Departamento de Ciência da COMPUTAÇÃO

UDESC Joinville

André Tavares da Silva

andre.silva@udesc.br

https://www.vulkanprogrammingguide.com/

Diferença da Vulkan para OpenGL

- Mais informações de baixo nível devem ser fornecidas (por você!) na aplicação, em vez do *driver*;
- Nenhum “estado atual” é preservada no *driver*;
- Tudo considerado obsoleto (*deprecated*) no OpenGL são obsoletas de fato na Vulkan (inexistem): transformações de pipeline integradas, begin-vertex*-end, etc.
- Você deve gerenciar suas próprias transformações;
- Todas as funcionalidades de transformação, cor e textura devem ser feitas em *shaders*;
- *Shaders* são "pré-compilados" fora do aplicativo. O processo de compilação é concluído durante o processo de construção do pipeline em tempo de execução;

Diferença da Vulkan para OpenGL

- No OpenGL, seu "estado de pipeline" é a combinação de quaisquer atributos gráficos atuais são: cor, transformações, texturas, shaders, etc.
- É muito caro mudar o estado on-the-fly um item por vez no OpenGL;
- Vulkan força você a definir todas as suas variáveis de estado de uma só vez em uma estrutura “Pipeline State Object” (PSO) e invoca a PSO de uma vez sempre que quiser usar essa combinação de estado;
- Potencialmente, você pode ter milhares desses PSO.

Diagrama de Blocos Geral

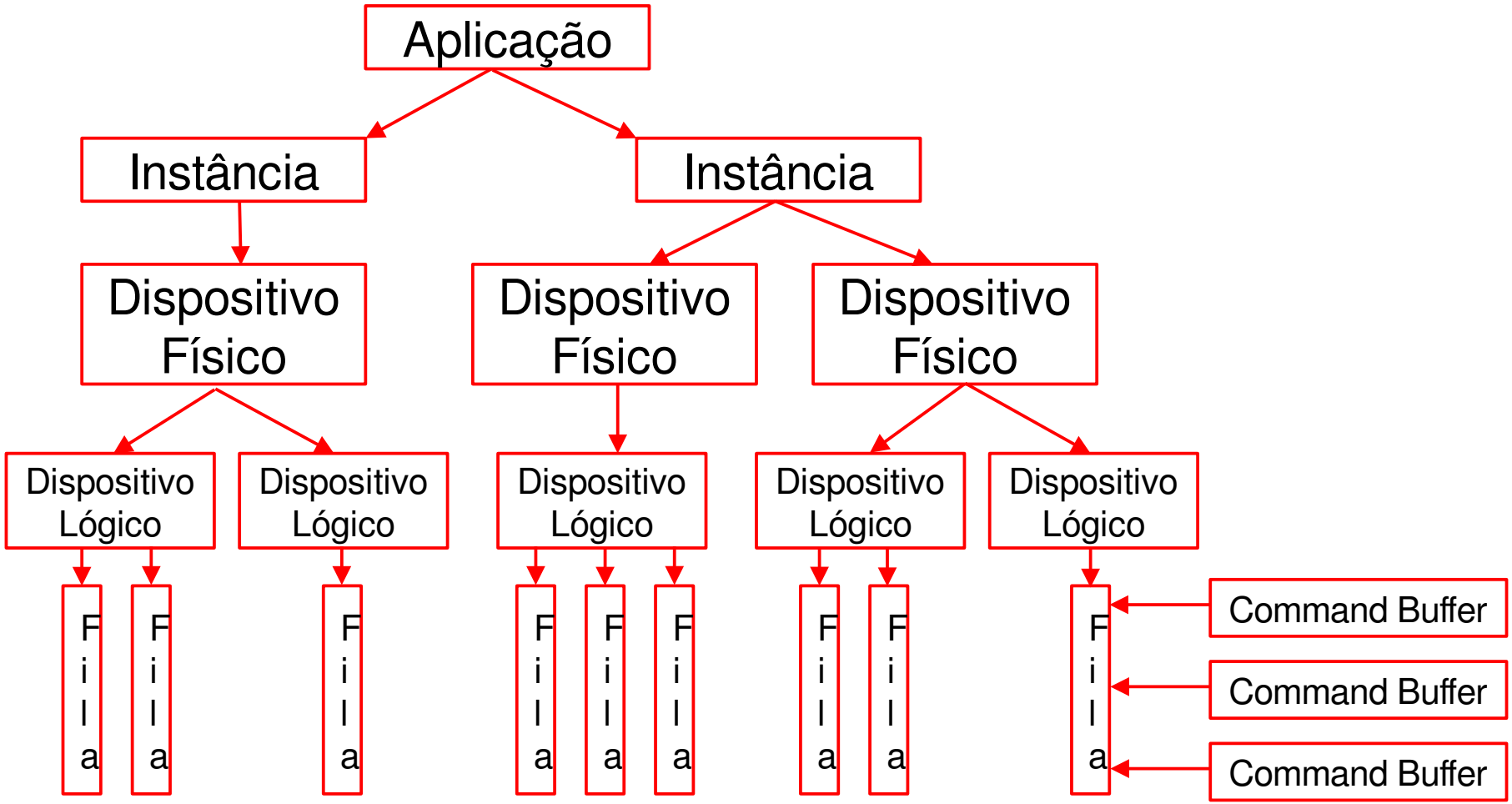
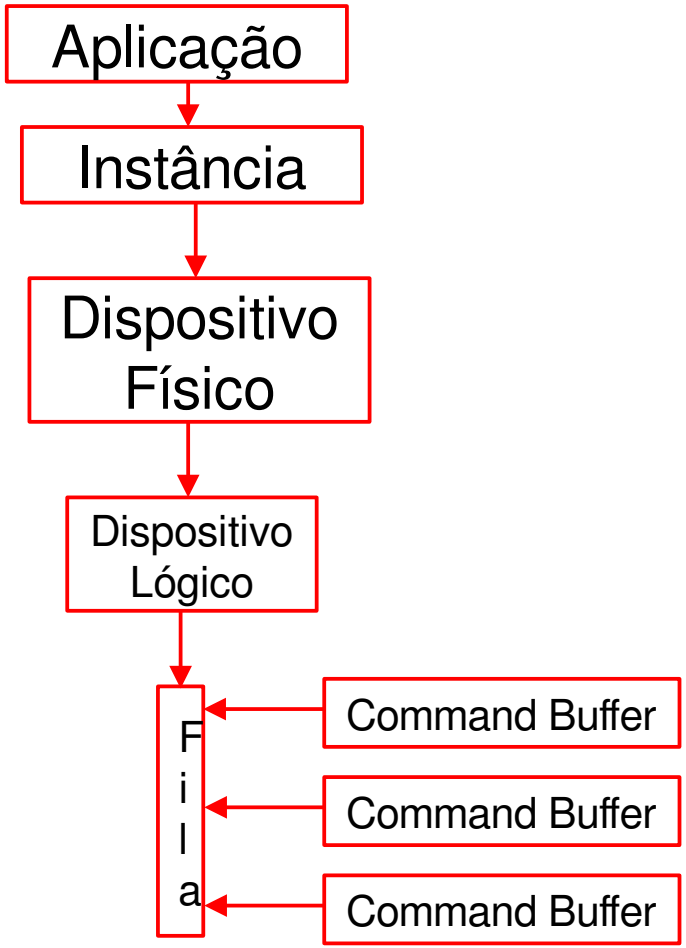


Diagrama de Blocos Típico



Passos para rendering usando Vulkan

1. Criar a instância Vulkan
2. Configurar chamada de depuração (Debug Callbacks)
3. Crie a superfície
4. Liste os dispositivos físicos
5. Escolha o dispositivo físico certo
6. Crie o dispositivo lógico
7. Crie a Window Surface
8. Crie a Swap Chain
9. Leia os Shaders
10. Crie Fixed Functions
11. Crie Render Passes
12. Crie os Framebuffers
13. Crie Command Buffers
14. Rendering
15. Múltiplos frames in-flight

Código Base GLFW


```cpp
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>

#include <iostream>
#include <stdexcept>
#include <cstdlib>

const uint32_t WIDTH = 800;
const uint32_t HEIGHT = 600;

class HelloTriangleApplication {
public:
    void run() {
        initWindow();
        initVulkan();
        mainLoop();
        cleanup();
    }

private:
    GLFWwindow* window;

    void initWindow() {
        glfwInit();

        glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
        glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);

        window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
    }

    void initVulkan() {

    }

    void mainLoop() {
        while (!glfwWindowShouldClose(window)) {
            glfwPollEvents();
        }
    }

    void cleanup() {
        glfwDestroyWindow(window);

        glfwTerminate();
    }
};
```

```cpp
int main() {
    HelloTriangleApplication app;

    try {
        app.run();
    } catch (const std::exception& e) {
        std::cerr << e.what() << std::endl;
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Fonte: https://vulkan-tutorial.com

Código Base GLFW

GLFW lê automaticamente os cabeçalhos Vulkan

```cpp
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>

#include <iostream>
#include <stdexcept>
#include <cstdlib>

const uint32_t WIDTH = 800;
const uint32_t HEIGHT = 600;

class HelloTriangleApplication {
public:
    void run() {
        initWindow();
        initVulkan();
        mainLoop();
        cleanup();
    }

private:
    GLFWwindow* window;

    void initWindow() {
        glfwInit();

        glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
        glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);

        window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
    }

    void initVulkan() {

    }

    void mainLoop() {
        while (!glfwWindowShouldClose(window)) {
            glfwPollEvents();
        }
    }

    void cleanup() {
        glfwDestroyWindow(window);

        glfwTerminate();
    }
};
```

```cpp
int main() {
    HelloTriangleApplication app;

    try {
        app.run();
    } catch (const std::exception& e) {
        std::cerr << e.what() << std::endl;
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Fonte: https://vulkan-tutorial.com

Código Base GLFW


```cpp
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>

#include <iostream>
#include <stdexcept>
#include <cstdlib>

const uint32_t WIDTH = 800;
const uint32_t HEIGHT = 600;

class HelloTriangleApplication {
public:
    void run() {
        initWindow();
        initVulkan();
        mainLoop();
        cleanup();
    }

private:
    GLFWwindow* window;

    void initWindow() {
        glfwInit();

        glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
        glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);

        window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
    }

    void initVulkan() {

    }

    void mainLoop() {
        while (!glfwWindowShouldClose(window)) {
            glfwPollEvents();
        }
    }

    void cleanup() {
        glfwDestroyWindow(window);

        glfwTerminate();
    }
};
```

