Universidad de Huelva

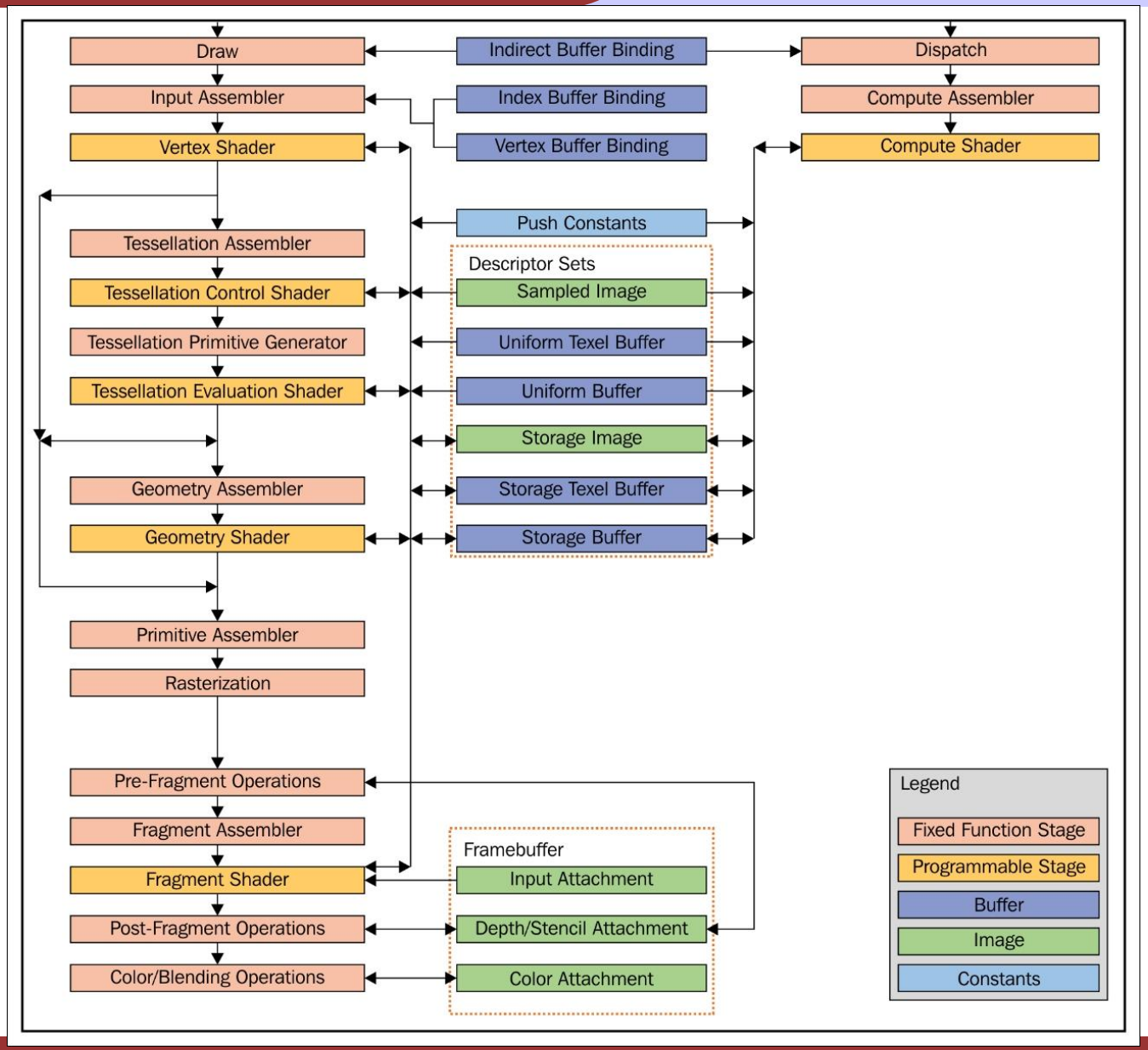
Tema 5

Vulkan – Pipeline

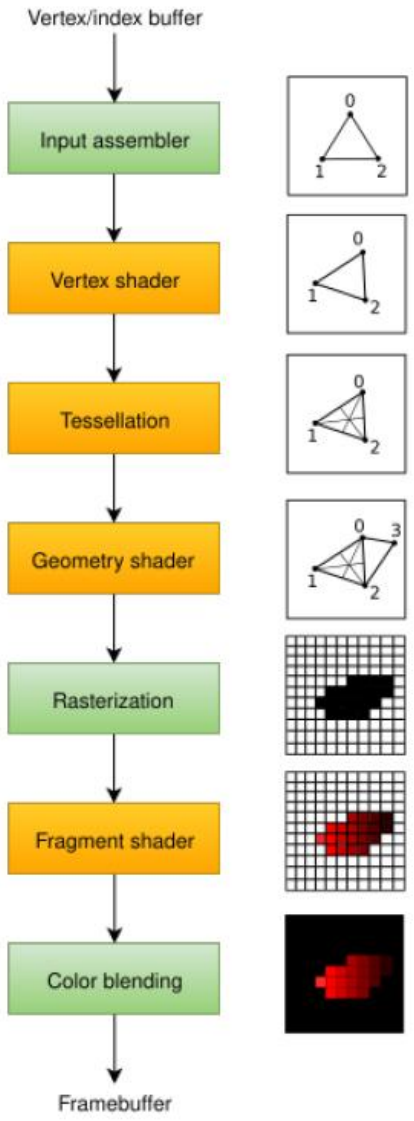
Índice

Índice

- La generación de imágenes en una tarjeta gráfica se realiza siguiendo un conjunto de etapas que se denomina *pipeline* de renderizado.
- Estas etapas pueden ser programables (denominadas *shaders*) o de funcionalidad fija.
- Las etapas programables se configuran mediante unos programas que deben ser cargados en el dispositivo. Vulkan utiliza el lenguaje intermedio SPIR-V para la definición de estos programas.
- Las etapas de funcionalidad fija se deben configurar mediante parámetros en el proceso de definición del *pipeline*.
- La siguiente imagen describe las diferentes etapas del *pipeline* utilizadas en Vulkan.


Draw
Input Assembler
Vertex Shader
Tessellation Assembler
Tessellation Control Shader
Tessellation Primitive Generator
Tessellation Evaluation Shader
Geometry Assembler
Geometry Shader
Primitive Assembler
Rasterization
Pre-Fragment Operations
Fragment Assembler
Fragment Shader
Post-Fragment Operations
Color/Blending Operations
Indirect Buffer Binding
Index Buffer Binding
Vertex Buffer Binding
Push Constants
Descriptor Sets
Sampled Image
Uniform Texel Buffer
Uniform Buffer
Storage Image
Storage Texel Buffer
Storage Buffer
Framebuffer
Input Attachment
Depth/Stencil Attachment
Color Attachment
Dispatch
Compute Assembler
Compute Shader
Legend
Fixed Function Stage
Programmable Stage
Buffer
Image
Constants

- La siguiente imagen muestra el proceso de forma simplificada.
- En primer lugar se leen los vértices que forman la malla de entrada y en el vertex shader se transforman los atributos de estos vértices.
- Las etapas de teselado y geometría permiten transformar las primitivas o crear nuevas.
- A continuación se deciden que píxeles forman parte de cada primitiva y se decide su color en el fragment shader. Por último se mezclan los pixeles de las diferentes primitivas.



