Architektur und Programmierung von Grafik- und Koprozessoren

Programmieren mit dem Vulkan API

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2018

Vulkan Grafik-Pipeline

vkCreateShaderModule

Parameter

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

2.) `pCreateInfo (VkShaderModuleCreateInfo const*)`
Shader Module Create Info struct

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

4.) `pShaderModule (VkShaderModule*)`
Das zurückgegebene Shader Module Objekt

Vulkan Grafik-Pipeline

vkCreateShaderModule

```
std::vector<char> spirv = readSpirV(filename);

VkShaderModuleCreateInfo sminfo = {};
sminfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
sminfo.codeSize = code.size();
sminfo.pCode = reinterpret_cast<uint32_t const*>(
    code.data());

VkResult res = vkCreateShaderModule(device_, &sminfo,
                nullptr, &shader_module_);
if (res != VK_SUCCESS) {
    ...
}
```

Vulkan Grafik-Pipeline

Zuordnung von Shadern zu Pipeline Stages

Geschieht mit Hilfe von `VkPipelineShaderStageCreateInfo` Objekt.

```
VkPipelineShaderStageCreateInfo fsinfo = {};
fsinfo.sType
  = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
fsinfo.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
fsinfo.module = frag_shader_module_;
fsinfo.pName = "main";
```

Vulkan Grafik-Pipeline

- OpenGL: Default State für Fixed-Function Pipeline Phasen.
- Vulkan: explizites API, jeder Pipeline State muss explizit gesetzt sein, keine Defaults.
- `VkXXXCreateInfo structs` für jede Pipeline Phase.
  - Programmierbare Phasen gerade behandelt, `VkShaderModule`.

Vulkan Grafik-Pipeline

Fixed-Function Pipeline (1/3)

Vertex Specification

```
VkPipelineVertexInputStateCreateInfo viinfo = {};
viinfo.sType =
 VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
```

- Hier werden Vertex Buffer und Vertex Attribut-Buffer aufgelistet. Hier: ganz einfache Pipeline ohne Vertex Buffer.

Primitive Assembly

```
VkPipelineInputAssemblyStateCreateInfo iainfo = {};
iainfo.sType =
VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO
;
iainfo.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
```

- Andere Topologien, z. B. Triangle-Strips.

Vulkan Grafik-Pipeline

Fixed-Function Pipeline (2/3)

Scan Conversion

```
VkPipelineRasterizationStateCreateInfo rinfo = {};
rinfo.sType =
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
rinfo.polygonMode = VK_POLYGON_MODE_FILL;
rinfo.cullMode = VK_CULL_MODE_BACK_BIT;
rinfo.frontFace = VK_FRONT_FACE_CLOCKWISE;
```

- Weitere Optionen, z. B. `lineWidth`, falls `polygonMode` `VK_POLYGON_MODE_LINE` oder (!) `VK_POLYGON_MODE_POINT`.

Vulkan Grafik-Pipeline

Fixed-Function Pipeline (3/3)

Render Output

- Für jedes Framebuffer Attachment:

```
VkPipelineColorBlendAttachmentState cbas = {};
cbas.colorWriteMask
    = VK_COLOR_COMPONENT_R_BIT
    | VK_COLOR_COMPONENT_G_BIT
    | VK_COLOR_COMPONENT_B_BIT
    | VK_COLOR_COMPONENT_A_BIT;
cbas.blendEnable = VK_FALSE;
```

- Global für alle Framebuffer Attachments:

```
VkPipelineColorBlendStateCreateInfo roinfo = {};
roinfo.sType =
 VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
roinfo.logicOpEnable = VK_FALSE;
```

Vulkan Grafik-Pipeline

VkPipelineColorBlendStateCreateInfo

```
struct VkPipelineColorBlendAttachmentState
{
    VkBool32                blendEnable;
    VkBlendFactor           srcColorBlendFactor;
    VkBlendFactor           dstColorBlendFactor;
    VkBlendOp               colorBlendOp;
    VkBlendFactor           srcAlphaBlendFactor;
    VkBlendFactor           dstAlphaBlendFactor;
    VkBlendOp               alphaBlendOp;
    VkColorComponentFlags   colorWriteMask;
};
```

Vulkan Grafik-Pipeline

Viewport und Scissor Rechteck

```
VkViewport viewport = {};
viewport.x = 0.0f;
viewport.y = 0.0f;
viewport.width = (float)WIDTH; // same as swapchain
viewport.height = (float)HEIGHT; // same as swapchain
viewport.minDepth = 0.0f;
viewport.maxDepth = 1.0f;
```

- Mit Scissor Rechteck kann man Teile des Viewports vom Rendern ausnehmen. Meistens nicht benötigt, muss trotzdem explizit gesetzt sein.

```
VkRect2D scissor = {};
scissor.offset = {0, 0};
scissor.extent = { WIDTH, HEIGHT };
```

Vulkan Grafik-Pipeline

Viewport und Scissor Rechteck

Viewport und Scissor werden in gemeinsames State Objekt kompiliert.

```
VkPipelineViewportStateCreateInfo vpscinfo = {};
vpscinfo.sType
  = VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
vpscinfo.viewportCount = 1;
vpscinfo.pViewports = &viewport;
vpscinfo.scissorCount = 1;
vpscinfo.pScissors = &scissor;
```

