Universidad de Huelva

Tema 2

Vulkan – Instancias y dispositivos

Índice

Índice

- Vulkan es una biblioteca desarrollada por Khronos Group sobre la experiencia acumulada en la programación de OpenGL.
- Se trata de un API para explotar las capacidades de los dispositivos gráficos o de computación paralela.
- El objetivo fundamental es la rapidez de las aplicaciones. Por este motivo Vulkan se ha programado en C (en lugar de utilizar un paradigma de programación orientada a objetos, que produce código más lento en tiempo de ejecución).
- Para aumentar la velocidad de ejecución las funciones de Vulkan no realizan comprobaciones sobre los datos sino que presuponen que son correctos. Para introducir comprobaciones es necesario añadir capas de validación.

- Vulkan es extremadamente detallista en cuanto a la información que hay que suministrar a sus funciones. Esto da una enorme flexibilidad a los programadores, pero también dificulta mucho la programación de cualquier aplicación simple.
- La generación de un simple triángulo requiere una enorme cantidad de código con gran nivel de detalle en cuanto a la configuración del proceso de generación gráfica.
- Desde un principio Vulkan está pensado para ser ejecutado en múltiples hilos (*multithreading*). Esta es una de las características en las que más se ha avanzado respecto de OpenGL (que realiza toda la ejecución sobre la CPU en un único hilo).

- La biblioteca no está planteada sólo para generar gráficos sino que admite otros tipos de dispositivos como los DSPs.
- Es posible utilizar Vulkan en aplicaciones que no tengan nada que ver con la generación de gráficos (como minado de criptomonedas, deep learning o simulaciones científicas).
- La generación de gráficos se contempla como una extensión sobre el núcleo principal de la API.

- Por motivos de eficiencia Vulkan está desarrollado en C. Toda la biblioteca consiste en la definición de una serie de estructuras, de valores constantes y de funciones.
- Las funciones de Vulkan comienzan con el prefijo ***vk*** y estilo *camel-case*. y suelen generar como resultado el tipo de dato *VkResult*, definido como una enumeración (*enum*). Si la función se ejecuta sin problema el resultado es VK_SUCCESS que corresponde al valor 0. Todos los resultados erróneos corresponden a valores negativos.
- Los valores a devolver por las funciones se pasan como punteros.
- Las constantes se definen en letra mayúscula y comienzan con el prefijo VK_

- Las estructuras de Vulkan comienzan con el prefijo ***Vk*** y se nombran en estilo *camel-case*.
- La mayoría de las funciones de Vulkan utilizan como primer argumento la referencia al dispositivo lógico utilizado (*VkDevice*). Algunas utilizan como primer argumento la instancia (*VkInstance*).
- Las funciones de creación de estructuras suelen utilizar una estructura asociada con el nombre ***Vk…CreateInfo***.
- Estas estructuras siempre comienzan con un campo común llamado *sType* que indica el tipo de estructura, un campo *pNext* que permite que en futuras versiones se amplíe la estructura con un puntero a información adicional y campo *flags* con algunas opciones que podrán ampliarse también en futuras versiones.

- Vulkan permite utilizar gestores de memoria dinámica propios. De esta forma, la creación y liberación de estructuras se puede personalizar por medio de un objeto de la clase *VkAllocationCallbacks*.
- En las funciones de la biblioteca que admiten este campo se puede introducir un valor nulo (*nullptr*) para indicar que se utilice el gestor de memoria por defecto.

```
typedef struct VkAllocationCallbacks {
    void*                                   pUserData;
    PFN_vkAllocationFunction                pfnAllocation;
    PFN_vkReallocationFunction              pfnReallocation;
    PFN_vkFreeFunction                      pfnFree;
    PFN_vkInternalAllocationNotification    pfnInternalAllocation;
    PFN_vkInternalFreeNotification          pfnInternalFree;
} VkAllocationCallbacks;
```

Índice

- Una instancia es una estructura que inicializa la biblioteca de Vulkan para poder ejecutar sus funciones en una aplicación.
- Las instancias se describen por medio de estructuras ***VkInstance***. Básicamente se trata de una estructura de datos que encapsula un conjunto de punteros a las funciones de la biblioteca.
- Al crear la instancia se indican también las opciones que se quieren incluir en la aplicación. Estas opciones pueden ser capas y extensiones.