- Para describir un proceso de renderizado en Vulkan es necesario crear un objeto *VkPipeline* por medio de la función *vkCreateGraphicsPipelines()*.
- El argumento *pipelineCache* es opcional y se utiliza para aprovechar información previa para acelerar la construcción de estos objetos.

```
VkResult vkCreateGraphicsPipelines (
    VkDevice                                    device,
    VkPipelineCache                             pipelineCache,
    uint32_t                                    createInfoCount,
    const VkGraphicsPipelineCreateInfo*         pCreateInfos,
    const VkAllocationCallbacks*                pAllocator,
    VkPipeline*                                 pPipelines);
```

```
typedef struct VkGraphicsPipelineCreateInfo {
    VkStructureType                                 sType;
    const void*                                     pNext;
    VkPipelineCreateFlags                           flags;
    uint32_t                                        stageCount;
    const VkPipelineShaderStageCreateInfo*          pStages;
    const VkPipelineVertexInputStateCreateInfo*     pVertexInputState;
    const VkPipelineInputAssemblyStateCreateInfo*   pInputAssemblyState;
    const VkPipelineTessellationStateCreateInfo*    pTessellationState;
    const VkPipelineViewportStateCreateInfo*        pViewportState;
    const VkPipelineRasterizationStateCreateInfo*   pRasterizationState;
    const VkPipelineMultisampleStateCreateInfo*     pMultisampleState;
    const VkPipelineDepthStencilStateCreateInfo*    pDepthStencilState;
    const VkPipelineColorBlendStateCreateInfo*      pColorBlendState;
    const VkPipelineDynamicStateCreateInfo*         pDynamicState;
    VkPipelineLayout                                layout;
    VkRenderPass                                    renderPass;
    uint32_t                                        subpass;
    VkPipeline                                      basePipelineHandle;
    int32_t                                         basePipelineIndex;
} VkGraphicsPipelineCreateInfo;
```

- El campo *sType* es VK_GRAPHICS_PIPELINE_CREATE_INFO.
- El campo *pNext* en la versión actual es *nullptr*.
- El campo *flags* puede ser nulo o alguno de los siguientes valores:
  - VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT: indica que no es necesario optimizar el pipeline, de manera que se acelere la creación aunque el resultado sea un pipeline más lento en ejecución.
  - VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT: indica que se pueden crear nuevos pipelines derivados de este, de manera que el intercambio entre ellos pueda hacerse más rápido.
  - VK_PIPELINE_CREATE_DERIVATIVE_BIT: indica que el pipeline a crear es un derivado de un pipeline anterior.

- Para destruir el objeto *pipeline* se utiliza la función *vkDestroyPipeline()*.

```
void vkDestroyPipeline (
        VkDevice                                    device,
        VkPipeline                                  pipeline,
        const VkAllocationCallbacks*                pAllocator);
```

- El resto de la estructura *VkGraphicsPipelineCreateInfo* está formado por campos definidos por nuevas estructuras dedicadas a describir la configuración de cada una de las etapas del renderizado.

Índice

- El campo *stageCount* de la estructura *VkGraphicsPipelineCreateInfo* indica el número de *shaders* que contiene el pipeline. Para configurar correctamente un proceso de renderizado se necesitan al menos el *Vertex Shader* y el *Fragment Shader*.
- El campo *pStages* es un array de estructuras de tipo *VkPipelineShaderStageCreateInfo* que contienen las definiciones de los *shaders*. El número de estructuras es el indicado en *stageCount*.

- La estructura *VkPipelineShaderStageCreateInfo* tiene el siguiente contenido.
- El campo *sType* de la estructura debe tener el valor VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO
- Los campos *pNext* y *flags* deben ser nulos.

```
typedef struct VkPipelineShaderStageCreateInfo {
    VkStructureType                     sType;
    const void*                         pNext;
    VkPipelineShaderStageCreateFlags    flags;
    VkShaderStageFlagBits               stage;
    VkShaderModule                      module;
    const char*                         pName;
    const VkSpecializationInfo*         pSpecializationInfo;
} VkPipelineShaderStageCreateInfo;
```

- El campo *module* contiene la referencia al objeto *VkShaderModule*.
- El campo *pName* contiene el nombre de la función principal del shader. Típicamente es "*main*".
- El campo *stage* indica el tipo de shader. Los valores incluidos son
  - VK_SHADER_STAGE_VERTEX_BIT
  - VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT
  - VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT
  - VK_SHADER_STAGE_GEOMETRY_BIT
  - VK_SHADER_STAGE_FRAGMENT_BIT
- La extensión *VK_KHR_ray_tracing* incluye muchos otros valores

- Para utilizar un shader dentro de un pipeline es necesario crear un objeto *VkShaderModule*. Para crear estos objetos es necesario que el shader esté ya compilado para poder cargarlo como fichero SPIR-V.
- Para crear este objeto se utiliza la función *vkCreateShaderModule()*. Para destruir estos objetos se usa la función *vkDestroyShaderModule()*.

```
VkResult vkCreateShaderModule (
    VkDevice                        device,
    const VkShaderModuleCreateInfo* pCreateInfo,
    const VkAllocationCallbacks*    pAllocator,
    VkShaderModule*                 pShaderModule);

void vkDestroyShaderModule (
    VkDevice                        device,
    VkShaderModule                  shaderModule,
    const VkAllocationCallbacks*    pAllocator);
```

- La información necesaria para crear el objeto se introduce en la estructura *VkShaderModuleCreateInfo*.
- El campo *sType* debe tener el valor VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO.
- Los campos *pNext* y *flags* deben dejarse a nulo.
- El código SPIR-V del shader se incluye en el campo *pCode*. El campo *codeSize* contiene el tamaño en bytes del código.

```
typedef struct VkShaderModuleCreateInfo {
    VkStructureType              sType;
    const void*                  pNext;
    VkShaderModuleCreateFlags    flags;
    size_t                       codeSize;
    const uint32_t*              pCode;
} VkShaderModuleCreateInfo;
```

Índice

- La configuración del proceso de renderizado incluye la descripción de la etapa de entrada de datos utilizados en el *Vertex Shader*. Estos datos corresponden a los atributos de los vértices que se han definido como entradas del Vertex Shader.
- La descripción de los atributos de entrada se realiza por medio de una estructura *VkPipelineVertexInputStateCreateInfo* que está incluida como uno de los campos de *VkGraphicsPipelineCreateInfo*. Esta estructura incluye una lista de enlaces a memoria y una lista de atributos de los vértices.

