
Universidad de Huelva

# Tema 4

# Vulkan - Renderizado

# Índice

# Índice

- Respecto a la imagen a mostrar en una superficie, Vulkan ha sustituido el concepto de doble buffer utilizado por OpenGL por el de una cadena de intercambio de imágenes o *swapchain*.
- En un doble buffer existe un buffer de trabajo sobre el que se genera la nueva imagen a mostrar y un buffer activo con la imagen que se está mostrando.
- Una *swapchain* contiene múltiples imágenes, de manera que mientras una de ellas se encuentra en modo presentación Vulkan permite trabajar en paralelo con el resto.
- Mientras que OpenGL generaba el doble buffer de forma automática, en Vulkan es necesario programar de forma detallada la construcción de la *swapchain*.

- Para utilizar *swapchains* hay que generar el dispositivo lógico con la extensión *VK_KHR_swapchain*.
- Al crear la estructura *VkSwapchainKHR* se construyen también los buffers de las imágenes que la forman.

```
VkResult vkCreateSwapchainKHR(
    VkDevice                                device,
    const VkSwapchainCreateInfoKHR*         pCreateInfo,
    const VkAllocationCallbacks*            pAllocator,
    VkSwapchainKHR*                         pSwapchain);
```

```
typedef struct VkSwapchainCreateInfoKHR {
    VkStructureType                  sType;
    const void*                      pNext;
    VkSwapchainCreateFlagsKHR        flags;
    VkSurfaceKHR                     surface;
    uint32_t                         minImageCount;
    VkFormat                         imageFormat;
    VkColorSpaceKHR                  imageColorSpace;
    VkExtent2D                       imageExtent;
    uint32_t                         imageArrayLayers;
    VkImageUsageFlags                imageUsage;
    VkSharingMode                    imageSharingMode;
    uint32_t                         queueFamilyIndexCount;
    const uint32_t*                  pQueueFamilyIndices;
    VkSurfaceTransformFlagBitsKHR    preTransform;
    VkCompositeAlphaFlagBitsKHR      compositeAlpha;
    VkPresentModeKHR                 presentMode;
    VkBool32                         clipped;
    VkSwapchainKHR                   oldSwapchain;
} VkSwapchainCreateInfoKHR;
```

- El valor del campo *sType* debe ser VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR
- Los campos *pNext* y *flags* están reservados para posibles modificaciones futuras. De momento deben asignarse a *nullptr* y cero, respectivamente.
- El campo *surface* se refiere a la superficie sobre la que presentaremos las imágenes.
- El campo *minImageCount* contiene el número de imágenes en el *swap chain*. Para saber si el dispositivo físico permite utilizar un cierto número de imágenes se pueden consultar sus propiedades con la función *vkGetPhysicalDeviceSurfaceCapabilitiesKHR()*.

- El campo *imageFormat* describe el formato en que se almacenan los píxeles de la imagen. Existen numerosos formatos compatibles con Vulkan, por ejemplo VK_FORMAT_B8G8R8A8_UNORM.
- El campo *imageColorSpace* solo admite de momento el valor VK_COLOR_SPACE_SRGB_NONLINEAR_KHR.
- El campo *imageExtent* contiene el tamaño de las imágenes y el campo *imageArrayLayers* el número de capas. En Vulkan las imágenes son siempre tridimensionales, aunque generalmente se utilice una única capa (o dos, para imágenes estereoscópicas).
- El campo *imageUsage* define el uso que se le va a dar a las imágenes de la swapchain (VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT), que en este caso es para describir la imagen a mostrar en la superificie.

- El campo *imageSharingMode* sirve para definir si las imágenes se van a gestionar desde una única cola (VK_SHARING_MODE_EXCLUSIVE) o si se va a utilizar una cola para generar la imagen y otra para presentarla (VK_SHARING_MODE_CONCURRENT).
- Los campos *queueFamilyIndexCount* y *pQueueFamilyIndices* sirven para introducir las familias de colas que van acceder a las imágenes.
- Lo normal es que el solo haya una familia de colas y que el modo sea exclusivo.

