Architektur und Programmierung von Grafik- und Koprozessoren

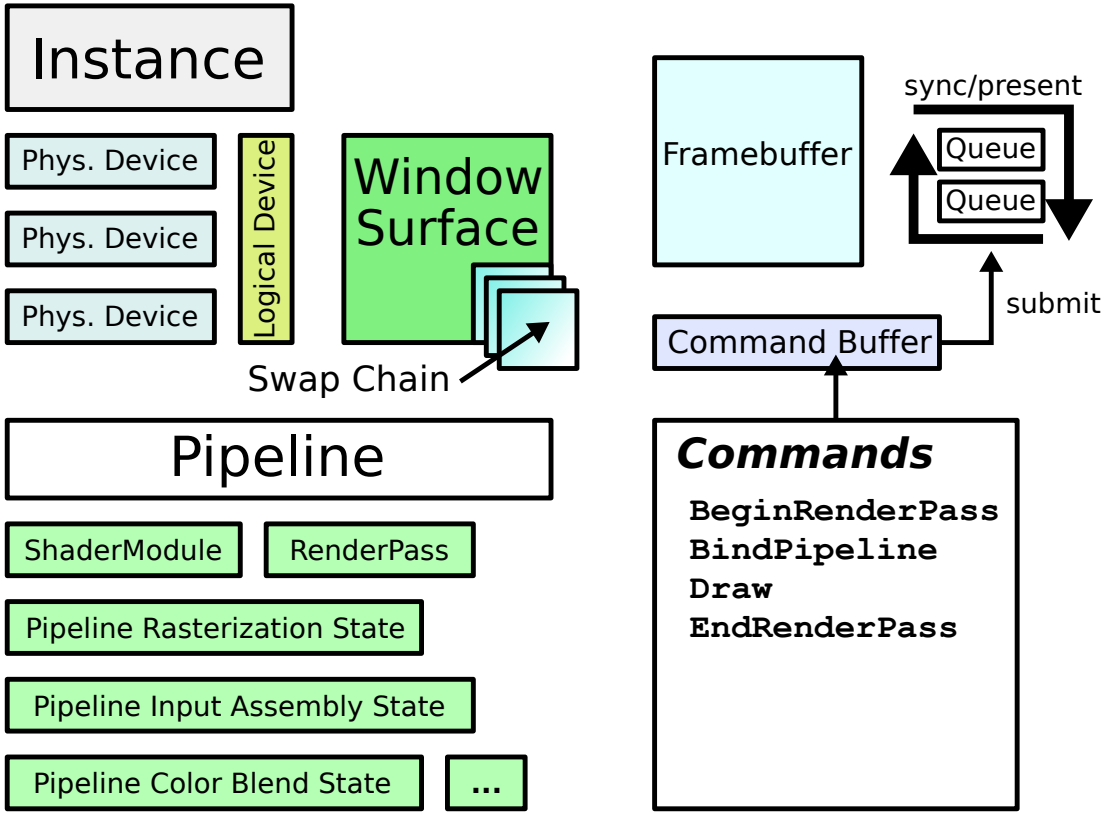
Programmieren mit dem Vulkan API

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2018

Vulkan Objektmodell Überblick




```
VkInstance
VkDevice
VkPhysicalDevice
VkPipeline
VkShaderModule
VkPipelineInput-
Assembly-
StateCreateInfo
VkCommandBuffer
VkQueue
VkFramebuffer
...
```

Vulkan Funktionsaufrufsemantik

Ressourcen

- Paare von Funktionen `vkCreateXXX()` und `vkDestroyXXX()`.
  - Vulkan Instanz, Vulkan Image, Vulkan Logical Device, Vulkan Pipeline, ...
- Übergebe Parameter an `vkCreateXXX()` mit `struct VkXXXCreateInfo`.
- Tipp: schreibe einfache RAII Klassen, die im Destruktor `vkDestroyXXX()` aufrufen.
- Hier `vkDestroyXXX()` der Kürze halber in den Beispielen nicht mit aufgeführt.

