Índice

Familiarización con Vulkan:

El método que se utilizado para realizar esta tarea está completamente basado en el tutorial básico desarrollado por Khronos Group: https://vulkan-tutorial.com

Se asume que el sistema operativo empleado por el usuario es Windows 10 y el IDE Visual Studio 2019. En caso contrario, Khronos Group ofrece esta información para desarrolladores con distintos requerimientos para establecer su ambiente de desarrollo para Vulkan.

1. Establecer ambiente de desarrollo

1.1. Descargar el SDK de Vulkan:

El SDK se puede descargar de https://vulkan.lunarg.com/

Esta librería (desarrollada por LunarG) promueve un ecosistema unificado de Vulkan e incluye, entre muchas cosas, los encabezados, las capas de validación estándar, las herramientas de depuración y un cargador para las funciones de Vulkan. Es esencial que los "drivers" de la tarjeta gráfica, incluyan el tiempo de ejecución de Vulkan y asegurarse de que sean compatibles con Vulkan. Para corroborar que funcione el SDK sea compatible con el sistema, se sugiere ejecutar vkcube.exe, ubicado dentro del directorio donde se encuentra el SDK, bajo el folder llamado Bin. La Figura 14 muestra el ejecutable del lado izquierdo y la ubicación del archivo en el derecho.

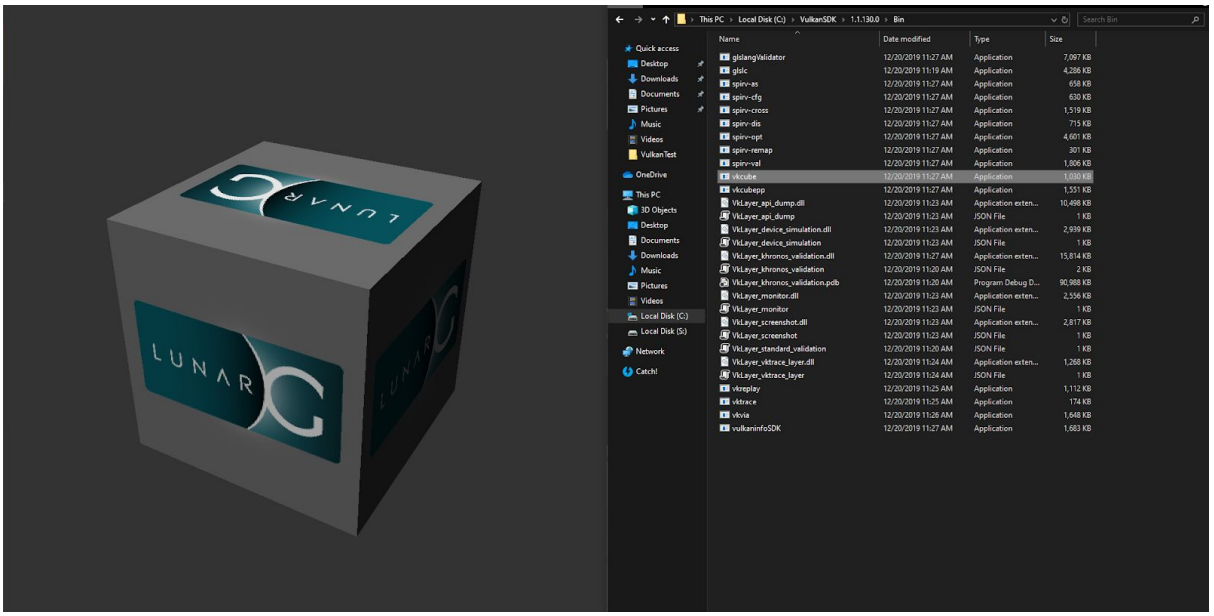


Figura 14. vkcube.exe

1.2.Descargar la librería GLFW:

GLFW es una biblioteca multiplataforma de código abierto para el desarrollo de OpenGL, OpenGL ES y Vulkan en el escritorio. Proporciona una API simple para crear ventanas, contextos y superficies, recibir entradas y eventos. Inicialmente, esta herramienta será de utilidad para crear ventanas que sean compatibles con Windows, MacOS y Linux. Khronos sugiere extraer esta carpeta en una ubicación conveniente. En su tutorial, crean una carpeta debajo de la carpeta de Visual Studio llamada “Libraries” o Librerías; es en esta carpeta donde extraen los archivos de GLFW. Es importante descargar el paquete de 32 o 64 bits dependiendo de la capacidad del procesador y sistema operativo. Para este ejemplo se estará trabajando con binarios de 64 bits.

Esta librería se puede descargar de https://www.glfw.org/

1.3. Descargar la librería GLM:

GLM es una es una biblioteca matemática escrita en C ++ para software de gráficos (comúnmente OpenGL) basada en las especificaciones de OpenGL Shading Language (GLSL). Se puede utilizar también en Vulkan y servirá para realizar operaciones de álgebra lineal. Se recomienda extraer esta biblioteca en la misma carpeta en la que se depositó GLFW.

Esta librería se puede descargar de https://glm.g-truc.net/0.9.9/index.html

El orden de ambas bibliotecas debería asemejarse al que se muestra en la Figura 15.

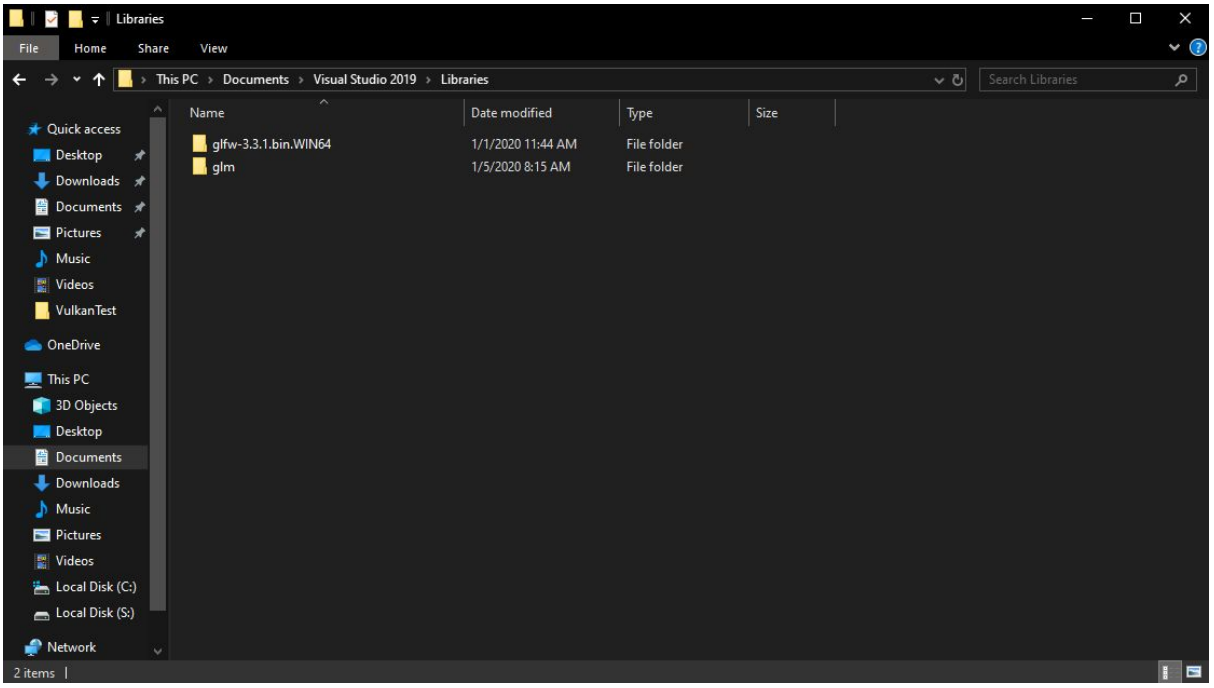


Figura 15. Estructura de carpetas

2. Abrir Ventana con Vulkan:

2.1. Crear Proyecto y Configurar Propiedades:

Para empezar el proyecto, después de ejecutar Visual Studio, se debe entonces crear un proyecto vacío utilizando *Windows Desktop Wizard*. Lo primero que se debe hacer, es vincular las librerías descargadas al proyecto para poder utilizarlas. Sin embargo, para poder comprobar que esto se realizó correctamente se necesita incluir primero en la carpeta *Source Files* (ubicada en el inspector) un archivo de tipo *cpp* llamado *main.* Posteriormente se tiene que abrir la ventana de propiedades del proyecto en la cual se deben realizar los siguientes pasos:

A. Cambiar la configuración a *All Configurations* (todas las configuraciones).

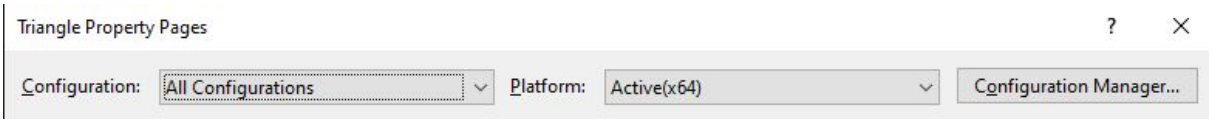


Figura 16. Establecimiento de configuraciones

B. Agregar los directorios de encabezados para Vulkan, GLFW y GLM en la opción de *Additional Include Directories* en sección general de *C/C++* (Figura 17):

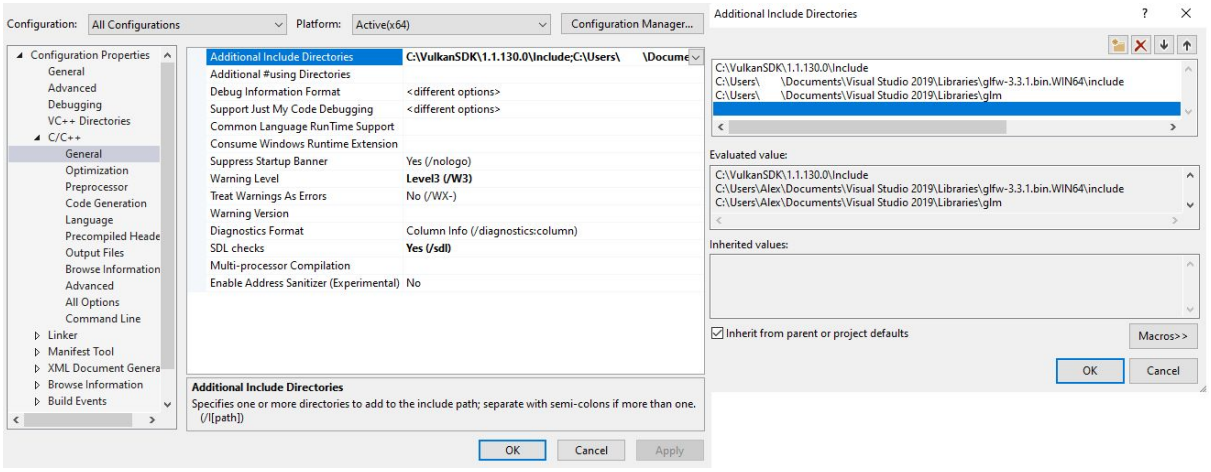


Figura 17. Incluir directorios de encabezado

C. Agregar las ubicaciones de los archivos de objetos para Vulkan y GLFW en la opción de *Additional Library Directories* dentro de la sección general de Linker (Figura 18):

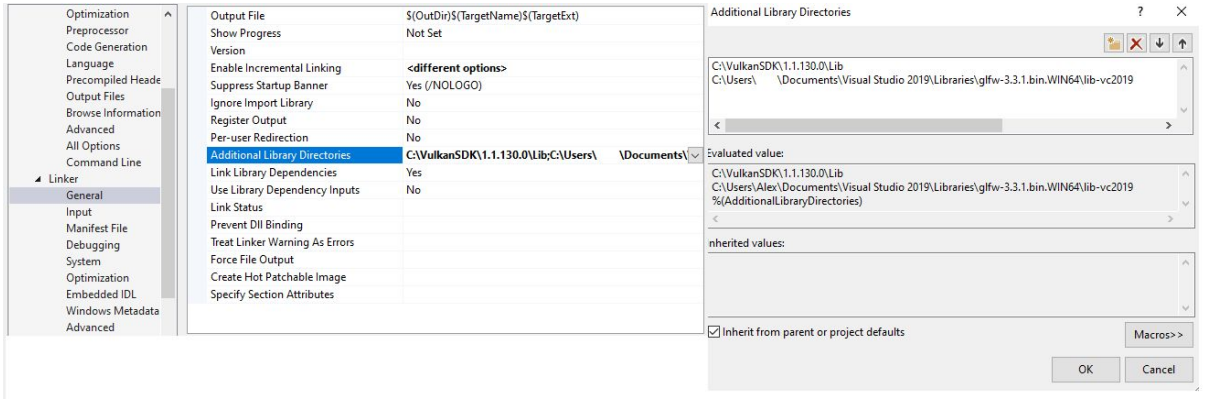


Figura 18. Ubicaciones de los archivos de objetos

D. Ingresar los nombres de los archivos de objetos Vulkan y GLFW: en la opción de *Additional Dependencias* dentro de la sección de *Input* de Linker (Figura 19):

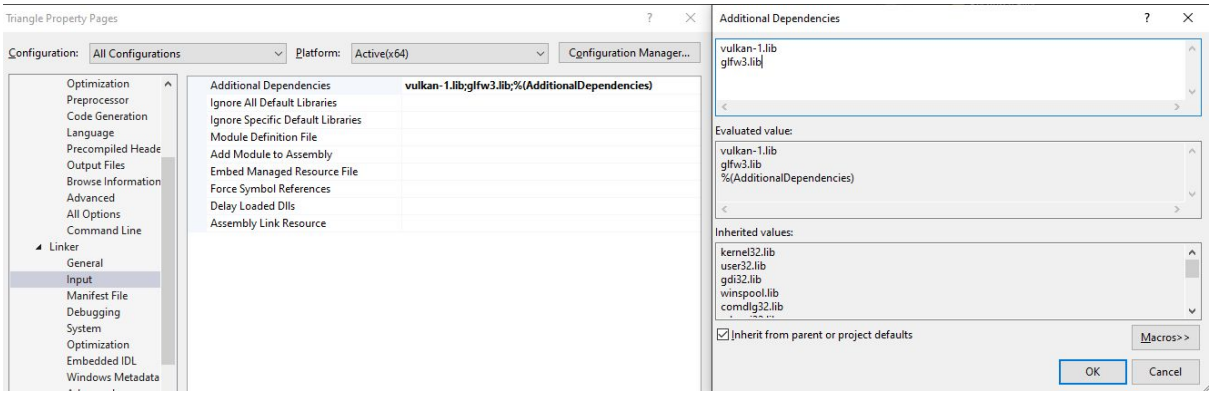


Figura 19. Nombres de archivos de objetos

E. Finalmente, cambiar el compilador para que sea compatible con las características de C ++ 17 en al opción de *C++ Language Standard* dentro de la sección de *Language* de *C/C++* (Figura 20):