- El campo *preTransform* sirve para configurar una transformación de las imágenes (rotación de 90º, de 180º, de 270º, voltear horizontalmente, voltear y rotar, …). Las opciones se definen en la enumeración *VkSurfaceTransformFlagBitsKHR* . Se suele dejar a nulo para no indicar ninguna transformación.

```
typedef enum VkSurfaceTransformFlagBitsKHR {
    VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR = 0x00000001,
    VK_SURFACE_TRANSFORM_ROTATE_90_BIT_KHR = 0x00000002,
    VK_SURFACE_TRANSFORM_ROTATE_180_BIT_KHR = 0x00000004,
    VK_SURFACE_TRANSFORM_ROTATE_270_BIT_KHR = 0x00000008,
    VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_BIT_KHR = 0x00000010,
    VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_90_BIT_KHR = 0x00000020,
    VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_180_BIT_KHR = 0x00000040,
    VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_270_BIT_KHR = 0x00000080,
    VK_SURFACE_TRANSFORM_INHERIT_BIT_KHR = 0x00000100,
} VkSurfaceTransformFlagBitsKHR;
```

- El campo *compositeAlpha* indica como tratar las transparencias en la imagen. De esta forma se pueden presentar imágenes en ventanas que mantengan transparencias. Si queremos un tratamiento opaco el valor debe ser VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR.
- El campo *presentMode* fija el modo de presentación, que puede ser:
  - VK_PRESENT_MODE_IMMEDIATE_KHR. Las imágenes se muestran de manera inmediata, lo que puede generar *tearing* (imágenes troceadas).
  - VK_PRESENT_MODE_FIFO_KHR: Las imágenes completadas se introducen en una cola. Si la cola se llena y no se han mostrado las imágenes la aplicación debe pararse. Si la cola se vacía la última imagen se repite.
  - VK_PRESENT_MODE_FIFO_RELAXED_KHR: Si la cola se vacía, la nueva imagen se muestra en el momento en que esté lista (provoca tearing).
  - VK_PRESENT_MODE_MAILBOX_KHR: Si la cola se llena la aplicación machaca la última imagen de la cola.

- El campo *clipped* indica como debe comportarse la aplicación cuando está oculta, por ejemplo porque exista otra ventana que la esté tapando. Si este campo es VK_TRUE significa que no importa el color de los pixeles ocultos.
- El último campo, *oldSwapchain*, permite almacenar la *swapchain* anterior cuando se sustituye por una nueva (por ejemplo, cuando se modifica el tamaño de la ventana).

- Al crear el objeto *VkSwapchainKHR* se crean el conjunto de imágenes de la cadena.
- Las imágenes son estructuras de tipo *VkImage*. Para obtener la lista de imágenes se utiliza la función *vkGetSwapchainImagesKHR()*.
- Normalmente se hacen dos pasadas. La primera para obtener el número de imágenes y la segunda para obtener la lista.

```
VkResult vkGetSwapchainImagesKHR(
        VkDevice                    device,
        VkSwapchainKHR              swapchain,
        uint32_t*                   pSwapchainImageCount,
        VkImage*                    pSwapchainImages);
```

- Para poder acceder a una imagen y generar su contenido es necesario crear un objeto *VkImageView* asociado a cada objeto *VkImage*. Esto quiere decir que para que la aplicación pueda generar las imágenes almacenadas en la *swapchain* es necesario crear una lista de vistas asociadas a cada imagen.