Vulkan Grafik-Pipeline

Dynamischer Zustand

Ändert sich etwa der Viewport nie, muss dieser auch nicht dynamisch sein. Andernfalls kann man dies explizit einstellen, um nicht jedes Mal die Pipeline neu erstellen zu müssen:

```
VkDynamicState dynstates[] = {
    VK_DYNAMIC_STATE_VIEWPORT
};

VkPipelineDynamicStateCreateInfo dsinfo = {};
dsinfo.sType
  = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
dsinfo.dynamicStateCount = 1;
dsinfo.pDynamicStates = dynstates;
```

Vulkan Grafik-Pipeline

Dynamischer Zustand

- Einige weitere Zustände können alternativ als `VkDynamicState` organisiert werden:
  - `VK_DYNAMIC_STATE_VIEWPORT`
  - `VK_DYNAMIC_STATE_SCISSOR`
  - `VK_DYNAMIC_STATE_LINE_WIDTH`
  - `VK_DYNAMIC_STATE_DEPTH_BIAS`
  - `VK_DYNAMIC_STATE_BLEND_CONSTANTS`
  - `VK_DYNAMIC_STATE_DEPTH_BOUNDS`
  - `VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK`
  - `VK_DYNAMIC_STATE_STENCIL_WRITE_MASK`
  - `VK_DYNAMIC_STATE_STENCIL_REFERENCE`
  - `VK_DYNAMIC_STATE_STENCIL_W_SCALING_NV`
  - `VK_DYNAMIC_STATE_DISCARD_RECTANGLE_EXT`
  - `VK_DYNAMIC_STATE_SAMPLE_LOCATIONS_EXT`
- Sonst gilt: Zustandsänderung $\Rightarrow$ Pipeline muss neu erstellt werden.

Vulkan Grafik-Pipeline

Über *Pipeline Layouts* wird die Kommunikation zwischen der Grafik-Pipeline und den Shadern erzeugt. Jede uniforme Variable wird (später) mit einem *Descriptor Set Layout* assoziiert. Mit Hilfe eines Pipeline Layout Objekts kann aus der Grafik-Pipeline heraus auf das Descriptor Set zugegriffen werden.

Selbst wenn die verwendeten Shader keine uniformen Variablen verwenden, muss beim Erzeugen der Grafik-Pipeline ein Pipeline Layout mit der Funktion `vkCreatePipelineLayout` erstellt werden.

Die Applikation sollte das Pipeline Layout Objekt persistent halten - z. B. als Member Variable.

Vulkan Grafik-Pipeline

vkCreatePipelineLayout

Parameter

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

2.) `pCreateInfo (VkPipelineLayoutCreateInfo const*)`
Pipeline Layout Create Info struct

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

4.) `pPipelineLayout (VkPipelineLayout*)`
Das zurückgegebene Pipeline Layout Objekt

Vulkan Grafik-Pipeline

vkCreatePipelineLayout

```
VkPipelineLayoutCreateInfo plinfo = {};
plinfo.sType
  = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
// Set num descriptor set layouts
plinfo.setLayoutCount = ..;
// Set pointer to num descriptor set layouts
plinfo.pSetLayouts = ..;
VkResult res = vkCreatePipelineLayout(device_,
        &pinfo, nullptr, &pipeline_layout_);
if (res != VK_SUCCESS) {
    ...
}
```

Vulkan Grafik-Pipeline

Render Passes

- Der Vorgang, mit Vulkan in den Framebuffer zu rendern, wird über *Render Passes* abgebildet.
- Render Passes setzen sich aus *Subpasses* zusammen, z. B. kann ein Render Pass aus einer Reihe von Post-Processing Filtern bestehen, die alle in einem separatem Subpass implementiert sind.
  - $\Rightarrow$ Optimierungspotential, z. B. kann Vulkan Speicheroperationen von Subpasses zusammenführen und dadurch Speicherbandbreite sparen.
- Jedem Subpass werden Color Buffer sowie ggf. Depth/Stencil Buffer als *Attachments* mitgegeben.

Vulkan Grafik-Pipeline

Attachments

```
VkAttachmentDescription col_att = {};
// Same color format used for vkCreateSwapchainKHR
col_att.format          = color_format;
col_att.samples         = VK_SAMPLE_COUNT_1_BIT;
col_att.loadOp          = VK_ATTACHMENT_LOAD_OP_CLEAR;
col_att.storeOp         = VK_ATTACHMENT_STORE_OP_STORE;
col_att.stencilLoadOp   = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
col_att.stencilStoreOp  = VK_ATTACHMENT_STORE_OP_DONT_CARE;
col_att.initialLayout   = VK_IMAGE_LAYOUT_UNDEFINED;
col_att.finalLayout     = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
```

Vulkan Grafik-Pipeline

Subpasses und Referenzen auf Attachments

- Subpasses erhalten *Referenzen* auf Attachments.

```
// Create attachment reference
VkAttachmentReference col_att_ref = {};
col_att_ref.attachment = 0;
col_att_ref.layout
  = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

// Create subpass
VkSubpassDescription subpass = {};
// Alternative: BIND_POINT_COMPUTE (currently
// not compatible with graphics pipeline!)
subpass.pipelineBindPoint
  = VK_PIPELINE_BIND_POINT_GRAPHICS;
subpass.colorAttachmentCount = 1;
subpass.pColorAttachments = &col_att_ref;
```

Vulkan Grafik-Pipeline

Subpasses und Referenzen auf Attachments

```
col_att_ref.attachment = 0;
col_att_ref.layout
  = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
```

