Universidad de Huelva

Tema 3

Vulkan - Buffers

Índice

Índice

- Un buffer es una zona de memoria de la tarjeta gráfica en la que se almacenan datos a utilizar en el proceso de renderizado.
- Los buffers se utilizan para almacenar los valores de los atributos de los vértices, los valores de los índices, las variables uniformes, las texturas, …
- Para crear un buffer hay que indicar el uso que va a tener (almacenar atributos, indices, …) y si se va a compartir entre varias colas o en modo exclusivo.

- Los buffers se definen por medio de estructuras *VkBuffer* y se crean mediante la función *vkCreateBuffer()*.

```
VkResult vkCreateBuffer (
        VkDevice                    device,
        const VkBufferCreateInfo*   pCreateInfo,
        const VkAllocationCallbacks* pAllocator,
        VkBuffer*                   pBuffer);
```

- La información para crear el buffer se introduce en la estructura *VkBufferCreateInfo*.
- La estructura contiene los campos habituales *sType*, *pNext* y *flags*.
- El campo *size* contiene el tamaño del buffer en bytes .

```
typedef struct VkBufferCreateInfo {
    VkStructureType              sType;
    const void*                  pNext;
    VkBufferCreateFlags          flags;
    VkDeviceSize                 size;
    VkBufferUsageFlags           usage;
    VkSharingMode                sharingMode;
    uint32_t                     queueFamilyIndexCount;
    const uint32_t*              pQueueFamilyIndices;
} VkBufferCreateInfo;
```

- El campo *usage* indica para qué se va a utilizar el buffer. Puede tomar los valores:
  - VK_BUFFER_USAGE_TRANSFER_SRC_BIT
  - VK_BUFFER_USAGE_TRANSFER_DST_BIT
  - VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT
  - VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT
  - VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT
  - VK_BUFFER_USAGE_STORAGE_BUFFER_BIT
  - VK_BUFFER_USAGE_INDEX_BUFFER_BIT
  - VK_BUFFER_USAGE_VERTEX_BUFFER_BIT
  - VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT
  - VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT

- El campo *sharingMode* indica si el buffer se va a compartir entre colas (VK_SHARING_MODE_CONCURRENT) o solo se usará en una cola (VK_SHARING_MODE_EXCLUSIVE).
- El campo *pQueueFamilyIndices* indica los índices de las familias de colas que pueden utilizar el buffer.

- Para destruir un buffer se utiliza la función *vkDestroyBuffer()*.

```
void vkDestroyBuffer (
        VkDevice                    device,
        VkBuffer                    buffer,
        const VkAllocationCallbacks* pAllocator);
```

Índice

- El objeto *VkBuffer* contiene la descripción del buffer, pero para ubicarlo en memoria es necesario crear una estructura paralela de tipo *VkDeviceMemory*, que describe la memoria ocupada por el buffer.
- Para alojar la memoria se utiliza la función *vkAllocateMemory()*.

```
VkResult vkAllocateMemory (
    VkDevice                    device,
    const VkMemoryAllocateInfo* pAllocateInfo,
    const VkAllocationCallbacks* pAllocator,
    VkDeviceMemory*             pMemory);
```

- Para indicar las características de la memoria a reservar se utiliza la estructura *VkMemoryAllocateInfo*.
- El campo *pNext* se deja a nulo y el camnpo *sTyppe* debe tener el valor VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO.
- El campo *allocationSize* define el tamaño a reservar en bytes.
- El campo *memoryTypeIndex* indica el tipo de memoria de entre los diferentes tipos soportados por el dispositivo.

```
typedef struct VkMemoryAllocateInfo {
    VkStructureType     sType;
    const void*         pNext;
    VkDeviceSize        allocationSize;
    uint32_t            memoryTypeIndex;
} VkMemoryAllocateInfo;
```

- Las propiedades de los distintos tipos de memoria del dispositivo se obtienen con la función *vkGetPhysicalDeviceMemoryProperties()*.

```
void vkGetPhysicalDeviceMemoryProperties (
    VkPhysicalDevice                    physicalDevice,
    VkPhysicalDeviceMemoryProperties*   pMemoryProperties);
```

- La función almacena la información sobre la estructura referenciada en el argumento *pMemoryProperties*.

```
typedef struct VkPhysicalDeviceMemoryProperties {
    uint32_t            memoryTypeCount;
    VkMemoryType        memoryTypes[VK_MAX_MEMORY_TYPES];
    uint32_t            memoryHeapCount;
    VkMemoryHeap        memoryHeaps[VK_MAX_MEMORY_HEAPS];
} VkPhysicalDeviceMemoryProperties;
```