- Al liberar los recursos de la aplicación es importante destruir la *swapchain* por medio de la función *vkDestroySwapchainKHR()*. Esto destruye también las imágenes contenidas en la estructura.

```
void vkDestroySwapchainKHR(
    VkDevice                    device,
    VkSwapchainKHR              swapchain,
    const VkAllocationCallbacks* pAllocator);
```

- A la hora de generar una imagen hay que tener en cuenta que el proceso se va a dividir en el dibujo de varios objetos que, a su vez, están formado por un conjunto de primitivas que se tratan de manera independiente. Esto provoca que cada pixel de la imagen pueda formar parte de varias primitivas (por ejemplo, cuando un objeto se situa por delante de otro).
- Para escribir en una posición hay que saber si ese pixel ya ha sido generado con anterioridad y si corresponde a un objeto que está más lejos o más cerca. Esto se consigue utilizando un array paralelo en el que para cada pixel se almacena la profundidad del punto que se dibuja en el. En esta imagen paralela se almacena también otra información llamada *stencil*.

- El resultado final del proceso de renderizado es la escritura sobre una estructura denominada *VkFramebuffer* que almacena las imágenes a generar y los buffers de profundidad y plantilla.
- Generalmentte la salida del proceso de renderizado para cada pixel es un única variable que contiene el color de ese pixel. Sin embargo el *pipeline* de renderizado puede definir más variables de salida, cada una de las cuales deberá almacenarse en una estructura contenida en el *framebuffer*.
- Es responsabilidad del programador la construcción de los objetos *VkImage*, *VkDeviceMemory* y *VkImageView* de cada estructura.
- El *framebuffer* destinado a generar la imagen a mostrar en la superficie toma el objeto *VkImage* creado en la *swapchain*.

- Al trabajar sobre un *swapchain*, hay que definir un *framebuffer* para cada *VkImageView* del *swapchain*.
- Para crear un ojeto *VkFramebuffer* se utiliza *vkCreateFramebuffer()*.

```
VkResult vkCreateFramebuffer (
    VkDevice                                device,
    const VkFramebufferCreateInfo*          pCreateInfo,
    const VkAllocationCallbacks*            pAllocator,
    VkFramebuffer*                          pFramebuffer);
```

- La estructura *VkFramebufferCreateInfo* contiene la información necesaria para crear el *framebuffer*.
- Esta estructura contiene la referencia al *renderpass*, al *imageview*, al tamaño de la imagen y al número de capas utilizado.

```
typedef struct VkFramebufferCreateInfo {
        VkStructureType sType;
        const void* pNext;
        VkFramebufferCreateFlags flags;
        VkRenderPass renderPass;
        uint32_t attachmentCount;
        const VkImageView* pAttachments;
        uint32_t width;
        uint32_t height;
        uint32_t layers;
} VkFramebufferCreateInfo;
```

- Al liberar los recursos de la aplicación hay que destruir los objetos framebuffer por medio de la función *vkDestroyFramebuffer()*.

```
void vkDestroyFramebuffer (
        VkDevice                        device,
        VkFramebuffer                   framebuffer,
        const VkAllocationCallbacks*    pAllocator);
```

- Todo proceso de dibujo en Vulkan debe estar contenido en un objeto *VkRenderPass*, que define el conjunto de pasadas del pipeline necesario para generar el dibujo. Estas pasadas se conocen como subpasses.
- Incluso si solo se realiza una pasada es necesario crear un objeto *VkRenderPass*. Para crear el objeto *VkRenderPass* se utiliza la función *vkCreateRenderPass()*.

```
VkResult vkCreateRenderPass (
    VkDevice                                    device,
    const VkRenderPassCreateInfo*               pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkRenderPass*                               pRenderPass);
```

- La configuración del *renderpass* se introduce en una estructura de tipo *VkRenderPassCreateInfo*.
- El campo *sType* debe tener el valor VK_STRUCTURE_TYPE_RENDERPASS_CREATE_INFO
- Los campos *pNext* y *flags* por el momento deben ser nulos.

```
typedef struct VkRenderPassCreateInfo {
    VkStructureType                 sType;
    const void*                     pNext;
    VkRenderPassCreateFlags         flags;
    uint32_t                        attachmentCount;
    const VkAttachmentDescription*  pAttachments;
    uint32_t                        subpassCount;
    const VkSubpassDescription*     pSubpasses;
    uint32_t                        dependencyCount;
    const VkSubpassDependency*      pDependencies;
} VkRenderPassCreateInfo;
```