- Una capa (*layer*) es un componente de Vulkan que permite sustituir algunas funciones de la biblioteca por otras con funcionalidades adicionales. De esta forma la llamada a una función puede incorporar más comprobaciones.
- Generalmente se utiliza durante la etapa de desarrollo de las aplicaciones para detectar posibles errores. Para ello se crea la instancia añadiendo una capa de validación.
- Una extensión es un componente de Vulkan que permite añadir nuevas funciones y estructuras al comportamiento básico. Estas extensiones pueden añadirse a la instancia general o a dispositivos concretos.

- Las instancias se crean por medio de la función ***vkCreateInstance()***.

```
VkResult vkCreateInstance (
        const VkInstanceCreateInfo*          pCreateInfo,
        const VkAllocationCallbacks*         pAllocator,
        VkInstance*                          pInstance);
```

- La información necesaria para crear la instancia se introduce mediante una estructura ***VkInstanceCreateInfo***.

```
typedef struct VkInstanceCreateInfo {
    VkStructureType              sType;
    const void*                  pNext;
    VkInstanceCreateFlags        flags;
    const VkApplicationInfo*     pApplicationInfo;
    uint32_t                     enabledLayerCount;
    const char* const*           ppEnabledLayerNames;
    uint32_t                     enabledExtensionCount;
    const char* const*           ppEnabledExtensionNames;
} VkInstanceCreateInfo;
```

- En el caso de *VkInstanceCreateInfo* el campo *sType* debe tener el valor VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO.
- El campo *flags* se ha reservado para futuras versiones y de momento debe dejarse a cero.
- El campo *pApplicationInfo* es un puntero a una estructura adicional que describe la aplicación que estamos construyendo.
- El campo *enabledLayerCount* contiene el número de capas que se desean incluir en la instancia y *ppEnabledLayerNames* es el array con los nombres de esas capas.
- El campo *enabledExtensionCount* contine el número de extensiones a incluir en la instancia y *ppEnabledExtensionNames* es un array con los nombres de estas extensiones.

- La estructura *VkApplicationInfo* permite describir la aplicación.
- El campo sType de esta estructura debe contener el valor VK_STRUCTURE_TYPE_APPLICATION_INFO.
- El resto de campos son de texto libre para indicar el nombre de la aplicación y la versión.

```
typedef struct VkApplicationInfo {
    VkStructureType    sType;
    const void*        pNext;
    const char*        pApplicationName;
    uint32_t           applicationVersion;
    const char*        pEngineName;
    uint32_t           engineVersion;
    uint32_t           apiVersion;
} VkApplicationInfo;
```

- La función *vkEnumerateInstanceLayerProperties()* permite obtener el número de capas disponible y las propiedades de estas capas.
- Ejecutando *vkEnumerateInstanceLayerProperties()* con un valor nulo en el puntero de propiedades se almacena en *pPropertyCount* el número de capas disponible. Una vez reservadas las estructuras necesarias se puede ejecutar de nuevo la función con la referencia al array a estas estructuras.

```
VkResult vkEnumerateInstanceLayerProperties (
    uint32_t*                   pPropertyCount,
    VkLayerProperties*          pProperties);
```

- Las propiedades de las capas se describen en estructuras de tipo *VkLayerProperties*.

```
typedef struct VkLayerProperties {
        char        layerName[VK_MAX_EXTENSION_NAME_SIZE];
        uint32_t    specVersion;
        uint32_t    implementationVersion;
        char        description[VK_MAX_DESCRIPTION_SIZE];
} VkLayerProperties;
```

- Para saber que extensiones de instancia están disponibles para cada capa se utiliza la función *vkEnumerateInstanceExtensionProperties()*.
- Para conocer las extensiones que no están vinculadas a una capa se utiliza el valor *nullptr* en el campo *pLayerName*.
- Con el valor *nullptr* en el campo *pProperties* se puede conocer el número de extensiones disponible.

```
VkResult vkEnumerateInstanceExtensionProperties (
    const char*                 pLayerName,
    uint32_t*                   pPropertyCount,
    VkExtensionProperties*      pProperties);
```