- Über `attachment` Index wird Framebuffer Attachment später im Fragment Shader identifiziert (vgl. minimaler Fragment Shader):

```
layout(location = 0) out vec4 out;

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

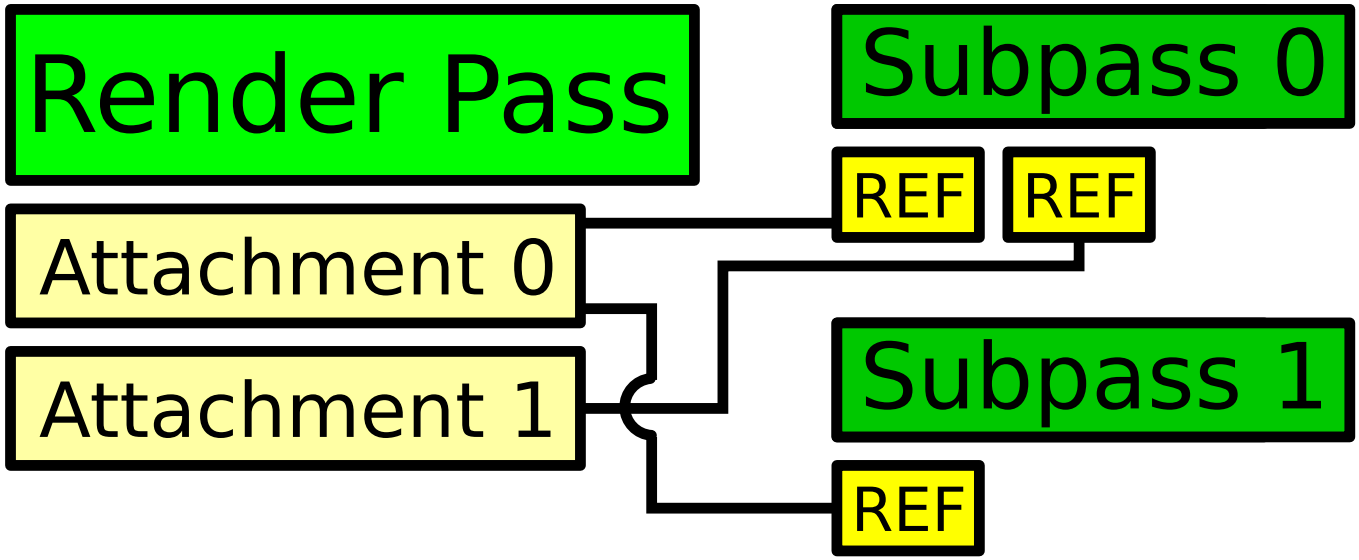
Vulkan Grafik-Pipeline

Mit `vkCreateRenderPass` werden Subpasses gebunden. die `VkRenderPass struct` speichert im wesentlichen einen Zeiger auf eine Liste von Subpasses und wird nach dem gewohnten Schema erzeugt.

Dem Render Pass wird beim Erzeugen die Liste der Attachments übergeben, die von den einzelnen Subpasses *referenziert* werden.

Das `VkRenderPass` Objekt sollte als Member Variable gespeichert werden (Attachments und Subpasses nicht notwendigerweise).

Vulkan Grafik-Pipeline



Vulkan Grafik-Pipeline

vkCreateRenderPass

Parameter

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

2.) `pCreateInfo (VkRenderPassCreateInfo const*)`
Render Pass Create Info struct

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

4.) `pRenderPass (VkRenderPass*)`
Das zurückgegebene Render Pass Objekt

Vulkan Grafik-Pipeline

vkCreateRenderPass

```
VkRenderPassCreateInfo rpinfo = {};
rpinfo.sType
  = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
rpinfo.attachmentCount = 1;
rpinfo.pAttachments = &col_att;
rpinfo.subpassCount = 1;
rpinfo.pSubpasses = &subpass;

VkResult res = vkCreateRenderPass(device_, &pinfo,
                 nullptr, &render_pass_);
if (res != VK_SUCCESS) {
    ...
}
```

Vulkan Grafik-Pipeline

Damit sind (schlussendlich) alle Vorbereitungen getroffen, um die Grafik-Pipeline erzeugen zu können. Das `VkPipeline` Objekt vereint alle Komponenten der Grafik-Pipeline:

- Shader Module (Vertex und Fragment Shader).
- Fixed-Function Pipeline Stages (Input Assembly, Scan Conversion, Viewport, Scissor etc., Blending).
- Pipeline Layout (definiert im wesentlichen Handles auf uniforme Variablen in den Shadern).
- Render Pass (beschreibt Render Subpasses und die Framebuffer Attachments, in die gezeichnet wird).

Mit `vkCreateGraphicsPipelines` wird (eine oder mehrere) Pipeline(s) erzeugt und als Member Variable(n) gespeichert.

Vulkan Grafik-Pipeline

vkCreateGraphicsPipelines

Parameter

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

2.) `pipelineCache (VkPipelineCache)`
Pipeline Cache für mehrere Programminstanzen

3.) `createInfoCount (uint32_t)`
Anzahl Create Info structs

4.) `pCreateInfos (VkGraphicsPipelineCreateInfo const*)`
Liste mit Pipeline Create Info structs

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

6.) `pPipelines (VkPipeline*)`
Liste der zurückgegebenen Pipeline Objekte

Vulkan Grafik-Pipeline

vkCreateGraphicsPipelines