Evita que GLFW crie contexto OpenGL

```cpp
int main() {
    HelloTriangleApplication app;

    try {
        app.run();
    } catch (const std::exception& e) {
        std::cerr << e.what() << std::endl;
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Fonte: https://vulkan-tutorial.com

Passos para rendering usando Vulkan

1. **Criar a instância Vulkan**
2. Configurar chamada de depuração (Debug Callbacks)
3. Crie a superfície
4. Liste os dispositivos físicos
5. Escolha o dispositivo físico certo
6. Crie o dispositivo lógico
7. Crie a Window Surface
8. Crie a Swap Chain
9. Leia os Shaders
10. Crie Fixed Functions
11. Crie Render Passes
12. Crie os Framebuffers
13. Crie Command Buffers
14. Rendering
15. Múltiplos frames in-flight

Instância

Cria uma Instância Vulkan

```
void initVulkan() {
    createInstance();
}

void createInstance() {
    VkApplicationInfo appInfo{};
    appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
    appInfo.pApplicationName = "Hello Triangle";
    appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
    appInfo.pEngineName = "No Engine";
    appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
    appInfo.apiVersion = VK_API_VERSION_1_0;

    VkInstanceCreateInfo createInfo{};
    createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
    createInfo.pApplicationInfo = &appInfo;

    uint32_t glfwExtensionCount = 0;
    const char** glfwExtensions;
    glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);

    createInfo.enabledExtensionCount = glfwExtensionCount;
    createInfo.ppEnabledExtensionNames = glfwExtensions;

    createInfo.enabledLayerCount = 0;

    if (vkCreateInstance(&createInfo, nullptr, &instance) != VK_SUCCESS) {
        throw std::runtime_error("failed to create instance!");
    }
}

void cleanup() {
    vkDestroyInstance(instance, nullptr);

    glfwDestroyWindow(window);

    glfwTerminate();
}
```

Fonte: https://vulkan-tutorial.com

Instância


```
void initVulkan() {
    createInstance();
}

void createInstance() {
    VkApplicationInfo appInfo{};
    appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
    appInfo.pApplicationName = "Hello Triangle";
    appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
    appInfo.pEngineName = "No Engine";
    appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
    appInfo.apiVersion = VK_API_VERSION_1_0;

    VkInstanceCreateInfo createInfo{};
    createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
    createInfo.pApplicationInfo = &appInfo;

    uint32_t glfwExtensionCount = 0;
    const char** glfwExtensions;
    glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);

    createInfo.enabledExtensionCount = glfwExtensionCount;
    createInfo.ppEnabledExtensionNames = glfwExtensions;

    createInfo.enabledLayerCount = 0;

    if (vkCreateInstance(&createInfo, nullptr, &instance) != VK_SUCCESS) {
        throw std::runtime_error("failed to create instance!");
    }
}

void cleanup() {
    vkDestroyInstance(instance, nullptr);

    glfwDestroyWindow(window);

    glfwTerminate();
}
```

Define quais extensões e camadas de validação serão usadas

Fonte: https://vulkan-tutorial.com

Instância


```
void initVulkan() {
    createInstance();
}

void createInstance() {
    VkApplicationInfo appInfo{};
    appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
    appInfo.pApplicationName = "Hello Triangle";
    appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
    appInfo.pEngineName = "No Engine";
    appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
    appInfo.apiVersion = VK_API_VERSION_1_0;

    VkInstanceCreateInfo createInfo{};
    createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
    createInfo.pApplicationInfo = &appInfo;

    uint32_t glfwExtensionCount = 0;
    const char** glfwExtensions;
    glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);

    createInfo.enabledExtensionCount = glfwExtensionCount;
    createInfo.ppEnabledExtensionNames = glfwExtensions;

    createInfo.enabledLayerCount = 0;

    if (vkCreateInstance(&createInfo, nullptr, &instance) != VK_SUCCESS) {
        throw std::runtime_error("failed to create instance!");
    }
}
```

Instâncias devem ser destruídas logo antes da saída do programa.

```
void cleanup() {
    vkDestroyInstance(instance, nullptr);

    glfwDestroyWindow(window);

    glfwTerminate();
}
```

Fonte: https://vulkan-tutorial.com

Instância

Define se o programa será compilado com depuração ou não

```
const uint32_t WIDTH = 800;
const uint32_t HEIGHT = 600;

const std::vector<const char*> validationLayers = {
    "VK_LAYER_KHRONOS_validation"
};

#ifdef NDEBUG
const bool enableValidationLayers = false;
#else
const bool enableValidationLayers = true;
#endif
```

```
    bool checkValidationLayerSupport() {
        uint32_t layerCount;
        vkEnumerateInstanceLayerProperties(&layerCount, nullptr);

        std::vector<VkLayerProperties> availableLayers(layerCount);
        vkEnumerateInstanceLayerProperties(&layerCount, availableLayers.data());

        for (const char* layerName : validationLayers) {
            bool layerFound = false;

            for (const auto& layerProperties : availableLayers) {
                if (strcmp(layerName, layerProperties.layerName) == 0) {
                    layerFound = true;
                    break;
                }
            }

            if (!layerFound) {
                return false;
            }
        }

        return true;
    }
```

Fonte: https://vulkan-tutorial.com

Instância


```
const uint32_t WIDTH = 800;
const uint32_t HEIGHT = 600;

const std::vector<const char*> validationLayers = {
    "VK_LAYER_KHRONOS_validation"
};

#ifdef NDEBUG
const bool enableValidationLayers = false;
#else
const bool enableValidationLayers = true;
#endif
```

Verifica se todas as camadas necessárias estão disponíveis

```
    bool checkValidationLayerSupport() {
        uint32_t layerCount;
        vkEnumerateInstanceLayerProperties(&layerCount, nullptr);

        std::vector<VkLayerProperties> availableLayers(layerCount);
        vkEnumerateInstanceLayerProperties(&layerCount, availableLayers.data());

        for (const char* layerName : validationLayers) {
            bool layerFound = false;

            for (const auto& layerProperties : availableLayers) {
                if (strcmp(layerName, layerProperties.layerName) == 0) {
                    layerFound = true;
                    break;
                }
            }

            if (!layerFound) {
                return false;
            }
        }

        return true;
    }
```

Fonte: https://vulkan-tutorial.com

Instância


```
std::vector<const char*> getRequiredExtensions() {
    uint32_t glfwExtensionCount = 0;
    const char** glfwExtensions;
    glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);

    std::vector<const char*> extensions(glfwExtensions, glfwExtensions + glfwExtensionCount);

    if (enableValidationLayers) {
        extensions.push_back(VK_EXT_DEBUG_UTILS_EXTENSION_NAME);
    }

    return extensions;
}
```

Exibe a lista de extensões baseadas no que está habilitado ou não na camada de validação.

Executar: 02_validation_layers

Passos para rendering usando Vulkan

1. Criar a instância Vulkan
2. Configurar chamada de depuração (Debug Callbacks)
3. Crie a superfície
4. **Liste os dispositivos físicos**
5. **Escolha o dispositivo físico certo**
6. Crie o dispositivo lógico
7. Crie a Window Surface
8. Crie a Swap Chain
9. Leia os Shaders
10. Crie Fixed Functions
11. Crie Render Passes
12. Crie os Framebuffers
13. Crie Command Buffers
14. Rendering
15. Múltiplos frames in-flight

Dispositivos Físicos

Se não tem nenhum dispositivo compatível, aborta a execução

```
void pickPhysicalDevice() {
    uint32_t deviceCount = 0;
    vkEnumeratePhysicalDevices(instance, &deviceCount, nullptr);

    if (deviceCount == 0) {
        throw std::runtime_error("failed to find GPUs with Vulkan support!");
    }

    std::vector<VkPhysicalDevice> devices(deviceCount);
    vkEnumeratePhysicalDevices(instance, &deviceCount, devices.data());

    for (const auto& device : devices) {
        if (isDeviceSuitable(device)) {
            physicalDevice = device;
            break;
        }
    }

    if (physicalDevice == VK_NULL_HANDLE) {
        throw std::runtime_error("failed to find a suitable GPU!");
    }
}
```

Fonte: https://vulkan-tutorial.com

Dispositivos Físicos


```
void pickPhysicalDevice() {
    uint32_t deviceCount = 0;
    vkEnumeratePhysicalDevices(instance, &deviceCount, nullptr);

    if (deviceCount == 0) {
        throw std::runtime_error("failed to find GPUs with Vulkan support!");
    }
```

Caso positivo, aloca um array para todos os VkPhysicalDevice

```
    std::vector<VkPhysicalDevice> devices(deviceCount);
    vkEnumeratePhysicalDevices(instance, &deviceCount, devices.data());

    for (const auto& device : devices) {
        if (isDeviceSuitable(device)) {
            physicalDevice = device;
            break;
        }
    }

    if (physicalDevice == VK_NULL_HANDLE) {
        throw std::runtime_error("failed to find a suitable GPU!");
    }
}
```

Fonte: https://vulkan-tutorial.com

Dispositivos Físicos


```
void pickPhysicalDevice() {
    uint32_t deviceCount = 0;
    vkEnumeratePhysicalDevices(instance, &deviceCount, nullptr);

    if (deviceCount == 0) {
        throw std::runtime_error("failed to find GPUs with Vulkan support!");
    }

    std::vector<VkPhysicalDevice> devices(deviceCount);
    vkEnumeratePhysicalDevices(instance, &deviceCount, devices.data());
```

Verifica se o dispositivo físico tem todos os requisitos

```
    for (const auto& device : devices) {
        if (isDeviceSuitable(device)) {
            physicalDevice = device;
            break;
        }
    }

    if (physicalDevice == VK_NULL_HANDLE) {
        throw std::runtime_error("failed to find a suitable GPU!");
    }
}
```

Fonte: https://vulkan-tutorial.com

Passos para rendering usando Vulkan

1. Criar a instância Vulkan
2. Configurar chamada de depuração (Debug Callbacks)
3. Crie a superfície
4. Liste os dispositivos físicos
5. Escolha o dispositivo físico certo
6. **Crie o dispositivo lógico**
7. Crie a Window Surface
8. Crie a Swap Chain
9. Leia os Shaders
10. Crie Fixed Functions
11. Crie Render Passes
12. Crie os Framebuffers
13. Crie Command Buffers
14. Rendering
15. Múltiplos frames in-flight

Dispositivos Lógicos e filas

Cria a fila e o dispositivo lógico

```
void createLogicalDevice() {
    QueueFamilyIndices indices = findQueueFamilies(physicalDevice);

    VkDeviceQueueCreateInfo queueCreateInfo{};
    queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
    queueCreateInfo.queueFamilyIndex = indices.graphicsFamily.value();
    queueCreateInfo.queueCount = 1;

    float queuePriority = 1.0f;
    queueCreateInfo.pQueuePriorities = &queuePriority;

    VkPhysicalDeviceFeatures deviceFeatures{};