- Para obtener las extensiones disponibles se suele ejecutar la función *vkEnumerateInstanceExtensionProperties()* dos veces.
- En una primera ejecución se pone *pProperties* a nulo para obtener el número de extensiones. A continuación se reserva la memoria para ese número de estructuras. Por último se vuelve a ejecutar la función para obtener las propiedades.

```
typedef struct VkExtensionProperties {
    char        extensionName[VK_MAX_EXTENSION_NAME_SIZE];
    uint32_t    specVersion;
} VkExtensionProperties;
```

- Para poder ejecutar una función de una extensión es necesario obtener en primer lugar un puntero a dicha función. Es importante tener en cuenta que en tiempo de compilación esa función podría no existir.
- Los punteros a las funciones de extensiones pertenecientes a instancias se obtienen con la función *vkGetInstanceProcAddr()*.
- Los punteros a las funciones de extensiones pertenecientes a dispositivos se obtienen con la función *vkGetDeviceProcAddr()*.

```
PFN_vkVoidFunction vkGetInstanceProcAddr (
        VkInstance              instance,
        const char*             pName);

PFN_vkVoidFunction vkGetDeviceProcAddr (
        VkDevice                device,
        const char*             pName);
```

- Para destruir un instancia se usa la función *vkDestroyInstance()*.

```
void vkDestroyInstance (
        VkInstance                      instance,
        const VkAllocationCallbacks*    pAllocator);
```

Índice

- Una vez creada la instancia de Vulkan es necesario seleccionar los dispositivos sobre el que queremos operar. Una instancia permite ejecutar código sobre varios dispositivos al mismo tiempo, por ejemplo, sobre varias tarjetas gráficas instaladas en la misma plataforma.
- Un dispositivo físico se representa mediante un valor de tipo ***VkPhysicalDevice***, que no es más que un descriptor (*handle*).

- Para obtener los dispositivos disponibles se ejecuta dos veces la función *vkEnumeratePhysicalDevices()*, una vez para obtener el número de dispositivos y la segunda para obtener los descriptores.

```
VkResult vkEnumeratePhysicalDevices (
    VkInstance          instance,
    uint32_t*           pPhysicalDeviceCount,
    VkPhysicalDevice*   pPhysicalDevices);
```

- A partir de la referencia a un dispositivo físico se pueden obtener sus propiedades (nombre, fabricante, versión, …), sus características (expresadas por una estructura con numerosos flags) y sus tipos de memoria.

```
void vkGetPhysicalDeviceProperties (
        VkPhysicalDevice                physicalDevice,
        VkPhysicalDeviceProperties*     pProperties);

void vkGetPhysicalDeviceFeatures (
        VkPhysicalDevice                physicalDevice,
        VkPhysicalDeviceFeatures*       pFeatures);

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

- Para enviar trabajos a los dispositivos se utilizan colas (*queue*). Los dispositivos pueden trabajar con varias colas en paralelo. Las colas se distribuyen en familias, que son colas con las mismas características.
- Dado un dispositivo, se pueden obtener las familias de colas soportadas utilizando *vkGetPhysicalDeviceQueueFamilyProperties()*.
- Esta función se suele ejecutar dos veces para obtener primero el número de familias de colas soportadas y luego las propiedades de estas familias.

```
void vkGetPhysicalDeviceQueueFamilyProperties (
        VkPhysicalDevice                physicalDevice,
        uint32_t*                       pQueueFamilyPropertyCount,
        VkQueueFamilyProperties*        pQueueFamilyProperties);
```

- Las propiedades de las familias de colas se expresan mediante la estructura *VkQueueFamilyProperties* .
- El campo *queueFlags* indica el uso que permite una determinada familia: operaciones gráficas, operaciones de computación, operaciones de transferencia…
- El campo *queueCount* indica el número de colas de esa familia que pueden ejecutarse en paralelo en el dispositivo.