- Los enlaces indican cuales son los buffers de los que se van a leer los atributos, es decir, los bloques de memoria de la GPU donde se encuentran almacenados los datos.
- La descripción de cada atributo incluye la posición del atributo en el Vertex Shader, el tipo de dato del atributo y el enlace que identifica el buffer en el que buscar esos datos.

Atributos de los vértices

```
in vec3 VPosition;

in vec3 VNormal;

in vec2 VTexCoord;

void main() { … }
```

buffer0		
V0	pos	1.0
		0.5
		3.7
	tex	0.2
		0.6
V1	pos	24.3
		1.2
		15.8
	tex	0.2
		0.65
…	…	…

buffer1		
V0	normal	0.707
		0.707
		0.0
V1	normal	0.0
		0.0
		1.0
V2	normal	0.5
		0.7
		0.5
…	…	…

Atributos de los vértices

```
in vec3 VPosition;

in vec3 VNormal;

in vec2 VTexCoord;

void main() { … }
```

buffer0		
V0	pos	1.0
		0.5
		3.7
	tex	0.2
		0.6
V1	pos	24.3
		1.2
		15.8
	tex	0.2
		0.65
…	…	…

buffer1		
V0	normal	0.707
		0.707
		0.0
V1	normal	0.0
		0.0
		1.0
V2	normal	0.5
		0.7
		0.5
…	…	…

Atributos de los vértices

```
in vec3 VPosition;

in vec3 VNormal;

in vec2 VTexCoord;

void main() { … }
```

buffer0		
V0	pos	1.0
		0.5
		3.7
	tex	0.2
		0.6
V1	pos	24.3
		1.2
		15.8
	tex	0.2
		0.65
…	…	…

buffer1		
V0	normal	0.707
		0.707
		0.0
V1	normal	0.0
		0.0
		1.0
V2	normal	0.5
		0.7
		0.5
…	…	…

- La estructura *VkPipelineVertexInputStateCreateInfo* tiene la siguiente descripción.

```
typedef struct VkPipelineVertexInputStateCreateInfo {
    VkStructureType                             sType;
    const void*                                 pNext;
    VkPipelineVertexInputStateCreateFlags       flags;
    uint32_t                                    vertexBindingDescriptionCount;
    const VkVertexInputBindingDescription*      pVertexBindingDescriptions;
    uint32_t                                    vertexAttributeDescriptionCount;
    const VkVertexInputAttributeDescription*    pVertexAttributeDescriptions;
} VkPipelineVertexInputStateCreateInfo;
```

- Los campos *sType*, *pNext* y *flags* tienen el significado habitual. La estructura describe una lista de "*bindings*" (enlaces) y una lista de atributos de los vértices.
- Los enlaces indican cuales son los buffers de los que se van a leer los atributos. El campo *binding* es un valor que permite identificar al buffer. El campo *stride* indica la cantidad de bytes que hay que tomar del buffer en cada lectura. El campo *inputRate* indica al Vertex Shader como leer el buffer (normalmente se lee en cada vértice VK_VERTEX_INPUT_RATE_VERTEX).

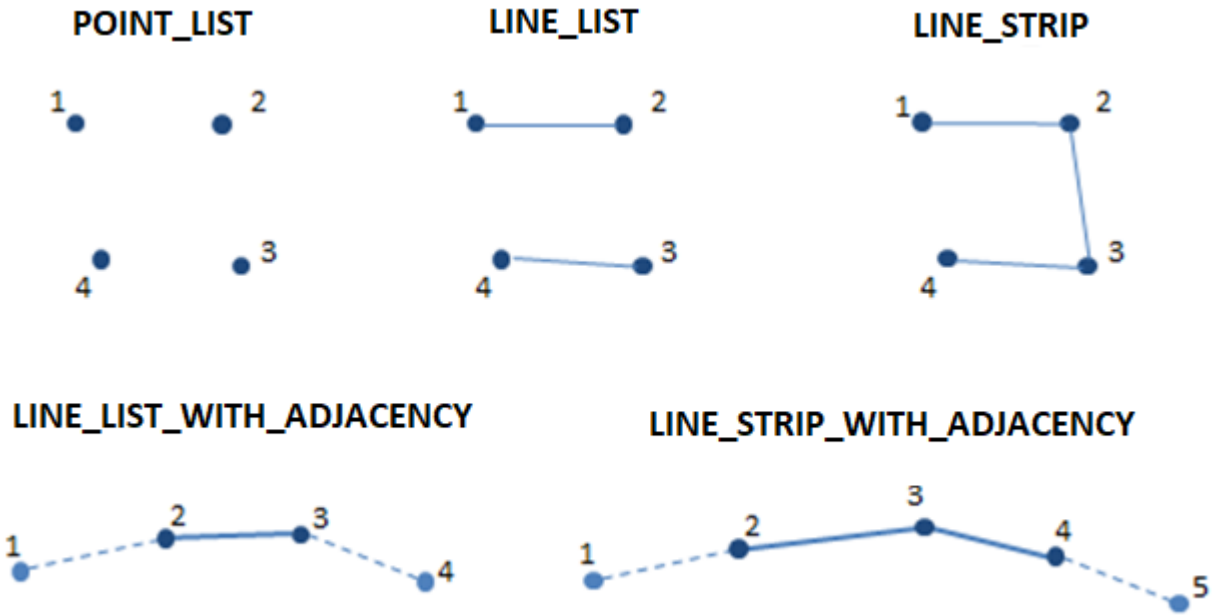
```
typedef struct VkVertexInputBindingDescription {
    uint32_t                    binding;
    uint32_t                    stride;
    VkVertexInputRate           inputRate;
} VkVertexInputBindingDescription;
```

- La descripción de cada atributo se incluye en estructuras de tipo *VkVertexInputAttributeDescription*.
- El campo *location* corresponde a la posición del atributo en el Vertex Shader. Normalmente se fija con un modificador *layout*.
- El campo *binding* permite identificar el buffer
- El campo *format* es el formato del atributo. Por ejemplo, un *vec3* tiene el formato VK_FORMAT_R32G32B32_SFLOAT.
- El campo *offset* es el desplazamiento en el bloque de lectura del buffer.

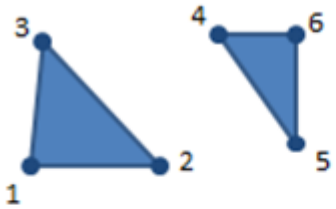
```
typedef struct VkVertexInputAttributeDescription {
    uint32_t        location;
    uint32_t        binding;
    VkFormat        format;
    uint32_t        offset;
} VkVertexInputAttributeDescription;
```

Índice

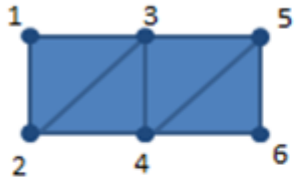
- El siguiente aspecto a configurar es la forma de las primitivas a ensamblar (puntos, líneas , triángulos o *patches*). Vulkan admite las siguientes topologías.



