber versteckt. Mit Vulkan müssen Kommandos explizit in Queues submittiert werden.

Queue Arten

Mit `vkGetPhysicalDeviceQueueFamilyProperties()` kann man herausfinden, welche *Arten von Queues* das physikalische Device unterstützt, und wieviele davon erzeugt werden können (mehr als eine selten sinnvoll, Queues sind leichtgewichtig und können von Threads geteilt werden).

Wir fragen die Arten von verfügbaren Queues ab, um zu entscheiden, ob das Device das richtige ist.

Queue Arten

Von Vulkan unterstützte Queue Arten sind (`VkQueueFamilyProperties.queueFlags :VkQueueFlagBits`):

`VK_QUEUE_GRAPHICS_BIT`
Queue unterstützt Grafik

`VK_QUEUE_COMPUTE_BIT`
Queue unterstützt Compute

`VK_QUEUE_TRANSFER_BIT` *Queue unterstützt Datentransfers (meistens DMA)*

`VK_QUEUE_SPARSE_BINDING_BIT`
Spezieller Support für Sparse Texture Images

`VK_QUEUE_PROTECTED_BIT`
Spezieller Support für Protected Memory (explizit nicht vom Host aus sichtbar)

Queues und Presentation Surfaces

Bei der Auswahl der Grafik Queue (und auch beim Auswählen des phys. Devices) muss (wenn wir ein Bild in einem Fenster anzeigen wollen) darauf geachtet werden, dass die Queue Family mit der `VkSurfaceKHR` Extension kompatibel ist:

```
vkGetPhysicalDeviceSurfaceSupportKHR(
        device,
        queue_num, // int, iterate over all queues
        surface,   // VkSurfaceKHR member object
        &result);
```

Physikalisches Device Aussuchen

Bei der Auswahl des Physikalischen Device ist also darauf zu achten, dass

- das Device die Features und Extensions unterstützt, die die Applikation benötigt.
- das Device die entsprechenden Queues zur Verfügung stellt.
- das Device ggf. die `VkSurfaceKHR` Extension zum Zeichnen in Fenster unterstützt.

Die meisten dedizierten Grafikkarten sind für Compute und Grafik geeignet. Es gibt aber auch integrierte GPUs oder reine Compute Karten, die nicht allen Kriterien entsprechen.

Physikalisches Device Aussuchen

In einer produktiven Applikation ist eine Routine sinnvoll, die aufgrund dieser Kriterien das beste physikalische Device aussucht.

```
VkPhysicalDevice pickBestPhysicalDevice() { /* ... */ }
```

Die Applikation sollte ein Handle auf das Device als Member speichern:

```
physical_device_ = pickBestPhysicalDevice();
```

- `pickBestPhysicalDevice()` sollte das passendste Ressourcen Handle unter denjenigen zurückgeben, die vorher mit `vkEnumeratePhysicalDevices()` aufgelistet wurden.

Logisches Device

Zum Schluss der Setup Phase muss logisches Device erzeugt werden.

Ein logisches Device kann auch *mehrere* physikalische Devices umfassen.

Über logisches Device werden u. a. die Command Queues angesteuert.

Instanzen und Logische Devices

- *Instanzen* und *logische Devices* sind die Schlüsselkonzepte von Vulkan, z. B. werden Layers und Extensions sowohl bzgl. der Instanz als auch bzgl. des logischen Devices initialisiert.
- Erzeugung des logischen Devices konzeptionell ähnlich wie Erzeugung der Vulkan Instanz.
- Initialisierung von Layers und Extensions semantisch genau wie bei Instanz, daher hier der Kürze halber ausgespart.

Erzeugen eines Logischen Devices

vkCreateDevice

Parameter

1.) `physicalDevice (VkPhysicalDevice)`
Das mit dem Device assoziierte phys. Device

2.) `pCreateInfo (VkDeviceCreateInfo const*)`
Struct mit Infos zum Erzeugen

3.) `pAllocator (VkAllocationCallbacks const*)`
Allocator für Host Speicher

4.) `pDevice (VkDevice*)`
zurückgegebenes Device

Erzeugen eines Logischen Devices

vkCreateDevice
(1/2)

```
//--- specify queues
VkDeviceQueueCreateInfo qinfo = {};
qinfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
// Init with queue index we found when
// choosing *physical* device
qinfo.queueFamilyIndex = GraphicsQueueIdx;
qinfo.queueCount = 1;
float prio = 1.0f;
qinfo.pQueuePriorities = &prio;

//--- specify features we need (nothing spec. for now)
VkPhysicalDeviceFeatures features = {};
```