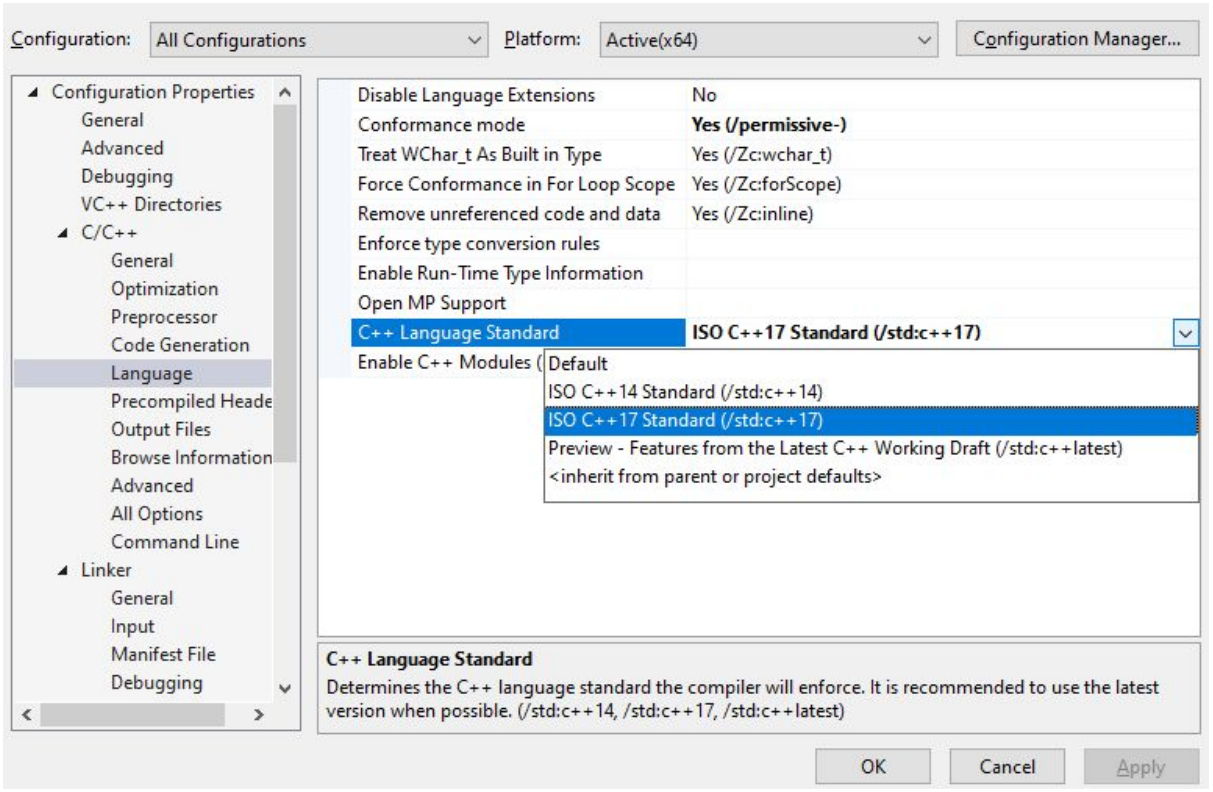


Figura 20. Establecer estándar del lenguaje C++:

2.2. Generar estructura e integrar GLFW:

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

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

const int WIDTH = 800;
const int HEIGHT = 600;

class HelloTriangleApplication { ... };

int main() {
    HelloTriangleApplication app;

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

    return EXIT_SUCCESS;
}
```

Figura 21. Estructura básica para abrir una ventana

Lo primero que se debe hacer es integrar GLFW para poder crear una ventana fácilmente y así poder observar lo que se está renderizando. Las primeras dos líneas de la Figura 21 son lo que permite hacer justo eso. *GLFW_INCLUDE_VULKAN* hace que el encabezado GLFW (*GLFW/glfw3.h)* incluya el ecnabezado de Vulkan por LunarG (*vulkan/vulkan.h*) además de cualquier encabezado OpenGL y OpenGL ES seleccionado. Además, agrega una función de inicialización y una llamada a ésta. El encabezado de Vulkan proporciona las funciones, estructuras y enumeraciones. Los encabezados *stdexcept* y *iostream* se incluyen para informar sobre errores y propagarlos. El encabezado *cstdlib* proporciona las macros EXIT_SUCCESS y EXIT_FAILURE. Cuando se usa el primer macro como argumento para

la salida de la función, significa que la aplicación fue exitosa y el caso opuesto se da para el segundo macro. Una última cosa a resaltar sobre la Figura 8 son las variables *WIDTH* y *HEIGHT* que se utilizarán para delimitar el tamaño de la ventana.

El programa en sí está envuelto en una clase donde se almacenan los objetos Vulkan como miembros privados y en donde se agregan las funciones para inicializar cada objeto, para renderizar cuadros y para designar o destruir objetos una vez que el ciclo principal termina. Esta declaración de esa clase y su función de ejecución se pueden observar en la Figura 21. La definición de esta clase se muestra en la Figura 22:

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

private:

    GLFWwindow* window;

    void initWindow() {
        glfwInit();  //Initializes GLFW
        glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API); //Avoids creating OpenGL Context
        glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE); //Prevents window resizing

        //Creates window: (Width, Height, Title, Monitor, OpenGL only)
        window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
    }
    //Initializes Vulkan Objects
    void initVulkan() {

    }
    //Loop to render frames
    void mainLoop() {
        //Keep application running by checking for events until an error occurs or the window is closed
        while (!glfwWindowShouldClose(window)) {
            glfwPollEvents();
        }
    }
    //Deallocates resources after main loop finishes
    void cleanup() {
        glfwDestroyWindow(window);
        glfwTerminate();
    }
};
```

Figura 22. Estructura de la clase principal del programa

La función *run()* funcionará como un miembro público de la clase para ejecutar las funciones principales del programa. En *initWindow()* se inicializa GLFW mediante *glfwInit()*. La función *glfwWindowHint()* debe incluir argumentos *GLFW_CLIENT_API* y *GLFW_NO_API* para evitar que se cree un contexto OpenGL ya que GLFW fue diseñado originalmente para crear un contexto OpenGL. La función *glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE)* simplemente evita el manejo de redimensionamiento de las ventanas ya que por ahora es innecesario de realizar y simplifica el proceso de crear una ventana. La función *glfwCreateWIndow()* se asigna al miembro privado *window* de tipo *GLFWwindow* y recibe como argumentos la anchura y altura de la ventana, su título, el monitor asignado y un quinto parámetro que sólo es relevante bajo un contexto OpenGL.

La función *mainLoop()* la cual va a funcionar como el ciclo principal del programa y simplemente mantiene la aplicación corriendo mientras no se detecten errores o se cierre la ventana. Estos sucesos se detectan mediante *glfwPollEvents()*. Finalmente, la función *cleanup()* se utilizará para destruir y desasignar los recursos; y lo primero que hará es destruir y terminar GLFW.

3. Dibujar un triángulo

3.1. Configuración:

3.1.1. Crear una instancia:

En este contexto, la instancia se refiere a la conexión entre la aplicación y la librería de Vulkan. Esencialmente, crear la instancia es el primer paso a tomar para inicializar la librería de Vulkan. Se debe entonces agregar una función dentro de *initVulkan()* llamada *createinstance()* y un miembro privado de la clase principal para identificar la instancia llamado *instance* de tipo *VkInstance.* La Figura 23 muestra el contenido de la función para crear una instancia:

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

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

Figura 23. Estructuras para crear instancia

De acuerdo a Khronos, la estructura *VkApplicationInfo* es opcional, pero puede ser útil para optimizar la aplicación. Adicionalmente, muchas estructuras en Vulkan requieren que se especifique su tipo por medio del miembro *sType* (structure type). Los demás miembros que conforman esta estructura básica son autoexplicativos. Sin embargo, la estructura *VkInstanceCreateInfo* es obligatoria ya que comunica al *driver* de Vulkan las extinciones globales y capas de validación que se van a utilizar.

Es necesario contar con una extensión que interactúe con el sistema de ventanas del sistema operativo de manera agnóstica. Afortunadamente, GLFW cuenta con funciones que devuelven las extensiones necesarias para hacer esto las cuales pueden usarse en la estructura. Esto se demuestra en la Figura 24:

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

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

createInfo.enabledLayerCount = 0;
```

Figura 24. Extensiones GLFW para la estructura de creación de instancia

Los parámetros *glfwExtensionCount* y *glfwExtensions* determinan las capas de validación globales a habilitar. El último miembro es un campo necesario para la estructura pero que por ahora basta con que permanezca vacío. Finalmente, como se muestra en la Figura 25, se emite *vkCreateInstance* para crear una instancia dentro de una sentencia condicional *if* que compruebe que se creó la instancia exitosamente:

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

Figura 25. Sentencia condicional para comprobar la creación de la instancia

En este punto, Khronos resalta el patrón general que siguen los parámetros de la función de creación de objetos en Vulkan que es:

- Apuntador a estructura con información de creación

- Apuntador a devoluciones de llamada del asignador personalizado
- Apuntador a la variable que almacena el "handle" para el nuevo objeto

Finalmente, se destruye la instancia antes de que termine el programa en el método de limpieza por medio de la función vkDestroy(), la cual recibe como parámetros el "handle" de la instancia y un apuntador nulo el cual representa una "callback" de apuntador opcional.

3.1.2. Capas de Validación:

Las capas de validación son componentes opcionales que se conectan a llamadas a funciones de Vulkan para aplicar operaciones adicionales. Son opcionales debido a que por defecto, el API de Vulkan minimiza la revisión de errores para disminuir lo más posible la sobrecarga a los *drivers* del GPU. Consecuentemente, errores simples pueden resultar en comportamientos indefinidos o bloqueos. Sin embargo, es recomendable incluirlas para la mayoría de los casos ya que evitan que la aplicación se rompa en distintos drivers por recaer en comportamiento indefinido. Las operaciones comúnmente utilizadas en las capas de validación incluyen:

- Verificación de los valores de los parámetros contra la especificación para detectar el mal uso.
- Seguimiento de la creación y destrucción de objetos para encontrar fugas de recursos.
- Verificación de la seguridad de los hilos rastreando los hilos desde donde se originan llamadas.

- Registrar cada llamada y sus parámetros en la salida estándar.
- Rastreando llamadas de Vulkan para perfilar y reproducir.

Estas capas de validación pueden ser personalizadas al gusto del desarrollador y se pueden deshabilitar para las versiones de lanzamiento. Mientras que Vulkan no viene con capas de validación incorporadas, LunarG incluye un buen conjunto de capas que verifican los errores más comunes. Toda la validación estándar útil se incluye en una capa incluida en el SDK que se conoce como *VK_LAYER_KHRONOS_validation.* Entonces se anexan las siguientes variables y vector al programa (Figura 26):

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

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

Figura 26. Variables de configuración

Estas variables permiten especificar las capas a validar y se utiliza el macro *NDEBUG* (not debug) para basar el valor de estas variables en el método de compilación del programa. Ahora se agrega una nueva función para revisar si todas las capas solicitadas están disponibles (Figura 27):

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

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

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

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

        if (!layerFound) {
            return false;
        }
    }

    return true;
}
```

Figura 27. Revisión de capas de validación

Finalmente, se modifica la instanciación de la estructura VkInstanceCreateInfo para incluir los nombres de la capa de validación y se agrega esta función al método de creación de instancia de la siguiente manera (Figura 28):

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

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

    createInfo.enabledLayerCount = 0;
```

```
void createInstance() {

    if (enableValidationLayers && !checkValidationLayerSupport()) {
        throw std::runtime_error("validation layers requested, but not available!");
    }
```

Figura 28. Sentencia condicional para revisar capas de validación y modificación de estructura de instanciación

El siguiente paso es configurar una "callback" en el programa para enviar mensajes y los detalles asociados a este. Se debe entonces configurar un mensajero de depuración con una "callback" utilizando la extensión *VK_EXT_DEBUG_UTILS_EXTENSION_NAME.*

Primero, se crea una función que devuelva la lista requerida de extensiones en función de si las capas de validación están habilitadas o no. Se debe notar que las extensiones especificadas por GLFW siempre son necesarias, pero la extensión de depuración de mensajería se agrega condicionalmente. Esta función se utiliza en el método de creación de instancia (Figura 29):

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

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

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

    return extensions;
}
```

```
auto extensions = getRequiredExtensions();
createInfo.enabledExtensionCount = glfwExtensionCount;
createInfo.ppEnabledExtensionNames = glfwExtensions;
```

Figura 29. Configuración para devolver extensiones requeridas

El siguiente paso es crear una función de "callback" de depuración con los macros necesarios para que esta tenga la firma necesaria para que Vulkan la llame. Esta función se observa de la siguiente manera (Figura 30):

```
static VKAPI_ATTR VkBool32 VKAPI_CALL debugCallback(
    VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity,
    VkDebugUtilsMessageTypeFlagsEXT messageType,
    const VkDebugUtilsMessengerCallbackDataEXT* pCallbackData,
    void* pUserData) {

    std::cerr << "validation layer: " << pCallbackData->pMessage << std::endl;

    return VK_FALSE;
}
```

Figura 30. "Callback" de depuración

Khronos especifica el significado de cada parámetro de la siguiente manera:

- messageSeverity:

    - VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT: mensaje de diagnóstico.
    - VK_DEBUG_UTILS_MESSAGE_SEVERITY_INFO_BIT_EXT: mensaje informativo como la creación de un recurso.
    - VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT: Mensaje sobre comportamiento que no es necesariamente un error, pero muy probablemente un error en su aplicación.
    - VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT: Mensaje sobre comportamiento que no es válido y puede causar bloqueos.
- messageType:
    - VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT: Ha ocurrido algún evento que no está relacionado con la especificación o el rendimiento.
    - VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT: Algo ha sucedido que viola la especificación o indica un posible error.
    - VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT: Uso potencial no óptimo de Vulkan.
- pCallbackData (contiene detalles del mensaje en sí):
    - pMessage: el mensaje de depuración como una cadena terminada en nulo.
    - pObjects: matriz de "handles" de objetos Vulkan relacionados con el mensaje.
    - objectCount: número de objetos en la matriz.