Vulkan Funktionsaufrufsemantik

CreateInfo struct

- `vkCreateXXX()` und `VkXXXCreateInfo` Parameter.
- Initialisierungsmuster immer gleich:
  - Initialisiere mit 0:
    `VkXXXCreateInfo info = {};`
  - *Reflection* mit ANSI-C API $\Rightarrow$ speichere Typdeskriptor:
    `info.sType = VK_STRUCTURE_TYPE_xxx_INFO;`
- Setze dann ressourcenspezifische Variablen.

Vulkan Funktionsaufrufsemantik

Enumeration von Ressourcen

- Einfache Enumeration: `vkGetXXX()`
  (z. B.: `vkGetPhysicalDeviceProperties`, gibt genau eine `struct` mit Eigenschaften zurück).
- Enumeration von Ressourcen mit *zweifachem* Funktionsaufruf falls *Liste*: `vkEnumerateXXX()`/`vkGetXXX()`:
  - Erster Funktionsaufruf: `vkEnumerateXXX(.., &count, nullptr)`.
  - Zweiter Funktionsaufruf: `vkEnumerateXXX(.., &count, xxx_pointer)`.
- Der erste Funktionsaufruf fragt ab, wie viele Instanzen es vom Ressourcentyp `XXX` gibt, der zweite Funktionsaufruf speichert die Instanzen in ein voralloziertes Array.

Vulkan Instanzen

Erstellt man ein Vulkan Instanz Objekt, initialisiert dies die Vulkan Library.

- Verbindungslayer zwischen Applikation und Vulkan Library.
- Enthält Informationen über die Applikation an den Treiber.
- Instanzen vs. Devices:
  - Instanzen und Devices Abstraktionen, verhalten sich sehr ähnlich: beide verfügen über Layer und Extensions.
  - Instanzen abstrahieren das gesamte Vulkan Ökosystem (Treiber, Loader etc.).
  - Devices abstrahieren ein oder mehrere physikalische Devices (GPUs).

Vulkan Instanzen

Immediate Mode APIs wie OpenGL hatten globalen State und mehrere Kontexte.

Vulkan Instanz kann als eine Art Kontext verstanden werden, der sich auf die gesamte Applikation bezieht. Device-spezifischer State wird über *logische Devices* verwaltet.

- z. B. werden über die Instanz *applikationsübergreifend* verfügbare Extensions verwaltet.
- Andere Extensions sind ggf. nur auf bestimmten Devices verfügbar, dafür gibt es logische Devices.

Zugriff auf Instanzobjekte von mehreren CPU-Threads aus muss extern synchronisiert werden, Instanz ist nicht thread-safe.

Erzeugen einer Vulkan Instanz

vkCreateInstance

Parameter

1.) `pCreateInfo (VkInstanceCreateInfo const*)`
Info struct zum Steuern, wie Instanz erzeugt wird

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

3.) `pInstance (VkInstance*)`
zurückgegebene Instanz

Erzeugen einer Vulkan Instanz

vkCreateInstance

```
VkApplicationInfo ainfo = {};
ainfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
ainfo.pApplicationName = "Vulkan Application";
ainfo.applicationVersion = VK_MAKE_VERSION(0, 0, 1);
ainfo.pEngineName = "";
ainfo.engineVersion = VK_MAKE_VERSION(0, 0, 1);
ainfo.apiVersion = VK_API_VERSION_1_1;

VkInstanceCreateInfo cinfo = {};
cinfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
cinfo.pApplicationInfo = &ainfo;

VkResult res = vkCreateInstance(&cinfo, nullptr,
                                &instance_);

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

Auflisten der Verfügbaren Vulkan Extensions

vkEnumerateInstanceExtensionProperties

Parameter

1.) `pLayerName (char const*)`
(optional): Layer Name

2.) `pPropertyCount (uint32_t*)`
Anzahl Extensions

3.) `pProperties (VkExtensionProperties*)`
Liste mit Extensions

Auflisten der Verfügbaren Vulkan Extensions

vkEnumerateInstanceExtensionProperties