- El campo *memoryTypeCount* indica el número de tipos de memoria diferente que contiene el dispositivo y la estructura *VkMemoryType* describe las propiedades de cada tipo de memoria.

```
typedef struct VkMemoryType {
    VkMemoryPropertyFlags    propertyFlags;
    uint32_t                 heapIndex;
} VkMemoryType;
```

- La reserva dinámica de memoria en el dispositivo utiliza montículos.

```
typedef struct VkMemoryHeap {
    VkDeviceSize             size;
    VkMemoryHeapFlags        flags;
} VkMemoryHeap;
```

- Las propiedadses de un tipo de memoria se describen como una combinación de flags con los posibles valores:
  - VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT: los cores del dispositivo pueden acceder a esta memoria.
  - VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT: la memoria es accesible desde la CPU.
  - VK_MEMORY_PROPERTY_HOST_COHERENT_BIT: se puede acceder simultáneamente a la memoria desde la CPU y desde el dispositivo.
  - VK_MEMORY_PROPERTY_HOST_CACHED_BIT: la memoria utiliza caché desde la CPU.
  - VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT: la memoria no se reserva realmente hasta su uso.

- Los requisitos de memoria del buffer se obtienen con la función *vkGetBufferMemoryRequirements()*.

```
void vkGetBufferMemoryRequirements (
        VkDevice                    device,
        VkBuffer                    buffer,
        VkMemoryRequirements*       pMemoryRequirements);
```

- Para elegir el tipo de memoria hay que encontrar un tipo que soporte las características requeridas por el buffer.

- La estructura *VkMemoryRequirements* indica el tamaño en bytes requerido por el buffer y el tipo de alineación.
- El campo *memoryTypeBits* contiene un bit para cada tipo de memoria del dispositivo (tipo 0 en bit 0, …). Si el bit está activo indica que ese tipo de memoria es compatible con los requisitos del buffer.
- El primer paso para aceptar un tipo de memoria es que su bit esté activo. A continuación hay que compara las propiedades de la memoria con los requisitos que queremos para el buffer.

```
typedef struct VkMemoryRequirements {
    VkDeviceSize        size;
    VkDeviceSize        alignment;
    uint32_t            memoryTypeBits;
} VkMemoryRequirements;
```

- Una vez reservada la memoria ya se pueden almacenar en ella los datos del buffer.
- La función *vkMapMemory()* obtiene un puntero a la memoria que se puede utilizar como referencia para copiar los datos.
- A partir del puntero se pueden volcar los datos con la función *memcpy()*.
- Es importante liberar el puntero con *vkUnmapMemory()* porque mientras la memoria está mapeada, la GPU no puede acceder a ella.

```
void* data;
vkMapMemory(device, bufferMemory, 0, bufferSize, 0, &data);
memcpy(data, bufferData,  bufferSize);
vkUnmapMemory(device, bufferMemory);
```

- Una vez creado el buffer, reservada la memoria y copiados en ella los datos, es necesario vincular el buffer a la memoria por medio de la función *vkBindBufferMemory()*.
- El campo *memoryOffset* indica en qué posición de la memoria se encuentra la infomación del buffer. Se puede utilizar una única memoria para enlazar varios buffers indicando posiciones de inicio diferentes.

```
VkResult vkBindBufferMemory (
        VkDevice            device,
        VkBuffer            buffer,
        VkDeviceMemory      memory,
        VkDeviceSize        memoryOffset);
```

- Para liberar un bloque de momoria se utiliza la función *vkFreeMemory()*.

```
void vkFreeMemory (
        VkDevice                    device,
        VkDeviceMemory              memory,
        const VkAllocationCallbacks* pAllocator);
```

Índice

- Los buffers describen un conjunto de datos almacenados en la memoria sin ninguna estructura. Los datos se almacenan y se leen en bloque.
- Las imágenes describen un conjunto de datos almacenados en la GPU con una estructura más compleja. Son datos multidimensionales, tienen un formato específico y pueden utilizarse en operaciones más complejas como filtrado, mezcla, tests de profundidad,…
- Las imágenes se defininen con estructuras *VkImage* y se crean con la función *vkCreateImage()*.

```
VkResult vkCreateImage (
    VkDevice                        device,
    const VkImageCreateInfo*        pCreateInfo,
    const VkAllocationCallbacks*    pAllocator,
    VkImage*                        pImage);
```