```
typedef struct VkQueueFamilyProperties {
    VkQueueFlags        queueFlags;
    uint32_t            queueCount;
    uint32_t            timestampValidBits;
    VkExtent3D          minImageTransferGranularity;
} VkQueueFamilyProperties;
```

Índice

- A partir de un dispositivo físico (***VkPhysicalDevice***) se puede construir un dispositivo lógico (***VkDevice***), que es una representación del dispositivo físico con un determinado estado.
- Se pueden construir varios dispositivos lógicos a partir de un mismo dispositivo físico.
- La mayoría de las funciones de Vulkan tienen como primer argumento el dispositivo lógico sobre el que se aplica la función.

- Para construir el dispositivo lógico se utiliza la función *vkCreateDevice()*.

```
VkResult vkCreateDevice (
        VkPhysicalDevice                    physicalDevice,
        const VkDeviceCreateInfo*           pCreateInfo,
        const VkAllocationCallbacks*        pAllocator,
        VkDevice*                           pDevice);
```

- La función utiliza una estructura *VkDeviceCreateInfo*, que se identifica indicando como campo *sType* el valor VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO.
- La estructura especifica las capas, las extensiones y las familias de colas y las características a utilizar en el dispositivo.

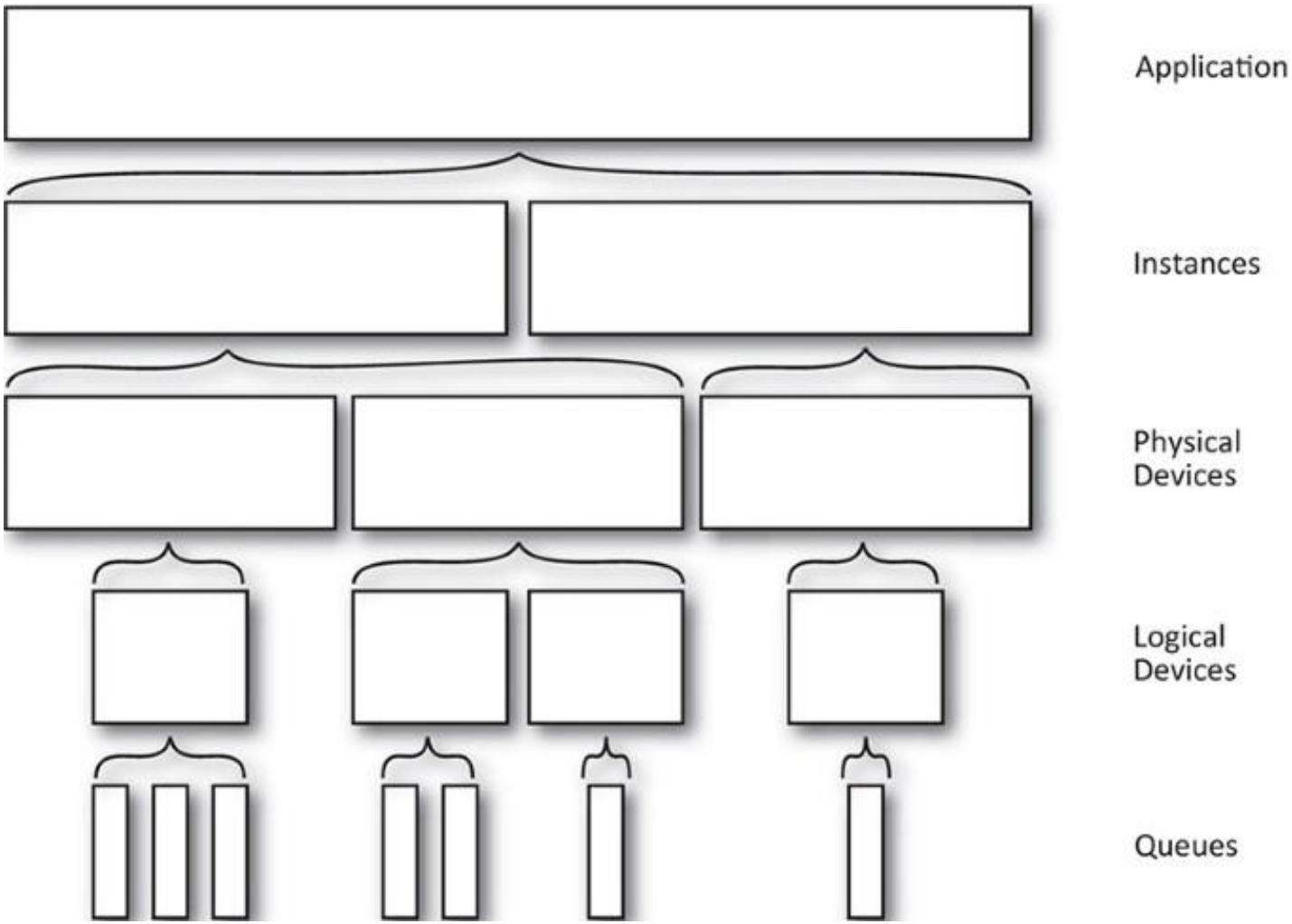
```
typedef struct VkDeviceCreateInfo {
    VkStructureType                 sType;
    const void*                     pNext;
    VkDeviceCreateFlags             flags;
    uint32_t                        queueCreateInfoCount;
    const VkDeviceQueueCreateInfo*  pQueueCreateInfos;
    uint32_t                        enabledLayerCount;
    const char* const*              ppEnabledLayerNames;
    uint32_t                        enabledExtensionCount;
    const char* const*              ppEnabledExtensionNames;
    const VkPhysicalDeviceFeatures* pEnabledFeatures;
} VkDeviceCreateInfo;
```

- Al construir el dispositivo lógico se pueden seleccionar las familias de colas (*queue*) que van a utilizarse sobre ese dispositivo.
- El campo *sType* de la estructura *VkDeviceQueueCreateInfo* toma el valor VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO.
- De momento no se han definido *flags* para esta estructura. El índice de la familia corresponde su posición en el listado de familias del dispositivo y *queueCount* se refiere al número de colas en paralelo.