- Cada *renderpass* debe describir un conjunto de *attachments* que definen la forma en la que se va a trabajar sobre los buffers donde se generará el resultado del renderizado.
- Estas ataduras indican las acciones a realizar al inicializar los buffers de profundidad y de stencil (pueden borrarse o mantener datos anteriores) y al finalizar el renderizado (pueden almacenarse los resultados o desecharse).
- El *renderpass* también describe las características de la imagen a generar, que debe ser compatible con las imágenes generadas en la cadena de intercambio (*swapchain*).

- El campo *pAttachments* contiene una lista describiendo estas ataduras. Para ello se utilizan estructuras de tipo *VkAttachmentDescription*.
- El campo *attachmentCount* indica el tamaño de esta lista.

```
typedef struct VkAttachmentDescription {
    VkAttachmentDescriptionFlags    flags;
    VkFormat                        format;
    VkSampleCountFlagBits           samples;
    VkAttachmentLoadOp              loadOp;
    VkAttachmentStoreOp             storeOp;
    VkAttachmentLoadOp              stencilLoadOp;
    VkAttachmentStoreOp             stencilStoreOp;
    VkImageLayout                   initialLayout;
    VkImageLayout                   finalLayout;
} VkAttachmentDescription;
```

- Los campos de tipo *VkAttachmentLoadOp* indican la acción a realizar en el proceso de carga de los buffers de trabajo. Puede ser:
  - VK_ATTACHMENT_LOAD_OP_LOAD
  - VK_ATTACHMENT_LOAD_OP_CLEAR
  - VK_ATTACHMENT_LOAD_OP_DONT_CARE
- Los campos de tipo *VkAttachmentStoreOp* indican la acción a realizar como resultado del renderpass. Puede ser:
  - VK_ATTACHMENT_STORE_OP_STORE
  - VK_ATTACHMENT_STORE_OP_DONT_CARE
- Los campos *loadOp* y *storeOp* se refieren a las acciones a realizar sobre el buffer de profundidad.
- Los campos *stencilLoadOp* y *stencilStoreOp* se refieren a las acciones a realizar sobre el stencil buffer .

- El campo *format* indica el formato de imagen a generar. Debe ser compatible con el formato asignado al objeto *VkImage*.
- El campo *samples* indica la configuración de multisampleado. Debe coincidir con el valor asignado en el pipeline.
- Los campos *initialLayout* y *finalLayout* describen la imagen a utilizar en el renderizado. Pueden tomar numerosos valores. Los más comunes son:
  - VK_IMAGE_LAYOUT_UNDEFINED
  - VK_IMAGE_LAYOUT_PRESENT_SRC_KHR

- El *renderpass* describe la entrada y salida global del proceso de renderizado. Los subpases deben describir las entradas y salidas de cada proceso de renderizado a realizar para generar la imagen completa. Esto incluye definir también las ataduras entre buffers de forma que la imagen de salida de un subpase se utilice como entrada de otro, o se pasen o inicialicen los datos de profundidad o de *stencil*.

- El campo *subpassCount* indica el número de subpases que formarán el *renderpass* mientras que el campo *pSubpasses* contiene la lista con las descripciones de estos subpases.
- La descripción de los subpasses indica como trabajar con el buffer de color, profundidad, stencil, …

```
typedef struct VkSubpassDescription {
    VkSubpassDescriptionFlags           flags;
    VkPipelineBindPoint                 pipelineBindPoint;
    uint32_t                            inputAttachmentCount;
    const VkAttachmentReference*        pInputAttachments;
    uint32_t                            colorAttachmentCount;
    const VkAttachmentReference*        pColorAttachments;
    const VkAttachmentReference*        pResolveAttachments;
    const VkAttachmentReference*        pDepthStencilAttachment;
    uint32_t                            preserveAttachmentCount;
    const uint32_t*                     pPreserveAttachments;
} VkSubpassDescription;
```