- pUserData contiene un apuntador que se especificó durante la configuración de la "callback" y le permite pasar datos propios.

La función devuelve un valor booleano que indica si la llamada de Vulkan que activó el mensaje de la capa de validación debe suspenderse. Si la "callback" devuelve verdadero, entonces la llamada se cancela con el error VK_ERROR_VALIDATION_FAILED_EXT.

Finalmente solo se debe crear el "handle" para la función de "callback". Esto se debe hacer mediante un “handle” que se crea y destruye explícitamente. Este callback es parte del mensajero de depuración y se pueden tener cuantos sean necesarios. Debajo de la instancia, se agrega un miembro de la clase para este “handle”. Dentro de la función de inicialización de Vulkan, después de la función de creación de instancia se agrega otra función de configuración del mensajero de depuración. Posteriormente, se debe crear una estructura con detalles del mensajero y su “callback”:

Nótese que la configuración utilizada en esta guía es una de varias otras posibilidades. Esta estructura se transfiere la función vkCreateDebugUtilsMessengerEXT para crear el objeto VkDebugUtilsMessengerEXT. Dado que esta función es una extensión, no se carga automaticamente y se debe buscar su ubicación utilizando vkGetInstanceProcAddr. dentro una función “proxy” la cual devuelve nullptr en caso de que no se haya podido cargar la función. Seguido de esto se puede crear el objeto de la extensión. Como el mensajero es específico a la instancia de Vulkan y sus capas, la instancia se tiene que especificar explícitamente como el primer argumento dentro de la función de creación. Este patrón se repite con otros objetos hijos. Este objeto también tiene que cargarse explícitamente y

limpiarse llamando vkDestroyDebugUtilsMessengerEXT mediante otra función “proxy” que se llama en la función general de limpieza.

Hasta este punto, es posible depurar todo excepto por la creación y destrucción de instancia. Para resolver esto, se debe crear un mensajero de depuración separado para estas funciones. Se logra pasando un apuntador a la estructura VkDebugUtilsMessengerCreateInfoEXT en el campo pNext. Asimismo, se extrae la población de la información de la creación de mensaje en una función separada llamada populateDebugMessengerCreateInfo para poder reutilizarse en la función de creación de instancia.

3.1.3. Dispositivos físicos y queue families:

Esta sección involucra el proceso de buscar y seleccionar una tarjeta gráfica en el sistema que admita las funciones que se necesitan. Para esto, se agrega a la función de inicialización una función de selección de dispositivo físico, llamada en el ejemplo pickPhysicalDevice. La tarjeta gráfica seleccionada se almacena en un “handle” llamado VkPhysicalDevice que se agrega como nuevo miembro de la clase. Este objeto se destruye implícitamente cuando la instancia se destruye. Similarmente, las extensiones, las tarjetas gráficas se enlistan preguntando por el número disponible. Se debe considerar el caso en el que no haya GPUs compatibles con Vulkan. De lo contrario, se debe asignar un arreglo que contenga todos los “handles” de los dispositivos y evaluar si funcionan para las operaciones a realizarse mediante una función booleana.

Es posible obtener las propiedades básicas mediante vkPhysicalDeviceProperties y sus atributos mediante vkGetPhysicalDeviceFeatures. Usando esto se puede aislar una tarjeta en el sistema que tenga capacidades exclusivas así como evaluar cómo se comparan las tarjetas de acuerdo a sus capacidades.

Avanzando, cada operación en Vulkan requiere que los comandos se envían a una cola. Hay diferentes colas que se originan de "queue families" y cada familia de colas permite un subconjunto de comandos. Por ejemplo, hay colas que solo permiten procesamiento de comandos de cálculo u otras que permiten comandos de transferencia de memoria. Se debe entonces revisar qué familias son compatibles con el dispositivo físico y que comandos son compatibles con estas familias. Se crea entonces una función que busque estas familias (en el ejemplo findQueueFamilies). En el ejemplo, se hace una estructura y se utilizan índices para estas familias para poblar la estructura. También se utiliza el "wrapper" std::optional para identificar un valor como no existente y así determinar que una familia no existe. Posteriormente, se utiliza vkGetPhysicalDeviceQueueFamilyProperties para obtener la lista de familias y la estructura VkQueueFamilyProperties para obtener información sobre la familia, como el tipo de operaciones que permite hacer y el número de colas basadas en la familia que se pueden crear.

3.1.4. Dispositivos lógicos y colas:

Después de seleccionar un dispositivo físico para usar, se necesita configurar un dispositivo lógico para interactuar con él. El proceso de creación del dispositivo lógico es similar al proceso de creación de instancias y describe las características que se desean utilizar. Se

deben especificar qué colas crear ahora al consultar qué las familias de colas están disponibles. Incluso se pueden crear múltiples dispositivos lógicos desde el mismo dispositivo físico si se tienen diferentes requisitos.

Primero se agrega un miembro de la clase llamado VkDevice y se agrega una función de creación de dispositivo lógico en la función de inicialización. Luego se especifica la estructura VkDeviceQueueCreateInfo que describe el número de colas que se desean para una familia de colas. Se puede también asignar prioridades a las colas para influenciar la calendarización de la ejecución del búfer de comandos utilizando puntos flotantes entre 0.0 y 0.1 (esto se requiere aunque haya solo una cola).

Después se especifican las características a utilizarse con vkPhysicalDeviceFeatures. Con esto se puede llenar la estructura VkDeviceCreateInfo. Primero se agregan apuntadores a la información de creación de colas y a las estructuras de las características del dispositivo. Luego se especifican extensiones y capas de validación específicas al dispositivo. Un ejemplo de estas extensiones es VK_KHR_swapchains, que permite presentar imágenes renderizadas desde el dispositivo a ventanas. Para asegurar compatibilidad con implementaciones más antiguas de VUlkan, se recomienda incluir los campos enabledLayerCount y ppEnabledLayerNames. Seguido de este paso, se instancia el dispositivo lógico llamando vkCreateDevice. Los parámetros de esta función son el dispositivo físico que funciona como interfaz, la cola y la información de uso que se especificó, los apuntadores de “callbacks” de asignación y un apuntador a variable para almacenar el “handle” del dispositivo lógico. El dispositivo se tiene que destruir en la función de limpieza con vkDestroyDevice. Ya que los

dispositivos lógicos no interactúan directamente con las instancias, no se incluyen como parámetro en la función de destrucción.

Finalmente, para establecer una interfaz con las colas, se agrega un miembro de la clase para almacenar un “handle” a las colas de gráficos (que son las únicas que se necesitan hasta este punto). Este miembro de tipo VkQueue, se limpia cuando se destruye el dispositivo y se obtiene mediante vkGetDeviceQueue. Los parámetros de esta función son el dispositivo lógico, la familia de colas, el índice de la cola y un apuntador a una variable para almacenar el “handle” de la cola. Para esta etapa, se utiliza el índice 0 ya que solo se está creando una cola de la familia de gráficos. A partir de ahora, se puede comenzar a utilizar la tarjeta gráfica.

3.2. Presentacion:

3.2.1. Superficie de ventana:

Aquí se utilizan las extensiones WSI (Windows System Integration) para interactuar con el sistema de ventanas. Como Vulkan es agnóstico, no puede formar un interfaz con el sistema de ventanas por sí solo. Una de las extensiones (VK_KHR_surface) expone un objeto VkSurfaceKHR que representa un tipo abstracto de superficie para presentar imágenes renderizadas. Esta superficie se respaldará por la ventana abierta con GLFW. No obstante, esta extensión es una de nivel de instancia se habilitó con glfwGetRequiredInstanceExtensions. La superficie se debe crear después de la instancia para evitar influenciar la selección de dispositivo físico.

Se empieza agregando un miembro de la clase de tipo VkSurfaceKHR. El objeto de este miembro es agnóstico pero su creación no lo es ya que depende del sistema de ventana. En Windows necesita los “handles” HWND y HMODULE. Se necesita una extensión específica a la plataforma la cual se incluyó con glfwGetRequiredInstanceExtensions (en Windows ésta es VK_KHR_win32_surface). En cualquier caso, GLFW maneja las diferencias entre plataformas mediante el “handle” glfwCreateWindowSurface. Para integrarlo, se crea una función para la creación de una superficie dentro de la inicialización y seguido de la llamada a la función de configuración del mensajero de depuración (setupDebugMessenger). Este “handle” solo recibe parámetros simples en lugar de estructuras e incluyen: la instancia, el apuntador a la ventana de GLFW, y apuntadores y asignaciones a la variable VkSurfaceKHR. Luego se destruye la superficie antes de la instancia mediante vkDestroySurfaceKHR.

Ahora se debe asegurar que los dispositivos en el sistema puedan presentar imágenes a la superficie creada. Para esto se debe extender la función isDeviceSuitable y encontrar la familia de colas que pueda presentar a la superficie creada. Se empieza modificando la estructura de índices de las familias y se agrega una familia para presentación. Luego se modifica la función que encuentra familias para buscar una capaz de presentar en la superficie de la ventana mediante la función vkGetPhysicalDeviceSurfaceSupportKHR. Luego se llama a esta función en el mismo ciclo que VK_QUEUE_GRAPHICS_BIT, se revisa el valor del booleano y se almacena su el índice de la cola. Esta implementación ofrece uniformidad. Sin embargo, se puede dar preferencia a un dispositivo físico que cuente con graficación y presentación en la misma cola para mejorar el rendimiento.

Por último, se crea la cola de presentación declarando un miembro de tipo VkQueue y se hacen múltiples estructuras VkDeviceQueueCreateInfo para crear una cola para la familia de presentación y graficación. El ejemplo propone un método en el que se crea un conjunto de todas las familias únicas que son necesarias para las colas requeridas mediante std::set. Luego se modifica VkDeviceCreateInfor para apuntar a un vector. Si las familias son las mismas, el índice solo se pasa una vez. Finalmente se agrega una llamada al "handle" de la cola.

3.2.2. Swap Chain:

El "swap chain" es una infraestructura que mantiene los buffers que se van a renderizar antes de visualizarlos en pantalla. Es más fácil pensar en esta infraestructura como una cola de imágenes esperando a presentarse en pantalla. El propósito es sincronizar la presentación de imágenes con la frecuencia de actualización de la pantalla.

El primer paso, como de costumbre, es revisar la compatibilidad del dispositivo con el "swap chain". Esto es relevante ya que puede haber casos en los que el dispositivo no es capaz de presentar directamente en pantalla (como lo es con tarjetas diseñadas para servidores). Por otro lado, dado que la presentación de imágenes está ligada al sistema de ventanas y a las superficies asociadas a estas ventanas, no forma parte del núcleo de Vulkan; por lo que se requiere la extensión VK_KHR_swapchain. Se debe entonces extender isDeviceSuitable nuevamente para revisar si es compatible esta extensión. Se hace notar que se utiliza un macro para esta extensión llamado VK_KHR_SWAPCHAIN_EXTENSION_NAME que permite que el compilador detecte faltas de ortografía. Se declara una lista de extensiones requeridas por el dispositivo y se crea una función (checkDeviceExtensionSupport) de apoyo

que servirá para enumerar las extensiones y confirmar que está presente la requerida. En el ejemplo, se utiliza un conjunto de . “strings” para representar las extensiones requeridas sin confirmar.

Para habilitar la extensión, se altera la estructura de creación de dispositivo lógico, cambiando la cuenta de 0 al tamaño de del vector que contiene el número de extensiones. Subsecuentemente, se revisan tres cosas para comprobar la compatibilidad con el “swap chain”:

- Capacidades de superficies básicas (min/máx número de imágenes y min/máx de anchura y altura de las imágenes).
- Formatos de superficie (formato de píxeles, color del espacio).
- Modos de presentación disponibles.

Estas propiedades se obtienen mediante estructuras y listas de estructuras. Se utiliza la función vkGetPhysicalDeviceSurfaceCapabilitiesKHR para obtener las capacidades básicas de la superficie. Al igual que las demás funciones de obtención de capacidades, esta recibe como los primeros dos parámetros el dispositivo y la superficie ya que son los componentes del núcleo del “swap chain”. Luego se obtienen los formatos de la superficie que al ser una lista de estructuras se logra mediante dos llamadas a la misma función (como se muestra en la Figura 31).

```
uint32_t formatCount;
vkGetPhysicalDeviceSurfaceFormatsKHR(device, surface, &formatCount, nullptr);

if (formatCount != 0) {
    details.formats.resize(formatCount);
    vkGetPhysicalDeviceSurfaceFormatsKHR(device, surface, &formatCount, details.formats.data());
}
```

Figura 31. Preguntando por formatos de superficie

Para obtener los modos de presentación se sigue la misma lógica que la anterior pero con la función vkGetPhysicalDeviceSurfacePresentModesKHR. Ahora se extiende nuevamente isDeviceSuitable para utilizar esta función y verificar que el el “swap chain” sea adecuado. Es importante hacer esto después de verificar que la extensión está disponible.

Ulteriormente, se escriben algunas funciones para optimizar el “swap chain”. Para esto, hay tres tipos de configuraciones que determinar:

- El formato de la superficie (profundidad del color).
- Modo de presentación (condiciones para intercambiar imágenes en la pantalla).
- Alcance de intercambio (resolución de imágenes en el “swap chain”).

La función para especificar el formato de la superficie (VkSurfaceFormatKHR) recibe como argumento el miembro formato de la estructura SwapChainSupportDetails. Cada entrada contiene un miembro format y colorSpace. El primero especifica los canales y tipos de color. Por ejemplo, VK_FORMAT_B8G8R8A8_SRGB indica que se almacenan canales B, G, R y alpha con un entero de 8 bits sin signo por un total de 32 bits por pixel. El segundo miembro indica si el espacio de color SRGB es compatible utilizando VK_COLOR_SPACE_SRGB_NONLINEAR_KHR.

Para el modo de presentación, existen cuatro modos:

- VK_PRESENT_MODE_IMMEDIATE_KHR: Las imágenes se presentan inmediatamente.
- VK_PRESENT_MODE_FIFO_KHR: El swap chain es una cola donde el monitor, al actualizarse, toma una imagen al frente de la cola y el programa inserta las imágenes

renderizadas al final de la cola. Si la cola está llena entonces el programa tiene que esperar. Esto funciona similarmente a vertical sync. El momento de la actualización se conoce como “vertical blank”. Este es el único modo cuya disponibilidad está garantizada.