    VkDeviceCreateInfo createInfo{};
    createInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;

    createInfo.pQueueCreateInfos = &queueCreateInfo;
    createInfo.queueCreateInfoCount = 1;

    createInfo.pEnabledFeatures = &deviceFeatures;

    createInfo.enabledExtensionCount = 0;

    if (enableValidationLayers) {
        createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
        createInfo.ppEnabledLayerNames = validationLayers.data();
    } else {
        createInfo.enabledLayerCount = 0;
    }

    if (vkCreateDevice(physicalDevice, &createInfo, nullptr, &device) != VK_SUCCESS) {
        throw std::runtime_error("failed to create logical device!");
    }

    vkGetDeviceQueue(device, indices.graphicsFamily.value(), 0, &graphicsQueue);
}
```

Fonte: https://vulkan-tutorial.com

Dispositivos Lógicos e filas


```
void createLogicalDevice() {
    QueueFamilyIndices indices = findQueueFamilies(physicalDevice);

    VkDeviceQueueCreateInfo queueCreateInfo{};
    queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
    queueCreateInfo.queueFamilyIndex = indices.graphicsFamily.value();
    queueCreateInfo.queueCount = 1;

    float queuePriority = 1.0f;
    queueCreateInfo.pQueuePriorities = &queuePriority;

    VkPhysicalDeviceFeatures deviceFeatures{};

    VkDeviceCreateInfo createInfo{};
    createInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;

    createInfo.pQueueCreateInfos = &queueCreateInfo;
    createInfo.queueCreateInfoCount = 1;

    createInfo.pEnabledFeatures = &deviceFeatures;

    createInfo.enabledExtensionCount = 0;

    if (enableValidationLayers) {
        createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
        createInfo.ppEnabledLayerNames = validationLayers.data();
    } else {
        createInfo.enabledLayerCount = 0;
    }

    if (vkCreateDevice(physicalDevice, &createInfo, nullptr, &device) != VK_SUCCESS) {
        throw std::runtime_error("failed to create logical device!");
    }

    vkGetDeviceQueue(device, indices.graphicsFamily.value(), 0, &graphicsQueue);
}
```

Para compatibilidade com versões anteriores da Vulkan

Fonte: https://vulkan-tutorial.com

Passos para rendering usando Vulkan

1. Criar a instância Vulkan
2. Configurar chamada de depuração (Debug Callbacks)
3. Crie a superfície
4. Liste os dispositivos físicos
5. Escolha o dispositivo físico certo
6. Crie o dispositivo lógico
7. **Crie a Window Surface**
8. Crie a Swap Chain
9. Leia os Shaders
10. Crie Fixed Functions
11. Crie Render Passes
12. Crie os Framebuffers
13. Crie Command Buffers
14. Rendering
15. Múltiplos frames in-flight

Window Surface

Chamada GLFW para criação da área da janela (Window Surface)

```
void createSurface() {
    if (glfwCreateWindowSurface(instance, window, nullptr, &surface) != VK_SUCCESS) {
        throw std::runtime_error("failed to create window surface!");
    }
}
```

Criando a fila de apresentação

```
QueueFamilyIndices findQueueFamilies(VkPhysicalDevice device) {
    QueueFamilyIndices indices;

    uint32_t queueFamilyCount = 0;
    vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, nullptr);

    std::vector<VkQueueFamilyProperties> queueFamilies(queueFamilyCount);
    vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, queueFamilies.data());

    int i = 0;
    for (const auto& queueFamily : queueFamilies) {
        if (queueFamily.queueFlags & VK_QUEUE_GRAPHICS_BIT) {
            indices.graphicsFamily = i;
        }

        VkBool32 presentSupport = false;
        vkGetPhysicalDeviceSurfaceSupportKHR(device, i, surface, &presentSupport);

        if (presentSupport) {
            indices.presentFamily = i;
        }

        if (indices.isComplete()) {
            break;
        }

        i++;
    }

    return indices;
}
```

Fonte: https://vulkan-tutorial.com

Passos para rendering usando Vulkan

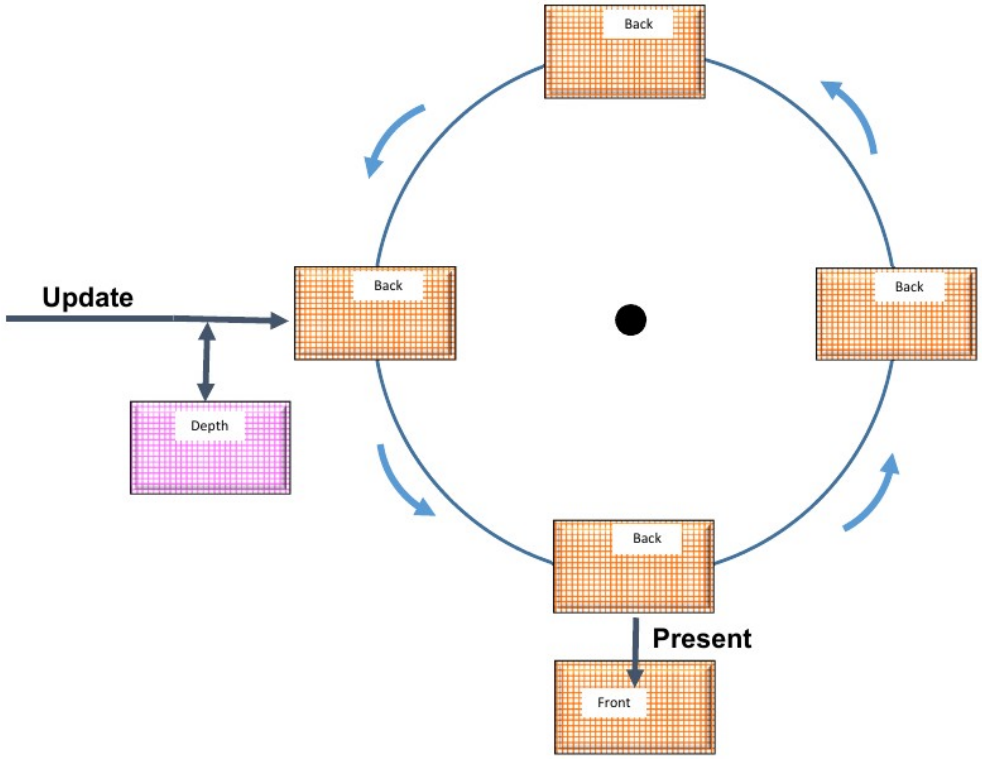
1. Criar a instância Vulkan
2. Configurar chamada de depuração (Debug Callbacks)
3. Crie a superfície
4. Liste os dispositivos físicos
5. Escolha o dispositivo físico certo
6. Crie o dispositivo lógico
7. Crie a Window Surface
8. **Crie a Swap Chain**
9. Leia os Shaders
10. Crie Fixed Functions
11. Crie Render Passes
12. Crie os Framebuffers
13. Crie Command Buffers
14. Rendering
15. Múltiplos frames in-flight

Swap Chain

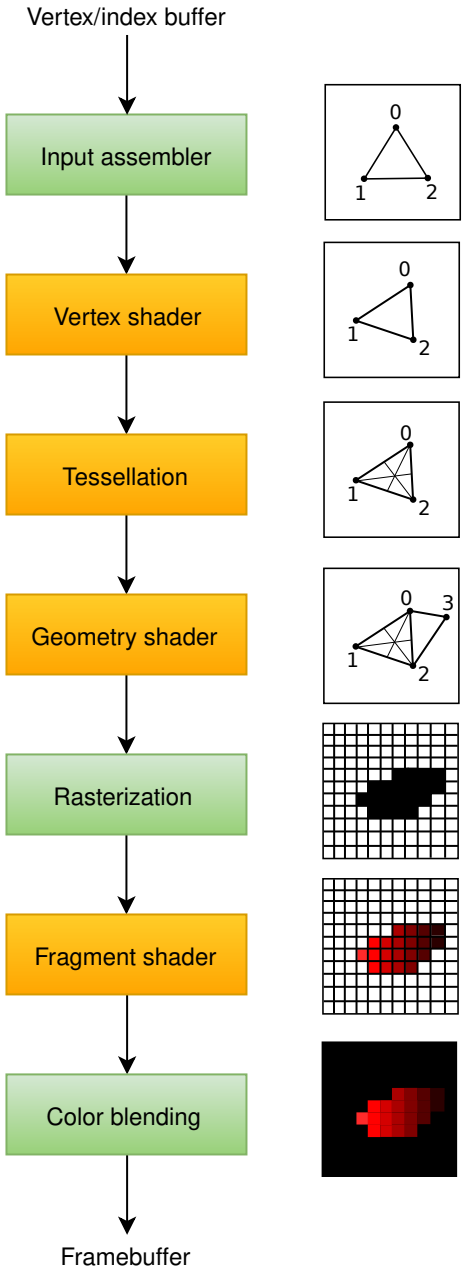
- O Vulkan não possui o conceito de "*framebuffer* padrão", portanto, requer uma infraestrutura que possua as imagens renderizadas antes de visualizar na tela. Essa infraestrutura é conhecida como *swap chain*.
- A *swap chain* é essencialmente uma fila de imagens que estão esperando para serem apresentadas na tela. O aplicativo irá adquirir tal imagem para desenhar nela e, em seguida, retorná-la à fila. O funcionamento e as condições para apresentar uma imagem da fila dependem de como a *swap chain* é configurada, mas a apresentação de imagens geralmente é sincronizada com com a taxa de atualização da tela.
- Ela é estruturada como uma lista circular chamada *ring buffer*.

Swap Chain

- Depois de criada a *swap chain*, o processo de uso é:
  1) Crie uma imagem (`createImageViews`);
  2) Renderize nela por meio de *command buffer*
  3) Retorne a imagem renderizada para a *swap chain*;
  4) Apresentar imagem para visualizador (copiar para *front buffer*).



Pipeline Básico



Fonte: https://vulkan-tutorial.com

Passos para rendering usando Vulkan

1. Criar a instância Vulkan
2. Configurar chamada de depuração (Debug Callbacks)
3. Crie a superfície
4. Liste os dispositivos físicos
5. Escolha o dispositivo físico certo
6. Crie o dispositivo lógico
7. Crie a Window Surface
8. Crie a Swap Chain
9. **Leia os Shaders**
10. Crie Fixed Functions
11. Crie Render Passes
12. Crie os Framebuffers
13. Crie Command Buffers
14. Rendering
15. Múltiplos frames in-flight

Shaders

- Diferente de GLSL e HLSL, os códigos dos *shaders* são especificados em *bytecode*;
- Esse formato *bytecode* é chamado de SPIR-V e é projetado para ser usado tanto na Vulkan quanto na OpenCL (ambos da Khronos);
- Para criação dos *bytecodes* existem compiladores que fazem esse serviço;
- Muitos usam o compilado da Google que pode ser baixado gratuitamente. É possível também incluir o compilador (incluído no SDK Vulkan) diretamente no código no formato de biblioteca para gerar o código SPIR-V em tempo de execução.

Vertex Shader

- Ele recebe na entrada os atributos como posição do mundo/universo, cor, coordenadas das normais e de textura;
- Processa cada vértice de entrada, gerando como saída a posição do vértice nas coordenadas do *frustum* (recorte);
- As coordenadas do *frustum* é um vetor de 4 dimensões (coordenadas homogêneas) normalizados entre -1 e +1 nas coordenadas da área de seleção, sendo o centro da "tela" (0,0), enquanto na coordenada Z usa a faixa [0,1].
- É possível apresentar um polígono usando diretamente as coordenadas do *frustum*. Mas normalmente essas coordenadas são armazenadas em um *vertex buffer*.

Vertex Shader

- Como isso não é trivial e ainda não vimos como fazer, para poder visualizar algo na tela, o próximo código inclui as coordenadas diretamente dentro do *vertex shader*:

```
#version 450

vec2 positions[3] = vec2[](
    vec2(0.0, -0.5),
    vec2(0.5, 0.5),
    vec2(-0.5, 0.5)
);

void main() {
    gl_Position = vec4(positions[gl_VertexIndex], 0.0, 1.0);
}
```

Fonte: https://vulkan-tutorial.com

Fragment Shader

- O triangulo formado no *vertex shader* vai preencher uma área com fragmentos (pixels e atributos);
- O *fragment shader* invoca esses fragmentos para produzir uma cor e profundidade.
- Um *fragment shader* para preencher o polígono (triângulo nesse caso) de vermelho:

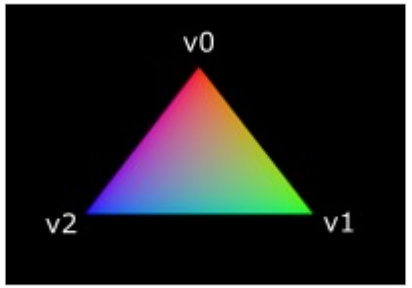
```
#version 450

layout(location = 0) out vec4 outColor;

void main() {
    outColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Fonte: https://vulkan-tutorial.com

Fragment Shader

- *Shader* para preencher o triângulo com cores:

```
vec3 colors[3] = vec3[](
    vec3(1.0, 0.0, 0.0),
    vec3(0.0, 1.0, 0.0),
    vec3(0.0, 0.0, 1.0)
);

layout(location = 0) out vec3 fragColor;

void main() {
    gl_Position = vec4(positions[gl_VertexIndex], 0.0, 1.0);
    fragColor = colors[gl_VertexIndex];
}
```




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

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

Fonte: https://vulkan-tutorial.com

Lendo *Shaders*

- Os arquivos de *bytecode* SPIR-V precisam ser carregados no programa para conectá-los ao pipeline gráfico em algum ponto:

```
static std::vector<char> readFile(const std::string& filename) {
        std::ifstream file(filename, std::ios::ate | std::ios::binary);

        if (!file.is_open()) {
            throw std::runtime_error("failed to open file!");
        }

        size_t fileSize = (size_t) file.tellg();
        std::vector<char> buffer(fileSize);

        file.seekg(0);
        file.read(buffer.data(), fileSize);

        file.close();

        return buffer;
}

void createGraphicsPipeline() {
        auto vertShaderCode = readFile("shaders/vert.spv");
        auto fragShaderCode = readFile("shaders/frag.spv");

        VkShaderModule vertShaderModule = createShaderModule(vertShaderCode);
        VkShaderModule fragShaderModule = createShaderModule(fragShaderCode);


        vkDestroyShaderModule(device, fragShaderModule, nullptr);
        vkDestroyShaderModule(device, vertShaderModule, nullptr);
}
```

Fonte: https://vulkan-tutorial.com

Passos para rendering usando Vulkan

1. Criar a instância Vulkan
2. Configurar chamada de depuração (Debug Callbacks)
3. Crie a superfície
4. Liste os dispositivos físicos
5. Escolha o dispositivo físico certo
6. Crie o dispositivo lógico
7. Crie a Window Surface
8. Crie a Swap Chain
9. Leia os Shaders
10. **Crie Fixed Functions**
11. Crie Render Passes
12. Crie os Framebuffers
13. Crie Command Buffers
14. Rendering
15. Múltiplos frames in-flight

Fixed functions

- Dynamic state
- Vertex input
- Input assembly
- Viewports / scissors
- Rasterizer
- Multisampling
- Depth and stencil testing
- Color blending
- Pipeline layout

O estado das variáveis são os mesmos em todos os estágios e possuem um valor padrão.

Fixed functions

- Dynamic state
- Vertex input
- Input assembly
- Viewports / scissors
- Rasterizer
- Multisampling
- Depth and stencil testing
- Color blending
- Pipeline layout

Dynamic state

- Embora a maior parte do estado do pipeline precise ser inserida nele, uma quantidade limitada de estados podem realmente ser alteradas sem recriar o pipeline (tamanho da *viewport*, largura da linha, etc);
- Caso queira usar o estado dinâmico mantendo essas propriedades de fora, será preciso definir uma estrutura `VkPipelineDynamicStateCreateInfo`:

```
std::vector<VkDynamicState> dynamicStates = {
    VK_DYNAMIC_STATE_VIEWPORT,
    VK_DYNAMIC_STATE_SCISSOR
};

VkPipelineDynamicStateCreateInfo dynamicState{};
dynamicState.sType =
VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
dynamicState.dynamicStateCount =
static_cast<uint32_t>(dynamicStates.size());
dynamicState.pDynamicStates = dynamicStates.data();
```

Fonte: https://vulkan-tutorial.com

Vertex input

- A estrutura **`VkPipelineVertexInputStateCreateInfo`** descreve o formato dos vértices passados pelo *vertex shader*.
- Ele descreve isso de duas maneiras:
  - Bindings: se é por vértice ou por instância (explicado em outro momento)
  - Descrição dos atributos: tipo de atributos passados para o *vertex shader*
- Por enquanto vamos deixar sem vértice (*hard coding*):

```
VkPipelineVertexInputStateCreateInfo vertexInputInfo{};
vertexInputInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
vertexInputInfo.vertexBindingDescriptionCount = 0;
vertexInputInfo.pVertexBindingDescriptions = nullptr; // Optional
vertexInputInfo.vertexAttributeDescriptionCount = 0;
vertexInputInfo.pVertexAttributeDescriptions = nullptr; // Optional
```

Fonte: https://vulkan-tutorial.com

Input assembly

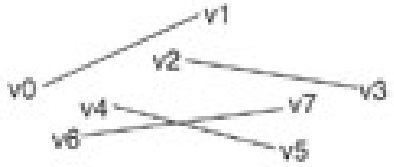
- A estrutura **`VkPipelineInputAssemblyStateCreateInfo`** descreve que tipo de primitiva será desenhada e se o reinício estará habilitado (ligação do último ao primeiro ponto do polígono):
  - VK_PRIMITIVE_TOPOLOGY_POINT_LIST: pontos
  - VK_PRIMITIVE_TOPOLOGY_LINE_LIST: linhas a cada 2 vértices
  - VK_PRIMITIVE_TOPOLOGY_LINE_STRIP: linha a cada último ponto e o próximo
  - VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST: um triângulo a cada três vértices
  - VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP: triângulo formado pelo primeiro, segundo e último vértices

```
VkPipelineInputAssemblyStateCreateInfo inputAssembly{};
inputAssembly.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
inputAssembly.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
inputAssembly.primitiveRestartEnable = VK_FALSE;
```

Fonte: https://vulkan-tutorial.com

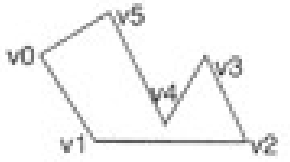
Lembrando (OpenGL)

GL_POINTS

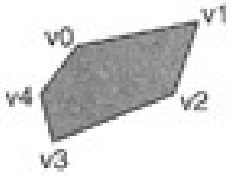


GL_LINES

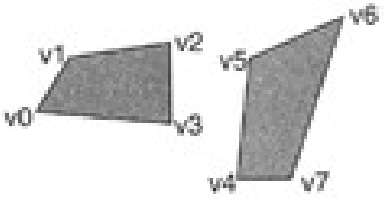
GL_LINE_STRIP



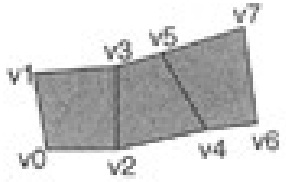
GL_LINE_LOOP



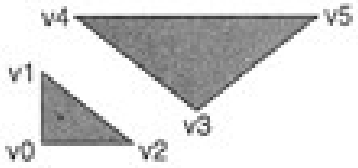
GL_POLYGON



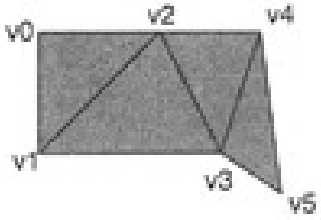
GL_QUADS



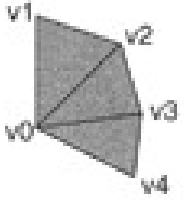
GL_QUAD_STRIP



GL_TRIANGLES



GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN

Viewports/scissors

- A *viewport* descreve a região do *framebuffer* que será renderizada. Geralmente (0,0) a (w,h);
- Os retângulos *scissors* definem em que região os *pixels* serão armazenados de fato. Qualquer pixel fora da região serão descartados pela rasterização;
- Para desenhar todo o *framebuffer*, deve-se especificar um retângulo cobrindo inteiramente ele:

```
VkRect2D scissor{};
scissor.offset = {0, 0};
scissor.extent = swapChainExtent;
VkPipelineViewportStateCreateInfo viewportState{};
viewportState.sType = VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
viewportState.viewportCount = 1;
viewportState.pViewports = &viewport;
viewportState.scissorCount = 1;
viewportState.pScissors = &scissor;
```

Fonte: https://vulkan-tutorial.com

Rasterizer

- O rasterizador pega a geometria formada pelos vértices do *vertex shader* e a transforma em fragmentos a serem coloridos pelo *fragment shader*;
- Ele realiza o teste de z-*buffer*, *backface culling* e teste das *scissors* usando a estrutura `VkPipelineRasterizationStateCreateInfo`:

```
VkPipelineRasterizationStateCreateInfo rasterizer{};
rasterizer.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
rasterizer.depthClampEnable = VK_FALSE;
rasterizer.rasterizerDiscardEnable = VK_FALSE;
rasterizer.polygonMode = VK_POLYGON_MODE_FILL;
rasterizer.lineWidth = 1.0f;
rasterizer.cullMode = VK_CULL_MODE_BACK_BIT;
rasterizer.frontFace = VK_FRONT_FACE_CLOCKWISE;
rasterizer.depthBiasEnable = VK_FALSE;
rasterizer.depthBiasConstantFactor = 0.0f; // Optional
rasterizer.depthBiasClamp = 0.0f; // Optional
rasterizer.depthBiasSlopeFactor = 0.0f; // Optional
```

Fonte: https://vulkan-tutorial.com

Multisampling

- A estrutura `VkPipelineMultisampleStateCreateInfo` configura *multisampling*, que é uma das maneiras de executar o *anti-aliasing,* combinando os resultados do *fragment shader* de vários polígonos que rasterizam para o mesmo pixel.
- Isso ocorre principalmente ao longo das bordas, onde ocorrem os artefatos mais perceptíveis.
- Como não é necessário executar o *fragment shader* várias vezes se apenas um polígono for mapeado para um pixel, é significativamente mais barato do que simplesmente renderizar para uma resolução mais alta e reduzir a escala.
- Ativá-lo requer habilitar um recurso de GPU.