Erzeugen eines Logischen Devices

vkCreateDevice (2/2)

```
// Wrap it all up
VkDeviceCreateInfo dcinfo = {};
dcinfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
dcinfo.pQueueCreateInfos = &qinfo;
dcinfo.queueCreateInfoCount = 1;
dcinfo.pEnabledFeatures = &features;
// We also set *device* specific validation layers
// (same as above for the instance)
dcinfo.enabledLayerCount = ..;
dcinfo.pEnabledLayerNames = ..;

// Finally.. create the logical device
VkResult res = vkCreateDevice(physical_device_, &dcinfo,
                              nullptr, &device_);
if (res != VK_SUCCESS) {
    ...
}
```

Queue Handles Verwalten

vkGetDeviceQueue

Wenn wir das logische Device erzeugt haben, müssen wir noch Handles auf die Queues (z. B. als Member Variablen) speichern, damit wir später darein submittieren können. Diese erhält man mittels der Funktion `vkGetDeviceQueue`.

Queue Handles Verwalten

vkGetDeviceQueue

Parameter

1.) `device (VkDevice)`
Das logische Device

2.) `queueFamilyIndex (uint32_t)`
Index der Queue Familie (vorher abgefragt)

3.) `queueIndex (uint32_t)`
Index innerhalb der Familie (i.d.R. 0)

4.) `pQueue (VkQueue*)`
zurückgegebenes Queue Handle

Queue Handles Verwalten

vkGetDeviceQueue

```
VkQueue graphics_queue_; // member variables
VkQueue present_queue_;
...
vkGetDeviceQueue(device_,
                 GraphicsQueueIdx, /* see above */
                 0, /* index within the queue, usually */
                    /* best to have one queue/family */
                 &graphics_queue_);

vkGetDeviceQueue(device_,
                 PresentQueueIdx, /* see above */
                 0, /* index within the queue, usually */
                    /* best to have one queue/family */
                 &present_queue_);
```

Struktur der Vulkan Applikationsklasse bis Hierher

```
struct VulkanApp {
    void init() {
        // Create instance
        // Activate validation layers + debug CB
        // Create WSI surface
        // Choose physical device
        // Create logical device, retrieve queue handles
    }

    void cleanup() {
        // Release resources, finally release instance
    }
private:
    VkInstance instance_;
    VkDebugReportCallbackEXT callback_;
    VkSurfaceKHR surface_;
    VkPhysicalDevice physical_device_;
    VkDevice device_;
    VkQueue graphics_queue_;
};
```

Swap Chains

Mit *Swap Chains* kann Double Buffering implementiert werden.

OpenGL: *Default Framebuffer* $\Rightarrow$ man muss nicht explizit einen Framebuffer auswählen, Zeichenbefehle zeichnen in den Default Framebuffer. Platformspezifische Befehle wie `glXSwapBuffers()` zum Vertauschen und Zeichnen von Buffer-Inhalten.

Vulkan: Abstraktionslayer für Buffer Swapping, Swap Chain verwaltet Bilder, Applikation kann ein Ressourcen-Handle für ein Bild bekommen und in dieses zeichnen.

Swap Chains

Bevor wir ein physikalisches Device auswählen, müssen wir prüfen, ob dieses Swap Chain Support hat. Dafür muss die Extension

`VK_KHR_swapchain`

zur Verfügung stehen. Diese muss in die Liste der benötigten Extensions (für das Device!) aufgenommen werden, etwa:

```
std::vector<char const*> required_device_extensions({
    VK_KHR_SWAPCHAIN_EXTENSION_NAME
    });
```

Swap Chains

Wenn unser physikalisches Device `VK_KHR_swapchain` unterstützt und wir ein logisches Device mit Swap Chain Support erstellt haben, müssen wir prüfen, ob die Swap Chain unseren Anforderungen genügt.