```
typedef struct VkDeviceQueueCreateInfo {
    VkStructureType              sType;
    const void*                  pNext;
    VkDeviceQueueCreateFlags     flags;
    uint32_t                     queueFamilyIndex;
    uint32_t                     queueCount;
    const float*                 pQueuePriorities;
} VkDeviceQueueCreateInfo;
```

- Para destruir un dispositivo lógico se usa la función *vkDestroyDevice()*.

```
void vkDestroyDevice (
        VkDevice                        device,
        const VkAllocationCallbacks*    pAllocator);
```

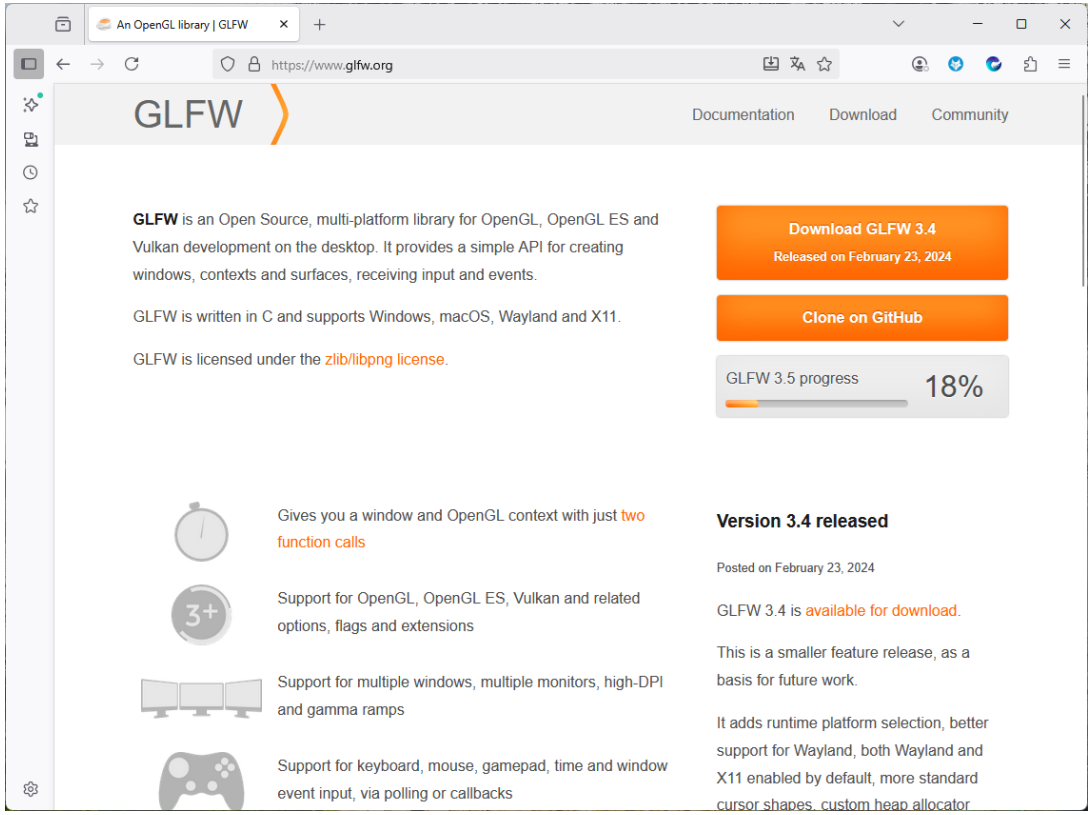

Application
Instances
Physical Devices
Logical Devices
Queues

Índice

- Vulkan es una API multiplataforma que permite desarrollar aplicaciones gráficas en diferentes sistemas operativos como MSWindows, Linux, Android, …
- Las ventanas son objetos que dependen de cada plataforma. Es el sistema operativo el responsable de crear y manejar las ventanas.
- El núcleo de Vulkan contiene las funciones dedicadas a configurar los procesos a ejecutar en las tarjetas gráficas y otros dispositivos de computación paralela. El proceso de mostrar el resultado en una ventana se conoce como presentación y no forma parte del núcleo.
- Para mostrar los gráficos en una ventana, Vulkan utiliza varias extensiones que se conocen comúnmente como WSI (Window System Integration).

- Vulkan no contiene funciones de control de las ventanas. Se limita a configurar la forma en la que las imágenes generadas en el dispositivo se van a presentar en la ventana, pero no incluye funciones de control sobre los eventos de la ventana (teclado, ratón, …).
- Para facilitar la gestión de ventanas se suele utilizar alguna API auxiliar. En este curso utilizaremos GLFW para controlar las ventanas, aunque existen muchas otras alternativas. Por ejemplo, FreeGLUT, SDL, SFML o Allegro.

- https://www.glfw.org/



- GLFW es una biblioteca de código abierto dedicada a la gestión de ventanas con soporte para OpenGL, OpenGL ES y Vulkan. Se trata de una biblioteca escrita en C adaptada a múltiples plataformas (MS-Windows, MacOS, X11, Wayland).