Fonte: https://vulkan-tutorial.com

Multisampling

- Podemos deixar `VkPipelineMultisampleStateCreateInfo` desabilitado caso desnecessário:

```
VkPipelineMultisampleStateCreateInfo multisampling{};
multisampling.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
multisampling.sampleShadingEnable = VK_FALSE;
multisampling.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
multisampling.minSampleShading = 1.0f; // Optional
multisampling.pSampleMask = nullptr; // Optional
multisampling.alphaToCoverageEnable = VK_FALSE; // Optional
multisampling.alphaToOneEnable = VK_FALSE; // Optional
```

Fonte: https://vulkan-tutorial.com

Depth and stencil testing

- Se você estiver usando um *buffer* de profundidade e/ou estêncil, também precisará configurar os testes de profundidade e estêncil usando **`VkPipelineDepthStencilStateCreateInfo`**.
- No no caso de não for desenhar nada, é possível passar um **`nullptr`** em vez de um ponteiro para a estrutura.

Fonte: https://vulkan-tutorial.com

Color blending

- Após um *fragment shader* retornar uma cor, ela precisa ser combinada com a cor que já está no *framebuffer*. Essa transformação é conhecida como *color blending* e há duas maneiras de fazer isso:
  - Fazer a média do valor antigo e o novo para produzir uma cor final
  - Combinar o valor antigo e o novo usando uma operação *bit* a *bit*
- Existem dois tipos de structs para configurar a mistura de cores. A primeira, **`VkPipelineColorBlendAttachmentState`**, contém a configuração por cada *framebuffer* e a segunda, **`VkPipelineColorBlendStateCreateInfo`**, contém as configurações globais de mesclagem de cores.

Fonte: https://vulkan-tutorial.com

Color blending

- Usando apenas um *framebuffer*:

```
VkPipelineColorBlendAttachmentState colorBlendAttachment{};
colorBlendAttachment.colorWriteMask = VK_COLOR_COMPONENT_R_BIT |
VK_COLOR_COMPONENT_G_BIT | VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;
colorBlendAttachment.blendEnable = VK_FALSE; // ou VK_TRUE
colorBlendAttachment.srcColorBlendFactor = VK_BLEND_FACTOR_ONE; // Optional
colorBlendAttachment.dstColorBlendFactor = VK_BLEND_FACTOR_ZERO; // Optional
colorBlendAttachment.colorBlendOp = VK_BLEND_OP_ADD; // Optional
colorBlendAttachment.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE; // Optional
colorBlendAttachment.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO; // Optional
colorBlendAttachment.alphaBlendOp = VK_BLEND_OP_ADD; // Optional
```

- Definindo as configurações globais de mesclagem:

```
VkPipelineLayoutCreateInfo pipelineLayoutInfo{};
pipelineLayoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
pipelineLayoutInfo.setLayoutCount = 0; // Optional
pipelineLayoutInfo.pSetLayouts = nullptr; // Optional
pipelineLayoutInfo.pushConstantRangeCount = 0; // Optional
pipelineLayoutInfo.pPushConstantRanges = nullptr; // Optional
```

Fonte: https://vulkan-tutorial.com

Passos para rendering usando Vulkan

1. Criar a instância Vulkan
2. Configurar chamada de depuração (Debug Callbacks)
3. Crie a superfície
4. Liste os dispositivos físicos
5. Escolha o dispositivo físico certo
6. Crie o dispositivo lógico
7. Crie a Window Surface
8. Crie a Swap Chain
9. Leia os Shaders
10. Crie Fixed Functions
11. **Crie Render Passes**
12. Crie os Framebuffers
13. Crie Command Buffers
14. Rendering
15. Múltiplos frames in-flight

Render Pass

- Antes de criar o *pipeline*, precisamos informar à Vulkan sobre os complementos do *framebuffer* que serão usados durante a renderização.
- Precisamos especificar quantos *buffers* de cor e profundidade existirão, quantas amostras usar para cada um deles e como seu conteúdo deve ser tratado durante as operações de renderização.
- Todas essas informações são encapsuladas em um objeto *render pass*, para o qual criaremos uma nova função **`createRenderPass`**.
- Chame esta função a partir de **`initVulkan`** antes de **`createGraphicsPipeline`**.

Fonte: https://vulkan-tutorial.com

Render Pass

- Tendo um único *buffer* de cor representado por uma das imagens da *swap chain*:

```
void initVulkan() {
    createInstance();
    setupDebugMessenger();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
}

...

void createRenderPass() {
    VkAttachmentDescription colorAttachment{};
    colorAttachment.format = swapChainImageFormat;
    colorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
}
```

Fonte: https://vulkan-tutorial.com

Render Pass

- Agora que o básico foi criado, podemos gerar o *render pass* em si:

```
VkRenderPass renderPass;
VkPipelineLayout pipelineLayout;

VkRenderPassCreateInfo renderPassInfo{};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
renderPassInfo.attachmentCount = 1;
renderPassInfo.pAttachments = &colorAttachment;
renderPassInfo.subpassCount = 1;
renderPassInfo.pSubpasses = &subpass;

if (vkCreateRenderPass(device, &renderPassInfo, nullptr, &renderPass) != VK_SUCCESS) {
    throw std::runtime_error("failed to create render pass!");
}
```

Fonte: https://vulkan-tutorial.com

Concluindo

- Agora podemos combinar todas as estruturas e objetos anteriores para criar o *pipeline* gráfico! Recapitulando:
  - Shaders: os módulos de *shaders* que definem a funcionalidade dos estágios programáveis do *pipeline* gráfico;
  - Fixed functions: todas as estruturas que definem a estrutura do *pipeline*, como tipo de primitiva a ser desenhada, rasterizador, *viewport* e *blend functions*
  - Layout: os valores referenciados pelo *shader* que podem ser atualizados no momento do desenho
  - *Render pass*: os encapsulamentos referenciados pelos estágios do *pipeline* e seus usos.

Fonte: https://vulkan-tutorial.com

Passos para rendering usando Vulkan

1. Criar a instância Vulkan
2. Configurar chamada de depuração (Debug Callbacks)
3. Crie a superfície
4. Liste os dispositivos físicos
5. Escolha o dispositivo físico certo
6. Crie o dispositivo lógico
7. Crie a Window Surface
8. Crie a Swap Chain
9. Leia os Shaders
10. Crie Fixed Functions
11. Crie Render Passes
12. **Crie os Framebuffers**
13. Crie Command Buffers
14. Rendering
15. Múltiplos frames in-flight

Framebuffers

- As ligações especificadas durante a criação da *render pass* são vinculadas ao serem agrupados em um objeto **`VkFramebuffer`**.
- Um objeto *framebuffer* faz referência a todos os objetos **`VkImageView`**.
- A imagem que temos que usar para a ligação depende de qual imagem a *swap chain* retorna quando recuperamos uma para apresentação.
- Isso significa que temos que criar um *framebuffer* para todas as imagens na *swap chain* e usar aquele que corresponde à imagem recuperada no momento do desenho.

Fonte: https://vulkan-tutorial.com

Framebuffers

- Cria-se então um vetor de *framebuffers* e em seguida iterar as imagens da *swap chain* criando *framebuffers* delas:

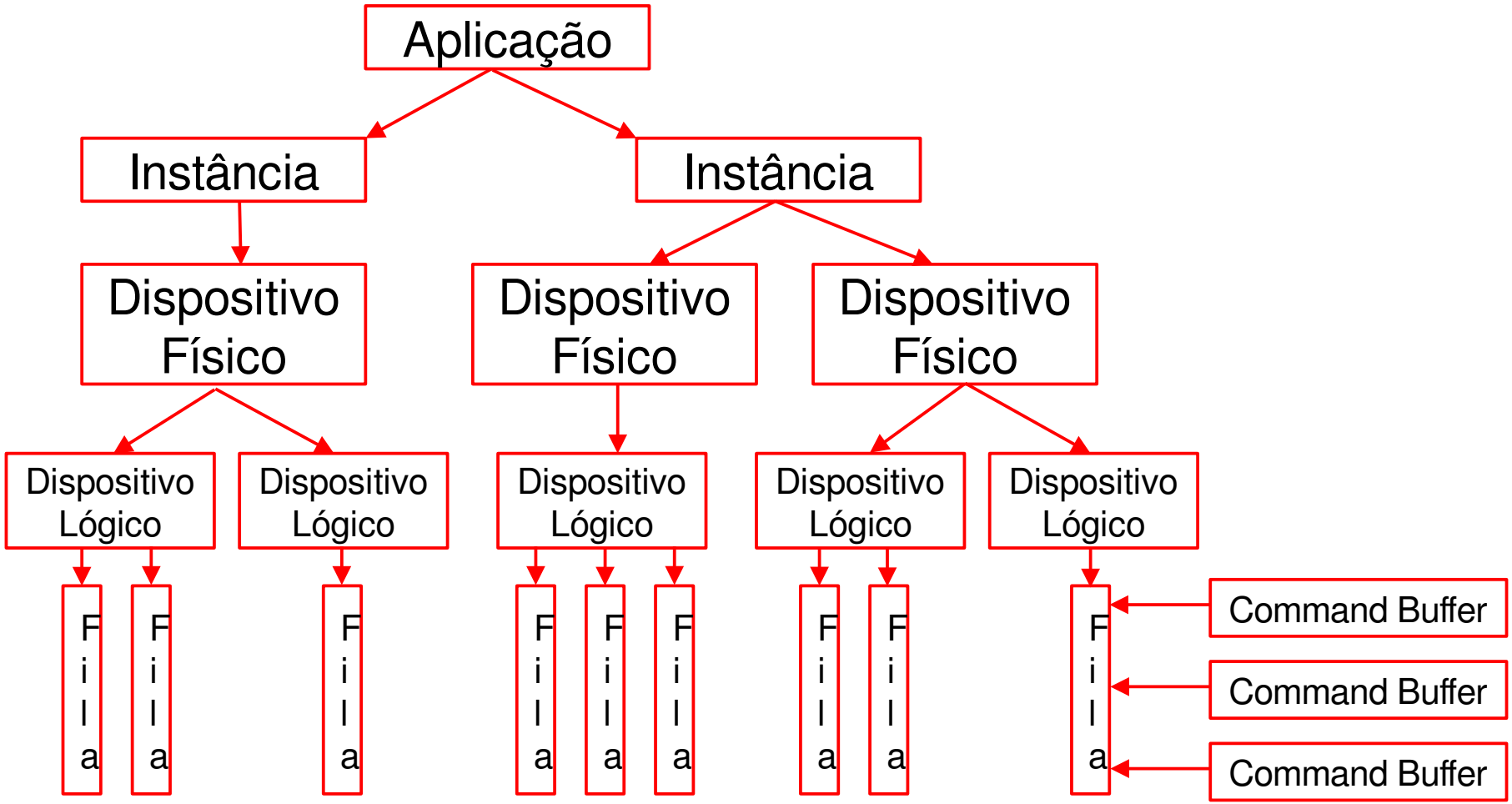
```
std::vector<VkFramebuffer> swapChainFramebuffers;

...
for (size_t i = 0; i < swapChainImageViews.size(); i++) {
    VkImageView attachments[] = {
        swapChainImageViews[i]
    };

    VkFramebufferCreateInfo framebufferInfo{};
    framebufferInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
    framebufferInfo.renderPass = renderPass;
    framebufferInfo.attachmentCount = 1;
    framebufferInfo.pAttachments = attachments;
    framebufferInfo.width = swapChainExtent.width;
    framebufferInfo.height = swapChainExtent.height;
    framebufferInfo.layers = 1;

    if (vkCreateFramebuffer(device, &framebufferInfo, nullptr,
&swapChainFramebuffers[i]) != VK_SUCCESS) {
        throw std::runtime_error("failed to create framebuffer!");
    }
}
```