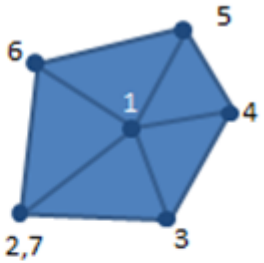
TRIANGLE_LIST



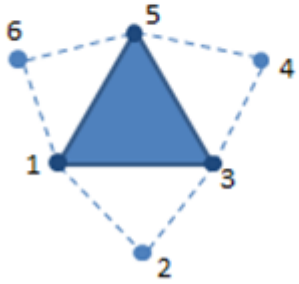
TRIANGLE_STRIP



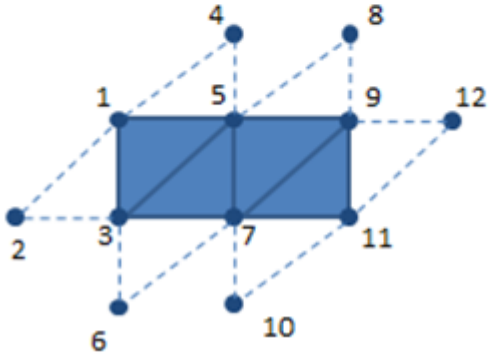
TRIANGLE_FAN



TRIANGLE_LIST_WITH_ADJACENCY



TRIANGLE_STRIP_WITH_ADJACENCY



- Además existe VK_PRIMITIVE_TOPOLOGY_PATCH_LIST que define la topología de puntos utilizada en la etapa de teselado

- El campo *pInputAssemblyState* es un puntero a una estructura de tipo *VkPipelineInputAssemblyStateCreateInfo* que permite describir la forma de las primitivas a dibujar.
- El campo sType debe tene el valor
  VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO
- En la versión actual los campos *pNext* y *flags* deben ser nulos.
- El campo *primitiveRestartEnable* se utiliza para trocear las listas.

```
typedef struct VkPipelineInputAssemblyStateCreateInfo {
    VkStructureType                            sType;
    const void*                                pNext;
    VkPipelineInputAssemblyStateCreateFlags    flags;
    VkPrimitiveTopology                        topology;
    VkBool32                                   primitiveRestartEnable;
} VkPipelineInputAssemblyStateCreateInfo;
```

- El campo *topology* puede tomar los valores siguientes:
  - VK_PRIMITIVE_TOPOLOGY_POINT_LIST
  - VK_PRIMITIVE_TOPOLOGY_LINE_LIST
  - VK_PRIMITIVE_TOPOLOGY_LINE_STRIP
  - VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST
  - VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP
  - VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN
  - VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY
  - VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY
  - VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY
  - VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY
  - VK_PRIMITIVE_TOPOLOGY_PATCH_LIST

Índice

- El campo *pTessellationState* es un puntero a una estructura de tipo *VkPipelineTessellationStateCreateInfo*.
- El campo *sType* debe tene el valor VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO
- En la versión actual los campos *pNext* y *flags* deben ser nulos.
- El único campo a completar es *patchControlPoints*, que indica el número de puntos que forman un PATCH.

```
typedef struct VkPipelineTessellationStateCreateInfo {
    VkStructureType                           sType;
    const void*                               pNext;
    VkPipelineTessellationStateCreateFlags    flags;
    uint32_t                                  patchControlPoints;
} VkPipelineTessellationStateCreateInfo;
```

Índice

- Se conoce como Viewport la descripción del tamaño de la imagen a generar. Normalmente coincide con el tamaño completo de la imagen a mostrar, aunque puede configurarse un proceso de renderizado que solo genere un trozo. La configuración del viewport también incluye información sobre el recorte a mostrar (*scissor*), que será el trozo del viewport que se copiará en la imagen.
- Conocido el viewport, las coordenadas de posición de los vértices se transforman en coordenadas de posición en la imagen.

- El campo *pViewportState* es un puntero a una estructura de tipo *VkPipelineViewportStateCreateInfo* que describe el tamaño de la imagen que se va a crear en el pipeline.
- El campo *sType* debe tene el valor

VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO

```
typedef struct VkPipelineViewportStateCreateInfo {
    VkStructureType                       sType;
    const void*                           pNext;
    VkPipelineViewportStateCreateFlags    flags;
    uint32_t                              viewportCount;
    const VkViewport*                     pViewports;
    uint32_t                              scissorCount;
    const VkRect2D*                       pScissors;
} VkPipelineViewportStateCreateInfo;
```

- El campo *viewportCount* indica cuantos tamaños se están configurando.
- Las estructuras *VkViewport* permiten definir la transformación entre coordenadas homogeneas y coordenadas de ventana.
- El campo *scissorCount* indica cuantos recortes se están configurando y *pScissors* contiene la lista de recortes. Un recorte define el rectángulo sobre la imagen 2D que se va a dibujar realmente.

```
typedef struct VkViewport {
        float x;
        float y;
        float width;
        float height;
        float minDepth;
        float maxDepth;
} VkViewport;
```

```
typedef struct VkRect2D {
        VkOffset2D offset;
        VkExtent2D extent;
} VkRect2D;
```

Índice

- El campo *pRasterizationState* es un puntero a una estructura de tipo *VkPipelineRasterizationStateCreateInfo*.

```
typedef struct VkPipelineRasterizationStateCreateInfo {
    VkStructureType                            sType;
    const void*                                pNext;
    VkPipelineRasterizationStateCreateFlags    flags;
    VkBool32                                   depthClampEnable;
    VkBool32                                   rasterizerDiscardEnable;
    VkPolygonMode                              polygonMode;
    VkCullModeFlags                            cullMode;
    VkFrontFace                                frontFace;
    VkBool32                                   depthBiasEnable;
    float                                      depthBiasConstantFactor;
    float                                      depthBiasClamp;
    float                                      depthBiasSlopeFactor;
    float                                      lineWidth;
} VkPipelineRasterizationStateCreateInfo;
```

Rasterización

- El campo *sType* debe ser VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO
- Los campos *pNext* y *flags* de momento deben ser nulos.
- El campo *depthClampEnable* permite activar la opción de depth clamp. Por defecto, cuando un fragmento se encuentra a una profundidad fuera de rango no se dibuja. Con esta opción se le asigna la profundidad límite y se sigue dibujando.
- El campo *rasterizerDiscardEnable*, si está activo, descarta toda la fase de rasterización de manera que el *pipeline* no llega a ejecutar el *Fragment Shader* ni a producir imágenes. Esto se utiliza para dirigir la salida de los shaders anteriores a memoria y reutilizarla posteriormente.