- Para que la ventana gráfica soporte Vulkan es necesario definir la macro GLFW_INCLUDE_VULKAN antes de importar la bibliteca de GLFW.
- Las funciones de la biblioteca dan soporte a múltiples ventanas, monitores y eventos de teclado, ratón, joystick e incluso mando de consolas.
- Para utilizar GLFW hay que comenzar ejecutando la función *glfwInit()*. Al terminar la aplicación debe ejecutarse *glfwTerminate()* para liberar todos los recursos que pueda haberse creado.

- La biblioteca define la estructura *GLFWwindow* para describir las ventanas.
- Para crear una ventana se utiliza la función *glfwCreateWindow()* y para destruirla se usa *glfwDestroyWindow()*.
- Las respuestas a los distintos eventos se configura por medio de punteros a funciones. Típicamente estas funciones deben acceder a la información de la aplicación que ha creado la ventana. Para ello se utiliza la función *glfwSetWindowUserPointer()* que permite almacenar una referencia a un objeto de cualquier tipo, lo que permite acceder posteriormente a este objeto por medio de la función *glfwGetWindowUserPointer()*.

Índice

- Vulkan ofrece una extensión llamada **VK_KHR_surface** que describe las funciones y estructuras genéricas necesarias para desarrollar el proceso de presentación.
- Esta extensión contiene la definición de la estructura ***VkSurfaceKHR***, que describe un superficie sobre la que se puede realizar la presentación de una imagen.
- Además de esta extensión genérica, en cada plataforma hay que incluir una extensión específica. El estándar de Vulkan ofrece ya numerosas extensiones para distintas plataformas :

  VK_KHR_win32_surface, VK_KHR_android_surface, VK_KHR_xcb_surface, VK_KHR_xlib_surface, VK_KHR_wayland_surface, VK_MVK_ios_surface, VK_MVK_macos_surface, VK_GGP_stream_descriptor_surface, VK_FUCHSIA_imagepipe_surface,, VK_EXT_directfb_surface.

- El uso de las funciones de las extensiones WSI es tan común que estas funciones se han declarado en la cabecera de Vulkan de forma que no hay que buscar los punteros a estas funciones a partir de su nombre.
- Para crear una superficie sobre la plataforma Windows se utiliza la función *vkCreateWin32SurfaceKHR()*.