Fonte: https://vulkan-tutorial.com

Passos para rendering usando Vulkan

1. Criar a instância Vulkan
2. Configurar chamada de depuração (Debug Callbacks)
3. Crie a superfície
4. Liste os dispositivos físicos
5. Escolha o dispositivo físico certo
6. Crie o dispositivo lógico
7. Crie a Window Surface
8. Crie a Swap Chain
9. Leia os Shaders
10. Crie Fixed Functions
11. Crie Render Passes
12. Crie os Framebuffers
13. **Crie Command Buffers**
14. Rendering
15. Múltiplos frames in-flight

Diagrama de Blocos Geral



Command Buffer

- Comandos na Vulkan, como operações de desenho e transferências de memória, não são executados diretamente usando chamadas de função.
- Todas as operações que for necessário executar devem ser registradas em *buffer objects*.
- A vantagem disso é que, quando estamos prontos para dizer à Vulkan o que queremos fazer, todos os comandos são enviados juntos e a Vulkan pode processar os comandos com mais eficiência, pois todos eles estão disponíveis juntos.
- Além disso, isso permite que a gravação do comando ocorra em várias *threads*, caso desejado.

Fonte: https://vulkan-tutorial.com

Command Pool

- Criando o *command pool*:

```
void createCommandPool() {
  QueueFamilyIndices queueFamilyIndices = findQueueFamilies(physicalDevice);

  VkCommandPoolCreateInfo poolInfo{};
  poolInfo.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
  poolInfo.flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;
  poolInfo.queueFamilyIndex = queueFamilyIndices.graphicsFamily.value();

  if (vkCreateCommandPool(device, &poolInfo, nullptr, &commandPool) != VK_SUCCESS) {
    throw std::runtime_error("failed to create command pool!");
  }
}
```

Fonte: https://vulkan-tutorial.com

Command Pool

- Criando o *command pool*;
- **Alocando** os *command buffers*:

```
void createCommandPool() {
  QueueFamilyIndices queueFamilyIndices = findQueueFamilies(physicalDevice);

  VkCommandPoolCreateInfo poolInfo{};
  poolInfo.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
  poolInfo.flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;
  poolInfo.queueFamilyIndex = queueFamilyIndices.graphicsFamily.value();

  if (vkCreateCommandPool(device, &poolInfo, nullptr, &commandPool) != VK_SUCCESS) {
    throw std::runtime_error("failed to create command pool!");
  }
}

void createCommandBuffer() {
  VkCommandBufferAllocateInfo allocInfo{};
  allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
  allocInfo.commandPool = commandPool;
  allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
  allocInfo.commandBufferCount = 1;

  if (vkAllocateCommandBuffers(device, &allocInfo, &commandBuffer) != VK_SUCCESS) {
    throw std::runtime_error("failed to allocate command buffers!");
  }
}
```

Fonte: https://vulkan-tutorial.com

recordCommandBuffer

- A função **`recordCommandBuffer`** grava os comandos a serem executados pelo *command buffer*. O **`VkCommandBuffer`** será passado como parâmetro, assim como o índice da imagem atual da *swap chain* que queremos gravar.

```
void recordCommandBuffer(VkCommandBuffer commandBuffer, uint32_t imageIndex) {
    VkCommandBufferBeginInfo beginInfo{};
    beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;

    if (vkBeginCommandBuffer(commandBuffer, &beginInfo) != VK_SUCCESS) {
        throw std::runtime_error("failed to begin recording command buffer!");
    }
```

Fonte: https://vulkan-tutorial.com

Iniciando o *Render Pass*

- O desenho começa iniciando a *render pass* com **`vkCmdBeginRenderPass`**. A *render pass* é configurada usando alguns parâmetros em uma estrutura **`VkRenderPassBeginInfo`**.

```
VkRenderPassBeginInfo renderPassInfo{};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
renderPassInfo.renderPass = renderPass;
renderPassInfo.framebuffer = swapChainFramebuffers[imageIndex];
renderPassInfo.renderArea.offset = {0, 0};
renderPassInfo.renderArea.extent = swapChainExtent;

VkClearValue clearColor = {{{0.0f, 0.0f, 0.0f, 1.0f}}};
renderPassInfo.clearValueCount = 1;
renderPassInfo.pClearValues = &clearColor;

vkCmdBeginRenderPass(commandBuffer, &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);
```

Fonte: https://vulkan-tutorial.com

Iniciando o *Render Pass*

- Os primeiros parâmetros são justamente o próprio render pass. Criamos um *framebuffer* para cada imagem da *swap chain* e vinculamos o *framebuffer* para a imagem da *swap chain* que queremos desenhar. Usando o parâmetro **`imageIndex`** que foi passado, podemos escolher o *framebuffer* correto para a imagem atual da *swap chain*.

```
VkRenderPassBeginInfo renderPassInfo{};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
renderPassInfo.renderPass = renderPass;
renderPassInfo.framebuffer = swapChainFramebuffers[imageIndex];
```

Fonte: https://vulkan-tutorial.com

Iniciando o *Render Pass*

- Os próximos dois parâmetros definem o tamanho da área de renderização. Os últimos parâmetros definem a cor de fundo, sendo preto com 100% de opacidade no exemplo abaixo.

```
VkRenderPassBeginInfo renderPassInfo{};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
renderPassInfo.renderPass = renderPass;
renderPassInfo.framebuffer = swapChainFramebuffers[imageIndex];
renderPassInfo.renderArea.offset = {0, 0};
renderPassInfo.renderArea.extent = swapChainExtent;

VkClearValue clearColor = {{{0.0f, 0.0f, 0.0f, 1.0f}}};
renderPassInfo.clearValueCount = 1;
renderPassInfo.pClearValues = &clearColor;

vkCmdBeginRenderPass(commandBuffer, &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);
```

Fonte: https://vulkan-tutorial.com

Comandos básicos de desenho

- Agora podemos vincular o pipeline gráfico. O segundo parâmetro especifica se o objeto de *pipeline* é gráfico ou de computação. Depois apontamos para o pipeline gráfico:

```
vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);
```

Fonte: https://vulkan-tutorial.com

Comandos básicos de desenho

- Conforme visto antes, especificamos a *viewport* e o *scissor state* para que esse *pipeline* seja dinâmico. Portanto, precisamos configurá-los no *command buffer* antes de enviar os comandos de desenho:

```
vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);

VkViewport viewport{};
viewport.x = 0.0f;
viewport.y = 0.0f;
viewport.width = static_cast<float>(swapChainExtent.width);
viewport.height = static_cast<float>(swapChainExtent.height);
viewport.minDepth = 0.0f;
viewport.maxDepth = 1.0f;
vkCmdSetViewport(commandBuffer, 0, 1, &viewport);

VkRect2D scissor{};
scissor.offset = {0, 0};
scissor.extent = swapChainExtent;
vkCmdSetScissor(commandBuffer, 0, 1, &scissor);
```

Fonte: https://vulkan-tutorial.com

Departamento de Ciência da COMPUTAÇÃO

Comandos básicos de desenho

UDESC Joinville

- Agora estamos prontos para enviar o comando de desenho para o triângulo:

```
vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);

VkViewport viewport{};
viewport.x = 0.0f;
viewport.y = 0.0f;
viewport.width = static_cast<float>(swapChainExtent.width);
viewport.height = static_cast<float>(swapChainExtent.height);
viewport.minDepth = 0.0f;
viewport.maxDepth = 1.0f;
vkCmdSetViewport(commandBuffer, 0, 1, &viewport);

VkRect2D scissor{};
scissor.offset = {0, 0};
scissor.extent = swapChainExtent;
vkCmdSetScissor(commandBuffer, 0, 1, &scissor);

vkCmdDraw(commandBuffer, 3, 1, 0, 0);
```

Fonte: https://vulkan-tutorial.com

Comandos básicos de desenho

- O *render pass* pode ser terminado e também encerramos a gravação do *command buffer*:

```
vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);

VkViewport viewport{};
viewport.x = 0.0f;
viewport.y = 0.0f;
viewport.width = static_cast<float>(swapChainExtent.width);
viewport.height = static_cast<float>(swapChainExtent.height);
viewport.minDepth = 0.0f;
viewport.maxDepth = 1.0f;
vkCmdSetViewport(commandBuffer, 0, 1, &viewport);

VkRect2D scissor{};
scissor.offset = {0, 0};
scissor.extent = swapChainExtent;
vkCmdSetScissor(commandBuffer, 0, 1, &scissor);

vkCmdDraw(commandBuffer, 3, 1, 0, 0);

vkCmdEndRenderPass(commandBuffer);

if (vkEndCommandBuffer(commandBuffer) != VK_SUCCESS) {
    throw std::runtime_error("failed to record command buffer!");
}
```

Fonte: https://vulkan-tutorial.com

Passos para rendering usando Vulkan

1. Criar a instância Vulkan
2. Configurar chamada de depuração (Debug Callbacks)
3. Crie a superfície
4. Liste os dispositivos físicos
5. Escolha o dispositivo físico certo
6. Crie o dispositivo lógico
7. Crie a Window Surface
8. Crie a Swap Chain
9. Leia os Shaders
10. Crie Fixed Functions
11. Crie Render Passes
12. Crie os Framebuffers
13. Crie Command Buffers
14. **Rendering**
15. Múltiplos frames in-flight

Princípios do *rendering*

- A função **`drawFrame`** será chamada a partir do *loop* principal para colocar o triângulo na tela. Vamos começar criando a função chamando ela a partir de de **`mainLoop`**:

```
void mainLoop() {
    while (!glfwWindowShouldClose(window)) {
        glfwPollEvents();
        drawFrame();
    }
}

...

void drawFrame() {

}
```

Fonte: https://vulkan-tutorial.com

Princípios do *rendering*

- A renderização de um quadro na Vulkan consiste em um conjunto comum de etapas:
  - Espere o quadro anterior terminar
  - Adquira uma imagem da *swap chain*
  - Grave o *command buffer* que desenham a cena nessa imagem
  - Envie o *command buffer* gravado
  - Apresentar a imagem da *swap chain*
- A função de desenho é mais complexa, mas por enquanto vamos considerar este *loop* de renderização simples.