- VK_PRESENT_MODE_FIFO_RELAXED_KHR: Similar al método anterior pero si la aplicación se atrasa y la cola está vacía en el ultimo vertical blank, en lugar de esperar al siguiente vertical blank, la imagen se transfiere inmediatamente al llegar.
- VK_PRESENT_MODE_MAILBOX_KHR: Otra variación del segundo modo. En lugar de bloquear la aplicación cuando se llena la cola, las imágenes en la cola se reemplazan con las más nuevas. Con este modo se puede utilizar triple buffering que permite evitar desgarres en la pantalla.

El modo se especifica mediante VkPresentModeKHR y en el ejemplo, se trata de confirmar la disponibilidad del cuarto método para aprovechar los beneficios de triple buffering. Si no se encuentra, se utiliza el segundo método.

Finalmente, el alcance de intercambio utiliza la función de tipo VkExtent2D. El rango de resoluciones posibles se define por la estructura VkSurfaceCapabilitiesKHR. Para emparejar la resolución de la ventana, se puede establecer el valor de anchura y altura en el miembro currentExtent como el valor máximo de uint32_t. Para esto se selecciona una resolución que encaje con la ventana dentro de los límites de minImageExtent y maxImageExtent. Las funciones max y min se utilizan para anclar los valores de anchura y altura entre los alcances mínimos y máximos permitidos. Esto requiere la inclusión <cstdint> y <algorithm>.

Es posible ahora crear el swap chain. Se crea entonces una función para esto en la inicialización después de la creación de dispositivos lógicos. Además de especificar las propiedades ya mencionadas, se tienen que elegir cuántas imágenes se pueden tener en el swap chain. Se indica el número mínimo de imágenes que se necesitan para que funcione la implementación más uno para evitar esperar a que el driver complete operaciones internas antes de poder adquirir otra imagen para renderizar. Por otro lado, para evitar alcanzar un valor máximo se utiliza 0. Luego se crea una estructura como de costumbre. Después de especificar la superficie, se especifican los detalles de las imágenes:

- imageArrayLayers: indica la cantidad de capas de las que consiste una imagen (siempre 1 excepto por aplicaciones estereoscópicas en 3D).
- imageUsage: campo de bits que especifica el tipo de operaciones que las imágenes utilizaran en el swap chain.

Luego se especifica cómo manejar las imágenes del swap chain a utilizarse a lo largo de múltiples familias de colas. Este es el caso si la familia de gráficos es diferente a las de presentación. Se dibujan las imágenes en el swap chain desde la cola de graficos y despues se envian a la cola de presentación. Khronos propone dos formas de manejar esto:

- VK_SHARING_MODE_EXCLUSIVE: Una imagen es poseída por una familia y se debe transferir la propiedad antes de usarse en otra familia (opción de alto rendimiento).
- VK_SHARING_MODE_CONCURRENT: Las imágenes se pueden utilizar a través de varias familias sin transferencia de propiedad explícita.

Por simplicidad, el ejemplo utiliza el modo concurrente que requiere que se especifique de forma anticipada las familias que comparten la propiedad de las imágenes usando los parámetros queueFamilyIndexCount y pQueueFamilyIndices. Sin embargo, se utiliza

comúnmente el modo exclusivo ya que la familia de gráficos y presentación son la misma en la mayoría del hardware. En el swap chain, se pueden especificar transformaciones compatibles. Si no se desean hacer, se utiliza currentTransform. El campo compositeAlpha se utiliza para mezclar el canal alpha con otras ventanas en el sistema de ventanas. Se utiliza VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR para ignorar esto. El miembro clipped se habilita para mejorar el rendimiento ignorando pixeles oscurecidos Finalmente, oldSwapChain se define como nulo ya que, por el momento, solo se va a crear un swap chain.

Para este punto, se agrega un miembro de la clase que almacene un objeto VkSwapchainKHR. Luego se crea el swap chain llamando la función vkCreateSwapchainKHR que recibe como parámetros el dispositivo lógico, la información de creación, asignadores, y apuntadores a la variable que almacena el handle. Luego se limpia usando vkDestroySwapchainKHR antes de destruir el dispositivo.

Lo último que queda es obtener los handles de VkImage. Se crea un miembro de la clase para almacenar estos handles. Luego se pide el número final de imágenes mediante vkGetSwapchainImagesKHR, y después se cambia el tamaño del contenedor; finalmente se llama de nuevo para obtener los handles. Posteriormente se almacena el formato y el alcance en variables miembros. La función de creación del swap chain de este ejemplo se muestra en la Figura 32:

```
void createSwapChain() {
    SwapChainSupportDetails swapChainSupport = querySwapChainSupport(physicalDevice);

    VkSurfaceFormatKHR surfaceFormat = chooseSwapSurfaceFormat(swapChainSupport.formats);
    VkPresentModeKHR presentMode = chooseSwapPresentMode(swapChainSupport.presentModes);
    VkExtent2D extent = chooseSwapExtent(swapChainSupport.capabilities);

    uint32_t imageCount = swapChainSupport.capabilities.minImageCount + 1;
    if (swapChainSupport.capabilities.maxImageCount > 0 && imageCount > swapChainSupport.capabilities.maxImageCount) {
        imageCount = swapChainSupport.capabilities.maxImageCount;
    }

    VkSwapchainCreateInfoKHR createInfo{};
    createInfo.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
    createInfo.surface = surface;

    createInfo.minImageCount = imageCount;
    createInfo.imageFormat = surfaceFormat.format;
    createInfo.imageColorSpace = surfaceFormat.colorSpace;
    createInfo.imageExtent = extent;
    createInfo.imageArrayLayers = 1;
    createInfo.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;

    QueueFamilyIndices indices = findQueueFamilies(physicalDevice);
    uint32_t queueFamilyIndices[] = { indices.graphicsFamily.value(), indices.presentFamily.value() };

    if (indices.graphicsFamily != indices.presentFamily) {
        createInfo.imageSharingMode = VK_SHARING_MODE_CONCURRENT;
        createInfo.queueFamilyIndexCount = 2;
        createInfo.pQueueFamilyIndices = queueFamilyIndices;
    }
    else {
        createInfo.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
    }

    createInfo.preTransform = swapChainSupport.capabilities.currentTransform;
    createInfo.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
    createInfo.presentMode = presentMode;
    createInfo.clipped = VK_TRUE;

    createInfo.oldSwapchain = VK_NULL_HANDLE;

    if (vkCreateSwapchainKHR(device, &createInfo, nullptr, &swapChain) != VK_SUCCESS) {
        throw std::runtime_error("failed to create swap chain!");
    }

    vkGetSwapchainImagesKHR(device, swapChain, &imageCount, nullptr);
    swapChainImages.resize(imageCount);
    vkGetSwapchainImagesKHR(device, swapChain, &imageCount, swapChainImages.data());

    swapChainImageFormat = surfaceFormat.format;
    swapChainExtent = extent;
}
```

Figura 32. Creación de Swap Chain

3.2.3. Image Views:

Es la vista hacia una imagen. Describe cómo acceder una imagen y a qué apartes de esa imagen acceder. Para utilizar los objetos VkImage (como los que están en el swap chain) se tiene que utilizar un objeto VkImageView, que como indica el nombre, describe la vista hacia una imagen. Entonces se crea una función dentro de la inicialización (llamada en el ejemplo

createImageViews) la cual servirá para crear una vista básica a una imagen por cada cada imagen en el swap chain. Asimismo, se crea un miembro de tipo vector que contenga los objetos de tipo VkImageView. Luego se cambia el tamaño de esta lista, igualando al de las imágenes en el swap chain para poder incluir todas las vistas a imágenes. Se crea un ciclo "for" que itere sobre todas las imágenes en el swap chain y que dentro del cual se especifican los parámetros de la estructura para la creación de vista de imágenes. Los parámetros viewType y Format indican cómo interpretar los datos de la imagen. Específicamente, viewType permite tratar imágenes como texturas 1D, 2D, 3D y mapas de cubos. Los parámetros de componentes, permiten mezclar los canales de color. Como ejemplo, esto podría servir si se quiere obtener una textura monocromática mapeando todos los canales al canal rojo (r). El mapeo estándar es VK_COMPONENT_SWIZZLE_IDENTITY. El campo subresourceRange describe el propósito de la imagen y que parte de esta puede ser accedida. Para crear la vista de imagen simplemente se llama la función de su creación (vkCreateImageView) y debido a que se creó explícitamente, se debe destruir con ciclo for similar. La Figura 33 muestra como se ve esta función:

```
void createImageViews() {
    swapChainImageViews.resize(swapChainImages.size());

    for (size_t i = 0; i < swapChainImages.size(); i++) {
        VkImageViewCreateInfo createInfo{};
        createInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
        createInfo.image = swapChainImages[i];
        createInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
        createInfo.format = swapChainImageFormat;
        createInfo.components.r = VK_COMPONENT_SWIZZLE_IDENTITY;
        createInfo.components.g = VK_COMPONENT_SWIZZLE_IDENTITY;
        createInfo.components.b = VK_COMPONENT_SWIZZLE_IDENTITY;
        createInfo.components.a = VK_COMPONENT_SWIZZLE_IDENTITY;
        createInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
        createInfo.subresourceRange.baseMipLevel = 0;
        createInfo.subresourceRange.levelCount = 1;
        createInfo.subresourceRange.baseArrayLayer = 0;
        createInfo.subresourceRange.layerCount = 1;

        if (vkCreateImageView(device, &createInfo, nullptr, &swapChainImageViews[i]) != VK_SUCCESS) {
            throw std::runtime_error("failed to create image views!");
        }
    }
}
```

Figura 33. Función de creación de vistas de imágenes

3.3. Pipeline de Gráficos:

3.3.1. Vision general:

Este pipeline gráfico (Figura 34) simplemente es la secuencia de pasos que toma los vértices y texturas de los meshes (mallas), y los conduce a los render targets (destinos de renderizado).

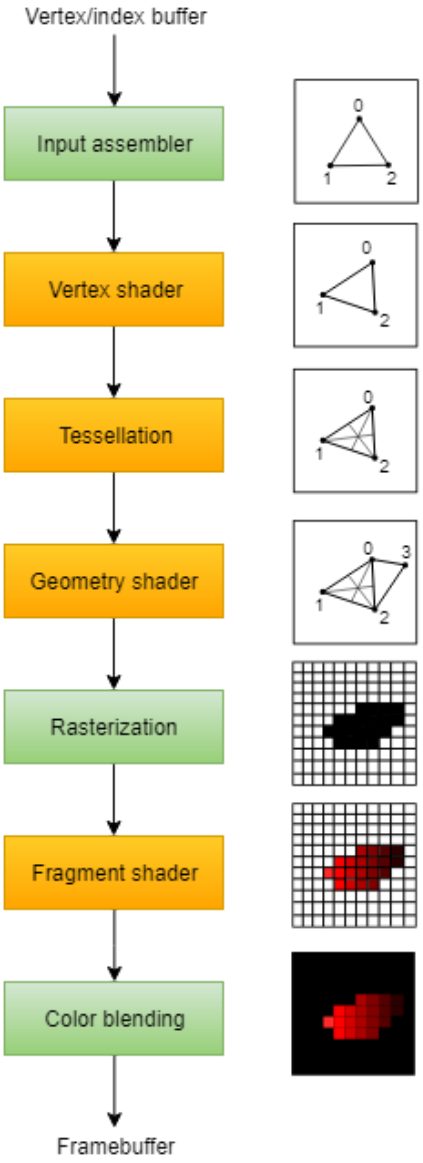


Figura 34. Pipeline Gráfico (Overvoorde, A., s.f.)

Overvoorde (s.f.) explica de la siguiente forma cada elemento del pipeline:

- El primer elemento (input assembler), recopila los datos de vértices sin procesar de los búferes especificados y también puede usar un búfer de índices para repetir ciertos elementos sin tener que duplicar los datos de vértice en sí.
- El vertex shader se ejecuta para cada vértice y generalmente se usa aplicar transformaciones para cambiar la posición del espacio del modelo al espacio de pantalla y para transmitir datos por vértice a la siguiente etapa del pipeline.
- Tessellation shader sirve para subdividir geometría con base a ciertas reglas para incrementar la calidad de la malla. Esto se llega a usar para que ciertas superficies parezcan menos planas cuando están muy próximas (ej. pared de ladrillos).
- El shader de geometría se ejecuta sobre cada primitiva (triángulo, linea punto) para descartar o regresar más de las que entraron. Funciona similar al shader anterior pero es más flexible. Se hace notar que su uso ha disminuido debido a que su rendimiento no es muy óptimo.
- La rasterización, discretiza las primitivas a fragmentos. Estos fragmentos son los pixeles que se llenan en el framebuffer. Los fragmentos que caen fuera de la pantalla se descartan mientras que los atributos devueltos por el shader de vértices se interpolan a través de los fragmentos. Normalmente, los fragmentos que se encuentran detrás de otros fragmentos se descartan gracias a pruebas de profundidad.
- El fragment shader se invoca por cada fragmento que sobrevive y determina a que framebuffer(s) está escrito y cuáles son sus atributos de color y profundidad. Hace esto mediante los datos interpolados del shader de vértices; también puede incluir coordenadas de texturas y normales para la iluminación.

- La última etapa mezcla distintos fragmentos mapeados al mismo pixel en el framebuffer. Los fragmentos se pueden sumar, sobreescribir o mezclar de acuerdo a su transparencia.

Overvoorde explica que las etapas verdes del pipeline se denominan como etapas de función fija lo cual implica que se pueden alterar sus operaciones utilizando parámetros pero la forma en que funcionan es predefinida. Por otro lado, las etapas naranjas son programables y permiten que uno suba su propio código a la tarjeta gráfica para aplicar operaciones personalizadas. Esto permite utilizar el shader de fragmentos para implementar cosas como texturas, luz, hasta ray tracing. Estos programas se ejecutan en varios núcleos del GPU simultáneamente para procesar varios objetos, como vértices y fragmentos de forma paralela. En Vulkan, este pipeline es casi completamente inmutable por lo que se debe recrear desde cero para poder cambiar shaders, enlazar diferentes framebuffers etc. La desventaja es que se debe crear una variedad de pipelines que representen cada combinación de los estados que se quieran utilizar en operaciones de renderización. Sin embargo, si se conocen todas las operaciones a realizar, el driver puede optimizar su ejecución. Algunas de las etapas programables son opcionales dependiendo de lo que se quieran hacer. Por ejemplo, las etapas de teselación y geometría no son necesarias si se está dibujando geometría sencilla.

Sin más, el primer paso es crear una función de creación del pipeline que se llama después de la creación de vista de imágenes.