```
VkGraphicsPipelineCreateInfo pplinfo = {};
pplinfo.sType
  = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
pplinfo.stageCount = 2;
// All the VkXXXCreateInfo Objects we created above..
pplinfo.pStages = shader_stages;
pplinfo.pVertexInputState = &viinfo;
pplinfo.pInputAssemblyState = &iainfo;
pplinfo.pViewportState = &vpscinfo;
pplinfo.pRasterizationState = &rinfo;
pplinfo.pColorBlendState = &roinfo;
// Pipeline layout and render pass handles
pplinfo.layout = pipeline_layout_;
pplinfo.renderPass = render_pass_;
VkResult res = vkCreateGraphicsPipelines(device_,
  VK_NULL_HANDLE,1,&pplinfo,nullptr,&graphics_pipeline_);
if (res != VK_SUCCESS) {
    ...
}
```

Vulkan Grafik-Pipeline

Pipeline Cache

Der Funktion `vkCreateGraphicsPipelines` kann ein Pipeline Cache Objekt übergeben werden (2. Parameter, im Beispiel `VK_NULL_HANDLE`), sodass die Pipeline nicht jedes mal neu kompiliert werden muss (z. B. SPIR-V nach Maschinencode).

Pipeline Cache kann als Datei gespeichert und auch über Programmausführungen hinweg wiederverwendet werden.

Struktur der Vulkan Applikationsklasse bis Hierher

(1/2): Member Variablen:

```
struct VulkanApp {
  //...
private:
  VkInstance instance_;VkDebugReportCallbackEXT callback_;
  VkSurfaceKHR surface_;VkPhysicalDevice physical_device_;
  VkDevice device_;VkQueue graphics_queue_;

  VkSwapchainKHR swapchain_;
  std::vector<VkImage>     swapchain_images_;
  std::vector<VkImageView> swapchain_image_views_;

  VkPipelineLayout pipeline_layout_;
  VkRenderPass render_pass_;
  VkPipeline graphics_pipeline_;
};
```

Struktur der Vulkan Applikationsklasse bis Hierher

(2/2): Operationen:

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

    void cleanup() {
        // Release resources, finally release instance
    }
};
```

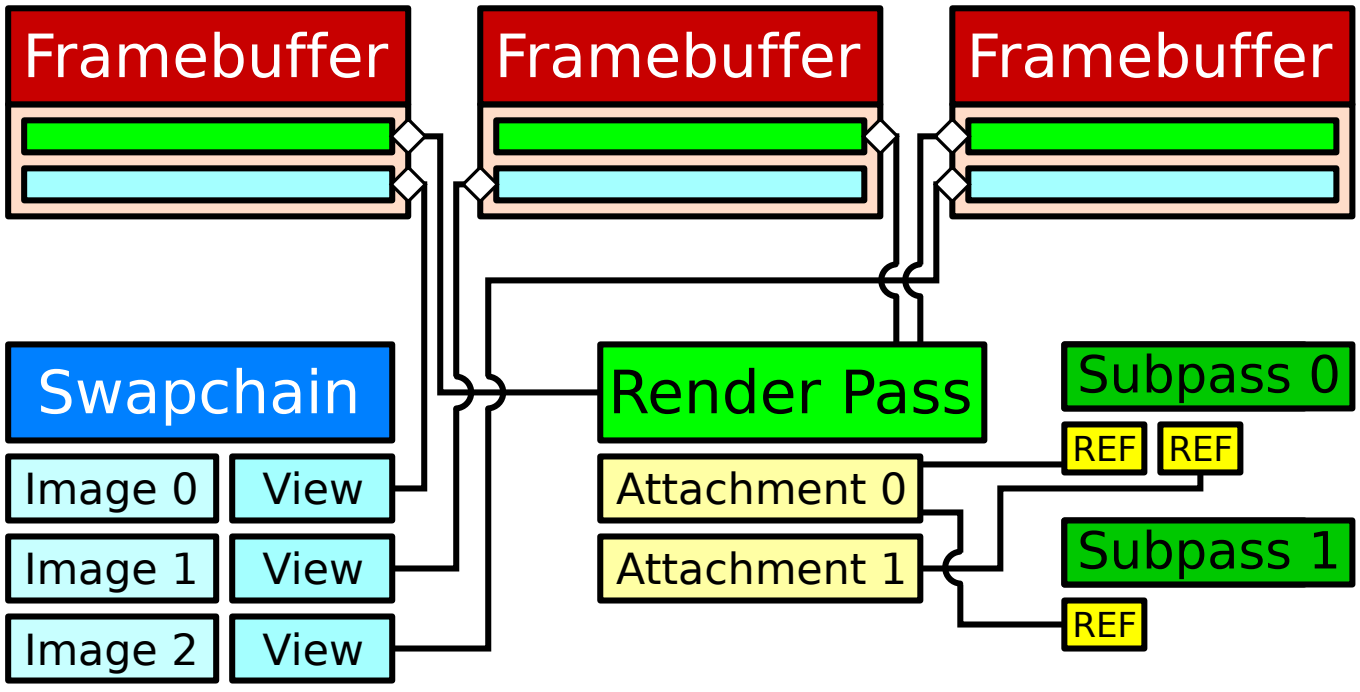
Zeichnen mit Vulkan

Nachdem die Grafik-Pipeline aufgesetzt wurde, müssen zum Zeichnen noch die folgenden Schritte durchgeführt werden:

- Erzeugen eines Framebuffers
  - Verbindet Image Views aus Swap Chain mit Attachments aus Render Pass.
- Erzeugen von *Command Pool* und *Command Buffer*, *Recording* der Zeichenkommandos in den Command Buffer.
- Synchronisierung und Submittieren des Command Buffers.