Fonte: https://vulkan-tutorial.com

Criando objetos de sincronização

- Vamos precisar de um semáforo para sinalizar que uma imagem foi adquirida do *swap chain* e está pronta para renderização, outro para sinalizar que a renderização terminou e a apresentação pode acontecer, e uma *fence* (bloqueio de memória) para garantir que apenas um quadro seja renderizado por vez.

```
VkSemaphore imageAvailableSemaphore;
VkSemaphore renderFinishedSemaphore;
VkFence inFlightFence;

void initVulkan() {
    createInstance();
    setupDebugMessenger();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createRenderPass();
    createGraphicsPipeline();
    createFramebuffers();
    createCommandPool();
    createCommandBuffer();
    createSyncObjects();
}
```

Fonte: https://vulkan-tutorial.com

Desenhando

- No início do quadro, queremos esperar até que o quadro anterior termine:

```
void drawFrame() {
  vkWaitForFences(device, 1, &inFlightFence, VK_TRUE, UINT64_MAX);
  vkResetFences(device, 1, &inFlightFence);
```

Fonte: https://vulkan-tutorial.com

Desenhando

- A próxima coisa que precisamos fazer é adquirir uma imagem do *swap chain*.

```
void drawFrame() {
  vkWaitForFences(device, 1, &inFlightFence, VK_TRUE, UINT64_MAX);
  vkResetFences(device, 1, &inFlightFence);

  uint32_t imageIndex;
  vkAcquireNextImageKHR(device, swapChain, UINT64_MAX, imageAvailableSemaphore,
VK_NULL_HANDLE, &imageIndex);
```

Fonte: https://vulkan-tutorial.com

Desenhando

- Com o **`imageIndex`** especificando a imagem *swap chain* a ser usada, agora podemos gravar o *command buffer*.

```
void drawFrame() {
  vkWaitForFences(device, 1, &inFlightFence, VK_TRUE, UINT64_MAX);
  vkResetFences(device, 1, &inFlightFence);

  uint32_t imageIndex;
  vkAcquireNextImageKHR(device, swapChain, UINT64_MAX, imageAvailableSemaphore,
VK_NULL_HANDLE, &imageIndex);

  vkResetCommandBuffer(commandBuffer, /*VkCommandBufferResetFlagBits*/ 0);
  recordCommandBuffer(commandBuffer, imageIndex);
```

Fonte: https://vulkan-tutorial.com

Desenhando

- Especifica quais semáforos esperar antes do início da execução e em qual(is) estágio(s) do pipeline esperar.

```
void drawFrame() {
  vkWaitForFences(device, 1, &inFlightFence, VK_TRUE, UINT64_MAX);
  vkResetFences(device, 1, &inFlightFence);

  uint32_t imageIndex;
  vkAcquireNextImageKHR(device, swapChain, UINT64_MAX, imageAvailableSemaphore,
VK_NULL_HANDLE, &imageIndex);

  vkResetCommandBuffer(commandBuffer, /*VkCommandBufferResetFlagBits*/ 0);
  recordCommandBuffer(commandBuffer, imageIndex);

  VkSubmitInfo submitInfo{};
  submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
  VkSemaphore waitSemaphores[] = {imageAvailableSemaphore};
  VkPipelineStageFlags waitStages[] = {VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT};
  submitInfo.waitSemaphoreCount = 1;
  submitInfo.pWaitSemaphores = waitSemaphores;
  submitInfo.pWaitDstStageMask = waitStages;
```

Fonte: https://vulkan-tutorial.com

Desenhando

- Os próximos dois parâmetros especificam quais *command buffers* realmente enviar para execução. Aqui envia um único *command buffer*.

```
VkSubmitInfo submitInfo{};
submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
VkSemaphore waitSemaphores[] = {imageAvailableSemaphore};
VkPipelineStageFlags waitStages[] = {VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT};
submitInfo.waitSemaphoreCount = 1;
submitInfo.pWaitSemaphores = waitSemaphores;
submitInfo.pWaitDstStageMask = waitStages;
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &commandBuffer;
```

Fonte: https://vulkan-tutorial.com

Desenhando

- Especificar quais semáforos sinalizar assim que o *command buffer*(s) terminar a execução (usando **`renderFinishedSemaphore`** nesse caso).

```
VkSubmitInfo submitInfo{};
submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
VkSemaphore waitSemaphores[] = {imageAvailableSemaphore};
VkPipelineStageFlags waitStages[] = {VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT};
submitInfo.waitSemaphoreCount = 1;
submitInfo.pWaitSemaphores = waitSemaphores;
submitInfo.pWaitDstStageMask = waitStages;
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &commandBuffer;

VkSemaphore signalSemaphores[] = {renderFinishedSemaphore};
submitInfo.signalSemaphoreCount = 1;
submitInfo.pSignalSemaphores = signalSemaphores;
```

Fonte: https://vulkan-tutorial.com

Desenhando

- Agora podemos enviar o *command buffer* para a fila de gráficos usando **`vkQueueSubmit`**.

```
VkSubmitInfo submitInfo{};
submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
VkSemaphore waitSemaphores[] = {imageAvailableSemaphore};
VkPipelineStageFlags waitStages[] = {VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT};
submitInfo.waitSemaphoreCount = 1;
submitInfo.pWaitSemaphores = waitSemaphores;
submitInfo.pWaitDstStageMask = waitStages;
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &commandBuffer;

VkSemaphore signalSemaphores[] = {renderFinishedSemaphore};
submitInfo.signalSemaphoreCount = 1;
submitInfo.pSignalSemaphores = signalSemaphores;
if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, inFlightFence) != VK_SUCCESS)
  throw std::runtime_error("failed to submit draw command buffer!");
```

Fonte: https://vulkan-tutorial.com

Desenhando

- A última etapa do desenho de um quadro é enviar o resultado de volta ao *swap chain* para que ele apareça na tela. Os dois primeiros parâmetros especificam quais semáforos esperar antes que a apresentação aconteça.

```
VkSubmitInfo submitInfo{};
submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
VkSemaphore waitSemaphores[] = {imageAvailableSemaphore};
VkPipelineStageFlags waitStages[] = {VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT};
submitInfo.waitSemaphoreCount = 1;
submitInfo.pWaitSemaphores = waitSemaphores;
submitInfo.pWaitDstStageMask = waitStages;
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &commandBuffer;

VkSemaphore signalSemaphores[] = {renderFinishedSemaphore};
submitInfo.signalSemaphoreCount = 1;
submitInfo.pSignalSemaphores = signalSemaphores;
if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, inFlightFence) != VK_SUCCESS)
  throw std::runtime_error("failed to submit draw command buffer!");

VkPresentInfoKHR presentInfo{};
presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
presentInfo.waitSemaphoreCount = 1;
presentInfo.pWaitSemaphores = signalSemaphores;
```