3.3.2. Modulos de Shaders:

En Vulkan, el código de los shaders se especifica en un formato bytecode llamado SPIR-V. Se distingue de otros formatos como GLSL y HLSL cuya sintaxis es legible por humanos. SPIR-V se puede usar para escribir gráficos y computar shaders. La ventaja de este formato está en que los compiladores pueden convertir el código del shader a código nativo de forma menos compleja. Esto ayuda a evitar riesgos como el GPU rechazando el código de shader o que surjan errores de compilación. EL compilador de Khronos se encarga de traducir GLSL a SPIR-V y verificar que el código de shader concuerde con los estándares para producir un binario SPIR-V. Este compilador se puede usar mediante glslangValidator.exe pero en el ejemplo se utiliza glslc.exe por Google ya que utiliza el mismo formato de parámetros que otros compiladores como GCC y Clang; e incluye funcionalidad extra como “includes”.

Para aclarar, GLSL en un lenguaje que utiliza una sintaxis estilo C. Sus programas utilizan una función main que se invoca para cada objeto. En lugar de usar parámetros como entradas y un valor de salida, utiliza variables globales para manejar salidas y entradas. GLSL cuenta con características para facilitar la programación de gráficos, como primitivas de vectores y matrices, y operaciones con estas como producto cruz, productos entre matrices y vectores, y reflexiones alrededor de un vector. El tipo de vector se llama vec junto con un número para indicar el número de elementos. También es posible crear un nuevo vector desde múltiples componentes (ej. vec3(1.0, 2.0, 3.0).xy) y los constructores de los vectores pueden tomar combinaciones de vectores y escalares (ej. vec3(vec2(1.0, 2.0), 3.0)). Con esto establecido, se comienza por escribir el shader de vértices.

3.3.3. Vertex Shader y Fragment Shader:

Como ya se citó, este shader toma los atributos de cada vértice entrante, como la posición en el mundo, el color, las coordenadas normales y de textura; y devuelve la posición final en coordenadas "clip" y los atributos para el shader de fragmentación. Como paréntesis, "clip coordinates" son un vector de cuatro dimensiones del shader de vértices hacen referencia a un sistema de coordenadas homogéneas en el pipeline y son el resultado de una transformación de proyección aplicada a las coordenadas en el espacio visible en donde se descartan fragmentos no visibles. Luego se les puede hacer una división de perspectiva (dividiendo el vector por su último elemento) resultando en coordenadas de dispositivo normalizadas que se mapean al framebuffer a un sistema coordenado de [-1, 1] por [-1, 1] (Figura 35). La Figura 36 demuestra esta transformación (entre otras).

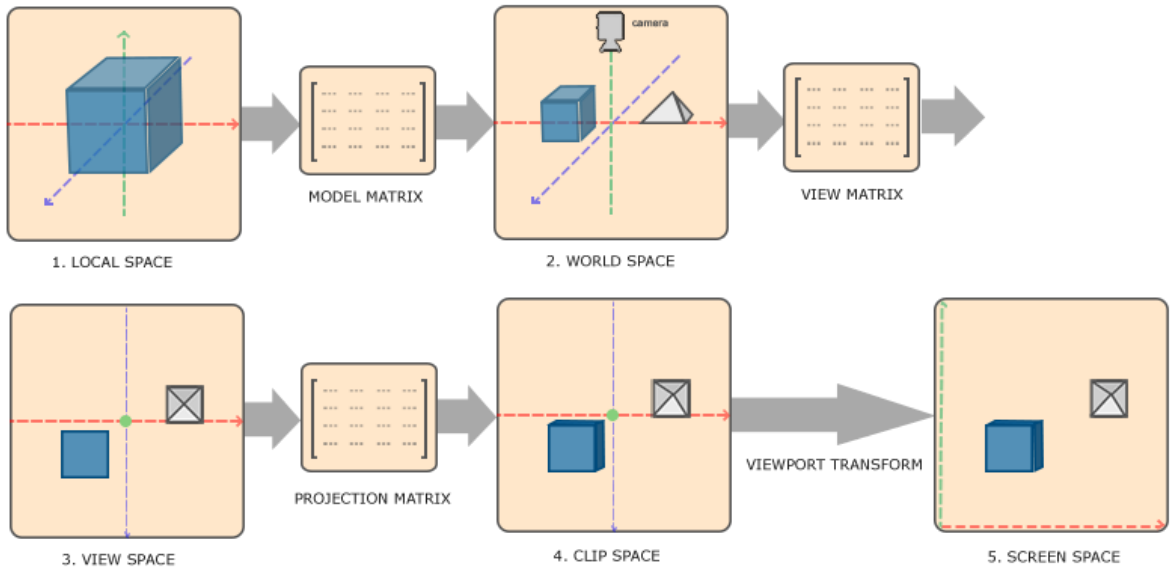


Figura 35. Transformaciones a sistemas de coordenadas (Vries J., s.f.)

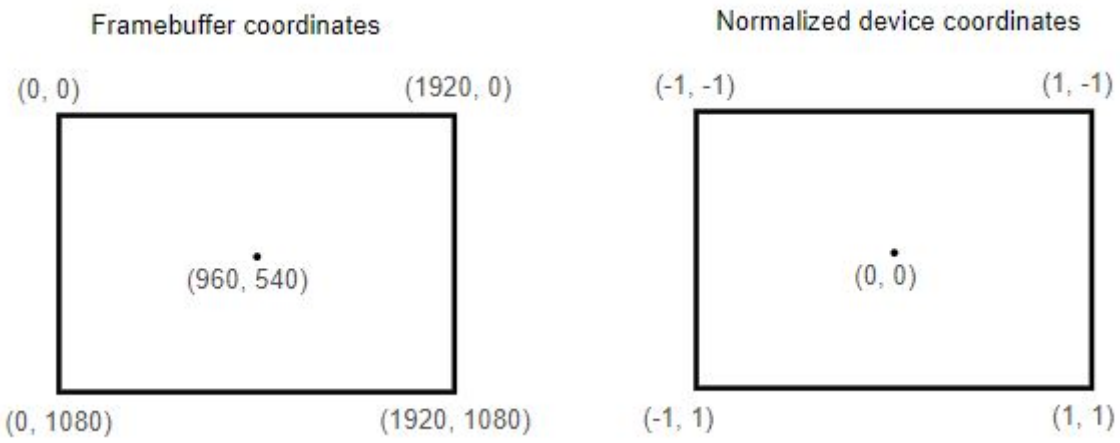


Figura 36. Transformación a coordenada de dispositivo normalizada (Overvoorde, A., s.f.)

En Vulkan, el eje Y está invertido y el rango de las coordenadas Z es de 0 a 1. En el shader, se pueden devolver coordenadas de dispositivo normalizadas definiendo el último componente de las coordenadas clip como 1 (así la división no cambia nada). Normalmente estas coordenadas se almacenan en el búfer de vértices pero en esta etapa del tutorial este paso es omitido por lo que las coordenadas se incluyen directamente en el shader. Una variable a utilizar es gl_VertexIndex que como describe, contiene el índice del vértice actual que usualmente se encuentra en el búfer de vértices. La posición de cada vértice se accede desde el arreglo constante en el shader y combinado con componentes z y w (tercero y cuarto respectivamente) falsos para producir una posición en las coordenadas “clip”. Asimismo, la variable gl_Position funciona como la salida del shader. También se incluye un arreglo que especifique los colores para cada vértice. Luego se transfieren estos valores al shader de fragmentación para que pueda interpolar los valores al framebuffer. Esto se hace mediante una salida especificada.

El triángulo formado por las posiciones del vertex shader llena un área de la pantalla con fragmentos. El fragment shader se invoca en estos fragmentos para producir un color y profundidad para el framebuffer(s). Similarmente al shader de vértices, el shader de fragmentación llama la función principal (main) por cada fragmento. En GLSL, los colores son vectores de cuatro componentes: canales de rojo, verde y alfa en un rango de 0 a 1. A diferencia del shader de vértices, no hay una variable para devolver el color del fragmento actual. Se tiene que especificar una variable de salida propia para cada framebuffer donde layout(location = 0) especifica el índice del framebuffer. La variable del color, se enlaza al primer y único índice del framebuffer (0). Para recibir los valores de colores, se define una variable de entrada que los reciba la cual se enlaza mediante los índices especificados por la directiva "location". La Figura 37 muestra cómo deberían verse estos shaders:

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) out vec3 fragColor;

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

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

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

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) in vec3 fragColor;

layout(location = 0) out vec4 outColor;

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

Figura 37. Shader de vértices (izquierda) y de fragmentos (derecha)

3.3.4. Compilando shaders:

El primer paso es crear un directorio en el directorio raíz del proyecto y almacenar en él los shaders en un archivo llamado shader.vert y otro llamado shader.frag. Para compilarlos usando glslc, se crea un archivo con la extensión ".bat" (en el ejemplo, compile.bat) y se

escribe (una línea para cada shader) la ubicación de glslc.exe en el SDK de Vulkan seguido del archivo donde se almacenó el shader seguido de "-o (shader).spv". Al ejecutar el archivo se crean los archivos SPIR-V. La Figura 38 muestra el contenido del archivo ".bat" y el resultado esperado tras ejecutarlo:

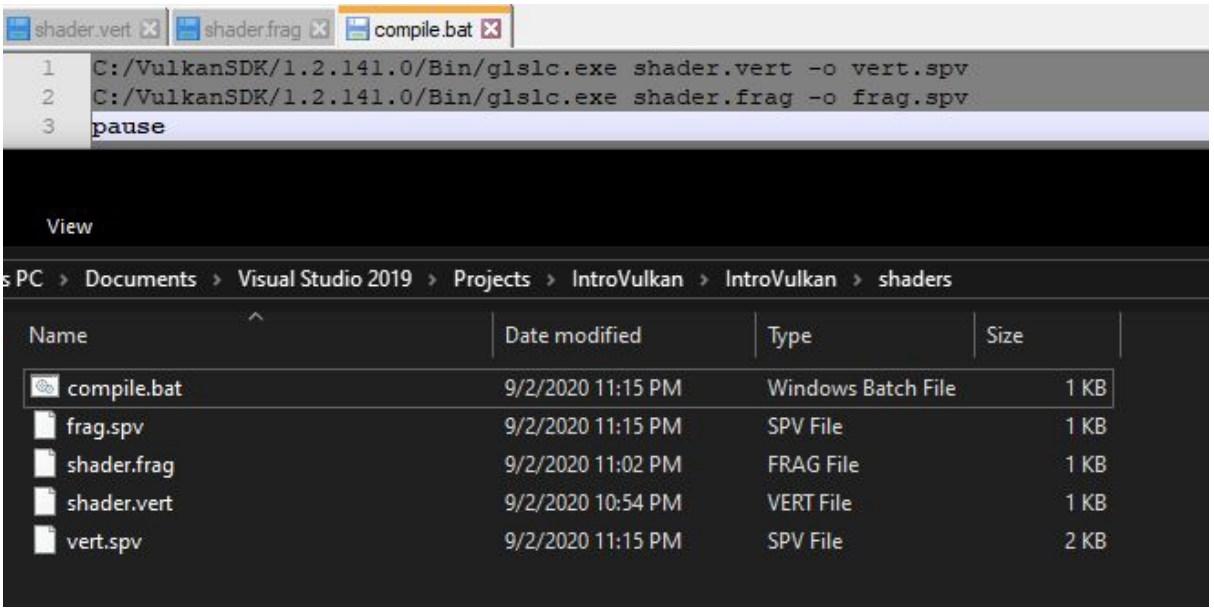


Figura 38. Archivo de compilación de shaders (superior) y directorio de shaders (inferior)

Los comandos descritos, le indican al compilador leer los archivos fuente GLSL y devolver el bytecode SPIR-V usando "-o" (de output). Este proceso se también se puede hacer dentro del programa principal usando la librería "libshaderc".

Para cargar los shaders, primero se deben envolver (wrap) en un objeto VkShaderModule. Se crea una función auxiliar de este tipo llamada createShaderModule que recibe un búfer con el bytecode y su tamaño. Esta información se especifica en la estructura VkShaderModuleCreateInfo. Debido a que el tamaño del bytecode se especifica en bytes pero su apuntador es un apuntador a "uint32_t" en lugar de uno a un "char", se hace un

“cast” mediante “reinterpret_cat”. Se hace notar que este proceso normalmente requiere que se asegure que los datos satisfacen los requerimientos de “uint_t”. Sin embargo, ya que los datos se están almacenando en un “std::vector”, el enlazador predeterminado se asegura de que los datos satisfagan los requisitos de alineación en el peor de los casos. Luego se crea el VkShaderModule mediante vkCreateShaderModule donde los parámetros son iguales a funciones de creación pasadas: dispositivo lógico, apuntador para crear información de la estructura, apuntadores opcionales a enlazadores personalizados y “handle” de variable de salida. El búfer del código se puede liberar inmediatamente después de crear el módulo de shader. Finalmente se regresa el módulo del shader.

Estos módulos son solo envolturas (wrapper) alrededor del bytecode del shader que se cargó anteriormente desde un archivo y que también envuelve las funciones en este. La compilación y enlazamiento de este bytecode a código de máquina (“machine code”) para su ejecución en el GPU no ocurre hasta que se crea el pipeline gráfico. Esto permite destruir los módulos de nuevo en cuanto se termina de crear el pipeline. Por esto, se hacen variables locales en la función de creación del pipeline en lugar de miembros de la clase y se limpian al final de la función llamando vkDestroyShaderModule.