Rasterización

- El campo *polygonMode* indica la forma de dibujar las primitivas y puede tomar los valores VK_POLYGON_MODE_FILL, VK_POLYGON_MODE_LINE y VK_POLYGON_MODE_POINT.
- El campo *cullMode* indica si se van a eliminar polígonos en función de la cara mostrada. Si toma el valor VK_CULL_MODE_FRONT_BIT los polígonos mostrados por la cara frontal se desechan. Si toma el valor VK_CULL_MODE_BACK_BIT se descartan los polígonos mostrados por la cara posterior. Un valor cero muestra todos los polígonos.
- El campo *frontFace* indica como identificar la cara frontal. Puede tener los valores VK_FRONT_FACE_COUNTER_CLOCKWISE o VK_FRONT_FACE_CLOCKWISE.

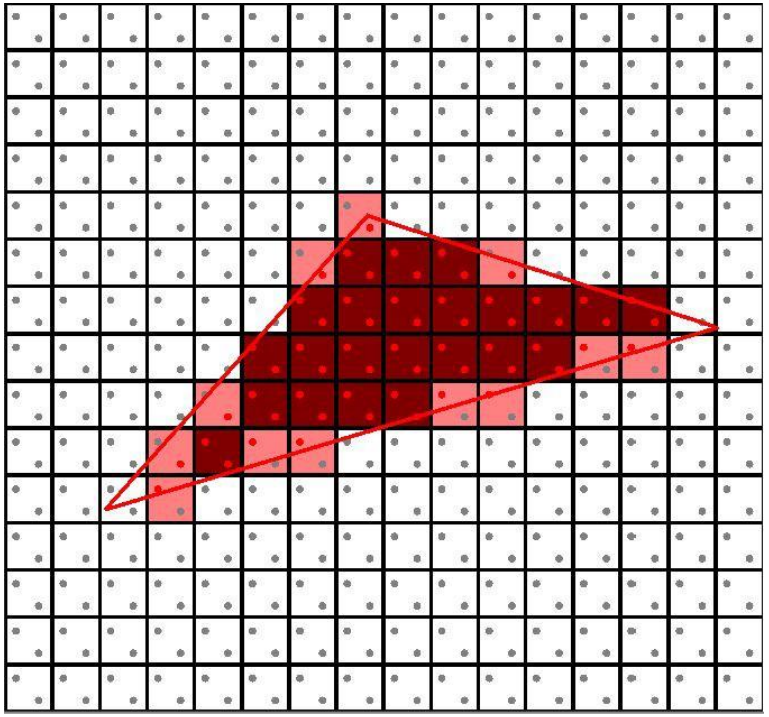
Rasterización

- Los campos *depthBiasEnable*, *depthBiasConstantFactor*, *depthBiasClamp* y *depthBiasSlopeFactor* se utilizan para añadir un desplazamiento en la profundidad de los fragmentos. Esto se utiliza para evitar un problema conocido como *z-fighting* o *depth-fighting* que afecta al dibujo de objetos que están en contacto.
- El campo *lineWidth* contiene el grosor de las líneas a dibujar. Esto se refiere tanto a las topologías de línea como a los polígonos dibujados en modo línea.

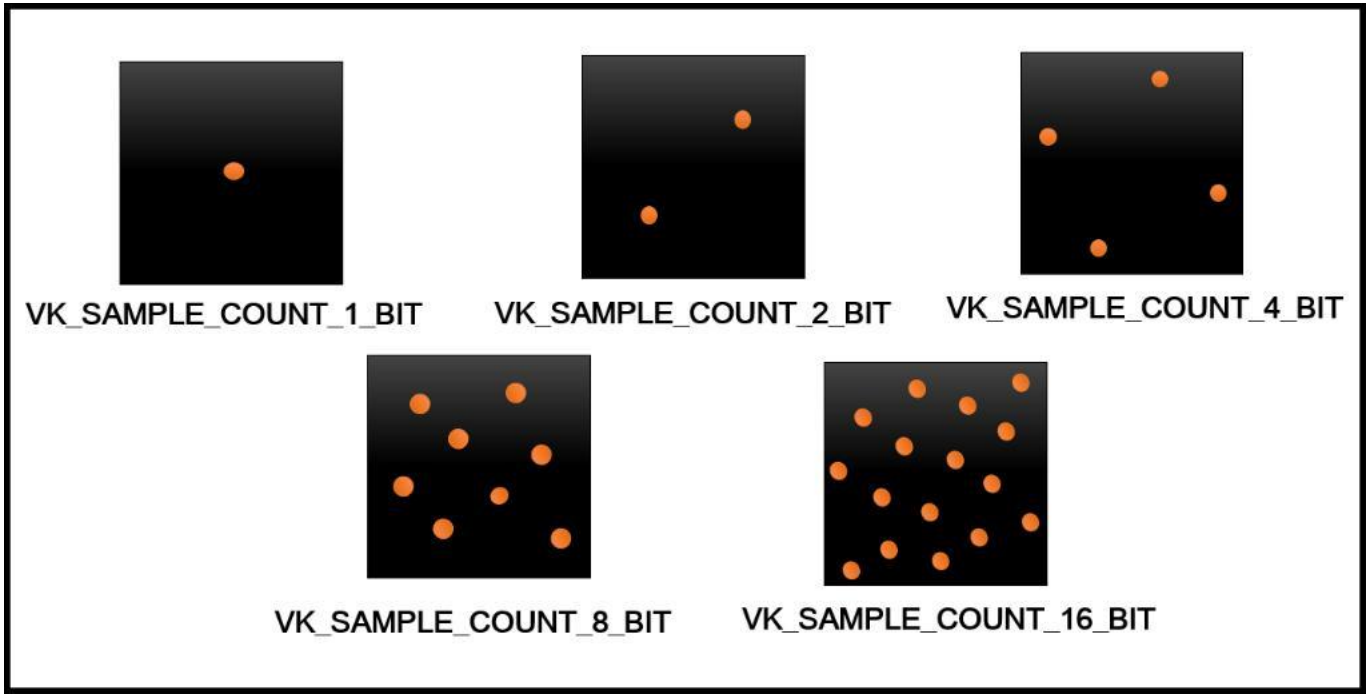
Índice

- Por defecto el color del pixel se calcula con la posición del centro del pixel, pero se pueden configurar procesos de multisampleado en los que el color del pixel es un promedio del cálculo realizado sobre varias posiciones.
- Existen dos formas de hacer esto: multisampleado y supersampleado.

- El multisampleado consiste en estudiar varias posiciones dentro del pixel para ver si pertenecen o no a la primitiva, pero solo se ejecuta el Fragment Shader sobre una posición.
- El supersampleado ejecuta el *Fragment Shader* sobre varias posiciones del pixel para obtener un color promedio.



- La configuración del proceso de sampleado permite elegir entre multisampleado y supersampleado y definir el número de puntos que se van a estudiar para cada pixel.