- La estructura *VkAttachmentReference* contiene la configuración de un enlace a una estructura del *framebuffer*. El campo attachment contiene el índice del buffer a enlazar y el campo layout la forma de enlazarlo.
- Esta forma puede ser de muchos tipos, por ejemplo:
  - VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
  - VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL

```
typedef struct VkAttachmentReference {
    uint32_t        attachment;
    VkImageLayout   layout;
} VkAttachmentReference;
```

- Para eliminar un RenderPass se utiliza la función *vkDestroyRenderPass()*.

```
void vkDestroyRenderPass (
        VkDevice                    device,
        VkRenderPass                renderPass,
        const VkAllocationCallbacks* pAllocator);
```

## Índice

- Para generar cada una de las imágenes del *swapchain* tendríamos que lanzar la ejecución del buffer de comandos asociado a cada una de ellas.
- La ejecución de un buffer de comandos es un proceso asíncrono. Esto quiere decir que la función de lanzamiento devuelve el control sin que realmente el proceso de renderizado se haya completado. El problema es que no podemos lanzar un proceso de renderizado sobre una imagen sin asegurarnos de que el renderizado anterior haya finalizado, es decir, hay que incluir mecanismos de sincronización.
- En Vulkan, esta sincronización es responsabilidad del programador.

- Hay dos formas de sincronizar eventos de la swapchain: semáforos y vallas (fences).
- Un semáforo permite definir puntos de sincronización internos en el dispositivo lógico. Esto permite sincronizar los procesos que se ejecutan de manera asíncrona en la GPU.
- Las vallas permiten definir puntos de sincronización entre la GPU y la CPU.

- Para representar una imagen sobre una superficie la GPU debe realizar dos pasos: generar la imagen y presentarla en la superficie. Para sincronizar esto se definen dos semáforos, uno para indicar que la imagen está disponible para renderizar sobre ella y otro para indicar que la imagen está renderizada y está disponible para presentarla.
- Para generar un semáforo se utiliza la función *vkCreateSemaphore()*. La estructura *VkSemaphoreCreateInfo* solo tiene los campos *sType*, *pNext* y *flags* que por el momento no tienen contenido.

```
VkResult vkCreateSemaphore (
    VkDevice                                device,
    const VkSemaphoreCreateInfo*            pCreateInfo,
    const VkAllocationCallbacks*            pAllocator,
    VkSemaphore*                            pSemaphore);
```

- Para sincronizar la generación de imágenes sobre la swapchain se utilizan vallas (*fences)*. Solo se va a lanzar un proceso de generación de una imagen si su valla está abierta. Solo abrimos la valla cuando hemos terminado de representar la imagen en una pasada anterior.
- Para crear una valla se utiliza la función *vkCreateFence()*. La estructura *VkFenceCreateInfo* por el momento solo tiene los campos *sType*, *pNext* y *flags* sin contenido.

```
VkResult vkCreateFence (
    VkDevice                                device,
    const VkFenceCreateInfo*                pCreateInfo,
    const VkAllocationCallbacks*            pAllocator,
    VkFence*                                pFence);
```

- El proceso de creación de la imagen tiene tres pasos: adquirir la imagen, generar la imagen y presentar la imagen.
- Para adquirir la imagen de una swapchain se utiliza la función *vkAcquireNextImageKHR()*. Esta función espera a que la siguiente imagen esté disponible para devolver el control almacenando en *pImageIndex* el índice de la imagen disponible.

```
VkResult vkAcquireNextImageKHR(
    VkDevice                    device,
    VkSwapchainKHR              swapchain,
    uint64_t                    timeout,
    VkSemaphore                 semaphore,
    VkFence                     fence,
    uint32_t*                   pImageIndex);
```

- El campo *timeout* se refiere al tiempo máximo de espera para obtener la imagen.
- El campo *semaphore* se refiere al semáforo que hay que esperar para poder renderizar en la imagen
- El campo *fence* indica la valla a abrir cuando sea posible renderizar la imagen.