Zeichnen mit Vulkan

Framebuffer



Zeichnen mit Vulkan

Framebuffer

- Framebuffer verbinden Swapchain Image Views und Render Passes.
- Für jedes Swapchain Image ein Framebuffer. Extent von Framebuffer und Swapchain Image sollte übereinstimmen.
- Render Passes werden an die Framebuffer gebunden, zu denen sie passen (Anzahl Attachments, Attachment Typ).

Zeichnen mit Vulkan

vkCreateFramebuffer

Parameter

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

2.) `pCreateInfo (VkRenderPassCreateInfo const*)`
Framebuffer Create Info struct

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

4.) `pFramebuffer (VkFramebuffer*)`
Das zurückgegebene Framebuffer Objekt

Zeichnen mit Vulkan

vkCreateFramebuffer

```
for (auto view : swapchain_image_views_) {
    // Simple example, only color
    VkImageView attachments[] = { view };
    VkFramebufferCreateInfo fbinfo = {};
    fbinfo.sType
      = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
    fbinfo.renderPass = render_pass_;
    fbinfo.attachmentCount = 1;
    fbinfo.pAttachments = attachments;
    fbinfo.width  = 512;
    fbinfo.height = 512;
    fbinfo.layers = 1;

    VkResult res = vkCreateFramebuffer(device_,
            &fbinfo, nullptr, &framebuffers_[i]);
    if (res != VK_SUCCESS) {
        ...
    }
}
```

Command Buffers

- OpenGL: Zeichenkommandos, Speichertransferkommandos etc. direkt durch C Funktionen.
- Vulkan: Kommandos werden zu einem Command Buffer Objekt *recorded*.
  - Idee ähnlich wie Kompilieren $\Rightarrow$ einmalig hoher Aufwand, später Wiederverwendung.
- Threads verwalten ihre eigenen Command Buffer.

Command Buffers

Funktionsaufrufsemantik für Ressourcenallokation

Ressourcenallokation in Vulkan über *Pool Objekte* oder *Heaps*. Allokation über Funktionen `vkAllocateXXX()`.

Man erstellt erst einen Pool oder Heap, kann dann aus diesem allozieren und muss die Ressourcen später mit `vkFreeXXX()` wieder an den Pool oder Heap freigeben.

Semantisch analog zu Objekterstellung mit `vkCreateXXX()` und `vkDestroyXXX()`. Command Buffer werden aus *Command Pools* erzeugt.

Command Buffers

vkCreateCommandPool

Parameter

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

2.) `pCreateInfo (VkCommandPoolCreateInfo const*)`
Command Pool Create Info struct

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

4.) `pCommandPool (VkCommandPool*)`
Das zurückgegebene Command Pool Objekt

Command Buffers

vkCreateCommandPool

```
VkCommandPoolCreateInfo fbinfo = {};
cpoinfo.sType
  = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
cpoinfo.queueFamilyIndex = GraphicsQueueIdx;
cpoinfo.flags = 0;

VkResult res = vkCreateCommandPool(device_, &cpoinfo,
             nullptr, &command_pool_);
if (res != VK_SUCCESS) {
    ...
}
```

Command Buffers

vkCreateCommandPool

`VkCommandPoolCreateInfo::flags`:

Zwei verschiedene Flags, können bitweise kombiniert werden:

`VK_COMMAND_POOL_CREATE_TRANSIENT_BIT`
Tipp an Treiber, dass Command Buffer häufig neu recorded werden.

`VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT`
Überschreibt den Default, sodass einzelne Command Buffer auch individuell neu recorded werden können.

Command Buffers

Ein Command Buffer pro Framebuffer / Image View $\Rightarrow$ wir speichern einen `std::vector<VkCommandBuffer>`.

Es gibt primäre und sekundäre Command Buffer. Primäre Command Buffer sind für das Setup von Render Passes verantwortlich. Sekundäre Command Buffer werden vom primären Command Buffer aufgerufen.

Command Buffer vererben keinen Zustand an sekundäre Command Buffer.

Command Buffers

vkAllocateCommandBuffers

Parameter

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

2.) `pAllocateInfo (VkCommandBufferAllocateInfo const*)`
Command Buffer Allocate Info struct

3.) `pCommandBuffers (VkCommandBuffer*)`
Liste der allozierten Command Buffer Objekte

Command Buffers

vkAllocateCommandBuffers

```
// Member vector
std::vector<VkCommandbuffer> command_buffers_;
command_buffers_.resize(
    swapchain_image_views_.size());

VkCommandBufferAllocateInfo ainfo = {};
ainfo.sType
 = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
ainfo.commandPool = command_pool_;
ainfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
ainfo.commandBufferCount = command_buffers_.size();

VkResult res = vkAllocateCommandBuffers(device_,
                &ainfo, command_buffers_.data());
if (res != VK_SUCCESS) {
    ...
}
```