- El campo *pMultisampleState* es un puntero a una estructura de tipo *VkPipelineMultisampleStateCreateInfo* que describe si se van a utilizar múltiples cálculos para cada pixel.

```
typedef struct VkPipelineMultisampleStateCreateInfo {
    VkStructureType                         sType;
    const void*                             pNext;
    VkPipelineMultisampleStateCreateFlags   flags;
    VkSampleCountFlagBits                   rasterizationSamples;
    VkBool32                                sampleShadingEnable;
    float                                   minSampleShading;
    const VkSampleMask*                     pSampleMask;
    VkBool32                                alphaToCoverageEnable;
    VkBool32                                alphaToOneEnable;
} VkPipelineMultisampleStateCreateInfo;
```

- El campo *rasterizationSamples* define el número de puntos que se van a estudiar para cada pixel. Puede tomar los valores:
  - VK_SAMPLE_COUNT_1_BIT
  - VK_SAMPLE_COUNT_2_BIT
  - VK_SAMPLE_COUNT_4_BIT
  - VK_SAMPLE_COUNT_8_BIT
  - VK_SAMPLE_COUNT_16_BIT
  - VK_SAMPLE_COUNT_32_BIT
  - VK_SAMPLE_COUNT_64_BIT
- El campo *sampleShadingEnable* permite elegir entre multisampleado (VK_FALSE) y supersampleado (VK_TRUE).

- El supersampleado no obliga a evaluar el *Fragment Shader* sobre todos los puntos, sino sobre un porcentaje mínimo que se define en el campo *minSampleShading* .
- El campo *pSampleMask* se utiliza para indicar cuales de los valores generados por el *Fragment Shader* al hacer supersampleado se van a almacenar en el framebuffer.
- El campo *alphaToCoverageEnable* permite activar un modo de sampleado en el que en la primera ejecución del *Fragment Shader* se almacena en el canal alpha un valor de covertura específico de cada pixel (como un *minSampleShading* generado para cada pixel).
- Si el campo anterior está activo el resultado final del canal alpha se puede calcular como el resto de componentes o como el valor uno si se activa el campo *alphaToOneEnable;*

Índice

- El test de profundidad consiste en comparar la profundidad (componente Z) del pixel a dibujar con la profundidad del pixel que se haya analizado anteriormente en la misma posición. Esto permite que los objetos más cercanos tapen a los más lejanos.
- La función de comparación es configurable. Normalmente se usa la función LESS o la función LESS_OR_EQUAL aunque hay muchas otras funciones disponibles que permiten generar otros efectos.
- Si se pasa el test, lo normal es almacenar el valor de profundidad del nuevo pixel pero también se puede configurar el renderizado de forma que no se dibuje el pixel.

- Otro posibilidad es comparar la profundidad con un rango de manera que se pase el test si la profundidad del pixel se encuentra entre unos valores máximo y mínimo.
- El test de plantilla (*stencil*) se realiza después del test de profundidad de manera opcional. Se puede configurar de forma distinta este test para los píxeles mostrados en una cara frontal o en una cara posterior.
- El test consiste en estudiar el valor almacenado en *stencil buffer* en la posición del pixel. Si el resultado es positivo se dibuja el pixel. Después del test se actualiza el valor del *stencil buffer*.
- Se pueden configurar diversas operaciones de comparación y diversas funciones de actualización.

- El campo *pDepthStencilState* es un puntero a una estructura *VkPipelineDepthStencilStateCreateInfo* define la forma en la que se realizan los test de profundidad (*depth*) y de plantilla (*stencil*).

```
typedef struct VkPipelineDepthStencilStateCreateInfo {
    VkStructureType                           sType;
    const void*                               pNext;
    VkPipelineDepthStencilStateCreateFlags flags;
    VkBool32                                  depthTestEnable;
    VkBool32                                  depthWriteEnable;
    VkCompareOp                               depthCompareOp;
    VkBool32                                  depthBoundsTestEnable;
    VkBool32                                  stencilTestEnable;
    VkStencilOpState                          front;
    VkStencilOpState                          back;
    float                                     minDepthBounds;
    float                                     maxDepthBounds;
} VkPipelineDepthStencilStateCreateInfo;
```

- El campo *sType* de la estructura debe tomar el valor VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_CREATE_INFO
- Los campos *pNext* y *flags* deben ser nulos.
- El test de profundidad consiste en comparar la profundidad (componente Z) del pixel a dibujar con la profundidad del pixel que se haya analizado anteriormente en la misma posición. Para activarlo se utiliza el campo *depthTestEnable*.
- La comparación a utilizar se especifica mediante el campo *depthCompareOp*. Suele utilizarse VK_COMPARE_OP_LESS o VK_COMPARE_OP_LESS_OR_EQUAL
- Si se pasa el test, lo normal es almacenar el valor de profundidad del nuevo pixel. Esto debe indicarse por medio del campo *depthWriteEnable*.

- Otro posibilidad es comparar la profundidad con un rango. Para activar esa opción se utiliza el campo *depthBoundsTestEnable* y se utilizan los campos *minDepthBounds* y *maxDepthBounds* para indicar el rango.
- El test de plantilla se realiza después del test de profundidad, activando el campo *stencilTestEnable*.
- Se puede configurar de forma distinta este test para los pixeles mostrados en una cara frontal (campo *front*) o en una cara posterior (campo *back*).
- La estuctura *VkStencilOpState* permite configurar tanto la operación de comparación como la acción a realizar si el test es positivo o negativo.

Índice

5.10 Color blend state

- Una vez superados los test de profundidad y de plantilla y obtenido el color del pixel por medio del Fragment Shader se puede escribir ese color en la posición del pixel. En ese momento existe un valor del pixel actual y un valor del pixel anterior. En este punto se puede decidir mezclar ambos colores para tener en cuenta la transparencia. De esta forma se pueden dibujar objetos traslúcidos.
- Existen diferentes funciones de mezcla y parámetros de configuración de estas funciones que deben definirse para describir esta etapa.

- El campo *pColorBlendState* es un puntero a una estructura de tipo *VkPipelineColorBlendStateCreateInfo* que describe la forma de mezclar colores entre píxeles con niveles de transparencia.

```
typedef struct VkPipelineColorBlendStateCreateInfo {
    VkStructureType                               sType;
    const void*                                   pNext;
    VkPipelineColorBlendStateCreateFlags          flags;
    VkBool32                                      logicOpEnable;
    VkLogicOp                                     logicOp;
    uint32_t                                      attachmentCount;
    const VkPipelineColorBlendAttachmentState*    pAttachments;
    float                                         blendConstants[4];
} VkPipelineColorBlendStateCreateInfo;
```

5.10 Color blend state

- En este caso el campo sType debe tomar el valor VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO
- Los campos *pNext* y *flags* deben ser nulos.
- Para activar la operación de mezcla hay que asignar VK_TRUE en el campo *logicOpEnable*.
- La forma de mezclar los componentes se especifica en un *VkPipelineColorBlendAttachmentState*. Estas configuraciones pueden utilizar constantes que se introducen en el campo *blendConstants*.