- Para generar la imagen hay que lanzar el buffer de comandos sobre una cola del dispositivo. Para esto se utiliza la función *vkQueueSubmit()*.
- El campo *fence* contiene la valla que se señalará al terminar el proceso de ejecución del buffer de comandos.

```
VkResult vkQueueSubmit (
    VkQueue                 queue,
    uint32_t                submitCount,
    const VkSubmitInfo*     pSubmits,
    VkFence                 fence);
```

- La estructura *VkSubmitInfo* describe la configuración del lanzamiento.

```
typedef struct VkSubmitInfo {
    VkStructureType                 sType;
    const void*                     pNext;
    uint32_t                        waitSemaphoreCount;
    const VkSemaphore*              pWaitSemaphores;
    const VkPipelineStageFlags*     pWaitDstStageMask;
    uint32_t                        commandBufferCount;
    const VkCommandBuffer*          pCommandBuffers;
    uint32_t                        signalSemaphoreCount;
    const VkSemaphore*              pSignalSemaphores;
} VkSubmitInfo;
```

- El campo *pCommandBuffers* indica los buffers a lanzar. Se pueden lanzar varios buffers en la misma llamada. El número de buffers se indica en *commandBufferCount*.
- El campo *pWaitSemaphores* indica los semáforos a los que hay que esperar para lanzar el renderizado. Pueden indicarse varios semáforos.
- El campo *pSignalSemaphores* indica los semáforos que hay que abrir al terminar el proceso de renderizado. Pueden indicarse varios semáforos.

- Para terminar el proceso hay que presentar la imagen renderizada. Para esto se utiliza la función *vkQueuePresentKHR()*.

```
VkResult vkQueuePresentKHR(
    VkQueue                 queue,
    const VkPresentInfoKHR* pPresentInfo);

typedef struct VkPresentInfoKHR {
    VkStructureType         sType;
    const void*             pNext;
    uint32_t                waitSemaphoreCount;
    const VkSemaphore*      pWaitSemaphores;
    uint32_t                swapchainCount;
    const VkSwapchainKHR*   pSwapchains;
    const uint32_t*         pImageIndices;
    VkResult*               pResults;
} VkPresentInfoKHR;
```

- La estructura *VkPresentInfoKHR* configura el proceso de presentación.
- El campo *pWaitSemaphores* contiene una lista de semáforos a los que hay que esperar para realizar la presentación. El número de semáforos se indica en *waitSemaphoreCount*.
- En una misma llamada se puede lanzar la presentación de varias imágenes. Esto puede ser util cuando una aplicación trabaja con varias ventanas. El campo *pSwapchains* contiene la lista de *swapchains* sobre las que presentar imágenes. El campo *pImageIndices* contiene el índice de la imagen a presentar de cada *swapchain*. El número de *swapchains* se indica en *swapchainCount*.

- Para lanzar un proceso de renderizado (una vez configuradas todas las estructuras anteriores) es necesario crear un buffer de comandos que ejecutar sobre una cola del dispositivo lógico.
- El primer paso para crear un buffer de comandos es generar un *Command Pool* (un generador de buffers de comandos). El primer argumento de la función *vkCreateCommandPool()* es el dispositivo lógico sobre el que se generarán los buffers.

```
VkResult vkCreateCommandPool (
    VkDevice                                device,
    const VkCommandPoolCreateInfo*          pCreateInfo,
    const VkAllocationCallbacks*            pAllocator,
    VkCommandPool*                          pCommandPool);
```

- La configuración del *command_pool* se describe por medio de la estructura *VkCommandPoolCreateInfo*, que tiene asociada como campo sType el valor VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO
- El campo *queueFamilyIndex* indica la familia de colas sobre las que se crearán los buffers de comandos.
- El campo *flags* tiene definidos dos bits:
  - VK_COMMAND_POOL_CREATE_TRANSIENT_BIT
  - VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT

```
typedef struct VkCommandPoolCreateInfo {
    VkStructureType                 sType;
    const void*                     pNext;
    VkCommandPoolCreateFlags        flags;
    uint32_t                        queueFamilyIndex;
} VkCommandPoolCreateInfo;
```