```
// Query num extensions
uint32_t cnt = 0;
vkEnumerateInstanceExtensionProperties(nullptr, &cnt,
                                       nullptr);
// List of extensions
std::vector<VkExtensionProperties> extensions(cnt);
vkEnumerateInstanceExtensionProperties(nullptr, &cnt,
                                       extensions.data());
for (auto& ext : extensions)
    cout << ext.extensionName << '\n';
```

Auflisten der Verfügbaren Vulkan Extensions

vkEnumerateInstanceExtensionProperties

Beispiel-Output (Ubuntu Linux 16.04):

```
VK_EXT_acquire_xlib_display
VK_EXT_debug_report
VK_EXT_direct_mode_display
VK_EXT_display_surface_counter
VK_KHR_display
VK_KHR_get_physical_device_properties2
VK_KHR_get_surface_capabilities2
VK_KHR_surface
VK_KHR_xcb_surface
VK_KHR_xlib_surface
VK_KHR_external_fence_capabilities
VK_KHR_external_memory_capabilities
VK_KHR_external_semaphore_capabilities
VK_EXT_debug_utils
```

Extensions Aktivieren

Extensions werden beim Aufruf von `vkCreateInstance` angefordert:

```
std::vector<char const*> extensions_desired({
    VK_EXT_DEBUG_REPORT_EXTENSION_NAME,
    VK_KHR_XCB_SURFACE_EXTENSION_NAME,
    VK_KHR_SURFACE_EXTENSION_NAME
});

for (auto ext : extensions) {
    for (auto ed : extensions_desired) {
        /* check if all extensions are present */
        if (strcmp(l.extensionName, ed) == 0) {
            /* found */
        }
    }
}
cinfo.ppEnabledExtensionNames = extensions_desired.data();
cinfo.enabledExtensionCount = extensions_desired.size();
// Later:
vkCreateInstance(&cinfo, nullptr, &instance_);
```

Extensions Aktivieren

Später, wenn wir die Instanz mit den gewünschten Extensions erstellt haben, müssen wir Funktionen, die die Extension evtl. bereitstellt, noch explizit laden. Dies passiert durch einen Funktionszeiger Initialisierungsmechanismus. **Achtung**: vergisst man beim Erstellen der Instanz eine Extension, gibt der Lademechanismus später für die entsprechenden Funktionen `nullptr` zurück!

Extensions Aktivieren

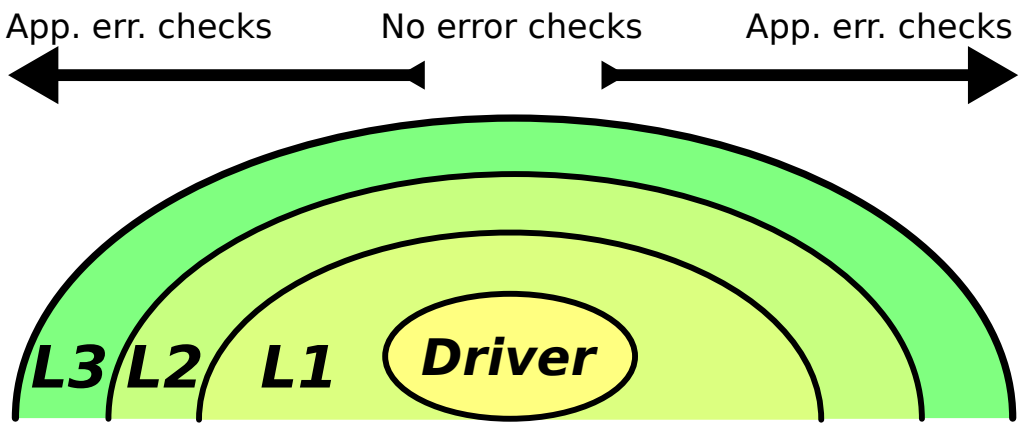
Extension Funktionen Laden mit Wrapper Funktionen:

```
VkResult myCreateXcbSurfaceKHR(
    VkInstance                          instance,
    VkXcbSurfaceCreateInfoKHR const*    pCreateInfo,
    VkAllocationCallbacks const*        pAllocator,
    VkSurfaceKHR*                       pSurface
    )
{
  auto func = (PFN_vkCreateXcbSurfaceKHR)
   vkGetInstanceProcAddr(instance,"vkCreateXcbSurfaceKHR");
  if (func != nullptr)
  {
    return func(instance,pCreateInfo,pAllocator,pSurface);
  }
  else
  {
    return VK_ERROR_EXTENSION_NOT_PRESENT;
  }
}
```

Fehlerbehandlung mit Validation Layers

- Kernprinzip von Vulkan ist minimaler Overhead durch Treiber, daher auch Fehlerbehandlung durch API und Treiber per default minimal.
- Vulkan API verlangt andererseits sehr explizite Programmierung, z. B. erwarten viele Funktionen, dass Parameter mittels **struct**s (z. B. **VkInstanceCreateInfo** von weiter oben) übergeben werden, die einzelne Parameter detailliert enumerieren. Es ist leicht, kleine Fehler zu machen.

Fehlerbehandlung mit Validation Layers



- Fast kein Error Checking durch Treiber.
  - Nur “fatal errors” (out-of-memory etc.) werden von Treiber an Applikation berichtet $\Rightarrow$ “thin API”.
- SDK kann Validation Layers einfügen (z. B. `VK_LAYER_LUNARG_standard_validation`).
- Applikation kann eigene Validation Layers einfügen.

Fehlerbehandlung mit Validation Layers

Prinzip: *wrappe* ANSI-C API Funktion, injiziere Error Checking / Error Handling Logik:

```
vkCreateInstance(VkInstanceCreateInfo const* pCreateInfo,
                 VkAllocationCallbacks const* pAllocator,
                 VkInstance* instance) {
    /* error checking */
    if (pCreateInfo == nullptr || instance == nullptr) {
        cerr << "...\n";
        return VK_ERROR_INITIALIZATION_FAILED;
    }

    /* call real vkCreateInstance() */
    return FP_vkCreateInstance(pCreateInfo, pAllocator,
                               instance);
}
```

Auflisten der Verfügbaren Vulkan Layers

vkEnumerateInstanceLayerProperties

Parameter

1.) `pPropertyCount (uint32_t*)`
Anzahl Layers

2.) `pProperties (VkLayerProperties*)`
Liste mit Layers

Auflisten der Verfügbaren Vulkan Layers

vkEnumerateInstanceLayerProperties

```
// Query num layers
uint32_t lcnt = 0;
vkEnumerateInstanceLayerProperties(&lcnt, nullptr);

// List of layers
std::vector<VkLayerProperties> layers(lcnt);
vkEnumerateInstanceLayerProperties(&lcnt, layers.data());
for (auto& layer : layers)
    cout << layer.layerName << '\n';
```

Auflisten der Verfügbaren Vulkan Layers

vkEnumerateInstanceLayerProperties

Beispiel-Output (Ubuntu Linux 16.04):

```
VK_LAYER_GOOGLE_unique_objects
VK_LAYER_GOOGLE_threading
VK_LAYER_LUNARG_core_validation
VK_LAYER_LUNARG_object_tracker
VK_LAYER_LUNARG_parameter_validation
VK_LAYER_LUNARG_standard_validation
```

Validation Layers Aktivieren

Validation Layers werden, wie Extensions, beim Aufruf von `vkCreateInstance` angefordert:

```
std::vector<char const*> layers_desired({
    "VK_LAYER_LUNARG_standard_validation"
});

for (auto l : layers) {
    for (auto ld : layers_desired) {
        /* check if all layers are present */
        if (strcmp(l.layerName, ld) == 0) {
            /* found */
        }
    }
}
cinfo.ppEnabledLayerNames = layers_desired.data();
cinfo.enabledLayerCount = layers_desired.size();
// Later:
vkCreateInstance(&cinfo, nullptr, &instance_);
```

Validation Layers Aktivieren

- Das reine *Einschalten* der Validation Layers reicht nicht, es muss eine Callback Funktion registriert werden, die Debug- und Statusmeldungen an die Applikation zurückgibt.
- Das geht mit der Extension `VK_EXT_debug_report`. Diese muss in die Liste der benötigten Extensions.

Validation Layers Aktivieren

PFN_vkCreateDebugReportCallbackEXT

Muss erst mit Extension-Lademechanismus geladen werden (setzt voraus, dass Extension `VK_EXT_debug_report` vorhanden):

```
VkResult myCreateDebugReportCallbackEXT(
    VkInstance instance,
    VkDebugReportCallbackCreateInfoEXT const* pCreateInfo,
    VkAllocationCallbacks const* pAllocator,
    VkDebugReportCallbackEXT* pCallback)
{
    auto f = (PFN_vkCreateDebugReportCallbackEXT)
        vkGetInstanceProcAddr(instance,
                              "vkCreateDebugReportCallbackEXT");
    if (f != nullptr)
        return f(instance, pCreateInfo,
                 pAllocator, pCallback);
    else
        /* error handling */
}
```

Validation Layers Aktivieren

PFN_vkDestroyDebugReportCallbackEXT

Muss erst mit Extension-Lademechanismus geladen werden (setzt voraus, dass Extension `VK_EXT_debug_report` vorhanden):

```
VkResult myDestroyDebugReportCallbackEXT(
    VkInstance instance,
    VkDebugReportCallbackEXT* pCallback,
    VkAllocationCallbacks const* pAllocator)
{
    auto f = (PFN_vkDestroyDebugReportCallbackEXT)
        vkGetInstanceProcAddr(instance,
                              "vkDestroyDebugReportCallbackEXT");
    if (f != nullptr)
        f(instance, callback, pAllocator);
    else
        /* extension not available */
}
```

Validation Layers Aktivieren

PFN_vkCreateDebugReportCallbackEXT

Parameter

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

2.) `pCreateInfo (VkDebugReportCallbackCreateInfoEXT const*)`
Info struct zum Erzeugen des Callbacks

3.) `pAllocationCallbacks (VkAllocationCallbacks const*)`
(optional): Host Speicherallokation

4.) `pCallback (VkDebugReportCallbackEXT*)`
Das zu erzeugende Callback Obekt

Validation Layers Aktivieren

PFN_vkCreateDebugReportCallbackEXT

(1/2)

```
VkDebugReportCallbackCreateInfoEXT cbinfo = {};
cbinfo.sType
 = VK_STRUCTURE_TYPE_DEBUG_REPORT_CALLBACK_CREATE_INFO_EXT;
cbinfo.flags = VK_DEBUG_REPORT_ERROR_BIT_EXT
             | VK_DEBUG_REPORT_WARNING_BIT_EXT;
cbinfo.pfnCallback = [](
        VkDebugReportFlagsEXT flags,
        VkDebugReportObjectTypeEXT obj_type,
        uint64_t obj,
        std::size_t location,
        int32_t code,
        char const* layer_prefix,
        char const* message,
        void* user_data) -> VkBool32 {
    std::cerr << message << '\n';
    return VK_FALSE;
};
```

Validation Layers Aktivieren

PFN_vkCreateDebugReportCallbackEXT

(2/2)