Para usar los shaders se deben asignar a una etapa específica del pipeline utilizando la estructura VkPipelineShaderStageCreateInfo en la función de creación del pipeline. Hay un valor de enumeración para cada etapa del pipeline que se debe utilizar. En la estructura se indica el módulo del shader que contiene el código y la función a invocar de este (conocida como *entrypoint*). Esto permite combinar múltiples shaders en un solo módulo y utilizar diferentes puntos de entrada para distinguir entre sus comportamientos. Se hace una

estructura para cada shader y se define un arreglo que contiene ambas el cual se va a usar para poder referenciarlas. La Figura 39 muestra cómo resulta la función de creación del pipeline y de los módulos de shaders:

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

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

    VkPipelineShaderStageCreateInfo vertShaderStageInfo{};
    vertShaderStageInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
    vertShaderStageInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;
    vertShaderStageInfo.module = vertShaderModule;
    vertShaderStageInfo.pName = "main";

    VkPipelineShaderStageCreateInfo fragShaderStageInfo{};
    fragShaderStageInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
    fragShaderStageInfo.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
    fragShaderStageInfo.module = fragShaderModule;
    fragShaderStageInfo.pName = "main";

    VkPipelineShaderStageCreateInfo shaderStages[] = { vertShaderStageInfo, fragShaderStageInfo };

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

VkShaderModule createShaderModule(const std::vector<char>& code) {
    VkShaderModuleCreateInfo createInfo{};
    createInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
    createInfo.codeSize = code.size();
    createInfo.pCode = reinterpret_cast<const uint32_t*>(code.data());

    VkShaderModule shaderModule;
    if (vkCreateShaderModule(device, &createInfo, nullptr, &shaderModule) != VK_SUCCESS) {
        throw std::runtime_error("failed to create shader module!");
    }

    return shaderModule;
}
```

Figura 39. Creación del pipeline de gráficos y de módulos de shader

3.3.5. Funciones fijas:

En esta etapa, se llenan las estructuras necesarias para configurar las operaciones de funciones fijas. Comenzando con la entrada de vértices, se utiliza la estructura

VkPipelineVertexInputStateCreateInfo que describe el formato de los de datos de vértices que se pasarán al shader de vértices. Esta descripción se hace de dos maneras:

- “Bindings”: es el espacio entre datos y si los datos son por vértice o por instancia.
- Descripciones de atributos: son el tipo de atributos que se transmiten al shader de vértices, a que binding se cargan y a que “offset”.

Por el momento, los datos de los vértices están codificados en el shader de vértices por lo que en la estructura se especifica que que no hay datos de vértices por cargar. La estructura tiene dos miembros: pVertexBindingDescriptions y pVertexAttributeDescriptions. Estos apuntan a un arreglo de estructuras que describen los detalles mencionados para cargar los datos de vértices. Esta estructura se agrega a la función en la que se crea el pipeline gráfico, después del arreglo de las etapas de shaders (shaderStages).

Avanzando al ensamblaje de entradas, se utiliza la estructura VkPipelineInputAssemblyStateCreateInfo que describe dos cosas: el tipo de geometría se va a dibujar desde los vértices y si el reinicio de primitivas debería habilitarse. La primera se especifica en el miembro “topology” y puede tener valores como:

- VK_PRIMITIVE_TOPOLOGY_POINT_LIST: puntos de vértices.
- VK_PRIMITIVE_TOPOLOGY_LINE_LIST: línea de cada 2 vértices sin reutilización.
- VK_PRIMITIVE_TOPOLOGY_LINE_STRIP: el vértice final de cada línea se usa como vértice inicial para la siguiente línea.
- VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST: triángulo de cada 3 vértices sin reutilización.

- VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP: el segundo y tercer vértice de cada triángulo se utilizan como los dos primeros vértices del siguiente triángulo.

Normalmente, los vértices se cargan del búfer de vértices por índice en un orden secuencial. No obstante, con un búfer de elementos (element buffer) se pueden especificar los índices a utilizar. Esto permite hacer optimizaciones como reutilizar vértices. Si se configura el miembro primitiveRestartEnable a VK_TRUE, es posible romper líneas y triángulos en los modos de topologia _STRIP utilizando un indice especial de 0xFFFF o 0xFFFFFFFF. En el ejemplo, como solo se tiene la intención de dibujar triángulos, la topología se define como lista de triángulos y la reiniciación de primitivas se deja en falso.

Continuando con el "viewport", este describe una región del framebuffer al cual la salida le renderiza. Por lo general esto es de cero a la anchura y altura. Como se planea usar las imágenes del swap chain como framebuffers, el tamaño del viewport se iguala a la anchura y altura de estas. Además, el viewport cuenta con los miembros minDepth y maxDepth que especifican los valores de rango de profundidad a utilizar en el framebuffer (normalmente de 0 a 1). No obstante, el minDepth puede ser mayor al maxDepth. Mientras el viewport describe una transformación desde la imagen al framebuffer, "scissor" son rectángulos que definen en qué región los pixeles se van a almacenar. Lo que quede afuera de este rectángulo se descarta por el rasterizador. Funcionan como un filtro, no una transformación. La Figura 40 ejemplifica esto.

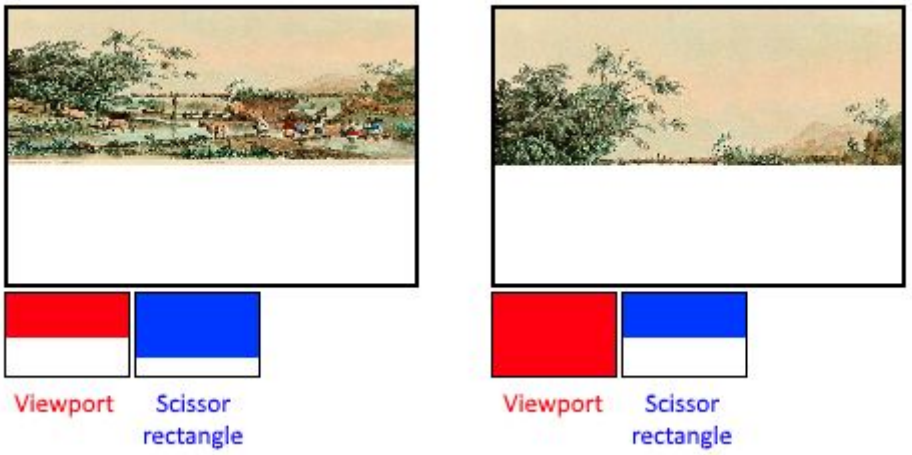


Figura 40. Viewport y Scissor (Overvoorde, A., s.f.)

Como se tiene la intención de dibujar en todo el framebuffer, el scissor se especifica para cubrirlo enteramente. Ulteriormente, el viewport y scissor deben ser combinados a un solo estado de viewport utilizando la estructura VkPipelineViewportStateCreateInfo. Como se pueden utilizar múltiples viewports y scissors en algunas tarjetas gráficas, los miembros de esta estructuran hacen referencia a un arreglo de ellos.

Transitando al rasterizador, este toma la geometría formada por los vértices en el shader de vértices y la convierte a fragmentos para colorearse por el shader de fragmentos. También realiza pruebas de profundidad (depth testing), face culling y pruebas de scissor, y se puede configurar para devolver fragmentos que llenen polígonos enteros o solo las esquinas (wireframe rendering). Todo esto se configura utilizando la estructura VkPipelineRasterizationStateCreateInfo. Si el miembro depthClampEnable se establece como verdadero, los fragmentos que sobrepasan el plano cercano y lejano se sujetan a estos en lugar de ser descartados. Esto sirve para hacer “”shadow maps”. El miembro rasterizerDiscardEnable se define como falso para que la geometría pueda pasar por el

rasterizador, esencialmente permite que haya una salida al framebuffer. El miembro polygonMode determina cómo los fragmentos se generan para geometría y tiene los siguientes modos básicos:

- VK_POLYGON_MODE_FILL: llena el área del polígono con fragmentos.
- VK_POLYGON_MODE_LINE: las esquinas del polígono se dibujan como líneas.
- VK_POLYGON_MODE_POINT: los vértices del polígono se dibujan como puntos.

El miembro lineWidth describe la anchura de las líneas en términos de número de fragmentos. El valor máximo de esto depende de la tarjeta gráfica y sobrepasar un valor de 1 requiere que se habilite la característica wideLine del GPU. El miembro cullMode determine el tipo de face culling a utilizar. Se puede deshabilitar, desechar las caras frontales, traseras o ambas. El campo frontFace especifica el orden de vértices para las caras consideradas como frontales y puede ser en el sentido del reloj o contrario al reloj. Finalmente, el rasterizador puede modificar valores de profundidad agregando valores constantes o sesgarlos basándose en la inclinación de un fragmento. Esto a veces se utiliza para shadow mapping pero es innecesario para este ejemplo.

La siguiente estructura a incluir en el pipeline es VkPipelineMultisampleStateCreateInfo que configura el “multisampling”, que es una de las formas de hacer anti-aliasing. Funciona combinando los resultados del shader de fragmentos de múltiples polígonos que se rasterizan al mismo pixel. Esto ocurre principalmente a lo largo de esquinas que es donde destacan los artefactos de “aliasing”. Como no tiene que ejecutar el shader de fragmentos múltiples veces en caso de que sólo un polígono se asigne un pixel, su costo es menor al de renderizar a una mayor resolución y luego hacer una reducción de escala. Habilitarlo requiere de un atributo del GPU y para este punto del programa se mantiene deshabilitado. Si se van a utilizar

buffers de profundidad o de plantilla, entonces también se deben configurar sus pruebas usando VkPipelineDepthStencilStateCreateInfo. Sin embargo, por ahora esta estructura también puede recibir un nullptr.

Después de que el shader de fragmentos devuelve un color, este necesita ser combinado con un color que ya está en el framebuffer. A esta transformación se le denomina como mezcla de colores (color blending). Tiene dos formas de realizarse:

- Combinar el valor viejo y el nuevo
- Combinar el valor viejo y el nuevo utilizando una operación “bitwise”

Para configurar la mezcla de colores se utiliza la estructura VkPipelineColorBlendAttachmentState que contiene la configuración por framebuffer adjunto y VkPipelineColorBlendStateCreateInfo que contiene la configuración global de la mezcla de colores. La primera estructura permite configurar la primera forma de realizar la mezcla de colores. Si el miembro blendEnable se establece como falso, el nuevo color del shader de fragmentos pasa sin modificarse. De lo contrario, se realizan dos operaciones de mezcla, una para los valores “rgb” y otra para el “alpha” para computar el nuevo color. El resultado pasa por el operador bitwise “AND” junto con el miembro “colorWriteMask” para determinar los canales que se pasan. Sin emabrgo, la forma más común de usar una mezcla de color es implementando una mezcla de alfas, donde el nuevo color se mezcla con el viejo basandose en su opacidad. El valor rgb del color final se puede obtener multiplicando el nuevo color por el nuevo valor alfa y sumando ese valor al producto entre el viejo color por la resta entre 1 y el nuevo valor alfa. El valor alfa del color final simplemente se asigna al alfa del nuevo color. Las operaciones posibles se pueden encontrar en las enumeraciones

VkBlendFactor y VkBlendOP en la especificación de Vulkan. Finalmente, la segunda estructura (VkPipelineColorBlendStateCreateInfo) hace referencia al arreglo de estructuras para todos los framebuffers y permite configurar constantes de la mezcla que se pueden usar como factores de mezcla en las operaciones de operaciones mencionadas previamente. Si se desea utilizar una combinación bitwise como método de mezcla se debe habilitar el miembro logicOpEnable. La operación bitwise se especifica en el campo logicOp. Esto deshabilitará el primer método de combinación. Además el miembro colorWirteMask de la primera estructura tendría que usarse para determinar el canal afectado en el framebuffer. Por el momento, ambos métodos se deshabilitan por lo que los colores de los fragmentos pasan sin modificarse.

Una cantidad limitada del estado que se especifican las estructuras previas puede ser modificada dinámicamente (sin recrear el pipeline) utilizando VkPipelineDynamicStateCreateInfo. Esto permite cambiar el estado de elementos como el viewport, anchura de las líneas, constantes de mezcla, etc. La configuración utilizada en este punto hace que los calores sean ignorados y que se tengan que especificar al dibujar.

La última función fija a especificar es el “pipeline layout” (diseño del pipeline). Se pueden utilizar valores uniformes en los shaders, que son globales, similarmente a variables de estado dinámicas que se pueden cambiar en tiempo de dibujo para alterar el comportamiento de los shaders sin recrearlos. Se usan comúnmente para pasar la matriz de transformación al shader de vértices, o para crear “texture samplers” en el shader de fragmentos. Estos valores uniformes tienen que especificarse durante la creación del pipeline creando un objeto VkPipelineLayout. Aunque estos valores no se utilicen, es requisito crear el pipeline layout,

aunque esté vacío. Por ende, se crea un miembro de la clase que contenga este objeto para poder referenciar en otras funciones. Luego se crea el objeto, especificando como de costumbre una estructura llamada VkPipelineLayoutCreateInfo en la función de creación del pipeline; y se destruye en la función de limpieza. Esta estructura específica "push constants" que son un método de pasar valores dinámicos a los shaders.

3.3.6. Render Passes:

Especifica cuántos buffers de color y profundidad habrá, cuántas muestras usar para cada uno de ellos y cómo se debe manejar su contenido durante las operaciones de renderizado. Esta información se envuelve en un objeto llamado "render pass" que requiere su propia función de creación la cual se llama en la función de inicialización antes de la creación del pipeline. Dentro de esta función se define una estructura que describe los elementos adjuntos al framebuffer, en este caso el búfer de color, representado por las imagenes en el swap chain. El formato del color en la estructura debe emparejarse con el formato en las imágenes del swap chain. Como no se está utilizando multisampling, "samples" se mantiene en 1. Los miembros loadOp y storeOp determinan qué hacer con los datos del elemento adjunto antes y después de renderizar. Las opciones del primero son:

- VK_ATTACHMENT_LOAD_OP_LOAD: Conservar el contenido existente de lo adjunto
- VK_ATTACHMENT_LOAD_OP_CLEAR: Limpia los valores a una constante al inicio
- VK_ATTACHMENT_LOAD_OP_DONT_CARE: El contenido inexistente se queda indefinido

Para este caso, se utiliza la operación de limpieza paara limpiar el framebuffer a negro antres de dibujar un nuevo frame. Para storeOp hay dos opciones:

- VK_ATTACHMENT_STORE_OP_STORE: Contenido renderizado se guarda en memoria y puede leerse después
- VK_ATTACHMENT_STORE_OP_DONT_CARE: El contenido del framebuffer permanece indefinido después de la operación de renderización

Como se desea ver el triángulo en pantalla, la opción a elegir es la de almacenamiento. Estos miembros aplican a datos de color y profundidad pero los miembros stencilLoadOp y stencilStoreOp aplican a datos de plantilla que para los propositos actuales son inecearios. Ahora, las texturas y los framebuffers se representan por objetos VkImage con un cierto formato de pixel. Empero, el "layout" de los pixeles en memoria puede cambiar dependiendo de lo que se esté haciendo con la imagen. Las configuraciones más comunes de esto son:

- VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL: Imágenes se usan como adjunto de color
- VK_IMAGE_LAYOUT_PRESENT_SRC_KHR: Imágenes a presentarse en el swap chain
- VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL: Imagenes a utilizarse como destino para una operación de copia de memoria

En fin, el miembro initLayout específica el layout que la imagen tendrá antes de comenzar el render pass y finalLayout específica el layout para transicionar automáticamente cuando el render pass termine. Dejar el layout inicial como indefinido basta ya que implica que no

importa el layout en el que la imagen estaba. Como se requiere que la imagen esté lista para presentación usando el swap chain después de renderizar, se utiliza la segunda configuración para el layout final.

Un render pass o pase de render, puede tener varios sub-pases (“subpasses”). Estas son operaciones de renderización subsecuentes que dependen del contenido de los framebuffers en pases pasados. Si estas operaciones se agrupan en un solo pase, Vulkan puede reordenarlas para conservar ancho de banda de memoria para mejorar el rendimiento. Para el triángulo es suficiente usar un subpass. Cada uno de estos hace referencia a uno o más elementos adjuntos descritos en la estructura previa. Estas referencias son estructuras de tipo VkAttachmentReference. Contienen un parámetro (“attachment”) que especifica el adjunto a referenciarse por su índice en el arreglo de descripciones de elementos adjuntos. Como el arreglo consiste de un solo VkAttachmentDescription, el índice es 0. El miembro layout, indica que layout debería tener el elemento adjunto durante un subpass que utilice esta referencia. Vulkan hará una transición automática del elemento adjunto a este layout cuando el subpass comience. Se planea usar el elemento adjunto como un búfer de color por lo que VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL para obtener el mejor rendimiento. El subpass se describe mediante la estructura VkSubpassDescription en donde se debe especificar que el subpass es de gráficos. Luego se especifica la referencia al adjunto de color. Se hace referencia al índice del adjunto en este arreglo desde el shader de fragmentos con la directiva layout(location = 0) out vec4 outColor. Se puede hacer referencia a los demás tipos de adjuntos por estos subpasses:

- pInputAttachments: Adjuntos que se leen desde un shader
- pResolveAttachments: Adjuntos usados para hacer multisampling a adjuntos de color

- pDepthStencilAttachment: Adjuntos para datos de profundidad y plantilla
- pPreserveAttachments: Adjuntos que no son usados por el subpass, pero cuyos datos deben ser preservados

Ya con esto, se puede crear el render pass creando una nueva variable que sostenga un objeto VkRenderPass. Este objeto se puede crear llenando la estructura VkRenderPassCreateInfo con un arreglo de adjuntos y subpasses. Se hace referencia a los objetos VkAttachmentReference usando los índices de este arreglo. Como se va utilizar el render pass a lo largo del programa, se debe limpiar al final, después del pipeline layout.

Finalmente con todo esto, se puede comenzar a definir la estructura VkGraficsPipelineCreateInfo al final de la función de la creación del pipeline pero antes de las destrucción de los módulos de shaders ya que son necesarios para su creación. Se comienza haciendo referencia a las estructuras VkPipelineShaderStageCreateInfo. Luego se hace referencia a todas la estructuras que describen las etapas de funciones fijas seguido del handle del pipeline layout y finalmente el render pass y el índice del subpass que el pipeline va a utilizar. Dos parámetros opcionales son basePipelineHandle y basePipelineIndex que permiten crear nuevos pipelines derivados del actual que facilitan y agilizan la creación de nuevos pipelines. Como hay un solo pipeline, el handle se mantiene nulo y se especifica un indice invalido. Ahora se crea un miembro de la clase que contenga el objeto VkPipeline y se crea como de costumbre con la función vkCreateGraphicsPipelines. El segundo parámetro de esta función hace referencia a un objeto opcional llamado VkPipelineCache que se puede usar para almacenar y reutilizar datos relevantes para crear un pipeline a traves de multiples llamadas a esta función y entre ejecuciones del programa si el “cache” está almacenado en

una carpeta. En esencia esto acelera la creación del pipeline. Como el pipeline se requiere para una operación de dibujo común, se destruye al final del programa.

3.4. Dibujo:

3.4.1. Framebuffers:

Los elementos adjuntos que se especificaron en la creación del render pass se atan envolviendolos en un objeto VkFramebuffer. Un objeto framebuffer hace referencia a todos los objetos VkImageView que representan los elementos adjuntos que en este caso es solo uno: el de color. Sin embargo,la imagen que se va a utilizar para el adjunto de color depende de que imagen devuelva el swap chain al momento de recuperar una para presentarla. Esto implica que hay que crear un framebuffer para todas las imágenes en el swap chain y utilizar una que corresponda a la imagen recuperada en tiempo de dibujo. Entonces se debe crear un miembro de la clase de tipo vector que contenga los framebuffers. Estos objetos se van a crear en una nueva función (createFramebuffers) y llamar en la inicialización. Lo primero que se hace es ajustar el tamaño del vector de buffers, igualando al de las vistas de imagen en el swap chain. Luego se recorre la lista de vistas de imagen y se crea un framebuffer de ellas. La estructura para crear los framebuffers debe especificar qué render pass necesita el framebuffer para ser compatible. Los parametros attachmentCount y pAttachments especifican los objetos VkImageView que deben atarse a las descripciones de adjuntos respectivas en el arreglo del render pass pAttachment. El campo “layers” se refiere al número de capas en los arreglos de imagen. Las imágenes del swap chain son imágenes singulares por lo que el número de capas es 1. Finalmente, se borran los framebuffers antes de las vistas de imagen y render pass de los que se basan pero al haber terminado el proceso de renderización. La Figura 41 expone cómo resulta la función de creación de los framebuffers.