Command Buffers

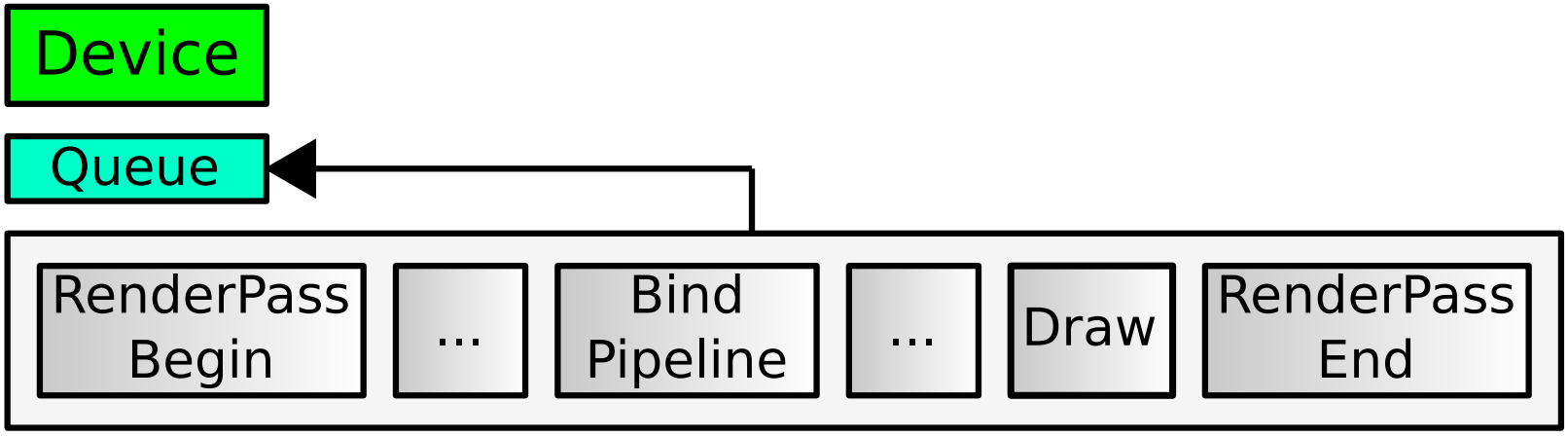


Abbildung: vgl.: Nvidia: “Engaging the Voyage to Vulkan” (https://developer.nvidia.com/engaging-voyage-vulkan)

Command Buffers

Command Buffer Recording

Kommandos werden durch Funktionen `vkCmdXXX()` aufgezeichnet.
Geben `void` zurück, da nur Recording, keine direkte Ausführung.

- Starte Recording mit `vkBeginCommandBuffer()` - für jeden Command Buffer:
  - Starte Render Pass mit `vkCmdBeginRenderPass`.
  - Binde Buffer, z. B. mit `vkCmdBindVertexBuffer()` oder `vkCmdBindIndexBuffer()`.
  - Binde Grafik-Pipeline mit `vkCmdPipeline()`.
  - Binde Layout für Shader Input Variablen mit `vkCmdBindDescriptorSets()`.
  - Zeichenkommandos: z. B. `vkCmdDraw()` oder `vkCmdDrawIndexed()`.
  - Beende Render Pass mit `vkCmdEndRenderPass()`.
- Beende Recording: `vkEndCommandBuffer()`

(Wir behandeln hier Buffer und Shader Input der Kürze halber nicht im Detail.)

Command Buffers

Command Buffer Recording - Recording Starten

```
for (int i = 0; i < command_buffers_.size(); ++i) {
    VkCommandBufferBeginInfo binfo = {};
    binfo.sType
      = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    VkResult res = vkBeginCommandBuffer(
        command_buffers_[i], &binfo);
    if (res != VK_SUCCESS) {
        ...
    }

    /* record commands here */

    res = vkEndCommandBuffers(command_buffers_[i]);
    if (res != VK_SUCCESS) {
        ...
    }
}
```

Command Buffers

Command Buffer Recording - Recording Starten

`VkCommandBufferBeginInfo::flags` (bitweise):

`VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT`
Command Buffer wird direkt nach Aufzeichung neu recorded.

`VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT`
Sekundärer Command Buffer als Teil eines einzelnen Render Passes.

`VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT`
Command Buffer kann resubmittiert werden, auch wenn die Ausführung einer früheren Instanz noch aussteht.

Command Buffers

Command Buffer Recording - Render Pass

```
VkClearValue clear_color = {
  0.0f, 0.0f, 0.0f, 0.0f };
// Begin render pass for cb[i]
VkRenderPassBeginInfo rbinfo = {};
rbinfo.sType
  = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
rbinfo = render_pass_;
rbinfo.framebuffer = framebuffers_[i];
rbinfo.renderArea.offset = { 0, 0 };
rbinfo.renderArea.extent = { 512, 512 };
rbinfo.clearValueCount = 1;
rbinfo.pClearValues = &clear_color;
vkCmdBeginRenderPass(command_buffers_, &rbinfo,
    VK_SUBPASS_CONTENTS_INLINE);

/* render pass commands here */

vkCmdEndRenderPass(command_buffers_[i]);
```

Command Buffers

`vkCmdBeginRenderPass()` dritter Parameter:

`VK_SUBPASS_CONTENTS_INLINE`
Kein sekundärer Command Buffer.

`VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS`
Render Pass Kommandos kommen aus sekundärem Command Buffer.