- `VkSurfaceCapabilitiesKHR`: Minimale und Maximale Anzahl an Bildern in der Swap Chain, minimale und maximale Dimensionen der Bilder.
- `VkSurfaceFormatKHR`: Pixel Format (Bits pro Farb- oder Tiefenkanal, Farbraum-Support z. B. für sRGB, etc.).
- `VkPresentModeKHR`: Modi steuern, wie das Zeichnen von Bildern mit Refresh Rate des Displays synchronisiert wird.

Swap Chains

vkGetPhysicalDeviceSurfaceCapabilitiesKHR

Parameter

1.) `physicalDevice (VkPhysicalDevice)`
Das physikalische Device

2.) `surface (VkSurfaceKHR)`
Rendering Surface

3.) `pSurfaceCapabilities (VkSurfaceCapabilitiesKHR*)`
Zurückgegebene struct mit Eigenschaften

Swap Chains

vkGetPhysicalDeviceSurfaceCapabilitiesKHR

```
VkSurfaceCapabilitiesKHR capabilities;
vkGetPhysicalDeviceSurfaceCapabilitiesKHR(
                 physical_device_,
                 surface_,
                 &capabilities);
std::cout << capabilities.minImageCount << ' '
          << capabilities.maximageCount << ' '
          << capabilities.currrentExtent << ' '
          ...
          << capabilities.minImageExtent << ' '
          << capabilities.maxImageExtent << ' ';
```

Swap Chains

vkGetPhysicalDeviceSurfaceFormatsKHR

Parameter

1.) `physicalDevice (VkPhysicalDevice)`
Das physikalische Device

2.) `surface (VkSurfaceKHR)`
Rendering Surface

3.) `pSurfaceFormatCounts`
Anzahl Formate

4.) `pSurfaceFormats (VkSurfaceFormatsKHR*)`
Zurückgegebene Liste mit unterstützten Formaten

Swap Chains

vkGetPhysicalDeviceSurfaceFormatsKHR

```
uint32_t fcount = 0;
vkGetPhysicalDeviceSurfaceFormatsKHR(
                    physical_device_,
                    surface_,
                    fcount,
                    nullptr);
std::vector<VkSurfaceFormatKHR> formats(fcount);
vkGetPhysicalDeviceSurfaceFormatsKHR(
                    physical_device_,
                    surface_,
                    fcount,
                    formats.data());
for (auto& f : formats) {
    std::cout << std::boolalpha
        << f.format == VK_FORMAT_R8G8B8_UNORM << ' '
        << f.format == VK_FORMAT_R8G8B8_SRGB  << ' '
        ...
        << '\n';
}
```

Swap Chains

vkGetPhysicalDeviceSurfacePresentModesKHR

Parameter

1.) `physicalDevice (VkPhysicalDevice)`
Das physikalische Device

2.) `surface (VkSurfaceKHR)`
Rendering Surface

3.) `pSurfacePresentModeCounts`
Anzahl Modi

4.) `pPresentModes (VkSurfacePresentModeKHR*)`
Zurückgegebene Liste mit unterstützten Modi

Swap Chains

vkGetPhysicalDeviceSurfacePresentModesKHR

```
uint32_t pcount = 0;
vkGetPhysicalDeviceSurfacePresentModesKHR(
                 physical_device_,
                 surface_,
                 pcount,
                 nullptr);
std::vector<VkSurfacePresentModeKHR> pmodes(fcount);
vkGetPhysicalDeviceSurfacePresentModesKHR(
                 physical_device_,
                 surface_,
                 pcount,
                 pmodes.data());
```

Swap Chains

Present Modes

Swap Chains verwalten intern Queue von Bildern. Present Mode entscheidet, wie Queue mit Vertical Refresh Rate des Displays synchronisiert wird.

- `VK_PRESENT_MODE_IMMEDIATE_KHR`: keine Synchronisation ⇒ kein Queueing, “Tearing” kann auftreten (Scanlines für zwei Bilder werden angezeigt).
- `VK_PRESENT_MODE_MAILBOX_KHR`: einelementige Queue, Synchronisation mit Vertical Blanking Periode des Displays.
- `VK_PRESENT_MODE_FIFO_KHR`: wie Mailboxing, aber mehrelementige Queue.
- `VK_PRESENT_MODE_FIFO_RELAXED_KHR`: wie FIFO, erlaubt aber der Applikation, *gelegentlich* einen *verspäteten* Frame nachzuliefern (dann Zeichnen, ohne dass auf nächstes Sync Signal gewartet wird). (vgl. `{WGL|GLX}_EXT_swap_control_tear`.)