```
VkResult vkCreateWin32SurfaceKHR(
    VkInstance                          instance,
    const VkWin32SurfaceCreateInfoKHR*  pCreateInfo,
    const VkAllocationCallbacks*        pAllocator,
    VkSurfaceKHR*                       pSurface);
```

- La información necesaria para crear una superficie sobre Windows incluye las referencias a los manejadores (*hinstance* y *hwnd*) asociados a la aplicación o módulo y a la ventana de la interfaz gráfica.
- El campo *sType* debe tener el valor VK_STRUCTURE_TYPE_DISPLAY_SURFACE_CREATE_INFO_KHR.
- Los campos *pNext* y *flags* no se utilizan por el momento.

```
typedef struct VkWin32SurfaceCreateInfoKHR {
    VkStructureType                 sType;
    const void*                     pNext;
    VkWin32SurfaceCreateFlagsKHR    flags;
    HINSTANCE                       hinstance;
    HWND                            hwnd;
} VkWin32SurfaceCreateInfoKHR;
```

- GLFW contiene funciones que facilitan la construcción superficies, como *glfwGetRequiredInstanceExtensions()*, que obtiene los nombres de las extensiones necesarias, o *glfwCreateWindowSurface()*, que genera la superficie sin necesidad de llamadas a funciones de la extensión.

```
VkResult glfwCreateWindowSurface (
        VkInstance                      instance,
        GLFWwindow*                     window,
        const VkAllocationCallbacks*    allocator,
        VkSurfaceKHR*                   surface);
```

- No todos los dispositivos físicos soportan la capacidad de presentar imágenes sobre una superficie. Para comprobarlo se puede utilizar una función genérica o una función específica de la plataforma utilizada. La comprobación se realiza sobre las familias de colas que soporta el dispositivo.

```
VkResult vkGetPhysicalDeviceSurfaceSupportKHR(
        VkPhysicalDevice    physicalDevice,
        uint32_t            queueFamilyIndex,
        VkSurfaceKHR        surface,
        VkBool32*           pSupported);

VkBool32 vkGetPhysicalDeviceWin32PresentationSupportKHR(
        VkPhysicalDevice    physicalDevice,
        uint32_t            queueFamilyIndex);
```

Índice

- Estructuras básicas:

VkAllocationCallbacks
VkInstance
VkPhysicalDevice
VkDevice
VkSurfaceKHR

- Estructuras que encapsulan información para las funciones

VkInstanceCreateInfo
VkApplicationInfo
VkDeviceCreateInfo
VkDeviceQueueCreateInfo
VkWin32SurfaceCreateInfoKHR

- Funciones de creación de estructuras:

vkCreateInstance()
vkCreateDevice()
vkCreateWin32SurfaceKHR()

- Funciones de destrucción de estructuras:

vkDestroyInstance()
vkDestroyDevice()
vkDestroySurfaceKHR()

- Funciones relacionadas con instancias:

vkEnumerateInstanceLayerProperties()
vkEnumerateInstanceExtensionProperties()
vkGetInstanceProcAddr()

- Funciones relacionadas con dispositivos:

vkEnumeratePhysicalDevices()
vkGetPhysicalDeviceProperties()
vkGetPhysicalDeviceFeatures()
vkGetPhysicalDeviceMemoryProperties()
vkGetPhysicalDeviceQueueFamilyProperties()
vkGetPhysicalDeviceSurfaceSupportKHR()
vkGetPhysicalDeviceWin32PresentationSupportKHR()
vkGetPhysicalDeviceSurfaceCapabilitiesKHR()
vkGetDeviceProcAddr()