Command Buffers

Command Buffer Recording - Grafik-Pipeline Binden

```
vkCmdBindPipeline(command_buffers_[i],
    VK_PIPELINE_BIND_POINT_GRAPHICS,
    graphics_pipeline_);
```

- Mögliche Werte für zweiten Parameter:
  - `VK_PIPELINE_BIND_POINT_COMPUTE`
  - `VK_PIPELINE_BIND_POINT_GRAPHICS`

Command Buffers

Command Buffer Recording - Zeichenkommands

```
vkCmdDraw(command_buffers_[i],
    vertex_count,
    1, /* default for instance count */,
    0, /* vertex offset */
    0, /* instance offset */
    );
```

Struktur der Vulkan Applikationsklasse bis Hierher

(1/2): Member Variablen:

```
struct VulkanApp {
  //...
private:
  VkInstance instance_;VkDebugReportCallbackEXT callback_;
  VkSurfaceKHR surface_;VkPhysicalDevice physical_device_;
  VkDevice device_;VkQueue graphics_queue_;
  VkSwapchainKHR swapchain_;
  std::vector<VkImage>     swapchain_images_;
  std::vector<VkImageView> swapchain_image_views_;
  VkPipelineLayout pipeline_layout_;
  VkRenderPass render_pass_; VkPipeline graphics_pipeline_;

  std::vector<VkFramebuffer> framebuffers_;

  VkCommandPool command_pool_;
  std::vector<VkCommandBuffer> command_buffer_;
};
```

Struktur der Vulkan Applikationsklasse bis Hierher

(2/2): Operationen:

```
struct VulkanApp {
    void init() {
        // Create instance
        // Activate validation layers + debug CB
        // Create WSI surface
        // Choose physical device
        // Create logical device, retrieve queue handles
        // Create swapchain, images and image views
        // Create graphics pipeline
        // Create framebuffers
        // Create and record command buffer
    }

    void cleanup() {
        // Release resources, finally release instance
    }
};
```

Rendering

Schlussendlich sind die folgenden Schritte nötig, um mit Hilfe des Command Buffers ein Bild zu zeichnen:

- Auswählen eines Images aus der SwapChain:
  `vkAcquireNextImageKHR()`
- Ausführen des Command Buffers mit diesem Image als Framebuffer Attachment:
  `vkQueueSubmit()`
- Returnieren des Images in die SwapChain zur Anzeige:
  `vkQueuePresentKHR()`

Rendering

Problem: die drei Vulkan Funktionen `vkAcquireNextImageKHR()`, `vkQueueSubmit()` und `vkQueuePresentKHR()` sind *asynchron* (d. h. der Funktionsaufruf kehrt direkt zurück) und ihre Reihenfolge ist nicht vorgegeben (d. h. ohne Synchronisation wird vielleicht erst `vkQueueSubmit()` und dann `vkAcquireNextImageKHR()` ausgeführt etc.)

$\Rightarrow$ wir müssen die Funktionsaufrufe explizit synchronisieren.

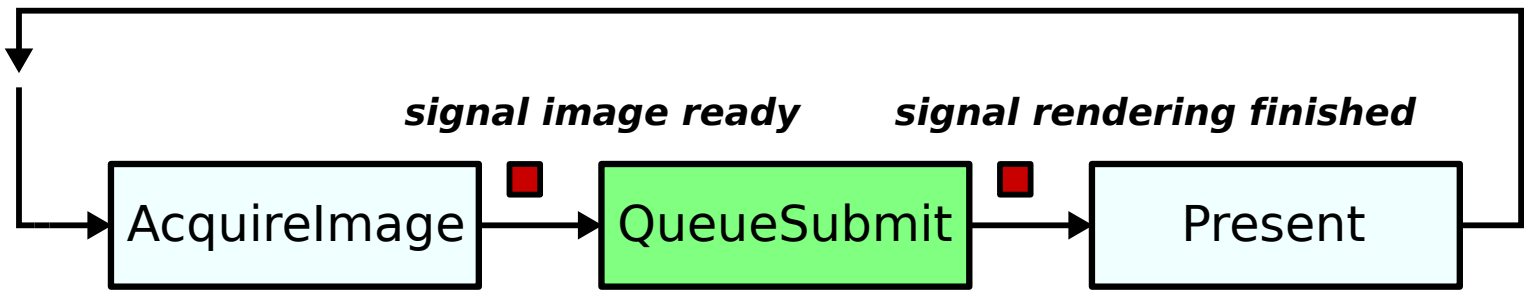
Rendering

Synchronisation

- **Fences**: (konzeptionell wie Barriers) dienen dazu, die Applikation zu synchronisieren. *Applikation* kann explizit an Fences warten (`vkWaitForFences()`), um sicherzustellen, dass Funktionsaufruf zurückgekehrt.
- **Semaphore**: dienen der Command Queue-internen Synchronisation (z. B. Zeichenkommandos und Anzeige ("Presentation")). Keine applikationsseitigen Synchronisationsprimitive (wie `vkWaitForFences()`).

Rendering

Synchronisation



Rendering

vkCreateSemaphore

Parameter

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

2.) `pCreateInfo (VkSemaphoreCreateInfo const*)`
Semaphore Create Info struct

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

4.) `pSemaphore (VkSemaphore*)`
Das zurückgegebene Semaphore Objekt

Rendering

vkCreateSemaphore

```
VkSemaphoreCreateInfo seminfo = {};
seminfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;

VkResult res = vkCreateSemaphore(device_, &seminfo,
        nullptr, &image_ready_semaphore_)

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

Rendering

vkCreateFence

Parameter

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

2.) `pCreateInfo (VkFenceCreateInfo const*)`
Fence Create Info struct

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

4.) `pFence (VkFence*)`
Das zurückgegebene Fence Objekt

Rendering

vkCreateSemaphore

```
VkSemaphoreCreateInfo finfo = {};
finfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;

VkResult res = vkCreateFence(device_, &finfo,
        nullptr, &fence_)