Fonte: https://vulkan-tutorial.com

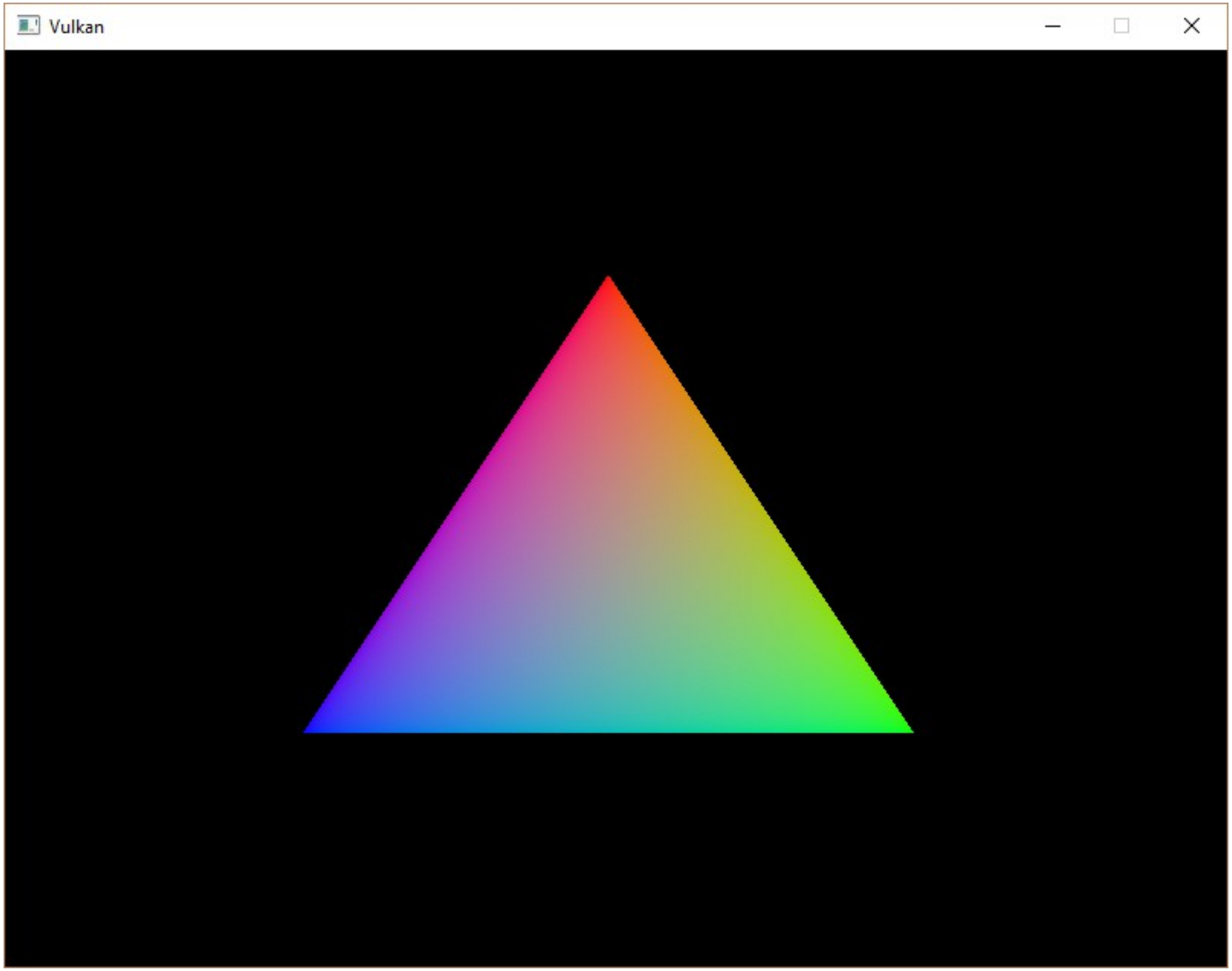
Desenhando

- O próximo parâmetro especifica quais *swap chains* apresentam as imagens e o índice da imagem para cada *swap chain*. A função **`vkQueuePresentKHR`** envia a solicitação para apresentar uma imagem ao *swap chain*.

```
  VkSubmitInfo submitInfo{};
  submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
  VkSemaphore waitSemaphores[] = {imageAvailableSemaphore};
  VkPipelineStageFlags waitStages[] = {VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT};
  submitInfo.waitSemaphoreCount = 1;
  submitInfo.pWaitSemaphores = waitSemaphores;
  submitInfo.pWaitDstStageMask = waitStages;
  submitInfo.commandBufferCount = 1;
  submitInfo.pCommandBuffers = &commandBuffer;

  VkSemaphore signalSemaphores[] = {renderFinishedSemaphore};
  submitInfo.signalSemaphoreCount = 1;
  submitInfo.pSignalSemaphores = signalSemaphores;
  if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, inFlightFence) != VK_SUCCESS)
    throw std::runtime_error("failed to submit draw command buffer!");

  VkPresentInfoKHR presentInfo{};
  presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
  presentInfo.waitSemaphoreCount = 1;
  presentInfo.pWaitSemaphores = signalSemaphores;
  VkSwapchainKHR swapChains[] = {swapChain};
  presentInfo.swapchainCount = 1;
  presentInfo.pSwapchains = swapChains;
  presentInfo.pImageIndices = &imageIndex;

  vkQueuePresentKHR(presentQueue, &presentInfo);
}
```

Fonte: https://vulkan-tutorial.com

Executando...



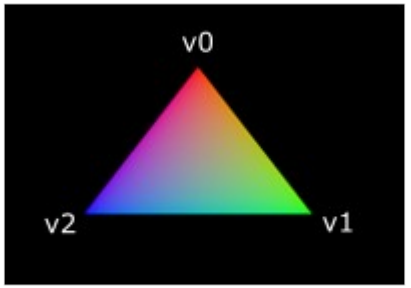
Executar: 15_hello_triangle

Fonte: https://vulkan-tutorial.com

Lembrando...

- *Fragment shader* para preencher o triângulo com cores:

```
vec3 colors[3] = vec3[](
    vec3(1.0, 0.0, 0.0),
    vec3(0.0, 1.0, 0.0),
    vec3(0.0, 0.0, 1.0)
);

layout(location = 0) out vec3 fragColor;

void main() {
    gl_Position = vec4(positions[gl_VertexIndex], 0.0, 1.0);
    fragColor = colors[gl_VertexIndex];
}
```




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

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

Fonte: https://vulkan-tutorial.com

Falha ao sair?

- Infelizmente, você verá que quando as camadas de validação estão habilitadas, o programa falha assim que você o fecha. Isso significa que quando saímos do loop principal, as operações de desenho e apresentação ainda podem estar acontecendo. Limpar recursos enquanto isso está acontecendo é uma péssima ideia. Para corrigir esse problema, devemos esperar que o dispositivo lógico termine as operações antes de sair do loop principal e destruir a janela:

```
void mainLoop() {
    while (!glfwWindowShouldClose(window)) {
        glfwPollEvents();
        drawFrame();
    }

    vkDeviceWaitIdle(device);
}
```

Fonte: https://vulkan-tutorial.com

Passos para rendering usando Vulkan

1. Criar a instância Vulkan
2. Configurar chamada de depuração (Debug Callbacks)
3. Crie a superfície
4. Liste os dispositivos físicos
5. Escolha o dispositivo físico certo
6. Crie o dispositivo lógico
7. Crie a Window Surface
8. Crie a Swap Chain
9. Leia os Shaders
10. Crie Fixed Functions
11. Crie Render Passes
12. Crie os Framebuffers
13. Crie Command Buffers
14. Rendering
15. **Múltiplos frames in-flight**

Frames in-flight

- No momento, nosso loop de renderização tem uma falha gritante. Somos obrigados a esperar que o quadro anterior termine antes de podermos começar a renderizar o próximo, o que resulta em inatividade desnecessária do *host*.
- Para corrigir isso é preciso permitir que vários quadros estejam em andamento ao mesmo tempo, ou seja, permitir que a renderização de um quadro não interfira na gravação do próximo.
- Qualquer recurso acessado e modificado durante a renderização deve ser duplicado. Portanto, precisamos de vários *command buffers*, semáforos e *fences*.

Fonte: https://vulkan-tutorial.com

Frames in-flight

- Comece adicionando uma constante no topo do programa que define quantos quadros devem ser processados simultaneamente:

```
const int MAX_FRAMES_IN_FLIGHT = 2;
```

Fonte: https://vulkan-tutorial.com

Frames in-flight

- Cada quadro deve ter seu próprio *command buffer*, conjunto de semáforos e *fences*. Renomeie e altere-os para serem vetores de objetos.

```
const int MAX_FRAMES_IN_FLIGHT = 2;

...

std::vector<VkCommandBuffer> commandBuffers;
...
std::vector<VkSemaphore> imageAvailableSemaphores;
std::vector<VkSemaphore> renderFinishedSemaphores;
std::vector<VkFence> inFlightFences;
```

Fonte: https://vulkan-tutorial.com

Frames in-flight

- Então precisamos criar vários *command buffers*. Renomeie **`createCommandBuffer`** para **`createCommandBuffers`**. Em seguida, precisamos redimensionar o vetor de buffers de comando para o tamanho de **`MAX_FRAMES_IN_FLIGHT`**.

```
const int MAX_FRAMES_IN_FLIGHT = 2;

...

std::vector<VkCommandBuffer> commandBuffers;
...
std::vector<VkSemaphore> imageAvailableSemaphores;
std::vector<VkSemaphore> renderFinishedSemaphores;
std::vector<VkFence> inFlightFences;

...

void createCommandBuffers() {
    commandBuffers.resize(MAX_FRAMES_IN_FLIGHT);
    ...
    allocInfo.commandBufferCount = (uint32_t) commandBuffers.size();

    if (vkAllocateCommandBuffers(device, &allocInfo, commandBuffers.data()) != VK_SUCCESS) {
        throw std::runtime_error("failed to allocate command buffers!");
    }
}
```

Fonte: https://vulkan-tutorial.com

Frames in-flight

- A função **`createSyncObjects`** deve ser alterada para criar todos os objetos:

```
void createSyncObjects() {
    imageAvailableSemaphores.resize(MAX_FRAMES_IN_FLIGHT);
    renderFinishedSemaphores.resize(MAX_FRAMES_IN_FLIGHT);
    inFlightFences.resize(MAX_FRAMES_IN_FLIGHT);

    VkSemaphoreCreateInfo semaphoreInfo{};
    semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;

    VkFenceCreateInfo fenceInfo{};
    fenceInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
    fenceInfo.flags = VK_FENCE_CREATE_SIGNALED_BIT;

    for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
        if (vkCreateSemaphore(device, &semaphoreInfo, nullptr, &imageAvailableSemaphores[i])
            != VK_SUCCESS ||
            vkCreateSemaphore(device, &semaphoreInfo, nullptr, &renderFinishedSemaphores[i])
            != VK_SUCCESS ||
            vkCreateFence(device, &fenceInfo, nullptr, &inFlightFences[i]) != VK_SUCCESS)
        {
            throw std::runtime_error("failed to create synchronization objects for a frame!");
        }
    }
}
```

Fonte: https://vulkan-tutorial.com

Frames in-flight

- Da mesma forma, todos eles também devem ser destruidos:

```
void cleanup() {
    for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++) {
        vkDestroySemaphore(device, renderFinishedSemaphores[i], nullptr);
        vkDestroySemaphore(device, imageAvailableSemaphores[i], nullptr);
        vkDestroyFence(device, inFlightFences[i], nullptr);
    }

    ...
}
```

Fonte: https://vulkan-tutorial.com

Frames in-flight

- Para usar os objetos certos a cada quadro, precisamos acompanhar o quadro atual. Usaremos um índice de quadro para esse fim:

```
uint32_t currentFrame = 0;
```

Fonte: https://vulkan-tutorial.com

Frames in-flight

- A função **`drawFrame`** agora pode ser modificada para usar os objetos certos:

```
uint32_t currentFrame = 0;

void drawFrame() {
  vkWaitForFences(device, 1, &inFlightFences[currentFrame], VK_TRUE, UINT64_MAX);
  vkResetFences(device, 1, &inFlightFences[currentFrame]);

  vkAcquireNextImageKHR(device, swapChain, UINT64_MAX,
                        imageAvailableSemaphores[currentFrame], VK_NULL_HANDLE, &imageIndex);
  ...
  vkResetCommandBuffer(commandBuffers[currentFrame],  0);
  recordCommandBuffer(commandBuffers[currentFrame], imageIndex);
  ...
  submitInfo.pCommandBuffers = &commandBuffers[currentFrame];
  ...
  VkSemaphore waitSemaphores[] = {imageAvailableSemaphores[currentFrame]};
  ...
  VkSemaphore signalSemaphores[] = {renderFinishedSemaphores[currentFrame]};
  ...
  if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, inFlightFences[currentFrame]) != VK_SUCCESS){
    throw std::runtime_error("failed to submit draw command buffer!");
  }
  ...
}
```

Fonte: https://vulkan-tutorial.com

Frames in-flight

- Não devemos esquecer de avançar para o próximo índice a cada quadro desenhado:

```
void drawFrame() {
    ...

    currentFrame = (currentFrame + 1) % MAX_FRAMES_IN_FLIGHT;
}
```

Fonte: https://vulkan-tutorial.com

To be continued…on the website

- Recriação de *swap chain*
- *Vertex buffer*
- *Uniform Buffers*
- Mapeamento de Textura
- *Depth Buffering*
- Leitura de modelos
- Mipmaps
- *Multisampling*
- *Compute shaders*