Índice

- El pipeline contiene muchísima información sobre la configuración de cada una de sus etapas. Si queremos utilizar configuraciones diferentes al dibujar distintas partes de la imagen necesitaríamos crear varios pipelines.
- Vulkan permite indicar que algunos de los aspectos configurados van a ser dinámicos y pueden ser modificados por comandos sin necesidad de tener que crear un pipeline diferente. Por ejemplo, se puede indicar que el grosor de las líneas es una información dinámica, de manera que se pueda modificar este parámetro sin necesidad de utilizar pipelines diferentes para cada grosor.

- El campo *pDynamicState* de la estructura *VkGraphicsPipelineCreateInfo* es un puntero a una estructura de tipo *VkPipelineDynamicStateCreateInfo*.

```
typedef struct VkPipelineDynamicStateCreateInfo {
    VkStructureType                         sType;
    const void*                             pNext;
    VkPipelineDynamicStateCreateFlags       flags;
    uint32_t                                dynamicStateCount;
    const VkDynamicState*                   pDynamicStates;
} VkPipelineDynamicStateCreateInfo;
```

- El campo *sType* de esta estructura debe tener el valor VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO
- Los campos *pNext* y *flags* deben estar a nulo.
- El campo *dynamicStateCount* indica el número de variables dinámicas que se va a configurar en el pipeline.
- El campo *pDynamicStates* contiene un array indicando las variables que se van a configurar de manera dinámica.
- Cada posible variable dinámica tiene asociado un comando para asignar su valor al pipeline.
- Existen extensiones, como *VK_EXT_extended_dynamic_state*, que permiten extender el conjunto de variables que se pueden definir como dinámicas.

Variables dinámicas

VkDynamicState	Función
VK_DYNAMIC_STATE_VIEWPORT	**vkCmdSetViewport()**
VK_DYNAMIC_STATE_SCISSOR	**vkCmdSetScissor()**
VK_DYNAMIC_STATE_LINE_WIDTH	**vkCmdSetLineWidth()**
VK_DYNAMIC_STATE_DEPTH_BIAS	**vkCmdSetDepthBias()**
VK_DYNAMIC_STATE_BLEND_CONSTANTS	**vkCmdSetBlendConstants()**
VK_DYNAMIC_STATE_DEPTH_BOUNDS	**vkCmdSetDepthBounds()**
VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK	**vkCmdSetStencilCompareMask()**
VK_DYNAMIC_STATE_STENCIL_WRITE_MASK	**vkCmdSetStencilWriteMask()**
VK_DYNAMIC_STATE_STENCIL_REFERENCE	**vkCmdSetStencilReference()**

Índice

- Para completar la configuración del proceso de renderizado hay que describir la forma de acceder a los valores de las variables uniformes definidas en los *shaders*. Por ejemplo, las matrices de transformación espacial, la información sobre el modelo de iluminación o las texturas son valores que se suelen introducir en el pipeline mediante este tipo de variables.
- La estructura *VkPipelineLayout* contiene la lista de objetos *VkDescriptorSetLayout* que describen los diferentes descriptor sets utilizados en el renderizado. También es la estructura donde se definen las *push constants* que se vayan a incluir.

- Las variables uniformes se pueden tratar de dos formas: como *push constants* y como descriptores.
- Las *push constants* se pueden asignar directamente desde el buffer de comandos. Para ello el renderizado puede incluir un único bloque descrito como *push_constant* por medio del modificador *layout*. Para asignar el valor del bloque de *push contants* dentro de un buffer de comandos se utiliza la función *vkCmdPushConstants()*.
- Los descriptores se definen y se cargan como buffers, de una forma parecida a los buffers de vértices o de índices.

- Las *push constants* permiten una asignación de valores más rápida, pero su uso está limitado a un único bloque con un tamaño máximo.
- Los descriptores, por su parte, no están limitados a uno por lo que pueden definirse varios descriptores en cada *shader* y compartirse algunos entre varios *shaders*.
- Los descriptores pueden ser de diferente tipo. Por ejemplo, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER describe un descriptor que se almacena en un buffer de variables uniformes; VK_DESCRIPTOR_TYPE_SAMPLER describe un descriptor que puede almacenar una textura, …

5.12 Pipeline layout

- Los descriptores se agrupan en conjuntos denominados *descriptor sets*. En una plantilla de renderizado pueden utilizarse varios descriptor sets.
- En los *shaders* se utiliza el modificador *layout* para identificar los conjuntos y los descriptores a los que pertenece un bloque uniforme. El valor *set* permite asociar el bloque a un conjunto y el valor *binding* indica el índice del descriptor dentro del conjunto.

- El campo *layout* de la estructura *VkGraphicsPipelineCreateInfo* es una estructura de tipo *VkPipelineLayout* que permite describir los *Descriptor Sets* y las *Push Constants* del proceso de renderizado.
- Para crear esta estructura se utiliza la función *vkCreatePipelineLayout()*.

```
VkResult vkCreatePipelineLayout (
    VkDevice                                device,
    const VkPipelineLayoutCreateInfo*       pCreateInfo,
    const VkAllocationCallbacks*            pAllocator,
    VkPipelineLayout*                       pPipelineLayout);
```

- El campo *setLayoutCount* contiene el número de *DescriptorSets* de la plantilla y el campo *pSetLayouts* la lista de objetos *VkDescriptorSetLayout*
- Las *push constants* deben estar incluidas en un único bloque. En la plantilla se indica a qué rangos de ese bloque tiene acceso cada shader.

```
typedef struct VkPipelineLayoutCreateInfo {
    VkStructureType                 sType;
    const void*                     pNext;
    VkPipelineLayoutCreateFlags     flags;
    uint32_t                        setLayoutCount;
    const VkDescriptorSetLayout*    pSetLayouts;
    uint32_t                        pushConstantRangeCount;
    const VkPushConstantRange*      pPushConstantRanges;
} VkPipelineLayoutCreateInfo;
```

- Para generar un objeto *VkDescriptorSetLayout* se utiliza la función *vkCreateDescriptorSetLayout()*, que utiliza una estructura *VkDescriptorSetLayoutCreateInfo*.

```
VkResult vkCreateDescriptorSetLayout (
        VkDevice                                    device,
        const VkDescriptorSetLayoutCreateInfo*      pCreateInfo,
        const VkAllocationCallbacks*                pAllocator,
        VkDescriptorSetLayout*                      pSetLayout);
```

- La estructura *VkDescriptorSetLayoutCreateInfo* contiene los campos habituales *sType*, *pNext* y *flags* y una lista de bindings que describen el contenido del conjunto de descriptores. El campo *bindingCount* indica el número de elementos del conjunto y el campo *pBindings* contiene la descripción de cada elemento (*VkDescriptorSetLayoutBinding*)

```
VkDescriptorSetLayoutCreateInfo {
    VkStructureType                    sType;
    const void*                        pNext;
    VkDescriptorSetLayoutCreateFlags   flags;
    uint32_t                           bindingCount;
    const VkDescriptorSetLayoutBinding* pBindings;
} VkDescriptorSetLayoutCreateInfo;
```