```
void createFramebuffers() {
    swapChainFramebuffers.resize(swapChainImageViews.size());

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

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

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

Figura 41. Creación de framebuffers

3.4.2. Command Buffers:

En Vulkan, se tienen que registrar todas las operaciones que se necesitan realizar en objetos de buffers de comandos en lugar de llamadas a funciones. La ventaja de esto está en que configurar los comandos de dibujo puede hacerse anticipadamente y en múltiples “threads”. Luego solo se le tiene que dar la instrucción a Vulkan para ejecutar los comandos en el ciclo principal.

Primero se tienen que crear “command pool” (alberca de comandos) antes de crear los buffers de comandos. Estas albercas administran la memoria que se usa para almacenar los búferes, los búferes de comandos se asignan desde estas. Entonces se hace un nuevo miembro de la clase de tipo VkCommandPool y una función para su creación que se llama después de la creación de los framebuffers. La estructura de su creación toma dos parametros: queueFamilyIndex y flags. Los búferes de comandos se ejecutan enviándolos a una de las colas de dispositivos, como las colas de presentación y gráficas recuperadas. Cada alberca puede asignar sólo un búfer de comandos que se envían sobre una cola de un solo tipo. Como

se van a registrar comandos de dibujo, se elige la familia de colas de gráficos. Hay dos banderas posibles para estas albercas:

- VK_COMMAND_POOL_CREATE_TRANSIENT_BIT: Sugerencia de que los búferes de comandos se vuelven a grabar con nuevos comandos con mucha frecuencia (esto puede cambiar el comportamiento de asignación de memoria).
- VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT: Permitir que los búferes de comando se vuelvan a grabar individualmente, sin esta bandera, todos deben restablecerse juntos.

Como solo se van a registrar los búferes de comandos al principio del programa y luego ejecutarlos varias veces en el ciclo principal, no se van a necesitar estas banderas. Entonces se crea la alberca de comandos usando vkCreateCommandPool. Como los comandos se van a utilizar a lo largo del programa para dibujar en pantalla, la alberca se debe destruir al final.

Ahora se pueden asignar los buffers de comandos y registrar comandos de dibujo en ellos. Como uno de los comandos de dibujo involucra enlazar el VkFramebuffer correcto, se tiene que registrar un búfer de comandos para cada imagen en el swap chain de nuevo. Por ello, se crea un vector de objetos VkCommandBuffer como miembros de la clase. Los buffers de comandos se liberan automáticamente cuando su alberca se destruya entonces no se tienen que destruir en la función de limpieza. Se crea entonces una función para crear los buffers de comandos que se llama después de la creación de las albercas y que asigna y registra los comandos por cada imagen del swap chain. Estos búfers se asignan con la función vkAllocateCommandBuffers que toma una estructura VkCommandBufferAllocateInfo como parámetro el cual especifica la alberca de comandos y el número de buffers a asignar. De esta

estructura, su miembro “level” especifica si los buffers de comandos asignados son principales o secundarios. En este contexto, principal implica que los comandos pueden enviarse a una cola para su ejecución pero no se pueden llamar de otro búfer de comandos. El nivel secundario indica que los comandos no pueden enviarse directamente pero sí se pueden llamar desde buffers de comandos principales. El nivel secundario no se necesita utilizar para los propósitos actuales.

Ahora se puede comenzar a registrar un búfer de comandos llamando vkBeginCommandBuffer con una estructura VkCommandBufferBeginInfo como argumento que especifica algunos detalles sobre cómo usar un búfer de comandos específico. Esto se hace mediante un ciclo “for” donde su condición es igual al tamaño del búfer de comandos. La estructura mencionada tiene un parámetro llamado “flags” que especifican cómo se van a usar estos buffers y cuenta con los siguientes valores (aunque ninguno es aplicable por el momento):

- VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT: El búfer de comando se volverá a grabar inmediatamente después de ejecutarlo una vez.
- VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT: Este es un búfer de comando secundario que estará completamente dentro de un solo render pass.
- VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT: El búfer de comando se puede volver a enviar mientras también está pendiente de ejecución.

El miembro pInheritanceInfo solo sirve para buffers de comandos secundarios. No obstante, especifica que estado heredar al llamar un buffers de comandos principales. Si el búfer de

comandos ya se registro una vez, una llamada a vkBeginCommandBuffer lo reinicia implícitamente.

Para dibujar se debe comenzar el render pass con vkCmdBeginRenderPass. Este pase se configura usando algunos parámetros en la estructura VkRenderPassBeginInfo. Sus primeros dos parámetros (además de sType) son el render pass y los elementos adjuntos a enlazar. Para esto se creó un famebuffer para cada imagen del swap chain que lo especifica como un adjunto de color. Los siguientes dos definen el tamaño del área de renderización. Esta área define donde el almacenamiento y carga de shaders toma lugar. Los píxeles fuera de esta región tendrán valores indefinidos. El tamaño del área debería ser igual al tamaño de los adjuntos para optimizar el rendimiento. Los últimos dos parámetros definen los valores claros a usar para VK_ATTACHMENT_LOAD_OP_CLEAR, que se usó como una operación de carga para el adjunto de color. En el ejemplo, el color claro se definio como negro con 100% de opacidad. Con esto ya se puede iniciar el render pass. Todas las funciones que registran comandos pueden ser reconocidas por el prefijo "vkCmd" y todas regresan "void" por lo que no hay manejo de errores hasta que se haya terminado el proceso de registrar.

El primer parámetro para cada comando siempre es el búfer de comandos para registrar el comando. El segundo especifica los detalles del render pass y el último controla como los comandos de dibujo dentro del render pass serán proporcionados. Puede tener uno de dos valores:

- VK_SUBPASS_CONTENTS_INLINE: Los comandos del render pass se incrustan en el búfer de comando principal y no se ejecutan los búferes de comando secundarios.

- VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS: Los comandos del render pass se ejecutan desde búferes de comandos secundarios.

Como no se utilizan los secundarios, la opción elegida en el ejemplo es la primera. Ahora se puede enlazar el pipeline mediante la función vkCmdBindPipeline. El segundo parámetro de esta función especifica si el objeto del pipeline es un pipeline de gráficos o cómputo. Con esto, Vulkan sabe qué operaciones ejecutar en el pipeline de gráficos y que elementos adjuntos utilizar en el shader de fragmentos. Ahora se puede dibujar un triángulo. Primero se utiliza vkCmdDraw con los siguientes parámetros:

- Búfer de comandos
- vertexCount: aunque no hay un búfer de vértices aún, se tienen que dibujar tres vértices.
- instanceCount: Usado para renderizado instanciado, de lo contrario el valor asignado es 1.
- firstVertex: Usado como un desplazamiento en el búfer de vértices, define el valor más bajo de gl_VertexIndex.
- firstInstance: Utilizado como un desplazamiento para renderización instanciada, define el valor más bajo de gl_InstanceIndex.