- Las propiedades de la imagen a crear se definen en una estructura *VkImageCreateInfo* .

```
typedef struct VkImageCreateInfo {
    VkStructureType          sType;
    const void*              pNext;
    VkImageCreateFlags       flags;
    VkImageType              imageType;
    VkFormat                 format;
    VkExtent3D               extent;
    uint32_t                 mipLevels;
    uint32_t                 arrayLayers;
    VkSampleCountFlagBits    samples;
    VkImageTiling            tiling;
    VkImageUsageFlags        usage;
    VkSharingMode            sharingMode;
    uint32_t                 queueFamilyIndexCount;
    const uint32_t*          pQueueFamilyIndices;
    VkImageLayout            initialLayout;
} VkImageCreateInfo;;
```

- El campo *pNext* debe estar a nulo. El campo *sType* debe tener el valor VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO.
- En este caso ya se han definido posible valores para el campo *flags*, aunque lo habitual es dejarlo a cero. Los bits definidos son:
  - VK_IMAGE_CREATE_SPARSE_BINDING_BIT,
  - VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT
  - VK_IMAGE_CREATE_SPARSE_ALIASED_BIT
  - VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT
  - VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT
- Los tres primeros permiten configurar imágenes en modo *sparse*. El bit de formato mutable se activa si la imagen va a soportar vistas definidas con un formato diferente. El bit *cube* se activa para utilizar la imagen para almacenar un cubemap (6 imágenes).

- El campo *imageType* indica el tipo de imagen a crear (1D, 2D o 3D). Puede tomar los siguientes valores
  - VK_IMAGE_TYPE_1D
  - VK_IMAGE_TYPE_2D
  - VK_IMAGE_TYPE_3D
- El campo *format* indica el formato utilizado para describir cada elemento de la imagen. *VkFormat* es una enumeración con muchísimas opciones. Por ejemplo, VK_FORMAT_R8G8B8A8_UNORM es un formato de 32 bits que dedica 8 bits a cada componente en formato normalizado sin signo (0 a 255) donde el byte menos significativo es la componente Red y el más significativo la componente Alpha.

- El campo *extent* indica el tamaño de la imagen. La estructura *VkExtent3D* contiene los campos *width*, *height* y *depth*. Todas las imágenes en Vulkan tienen tres dimensiones, aunque para imágenes 2D la profundidad es 1 y para imágenes 1D la altura y la profundidad son 1.
- El campo *mipLevels* indica el número de mipmaps de la imagen. Los mipmaps son versiones de la imagen de distinto tamaño. Por ejemplo, una imagen de 256x256 podría definirse con tres niveles: 256x256 (level 0), 128x128 (level 1) y 64x64 (level 2).
- Una estructura *VkImage* puede contener un array de imágenes. El campo *arrayLayers* indica el número de imágenes almacenadas en la estructura (lo normal es una).

- El campo *samples* indica el número de valores almacenados para cada pixel, lo que se utiliza en la opción de renderizado con multisampleado. Lo normal es uno pero puede tomar los valores 1, 2, 4, 8, 16, 32 y 64.
- El campo *tiling* indica el orden en el que se van a almacenar los píxeles en memoria. Los valores pueden ser VK_IMAGE_TILING_OPTIMAL para almacenarlos de manera compacta (dependiente del dispositivo), y VK_IMAGE_TILING_LINEAR para almacenarlos por fila de manera consecutiva. Generalmente se elige la opción óptima.

- El campo *usage* especifica el uso que se le va a dar a la imagen. En la primera versión de Vilkan, los valores de este campo pueden ser una combinación de los siguientes:
  - VK_IMAGE_USAGE_TRANSFER_SRC_BIT
  - VK_IMAGE_USAGE_TRANSFER_DST_BIT
  - VK_IMAGE_USAGE_SAMPLED_BIT
  - VK_IMAGE_USAGE_STORAGE_BIT
  - VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT
  - VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT
  - VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT
  - VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT
- Las nuevas versiones de Vulkan han ampliado mucho estas opciones.