- Una vez creado el *command_pool*, se puede utilizar para crear buffers de comandos. Para ello se utiliza la función *vkAllocateCommandBuffers()* que permite construir una lista de buffers.
- La configuración de esta función se realiza con una estructura *VkCommandBufferAllocateInfo*.

```
VkResult vkAllocateCommandBuffers (
        VkDevice                                    device,
        const VkCommandBufferAllocateInfo*          pAllocateInfo,
        VkCommandBuffer*                            pCommandBuffers);

typedef struct VkCommandBufferAllocateInfo {
        VkStructureType                 sType;
        const void*                     pNext;
        VkCommandPool                   commandPool;
        VkCommandBufferLevel            level;
        uint32_t                        commandBufferCount;
} VkCommandBufferAllocateInfo;
```

- El campo *sType* en este caso debe tomar el valor VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO.
- El campo *pNext* debe dejarse a nulo.
- El campo *commandPool* es una referencia al generador de buffers.
- El campo *level* indica el tipo de buffer, que puede ser primario o secundario (VK_COMMAND_BUFFER_LEVEL_PRIMARY o VK_COMMAND_BUFFER_LEVEL_SECONDARY). Un buffer secundario puede ser ejecutado desde un buffer primario.

- Una vez creado un buffer de comandos, se puede utilizar para lanzar un proceso de renderizado. Para ello hay que inicializarlo, llenarlo de comandos y lanzarlo.
- Para inicializar el buffer se utiliza la función *vkBeginCommandBuffer()*.

```
VkResult vkBeginCommandBuffer (
        VkCommandBuffer                 commandBuffer,
        const VkCommandBufferBeginInfo* pBeginInfo);

typedef struct VkCommandBufferBeginInfo {
        VkStructureType                                 sType;
        const void*                                     pNext;
        VkCommandBufferUsageFlags                       flags;
        const VkCommandBufferInheritanceInfo*           pInheritanceInfo;
} VkCommandBufferBeginInfo;
```

- El campo *sType* de la estructura *VkCommandBufferBeginInfo* debe tener el valor VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO
- El campo *pNext* debe estar a nulo.
- El campo *flags* puede tomar los siguientes valores:
  - VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT
  - VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT
  - VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT
  - El valor cero permite utilizar el buffer varias veces
- El campo *pInheritanceInfo* se utiliza para inicializar un buffer secundario y permite definir los estados que se heredan del buffer primario.

- Una vez inicializado el buffer de comandos hay que iniciar el *renderpass* por medio de la función *vkCmdBeginRenderPass ()*.
- La información de esta función incluye las referencias al *renderpass*, al *framebuffer*, al área de dibujo y a los valores utilizados para borrar.

```
void vkCmdBeginRenderPass (
        VkCommandBuffer                 commandBuffer,
        const VkRenderPassBeginInfo*    pRenderPassBegin,
        VkSubpassContents               contents);

typedef struct VkRenderPassBeginInfo {
        VkStructureType             sType;
        const void*                 pNext;
        VkRenderPass                renderPass;
        VkFramebuffer               framebuffer;
        VkRect2D                    renderArea;
        uint32_t                    clearValueCount;
        const VkClearValue*         pClearValues;
} VkRenderPassBeginInfo;
```

- El siguiente paso es enlazar el pipeline a utilizar por medio de la función *vkCmdBindPipeline()*.
- El campo *pipelineBindPoint* indica el tipo de pipeline que estamos utilizando y puede tomar los valores:
  - VK_PIPELINE_BIND_POINT_COMPUTE
  - VK_PIPELINE_BIND_POINT_GRAPHICS

```
void vkCmdBindPipeline (
        VkCommandBuffer             commandBuffer,
        VkPipelineBindPoint         pipelineBindPoint,
        VkPipeline                  pipeline);
```