Swap Chains

Wenn wir sichergestellt haben, dass das physikalische Device die gewünschte Swap Chain unterstützt, können wir sie mit `vkCreateSwapchainKHR` erzeugen. Dies passiert wie üblich durch Initialisierung einer Parameter `struct`.

Vulkan Swap Chains sind sehr starr, z. B. muss jedes Mal eine neue Swap Chain erzeug werden, wenn sich die Fenstergröße verändert.

Überprüfe vor Erzeugung, ob die von der Applikation präferierten Farbformate, Presentation Modes etc. zur Verfügung stehen (hier der Kürze halber weggelassen).

Swap Chains

vkCreateSwapchainKHR

Parameter

1.) `device (VkDevice)`
Das logische Device

2.) `pCreateInfo (VkSwapchainCreateInfoKHR const*)`
Swapchain Create Info struct

3.) `pAllocator (VkAllocationCallbacks const*)`
Allocator für Host Speicher

4.) `pSwapchain (VkSwapchainKHR*)`
Zurückgegebenes Swap Chain Objekt

Swap Chains

vkCreateSwapchainKHR

(1/2)

```
VkSwapchainCreateInfoKHR scinfo = {};
scinfo.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
scinfo.surface = surface_;
scinfo.minImageCount = preferred_image_count;
// TODO: check these two
scinfo.imageFormat = color_format_;
scinfo.imageColorSpace = VK_COLOR_SPACE_SRGB_NONLINEAR_KHR;
scinfo.imageExtent = { 512, 512 };
// Relevant for stereo rendering
scinfo.imageArrayLayers = 1;
scinfo.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
// assumes graphicsFamily == presentFamily
scinfo.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
// E.g. 90deg rotation
scinfo.preTransform = capabilities.currentTransform;
scinfo.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
```

Swap Chains

vkCreateSwapchainKHR

(2/2)

```
VkResult res = vkCreateSwapchainKHR(device_, &scinfo,
                                    nullptr, &swapchain_);

if (res != VK_SUCCESS) {
    ...
}
```

Swap Chains

Swap Chain Images

Um später auf die Bilddaten in der Swap Chain zugreifen zu können, brauchen wir zum einen *Handles* auf die Bilder und zum anderen *Image Views*. Image Views definieren, wie wir auf die Bilddaten zugreifen (ist das Bild eine 2D Textur? Bezieht es sich auf Farbe oder Tiefe? Ist Mip-Mapping für das Bild aktiviert? Sollen Farbkanäle vertauscht werden ("Swizzling")? etc.).

Nach Erzeugen der Swap Chain erhalten wir Bild-Handles mit `vkGetSwapchainImagesKHR()` und Image Views für die entsprechenden Bilder mit `vkCreateImageView()`.

Man erhält erst die Handles mit dem gewohnten 2-Step Funktionsaufruf, dann iteriert man über die Liste von Bild Handles und generiert Image Views daraus.

Swap Chains

vkGetSwapchainImagesKHR

Parameter

1.) `device (VkDevice)`
Das logische Device

2.) `swapchain (VkSwapchainKHR)`
Das Swapchain Objekt

3.) `pSwapchainImageCount (uint32_t*)`
Bildanzahl

4.) `pSwapchainImages (VkImage*)`
Zurückgegebene Liste mit Bild-Handles

Swap Chains

vkGetSwapchainImagesKHR

```
// Member
std::vector<VkImage> swapchain_images_;

//...
uint32_t icount = 0;
vkGetSwapchainImagesKHR(
        device_,
        swapchain_,
        &icount,
        nullptr
        );
swapchain_images_.resize(icount);
vkGetSwapchainImagesKHR(
        device_,
        swapchain_,
        &icount,
        swapchain_images_.data()
        );
```