- El campo *sharingMode* indica si se va a acceder a la imagen desde diferentes colas o solo desde una. Los valores pueden ser:
  - VK_SHARING_MODE_EXCLUSIVE
  - VK_SHARING_MODE_CONCURRENT.
- EL campo *pQueueFamilyIndices* es un puntero al array de colas que pueden acceder a la imagen en modo concurrente. El campo *queueFamilyIndexCount* indica el tamaño del array.
- Finalmente, las imágenes tienen un layout que está relacionado con el uso que se les va a dar. El campo *initialLayout* indica el layout al que se va a asignar la imagen al ser creada. Por ejemplo, si la imagen se va a generar en un proceso de renderizado se utiliza el valor VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL.

- Para destruir una imagen se utiliza la función *vkDestroyImage()*.

```
void vkDestroyImage (
        VkDevice                        device,
        VkImage                         image,
        const VkAllocationCallbacks*    pAllocator);
```

Índice

- Para poder acceder a una imagen y generar su contenido es necesario crear un objeto *VkImageView* asociado a cada objeto *VkImage*.
- Para crear una vista se utiliza la función *vkCreateImageView()*.

```
VkResult vkCreateImageView (
    VkDevice                                device,
    const VkImageViewCreateInfo*            pCreateInfo,
    const VkAllocationCallbacks*            pAllocator,
    VkImageView*                            pView);
```

- La información necesaria para crear una vista se almacena en una estructura *VkImageViewCreateInfo*. En este caso, el campo *sType* debe ser VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO.
- Los campos *pNext* y *flags* se han reservado para futuras modificaciones y de momento deben declararse nulos.

```
typedef struct VkImageViewCreateInfo {
    VkStructureType             sType;
    const void*                 pNext;
    VkImageViewCreateFlags      flags;
    VkImage                     image;
    VkImageViewType             viewType;
    VkFormat                    format;
    VkComponentMapping          components;
    VkImageSubresourceRange     subresourceRange;
} VkImageViewCreateInfo;
```

- El campo *image* contiene la referencia a la imagen asociada a la vista.
- El campo *viewType* indica el tipo de imagen que se quiere editar. Para una imagen asociada a una superficie lo lógico es usar el tipo VK_IMAGE_VIEW_TYPE_2D, pero existen tipos 1D, 2D, 3D, CUBE, 1D_ARRAY, 2D_ARRAY y CUBE_ARRAY que se utilizan en otros ámbitos.
- El formato de la vista debe ser compatible con el formato de la imagen. Generalmente se usa el mismo, aunque se consideran formatos compatibles si tienen el mismo número de bits por componente.

- El campo *components* describe el mapeo a realizar sobre los canales de color. La estructura *VkComponentMapping* contiene cuatro campos (r,g,b,a) que permiten referenciar estos canales. Estos campos deben recoger valores de la enumeración *VkComponentSwizzle*. Por ejemplo, el valor VK_COMPONENT_SWIZZLE_IDENTITY indica que el mapeo es al mismo canal, pero también se puede indicar valores como VK_COMPONENT_SWIZZLE_R para referirse a un canal concreto (el R en este caso) o VK_COMPONENT_SWIZZLE_ZERO para indicar un valor fijo (el cero en este caso).

- Las imágenes están distribuidas en capas y en niveles de mipmap. Una vista se vincula a una parte de las capas y niveles de mipmap de la imagen relacionada.
- El campo *subresourceRange* describe el fragmento de la imagen al que se quiere acceder por medio de una estructura *VkImageSubresourceRange*.

```
typedef struct VkImageSubresourceRange {
    VkImageAspectFlags      aspectMask;
    uint32_t                baseMipLevel;
    uint32_t                levelCount;
    uint32_t                baseArrayLayer;
    uint32_t                layerCount;
} VkImageSubresourceRange;
```

- Esta estructura contiene un campo *aspectMask* para seleccionar el contenido a utilizar:
  - VK_IMAGE_ASPECT_COLOR_BIT
  - VK_IMAGE_ASPECT_DEPTH_BIT
  - VK_IMAGE_ASPECT_STENCIL_BIT
  - VK_IMAGE_ASPECT_METADATA_BIT
- También tiene los campos *baseMipLevel* and *levelCount* para seleccionar los niveles de mipmap utilizados y los campos *baseArrayLayer* y *layerCount* para indicar las capas utilizadas.