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

...

vkWaitForFences(device_, 1, &fence_, VK_TRUE,
    std::numeric_limits<uint64_t>::max());
vkResetFences(device_, 1, &fence_);
```

Rendering

Als erstes akquirieren wir ein Bild aus der SwapChain und registrieren die `image_ready_` Semaphore mit dem Funktionsaufruf. Diese wird signalisiert, sobald das Bild akquiriert wurde:

```
uint32_t image_index = 0;
vkAcquireNextImageKHR(
    device_,
    swap_chain_,
    std::numeric_limits<uint64_t>::max(),//timeout
    image_ready_semaphore_,//semaphore to signal (optional)
    VK_NULL_HANDLE,//fence object to wait for (optional)
    &image_index
    );
```

- Mit der Variable `image_index` kann der richtige Command Buffer zum Submittieren ausgewählt werden.

Rendering

Dann submittieren wir den Command Buffer: **(1/2)**:

```
VkSubmitInfo subinfo = {};
subinfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;

VkSemaphore wait_semaphores[]
  = { image_ready_semaphore_};
VkPipelineStageFlags wait_stages[]
  = { VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT };
subinfo.waitSemaphoreCount = 1;
subinfo.pWaitSemaphores = wait_semaphores;
subinfo.pWaitDstStageMask = wait_stages;
subinfo.commandBufferCount = 1;
subinfo.pCommandBuffers = &command_buffers_[image_index];
VkSemaphore signal_semaphores[]
  = { render_finished_semaphore_ };
subinfo.signalSemaphoreCount = 1;
subinfo.pSignalSemaphores = signal_semaphores;
```

Rendering

Dann submittieren wir den Command Buffer: **(2/2)**:

```
VkResult res = vkQueueSubmit(graphics_queue_, 1,
                             &subinfo, VK_NULL_HANDLE);

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

- Synchronisation: *warte* auf `image_ready_semaphore_`, signalisiere (a.k.a. “notify”) `render_finished_semaphore_`.
- `vkQueueSubmit()` letzter Parameter (optional): Fence Objekt, an dem gewartet werden soll.

Presentation

Zu guter Letzt zeigen wir das gerenderte Bild an ("Presentation"):

```
VkPresentInfoKHR pinfo = {};
pinfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
// Semaphores signaled by vkQueueSubmit()
pinfo.waitSemaphoreCount = 1;
pinfo.pWaitSemaphores = signal_semaphores;

VkSwapchainKHR swap_chains[] = { swap_chain_ };
pinfo.swapchainCount = 1;
pinfo.pSwapchains = swap_chains;
pinfo.pImageIndices = &image_index;

VkResult res = vkQueuePresentKHR(present_queue_, &pinfo);

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

Struktur der Vulkan Applikationsklasse

(1/2): Member Variablen:

```
struct VulkanApp {
  //...
private:
  VkInstance instance_;VkDebugReportCallbackEXT callback_;
  VkSurfaceKHR surface_;VkPhysicalDevice physical_device_;
  VkDevice device_;VkQueue graphics_queue_;
  VkSwapchainKHR swapchain_;
  std::vector<VkImage>     swapchain_images_;
  std::vector<VkImageView> swapchain_image_views_;
  VkPipelineLayout pipeline_layout_;
  VkRenderPass render_pass_; VkPipeline graphics_pipeline_;
  std::vector<VkFramebuffer> framebuffers_;
  VkCommandPool command_pool_;
  std::vector<VkCommandBuffer> command_buffer_;

  VkSemaphore image_ready_semaphore_;
  VkSemaphore render_finished_semaphore_;
};
```

Struktur der Vulkan Applikationsklasse

(2/2): Operationen:

```
struct VulkanApp {
    void init() {
        // Create instance, device etc., init WSI
        // Create swapchain, images and image views
        // Create graphics pipeline and framebuffer
        // Create and record command buffer
    }

    void render_and_present() {
        // Acquire image
        // (sync) submit (sync)
        // Present
    }

    void cleanup() {
        // Release all resources
    }
};
```

Bemerkungen

- Das Programm verwendet keine Vertex Buffer. Dann wäre der Vortrag noch deutlich komplexer (Buffer erstellen, Speicher allozieren, Buffer an Speicher binden, Descriptor Sets und Descriptor Set Layouts erzeugen, um in Shader auf Buffer zugreifen zu können).
- Dennoch kann man mit einem einfachen Vertex Shader (z. B. Vertices hardcodiert als Konstanten eine einfache Ausgabe erzeugen.
- Das Programm bis hierher erstreckt sich mit Strukturierung auf etwa 1000 Zeilen Quellcode.

Recap

- Vulkan exponiert äußerst komplexes Objektmodell $\Rightarrow$ Kontrolle über alle Pipeline Stages auf der GPU.
- Vulkan Programmiermodell *explizit*, es gibt keine Defaults $\Rightarrow$ Code Komplexität und Fehleranfälligkeit.
- Einsatz von Vulkan sinnvoll, wenn Applikation CPU-bound ist.
- Objektmodell von Vulkan recht nah an GPU Architekturen $\Rightarrow$ geeignet, um GPU Architekturen besser zu verstehen.

Literaturempfehlungen

- Pawel Lapinski: Vulkan Cookbook: Work through recipes to unlock the full potential of the next generation graphics API - Vulkan: Solutions to next gen 3D graphics API (2017).
- Online Quelle: `https://vulkan-tutorial.com` (Alexander Overvoorde).