Swap Chains

vkCreateImageView

Parameter

1.) `device (VkDevice)`
Das logische Device

2.) `pCreateInfo (VkImageViewCreateInfo const*)`
Image View Create Info struct

3.) `pAllocator (VkAllocationCallbacks const*)`
Allocator für Host Speicher

4.) `pView (VkImageView*)`
Das zurückgegebene Image View Objekt

Swap Chains

vkCreateImageView

```
// Member, one view per image
std::vector<VkImageView> swapchain_image_views_;
swapchain_image_views_.resize(swapchain_images_.size());

for (int i = 0; i < swapchain_images_.size(); ++i)
{
  VkImageViewCreateInfo ivinfo = {};
  ivinfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
  ivinfo.image = swapchain_images_[i];
  ivinfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
  ivinfo.format = color_format_;
  //...
  VkResult res = vkCreateImageView(device_, &ivinfo,
           nullptr, swapchain_image_view_[i]);
  if (res != VK_SUCCESS) {
      ...
  }
}
```

Struktur der Vulkan Applikationsklasse bis Hierher

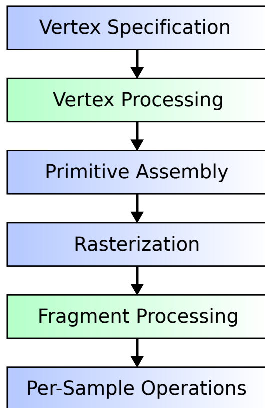
```
struct VulkanApp {
  void init() {
    // Create instance, layers, WSI surface, devices
    // Create swapchain for surface with suitable
    //      formats and presentation modes
    // obtain image handles and image views
  }

  void cleanup() {
    // Release resources, finally release instance
  }
private:
  VkInstance instance_;VkDebugReportCallbackEXT callback_;
  VkSurfaceKHR surface_;VkPhysicalDevice physical_device_;
  VkDevice device_;VkQueue graphics_queue_;
  // New:
  VkSwapchainKHR swapchain_;
  std::vector<VkImage>     swapchain_images_;
  std::vector<VkImageView> swapchain_image_views_;
};
```

Vulkan Grafik-Pipeline

Wir haben jetzt etwa den halben Weg hinter uns - als nächstes müssen wir die Grafik-Pipeline selbst aufsetzen, um tatsächlich Zeichnen zu können.

Vulkan Grafik-Pipeline



Reminder: Rasterisierungspipeline

- `VkPipeline` Objekt.
- Jeder Phase der Pipeline entspricht ein State Objekt in Vulkans Objektmodell.
  - ..**ShaderModule**..
  - ..**VertexInput**State..
  - ..**InputAssembly**State..
  - ..**Rasterization**State..
  - ..**Multisample**State..
  - ..**DepthStencil**State..
  - ..**ColorBlend**State..
  - ..
- Vulkan Pipeline starr, muss auch bei kleinen Änderungen neu erzeugt werden.

Vulkan Grafik-Pipeline

SPIR-V

Standard **P**ortable **I**ntermediate **R**epresentation for **V**ulkan.

Shader werden anders als in OpenGL nicht als Text in Binary gespeichert und zur Laufzeit kompiliert, sondern im Bytecode Format SPIR-V.

SPIR-V Code nahe an platformspezifischem Maschinencode.

Übersetze Shader Hochsprache nach SPIR-V, z. B. mit dem Tool `glslangValidator` von LunarG.

Shader Entwicklung für Vulkan in GLSL defacto Standard.

Vulkan Grafik-Pipeline

Minimaler Vertex Shader

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) in vec3 pos;

out gl_PerVertex {
    vec4 gl_Position;
};

void main() {
    gl_Position = vec4(pos, 1.0);
}
```

- "Eingebaute" Variablen im Shader, z. B.:
  - **Output** Position: `gl_Position`.
  - Vertex Index im Vertex Buffer: `gl_VertexId`.

Vulkan Grafik-Pipeline

Minimaler Fragment Shader

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) out vec4 out;

void main() {
    out = vec4(1.0, 1.0, 1.0, 1.0);
}
```

- Keine “eingebaute” Output Variable wie im Vertex Shader.
- `location = 0` bindet Output an Framebuffer 0.

Vulkan Grafik-Pipeline

SPIR-V

Der Befehl

```
glslangValidator -V frag_shade.frag
```

übersetzt beispielsweise den Fragment Shader aus der .frag Datei in SPIR-V Code (Option `-V`).

Dieser muss von Applikation als *Binärdatei* eingelesen werden. Mit dem Binärcode kann ein Objekt vom Typ `VkShaderModule` initialisiert werden.