```
if (myCreateDebugReportCallbackEXT(instance_,
        &cbinfo, nullptr, &callback_) != VK_SUCCESS)
{
    /* error handling */
}
```

Debug Callback

Debug Callback wird immer aufgerufen, wenn zur Laufzeit falsche Verwendung des APIs festgestellt wird (falsche Enumeration übergeben, Ressourcen in der falschen Reihenfolge freigegeben, etc.).

Validation Layers und Debug Extension sollte man an Build-Typ koppeln: eingeschaltet für Debug Builds, ausgeschaltet für Release Builds (z. B. mit `#ifdef NDEBUG`).

Debug Callback

Beispiel: Falsche Verwendung von vkCreateSwapchainKHR

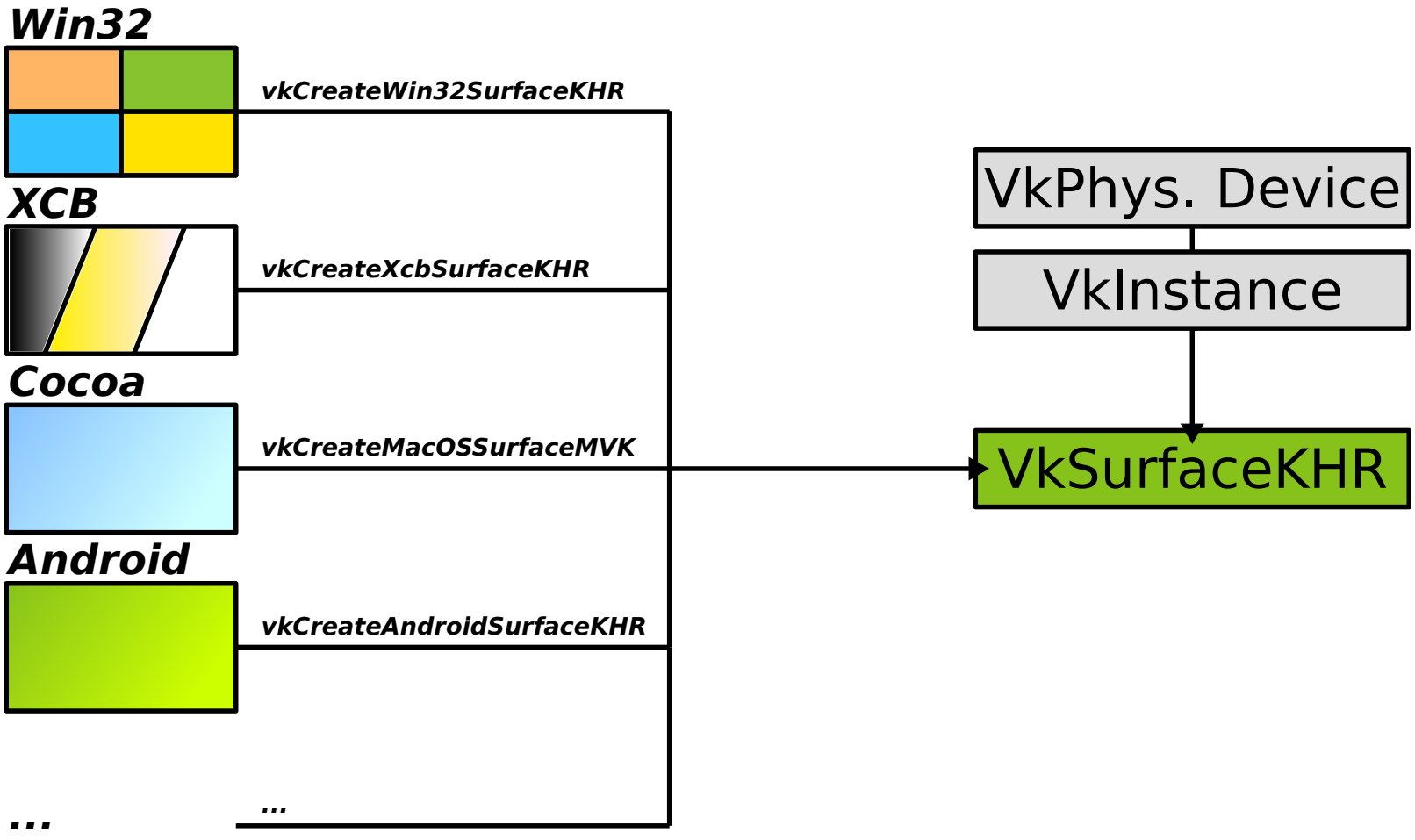
```
Object: 0x1f30500 (Type = 3) |
vkCreateSwapChainKHR() called with a non-supported
pCreateInfo->imageFormat (i.e. 37). The spec
valid usage text states 'imageFormat and
imageColorSpace must match the format and
colorSpace members, respectively, of one of
the VkSurfaceFormatKHR structures returned by
vkGetPhysicalDeviceSurfaceFormatsKHR for the
surface' (https://www.khronos.org/registry/-
vulkan/specs/1.0-extensions/html/vkspec.html-
#VUID-VkSwapchainCreateInfoKHR-imageFormat-01273)
```

Swap Chain wird mit nicht unterstütztem Farbformat verwendet. Ohne Validation Layer (oder OpenGL ohne Debug Extension) schwer zu debuggen (fehlerhaftes Bild, schwarzes Bild etc.).

Window System Integration (WSI)

- Vulkan ist cross-platform API, *interfaced* mit platformspezifischen Fenstern mittels Integration Layer Abstraktion.
- `VkSurfaceKHR` Objekt wird mit platformspezifischem Fenster assoziiert.
- Window System Integration (WSI) ist Extension, Zeichnen in Fenster ist keine Core Funktionalität.

Window System Integration (WSI)



Window System Integration (WSI)

PFN_vkCreateXXXSurfaceEEE

`vkCreateXXXSurfaceEEE` ist Extension $\Rightarrow$ muss genau wie `vkCreateDebugReportCallbackEXT` *geladen* werden (z. B. XCB für X11 Clients):