- A continuación hay que enlazar las entradas del pipeline. Para eso hay que utilizar las siguientes funciones:
  - *vkCmdBindVertexBuffers()*
  - *vkCmdBindIndexBuffer()*
  - *vkCmdBindDescriptorSets()*

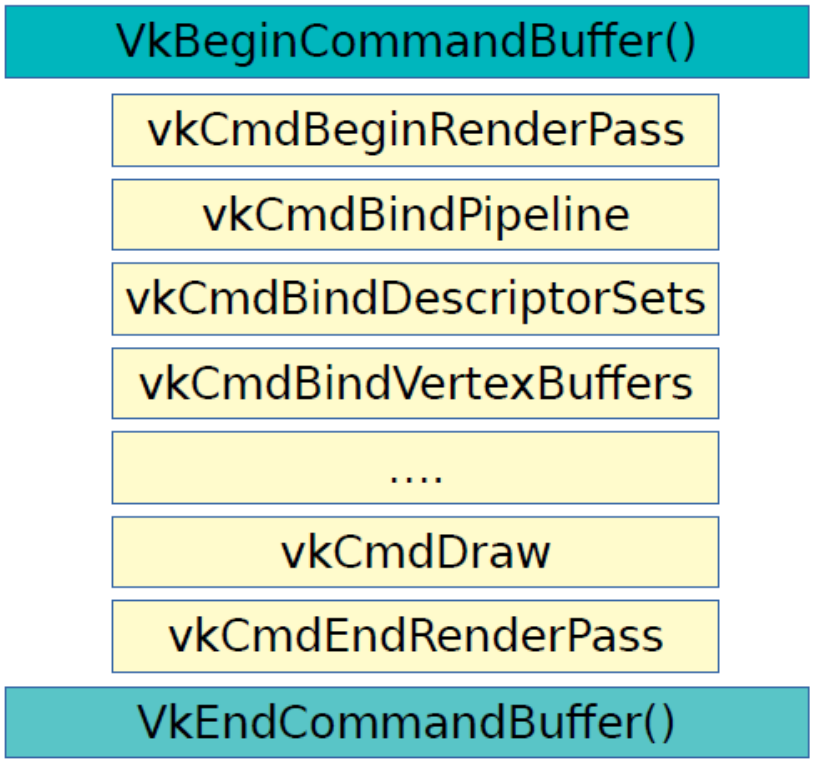
- Para lanzar el pipeline y realizar el dibujo se utilizan los comandos *vkCmdDraw()* o *vkCmdDrawIndexed()*. En el primer caso los vértices se leen secuencialmente. En el segundo caso se utiliza el buffer de índices para leer los vértices de forma indirecta.
- Para declarar el final de la descripción del *renderpass* se utiliza la función *vkCmdEndRenderPass().*
- Para finalizar la creación del buffer de comandos se utiliza la función *vkEndCommandBuffer().*

- Para declarar el final de la descripción del *renderpass* se utiliza la función *vkCmdEndRenderPass()*.
- Para finalizar la creación del buffer de comandos se utiliza la función *vkEndCommandBuffer()*.

```
void vkCmdEndRenderPass (
        VkCommandBuffer commandBuffer);

VkResult vkEndCommandBuffer (
        VkCommandBuffer commandBuffer);
```

- El esquema de generación de un buffer de comandos es el siguiente.



- Para destruir una lista de *command buffers* se utiliza la función *vkFreeCommandBuffers ().*
- Para destruir *un command pool* se utiliza la función *vkDestroyCommandPool().*

```
void vkFreeCommandBuffers (
        VkDevice                    device,
        VkCommandPool               commandPool,
        uint32_t                    commandBufferCount,
        const VkCommandBuffer*      pCommandBuffers);

void vkDestroyCommandPool (
        VkDevice                    device,
        VkCommandPool               commandPool,
        const VkAllocationCallbacks* pAllocator;
```