- La estructura *VkDescriptorSetLayoutBinding* describe cada elemento del descriptor set.
- Lo más importante de esta estructura es el campo *binding*, que corresponde al valor introducido en los shaders por medio del modificador *layout* y el campo *descriptorType* que define el tipo de descriptor que se está enlazando.

```
typedef struct VkDescriptorSetLayoutBinding {
    uint32_t                binding;
    VkDescriptorType        descriptorType;
    uint32_t                descriptorCount;
    VkShaderStageFlags      stageFlags;
    const VkSampler*        pImmutableSamplers;
} VkDescriptorSetLayoutBinding;
```

Índice

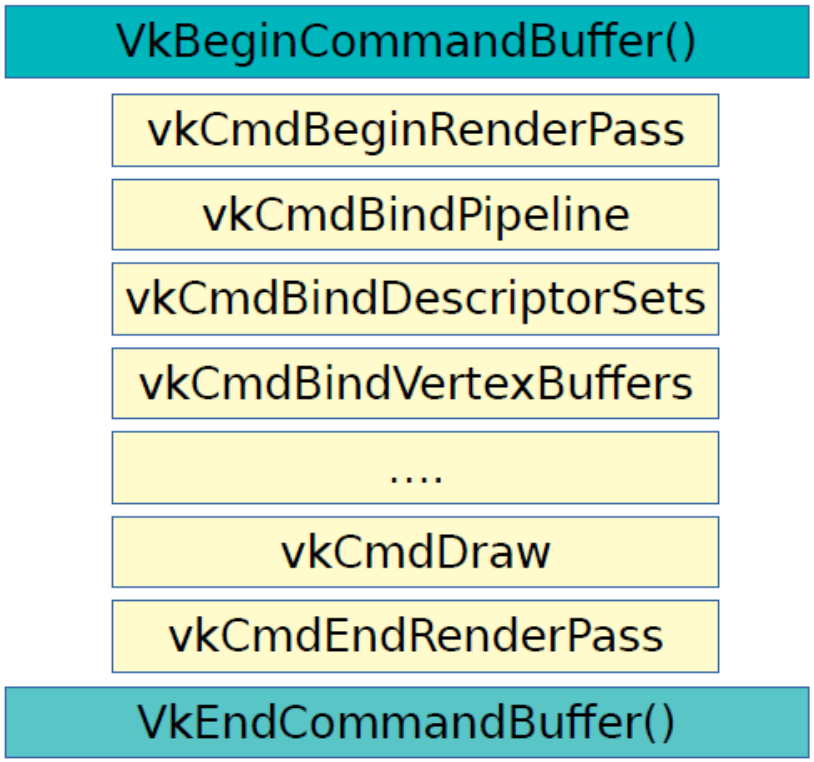
- Los descriptores se definen y se cargan como buffers, de una forma parecida a los buffers de vértices o de índices.
- Las *push constants* permiten una asignación de valores más rápida, pero su uso está limitado a un único bloque con un tamaño máximo.
- Los descriptores, por su parte, no están limitados a uno por lo que pueden definirse varios descriptores en cada *shader* y compartirse algunos entre varios *shaders*.
- Los descriptores pueden ser de diferente tipo. Por ejemplo, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER describe un descriptor que se almacena en un buffer de variables uniformes; VK_DESCRIPTOR_TYPE_SAMPLER describe un descriptor que puede almacenar una textura, …

- Los descriptores se agrupan en conjuntos denominados *descriptor sets*. En una plantilla de renderizado pueden utilizarse varios descriptor sets.
- En los *shaders* se utiliza el modificador *layout* para identificar los conjuntos y los descriptores a los que pertenece un bloque uniforme. El valor *set* permite asociar el bloque a un conjunto y el valor *binding* indica el índice del descriptor dentro del conjunto.

- Para generar descriptores se utiliza un *descriptor pool*, que se crea con la función *vkCreateDescriptorPool()*.
- Una vez creado el descriptor pool, se pueden crear los descriptor sets con la función *vkAllocateDescriptorSets()*.
- Los descriptor sets se almacenan en estructuras del tipo *VkDescriptorSet*.
- Para asignar los buffers a los descriptores hay que actualizar los descriptor sets por medio de la función *vkUpdateDescriptorSets()*.

Índice

- La manera de rellenar un buffer de comandos es la siguiente:



- La función *vkCmdBindPipeline()* permite asignar el pipeline a utilizar en el proceso de renderizado.
- 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()*

- La función *vkCmdBindVertexBuffers()* permite vincular los buffers de vértices a la entrada del pipeline.
- Como los atributos pueden estar almacenados en varios buffers, el parámetro *pBuffers* contiene una lista de buffers. El parámetro *bindingCount* indica el número de buffers a vincular.

```
void vkCmdBindVertexBuffers (
    VkCommandBuffer          commandBuffer,
    uint32_t                 firstBinding,
    uint32_t                 bindingCount,
    const VkBuffer*          pBuffers,
    const VkDeviceSize*      pOffsets);
```

- Cuando el dibujo se va a realizar de forma indexada hay que crear un buffer de índices y asociarle una memoria de igual forma que se hace con los atributos.
- Para indicar cual es el buffer de índices a utilizar en el comando DrawIndexed se utiliza la función *vkCmdBindIndexBuffer()*.
- El campo *indexType* indica el tipo de dato que utiliza el buffer, por ejemplo VK_INDEX_TYPE_UINT16 indica que los índices se almacenan como "short" (enteros de 16 bits).

```
void vkCmdBindIndexBuffer (
        VkCommandBuffer commandBuffer,
        VkBuffer        buffer,
        VkDeviceSize    offset,
        VkIndexType     indexType);
```

- El comando *vkCmdBindDescriptorSets()* indica la asignación de los descriptor sets al proceso de renderizado.

```
void vkCmdBindDescriptorSets (
        VkCommandBuffer             commandBuffer,
        VkPipelineBindPoint         pipelineBindPoint,
        VkPipelineLayout            layout,
        uint32_t                    firstSet,
        uint32_t                    descriptorSetCount,
        const VkDescriptorSet*      pDescriptorSets,
        uint32_t                    dynamicOffsetCount,
        const uint32_t*             pDynamicOffsets);
```

- 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.

```
void vkCmdDraw (
        VkCommandBuffer commandBuffer,
        uint32_t        vertexCount,
        uint32_t        instanceCount,
        uint32_t        firstVertex,
        uint32_t        firstInstance);

void vkCmdDrawIndexed (
        VkCommandBuffer commandBuffer,
        uint32_t        indexCount,
        uint32_t        instanceCount,
        uint32_t        firstIndex,
        int32_t         vertexOffset,
        uint32_t        firstInstance);
```