```
VkResult myCreateXcbSurfaceKHR(
    VkInstance instance,
    VkXcbSurfaceCreateInfoKHR* pCreateInfo,
    VkAllocationCallbacks const* pAllocator,
    VkSurfaceKHR* pSurface)
{
    auto f = (PFN_vkCreateXcbSurfaceKHR)
        vkGetInstanceProcAddr(instance,
                              "vkCreateXcbSurfaceKHR");
    if (f != nullptr)
        f(instance, callback, pAllocator, pSurface);
    else
        /* extension not available */
}
```

Window System Integration (WSI)

VkXcbSurfaceCreateInfoKHR

`VkXcbSurfaceCreateInfoKHR` speichert platformspezifische Informationen:

```
VkXcbSurfaceCreateInfoKHR sinfo = {};
sinfo.pType
  = VK_STRUCTURE_TYPE_XCB_SURFACE_CREATE_INFO_KHR;
sinfo.connection = viewer_.xcb_connection(); //
sinfo.window = viewer_.xcb_window(); //

VkResult res = myCreateXcbSurfaceKHR(instance_, &sinfo,
                                     nullptr, &surface_);
if (res != VK_SUCCESS) {
    ...
}
```

Window System Integration (WSI)

Das beschriebene Vorgehen setzt voraus, dass wir eine plattformspezifische Applikation mit Fenster erzeugt haben und auf plattformspezifische Handles (`HWND`, `HMODULE`, `xcb_window_t`, `xcb_connection_t` etc.) Zugriff haben.

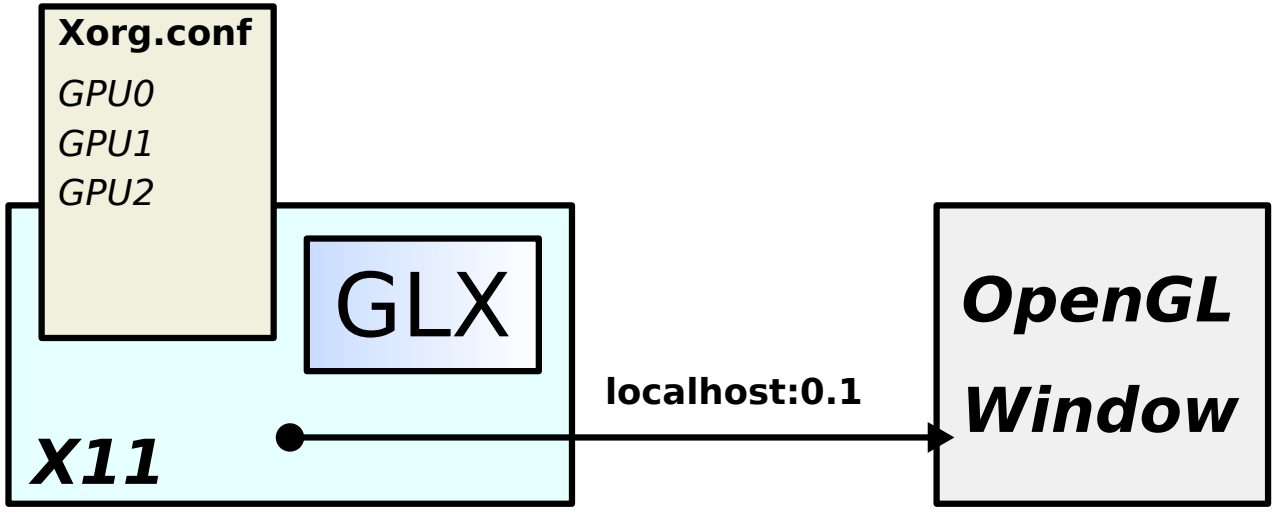
Prinzip für andere Betriebssysteme und/oder Fenstersysteme ähnlich.

Physikalische und Logische Devices

OpenGL: kein explizites physikalisches Device. Beispiel X11:

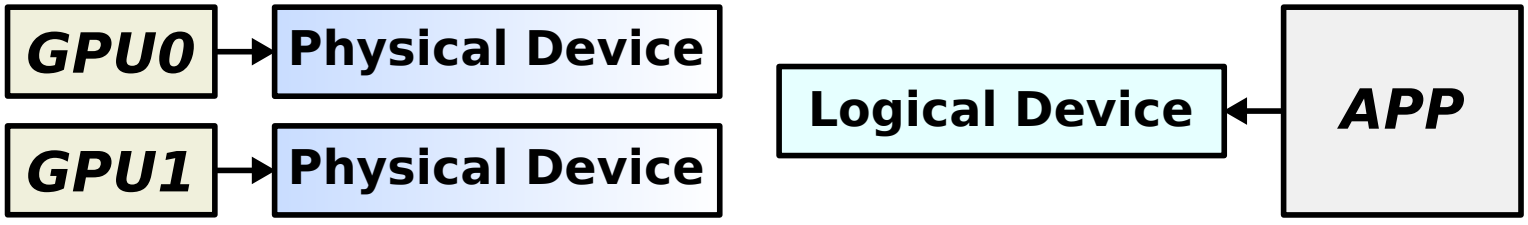
1. Konfiguriere X-Server, sodass Grafikkarte mit bestimmtem X-Screen assoziiert ist.
2. Erzeuge Fenster mit OpenGL Kontext auf diesem X-Screen.

Aus OpenGL heraus hat man keinen Zugriff auf Grafikkarte oder deren technische Spezifikation.



Physikalische und Logische Devices

Vulkan: API assoziiert Hardware mit *physikalischem Device*. Physikalisches Device kann explizit aufgrund der Spezifikation (z. B. Leistungsfähigkeit) ausgesucht werden. *Logisches Device* bildet Schnittstelle zur Applikation.



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.