- Para destruir una vista se utiliza la función *vkDestroyImageView()*.

```
void vkDestroyImageView (
        VkDevice                    device,
        VkImageView                 imageView,
        const VkAllocationCallbacks* pAllocator);
```

Índice

- Existen dos formas de acceder a las imágenes desde los shaders. La primera es acceder directamente como al contenido, lo que permite leer un píxel a partir de su posición. La segunda es acceder mediante un sampler, lo que permite otras operaciones como el filtrado o diversas transformaciones.
- Los samplers se describen con estructuras *VkSampler* y se crean con la función *vkCreateSampler()*.

```
VkResult vkCreateSampler (
        VkDevice                    device,
        const VkSamplerCreateInfo*  pCreateInfo,
        const VkAllocationCallbacks* pAllocator,
        VkSampler*                  pSampler);
```

```
typedef struct VkSamplerCreateInfo {
    VkStructureType              sType;
    const void*                  pNext;
    VkSamplerCreateFlags         flags;
    VkFilter                     magFilter;
    VkFilter                     minFilter;
    VkSamplerMipmapMode          mipmapMode;
    VkSamplerAddressMode         addressModeU;
    VkSamplerAddressMode         addressModeV;
    VkSamplerAddressMode         addressModeW;
    float                        mipLodBias;
    VkBool32                     anisotropyEnable;
    float                        maxAnisotropy;
    VkBool32                     compareEnable;
    VkCompareOp                  compareOp;
    float                        minLod;
    float                        maxLod;
    VkBorderColor                borderColor;
    VkBool32                     unnormalizedCoordinates;
} VkSamplerCreateInfo;
```

- El campo *pNext* debe ser nulo, el campo *flags* debe tener el valor 0 y el campo *sType* es VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO.
- Los campos *magFilter* y *minFilter* definen el proceso de filtrado al magnificar o minimizar la imagen. Los valores pueden ser
  - VK_FILTER_NEAREST
  - VK_FILTER_LINEAR
- El campo *mipmapMode* indica la forma de seleccionar el mipmap al acceder a la imagen. Puede ser
  - VK_SAMPLER_MIPMAP_MODE_NEAREST
  - VK_SAMPLER_MIPMAP_MODE_LINEAR

- Los campos *addressModeU*, *addressModeV* y *addressModeW* describen la forma de extender la imagen cuando se busca una coordenada fura del rango [0,1]. Los valores pueden ser
  - VK_SAMPLER_ADDRESS_MODE_REPEAT
  - VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT
  - VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE
  - VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER
  - VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE
- El campo *mipLodBias* define una constante que se suma en el proceso de selección del mipmap.
- El campo *anisotropyEnable* activa un modo en el que se tiene en cuenta la inclinación de las imágenes al interpolar las coordenadas (filtro anisotrópico). El campo *maxAnisotropy* establece un límite al grado de anisotropía utilizado.

- El campo *compareEnable* permite activar el modo de comparación. En este modo, en lugar de acceder al contenido de la imagen en una posición el sampleado compara ese contenido con el valor de la úlktima coordenada. La función de comparación se esxtablece en el campo *compareOp*.
- El sampler puede configurarse para restringir el rango de niveles de mipmap a utilizar. Los campos *minLod* y *maxLod* indican los límites de este rango.
- El campo *borderColor* almacena el valor a utilizar al extender una componente configurada en modo CLAMP_TO_BORDER.
- El campo *unnormalizedCoordinates* indica que el acceso a la imagen no utiliza coordenadas en el rango [0,1] sino la posición natural del pixel.

- Para destruir un sampler se utiliza la función *vkDestroySampler()*.

```
void vkDestroySampler (
        VkDevice                        device,
        VkSampler                       sampler,
        const VkAllocationCallbacks*    pAllocator);
```

Índice

- Estructuras básicas:

VkBuffer
VkDeviceMemory
VkImage
VkImageView
VkSampler

- Estructuras que encapsulan información para las funciones

VkBufferCreateInfo
VkMemoryAllocateInfo
VkImageCreateInfo
VkImageViewCreateInfo
VkSamplerCreateInfo

- Estructuras auxiliares

VkPhysicalDeviceMemoryProperties
VkMemoryType
VkMemoryHeap
VkMemoryRequirements
VkExtent3D
VkComponentMapping
VkImageSubresourceRange

- Funciones de creación de estructuras:

vkCreateBuffer()
vkAllocateMemory()
vkCreateImage()
vkCreateImageView()
vkCreateSampler()

- Funciones de destrucción de estructuras:

vkDestroyBuffer()
vkFreeMemory()
vkDestroyImage()
vkDestroyImageView()
vkDestroySampler()

- Otras funciones:

vkMapMemory()
vkUnmapMemory()
vkBindBufferMemory()
vkGetBufferMemoryRequirements ()
vkGetPhysicalDeviceMemoryProperties ()