Finalmente, el render pase de renderización se puede terminar utilizando vkCmdEndRenderPass y se termina registrando el búfer de comandos utilizando vkEndCommandBuffer. La Figura 42 expone el cómo resulta la función de creación del búfer de comandos.

```
void createCommandBuffers() {
    commandBuffers.resize(swapChainFramebuffers.size());

    VkCommandBufferAllocateInfo allocInfo{};
    allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    allocInfo.commandPool = commandPool;
    allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
    allocInfo.commandBufferCount = (uint32_t)commandBuffers.size();

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

    for (size_t i = 0; i < commandBuffers.size(); i++) {
        VkCommandBufferBeginInfo beginInfo{};
        beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;

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

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

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

        vkCmdBeginRenderPass(commandBuffers[i], &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);

        vkCmdBindPipeline(commandBuffers[i], VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);

        vkCmdDraw(commandBuffers[i], 3, 1, 0, 0);

        vkCmdEndRenderPass(commandBuffers[i]);

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

Figura 42 Creación del búfer de comandos

3.4.3. Renderización y Presentación:

Ahora se puede hacer la función del ciclo principal del programa (mainLoop) el cual va a llamar otra función nueva de dibujo (drawframe). Esta última realizará tres operaciones:

- Conseguir una imagen del swap chain.
- Ejecutar el búfer de comandos con esa imagen adjuntada en el framebuffer.

- Devolver la imagen al swap chain para presentarse.

Cada una de estas operaciones se ejecuta con una sola llamada de función pero de manera asincrónica. Las llamadas a la función regresan antes de que las operaciones se terminen y el orden de ejecución es indefinido. No obstante, cada una de las operaciones depende de que termine la anterior. Para combatir esto, existen dos métodos que sincronizan los eventos del swap chain: “fences” (cercas) y “semaphores” (semáforos). Ambos son objetos que pueden usarse para coordinar operaciones haciendo que una operación señale y otra espere a una cerca o semáforo para avanzar de un estado señalado o sin señalar. La diferencia radica en que los estados de las cercas pueden ser accesados desde el programa usando llamadas como vkWaitForFences mientras que en los semáforos no. Las cercas están diseñadas para sincronizar la aplicación con las operaciones de renderización. Los semáforos se usan para sincronizar operaciones dentro o a través de colas de comandos. En este caso es mejor usar semáforos ya que la intención es sincronizar las colas de operaciones de comandos de dibujo y presentación.

Se necesita un semáforo para señalar que una imagen ha sido adquirida y está lista para renderizarse, y otro para señalar que la renderización ha terminado y se puede presentar. Entonces se crean dos miembros para almacenar estos objetos de tipo semáforo (VkSemaphore). Entonces se agrega una función más dentro de la inicialización para crear semáforos (createSemaphores). Se llena una estructura (VkSemaphoreCreateInfo) y se crean ambos semáforos mediante vkCreateSemaphore. Finalmente se limpian al final del programa.

Para obtener una imagen del swap chain en la función de dibujo se usa vkAcquireNextImageKHR. Sus primeros dos parámetros son el dispositivo lógico y el swap chain del cual se están obteniendo las imágenes. El tercero especifica el "timeout" (tiempo de terminación) en nanosegundos para que una imagen esté disponible. Para esto se utiliza el valor máximo de un entero de 64 bits sin signo el cual deshabilita el timeout. Los siguientes dos parámetros especifican la sincronización de objetos que se van a señalar cuando el motor de presentación termine de usar la imagen. En ese momento, se puede empezar a dibujar. Aquí se va a utilizar imageAvailableSemaphore. El último parámetro especifica una variable para devolver el índice de la imagen del swap chain que se vuelve disponible. El índice se refiere al objeto VkImage en el arreglo swapChainImages. Este índice se va a utilizar para elegir el búfer de comandos correcto.

Envíos de colas y sincronización se configuran a través de parámetros en la estructura VkSubmitInfo. Sus primeros tres parámetros indican a qué semáforos esperar antes de comenzar ejecución y en qué etapa(s) del pipeline esperar. Como se quiere esperar con escribir colores a la imagen hasta que este disponible, se especifica la etapa del pipeline grafico que escribe al adjunto de color. Esto implica que la implementación puede empezar a ejecutar el shader de vértices mientras la imagen aún no está disponible. Cada entrada en el arreglo waitStages corresponde al semáforo con el mismo índice en pWaitSemaphores. Los siguientes parámetros especifican a que búferes de comandos enviar para ejecución. Se debería enviar el búfer de comandos que enlace la imagen del swap chain que se acaba de adquirir como adjunto de color. El parámetro signalSemaphoreCount y pSignalSemaphores especifican a qué semáforos señalar una vez que el o los búferes de comandos terminen de ejecutarse. Para esto se usa renderFinishedSemaphore. Luego se envía el búfer de comandos

a la cola de gráficos usando vkQueueSubmit. La función toma un arreglo de estructuras VkSubmitInfo como argumento para eficacia para cuando la carga de trabajo sea mucho mayor. El último parámetro hace referencia a una cerca opcional que se señala cuando el búfer de comandos termine ejecución. Este es nulo ya que se están usando semáforos.

Las transiciones entre layouts de imágenes se controlan automáticamente por dependencias de sub-pases, las cuales especifican dependencias de memoria y ejecución entre sub-pases. Ahora solo se tiene un sub-pase pero las operaciones antes y después también cuentan como sub-pases implícitos. Hay dos dependencias nativas que se encargan de la transición al principio y al final de un pase de renderización. No obstante, la transición final no ocurre en el tiempo correcto. Asume que la transición ocurre al principio del pipeline, antes de adquirir la imagen. Se puede entonces cambiar waitStages para imageAvailableSemaphore a VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT para asegurar que los pases de renderización no comiencen hasta que la imagen esté disponible; o se puede hacer que el pase de renderización espere a la etapa VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT.

Yendo por el segundo método, se tiene que especificar una estructura VkSubpassDependency en la función de creación de pase de renderización (createRenderPass). Los primeros dos parámetros de esta estructura especifican los índices de las dependencias y los sub-pases dependientes. El valor VK_SUBPASS_EXTERNAL se refiere al sub-pase implícito antes o después del pase de renderización dependiendo de si se especificó en srcSubpass o dstSubpass. El índice 0 se refiere al primer y único sub-pase. El campo dstSubpass siempre debe ser mayor a srcSubpass para prevenir ciclos en el gráfico de dependencia. Los

siguientes dos campos especifican las operaciones de espera y las etapas en las que estas operaciones ocurren. Se necesita esperar a que el swap chain termine de leer las imágenes antes de accederlo. Esto se puede lograr esperando la etapa de salida del adjunto de color. Las operaciones que deberian esperar estan en la etapa de adjunto de color que involucran escribir el adjunto de color. Estas configuraciones previenen que la transición ocurra hasta que sea necesario y permitido, que es cuando se quiere empezar a escribir colores. Finalmente, la estructura VkRenderPassCreateInfo cuenta con dos campos para especificar un arreglo de dependencias: dependencyCount y pDependencies.

El último paso para dibujar un cuadro es enviar los resultados de regreso al swap chain para que los muestre en pantalla. La presentación se configura por medio de la estructura VkPresentInfoKHR que se coloca al final de la función de dibujo. Sus primeros dos parámetros especifican qué semáforos esperar antes de poder presentar. Los siguientes dos indican a qué swap chains presentar imágenes y el índice de la imagen para cada swap chain. Casi siempre será a uno solo. Hay un miembro más que es opcional, llamado pResults. Permite especificar un arreglo de valores VkResult para revisar para cada swap chain individual si la presentación fue exitosa. Esto es innecesario para un solo swap chain porque se puede usar el valor de salida de la función actual. Posteriormente, se utiliza vkQueuePresentKHR para enviar la solicitud para presentar una imagen al swap chain. Para evitar que se limpien recursos mientras continúan las operaciones de dibujo, se usa vkDeviceWaitIdle el cual recibe como argumento el dispositivo lógico para esperar a que este termine operaciones antes de salir del ciclo principal.

Para este punto, la aplicación está enviando trabajo a la función de dibujo sin revisar que se termine por lo que pueden surgir errores en las capas de validación hacer crecer el uso de memoria. También se está utilizando los semáforos y los búferes de comandos para múltiples cuadros al mismo tiempo. Para resolver esto se puede extender la aplicación para permitir que múltiples cuadros estén "in-flight" (en vuelo) sin limitar la cantidad de trabajo que se acomula. Se comienza entonces agregando una constante en la cima del programa que define cuántos cuadros deberían procesarse simultáneamente. Cada marco debe tener sus propios semáforos y cada función de creación de semáforos debe modificarse para crearlos. Similarmente, todos deben destruirse. Para utilizar el par correcto de semáforos cada vez, se necesita rastrear el cuadro actual. Para esto se utiliza un índice de cuadro de tipo size_t. La función de dibujo entonces se puede modificar para usar los objetos correctos. También esta función debe avanzar al siguiente marco cada vez y para esto se utiliza un operador de módulo (%) que asegura que el índice de cuadros se repita después de cada MAX_FRAMES_IN_FLIGHT cuadros en cola.

En seguida, para realizar sincronización entre el CPU y el GPU, se pueden usar cercas ya que se puede esperar por ellas dentro del código. Entonces al igual que con los semáforos, se crea una cerca para cada cuadro (usando un vector). Por facilidad, en el ejemplo se crean las cercas junto con los semáforos por lo que la función en la que se crean los semáforos se renombra createSyncObject (creación de objetos de sincronización). El proceso de creación es similar al de semáforos (con vkCreateFence) y también deben destruirse en la función de limpieza (con vkDestroyFence). Al terminar esto, se cambia la función de dibujo para utilizar las cercas. La función vkQueueSubmit se usa ya que incluye un parámetro opcional para pasar la cerca que debe señalarse cuando el búfer de comandos termine de ejecutarse.

Esta señal se usa para indicar que se terminó un cuadro. Posteriormente, se cambia el principio de la función de dibujo para esperar a que el cuadro termine. Se usa entonces vkWaitForFences que toma un arreglo de cercas y espera a cualquiera o todas sean señaladas antes de devolver algo. El cuarto parámetro recibe VK_TRUE para indicar que se desea esperar por todas las cercas. El último parámetro también sirve como tiempo de terminación. Luego se reinicia la cerca a su estado sin señal con vkResetFences. Para evitar que el programa entre en un estado de espera perpetuo, se cambia la creación de cercas para inicializarse en un estado señalado. Finalmente, para evitar que se renderize a una imagen del swap chain que ya está en vuelo, se debe rastrear cada imagen en el swap chain sí un cuadro en vuelo la está utilizando. Este mapeo va a referirse a los cuadros en vuelo por sus cercas para poder inmediatamente tener un objeto de sincronización al cual esperar antes de que un nuevo cuadro pueda utilizar esa imagen. Para esto, se agrega una nueva lista para rastrear (imagesInFlight) y se prepara en createSyncObjects. Como al inicio ningún cuadro utiliza una imagen se debe inicializar explícitamente a “no fence” (sin cerca). Entonces se modifica drawFrame para que espere por cualquier cuadro previo que esté usando la imagen que se acaba de asignar para el nuevo marco. Al tener más llamadas a vkWaitForFences, la llamada a vkResetFences debe moverse a la posición previa al uso de la cerca. Ahora el método de sincronización se asegura de que no haya más de dos cuadros de trabajo en cola y que estos cuadros no utilicen accidentalmente la misma imagen. La Figura 43 muestra la función de creación de objetos de sincronización y la Figura 44 muestra la función de dibujo.

```
void createSyncObjects() {
    imageAvailableSemaphores.resize(MAX_FRAMES_IN_FLIGHT);
    renderFinishedSemaphores.resize(MAX_FRAMES_IN_FLIGHT);
    inFlightFences.resize(MAX_FRAMES_IN_FLIGHT);
    imagesInFlight.resize(swapChainImages.size(), VK_NULL_HANDLE);

    VkSemaphoreCreateInfo semaphoreInfo{};
    semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;

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

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

Figura 43. Creación de objetos de sincronización

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

    uint32_t imageIndex;
    vkAcquireNextImageKHR(device, swapChain, UINT64_MAX, imageAvailableSemaphores[currentFrame], VK_NULL_HANDLE, &imageIndex);

    if (imagesInFlight[imageIndex] != VK_NULL_HANDLE) {
        vkWaitForFences(device, 1, &imagesInFlight[imageIndex], VK_TRUE, UINT64_MAX);
    }
    imagesInFlight[imageIndex] = inFlightFences[currentFrame];

    VkSubmitInfo submitInfo{};
    submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;

    VkSemaphore waitSemaphores[] = { imageAvailableSemaphores[currentFrame] };
    VkPipelineStageFlags waitStages[] = { VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT };
    submitInfo.waitSemaphoreCount = 1;
    submitInfo.pWaitSemaphores = waitSemaphores;
    submitInfo.pWaitDstStageMask = waitStages;

    submitInfo.commandBufferCount = 1;
    submitInfo.pCommandBuffers = &commandBuffers[imageIndex];

    VkSemaphore signalSemaphores[] = { renderFinishedSemaphores[currentFrame] };
    submitInfo.signalSemaphoreCount = 1;
    submitInfo.pSignalSemaphores = signalSemaphores;

    vkResetFences(device, 1, &inFlightFences[currentFrame]);

    if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, inFlightFences[currentFrame]) != VK_SUCCESS) {
        throw std::runtime_error("failed to submit draw command buffer!");
    }

    VkPresentInfoKHR presentInfo{};
    presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;

    presentInfo.waitSemaphoreCount = 1;
    presentInfo.pWaitSemaphores = signalSemaphores;

    VkSwapchainKHR swapChains[] = { swapChain };
    presentInfo.swapchainCount = 1;
    presentInfo.pSwapchains = swapChains;

    presentInfo.pImageIndices = &imageIndex;

    vkQueuePresentKHR(presentQueue, &presentInfo);

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

Figura 44. Función de dibujo

3.4.4. Recreación del swap chain

Para este punto, es posible que la superficie de la ventana cambie y comprometa la compatibilidad del swap chain con ella por lo que estos eventos deben atraparse y el swap chain debe recrearse. Por lo tanto se hace una función para recrear el swap chain que llame la función de creación del swap chain y todas las funciones de creación para los objetos que dependen del swap chain o el tamaño de la ventana. Estas incluyen la creación de vistas de imágenes, pases de renderización, el pipeline gráfico, framebuffers y búferes de comandos. Asimismo, antes que nada, se llama a vkDeviceWaitIdle para evitar tocar recursos que sigan en uso. Para asegurar que las versiones viejas de estos objetos se limpien antes de recrearlas, parte del código de limpieza se mueve a una función separada (cleanupSwapChain) que se llama desde la función de recreación. Esta nueva función se puede llamar desde la función original de limpieza. Para las albercas de comandos se utiliza vkFreeCommandBuffers para poder utilizar la alberca existente y asignar nuevos búferes de comandos. Para manejar cambios de tamaño de ventana, se necesita pedir el tamaño del framebuffer para asegurar que las imágenes del swap chain tienen el (nuevo) tamaño correcto. Se usa entonces la función choosesSwapExtent. Ahora se debe averiguar cuándo la recreación del swap chain es necesaria. Las funciones vkAcquireNextImageKHR y vkQueuePresentKHR regresan valores que indican esto:

- VK_ERROR_OUT_OF_DATE_KHR: El swap chain se ha vuelto incompatible con la superficie y ya no se puede utilizar para renderizar. Suele ocurrir después de un cambio de tamaño de ventana.
- VK_SUBOPTIMAL_KHR: El swap chain aún se puede usar para presentar con éxito en la superficie, pero las propiedades de la superficie ya no coinciden exactamente.

Si el swap chain está desactualizado al intentar adquirir una imagen, entonces ya no es posible presentarle. En este caso se debe recrear el swap chain inmediatamente e intentar de nuevo en la siguiente llamada a la función de dibujo. VK_SUCCESS y VK_SUBOPTIMAL_KHR se consideran como devoluciones exitosas. La función vkQueuePresentKHR regresa los mismos valores con el mismo significado. En este caso se recrea el swap chain si es subóptimo para tratar de obtener el mejor resultado posible.

Aunque muchos drivers y plataformas activan VK_ERROR_OUT_OF_DATE_KHR automáticamente después de cambiar el tamaño de la ventana, esto no es garantía. Para manejar esto explícitamente se agrega un nuevo miembro variable que indique un cambio en tamaño. Se modifica la función de dibujo para revisar esta bandera. Esto se debe hacer después de vkQueuePresentKHR para asegurar que los semáforos se encuentran en estados consistentes. Para detectar cambios en tamaño se usa la función glfwSetFramebufferSizeCallback para configurar una devolución de llamada ("callback"). Se crea una función estática para esto ya que GLFW no sabe llamar de forma adecuada a una función miembro con el apuntador "this" correcto hacia la instancia de la aplicación. No obstante, hay una referencia al GLFWwindow en esta devolución y hay otra función de GLFW que permite almacenar apuntadores arbitrarios en ella: glfwSetWindowUserPointer. Este valor se puede obtener desde el callback con glfwGetWindowUserPointer para configurar una bandera adecuadamente. Para manejar minimizaciones de ventana que resulten en el búfer de cuadros teniendo un tamaño de 0, este se puede poner en pausa hasta que la ventana esté en primer plano. Entonces se extiende de nuevo la función de recreación del swap chain de forma que la llamada inicial a glfwGetFramebufferSize maneje el caso en

el que el tamaño ya es correcto y glfwWaitEvents no tenga nada sobre qué esperar. Ejecutar el programa resulta en la Figura 45.

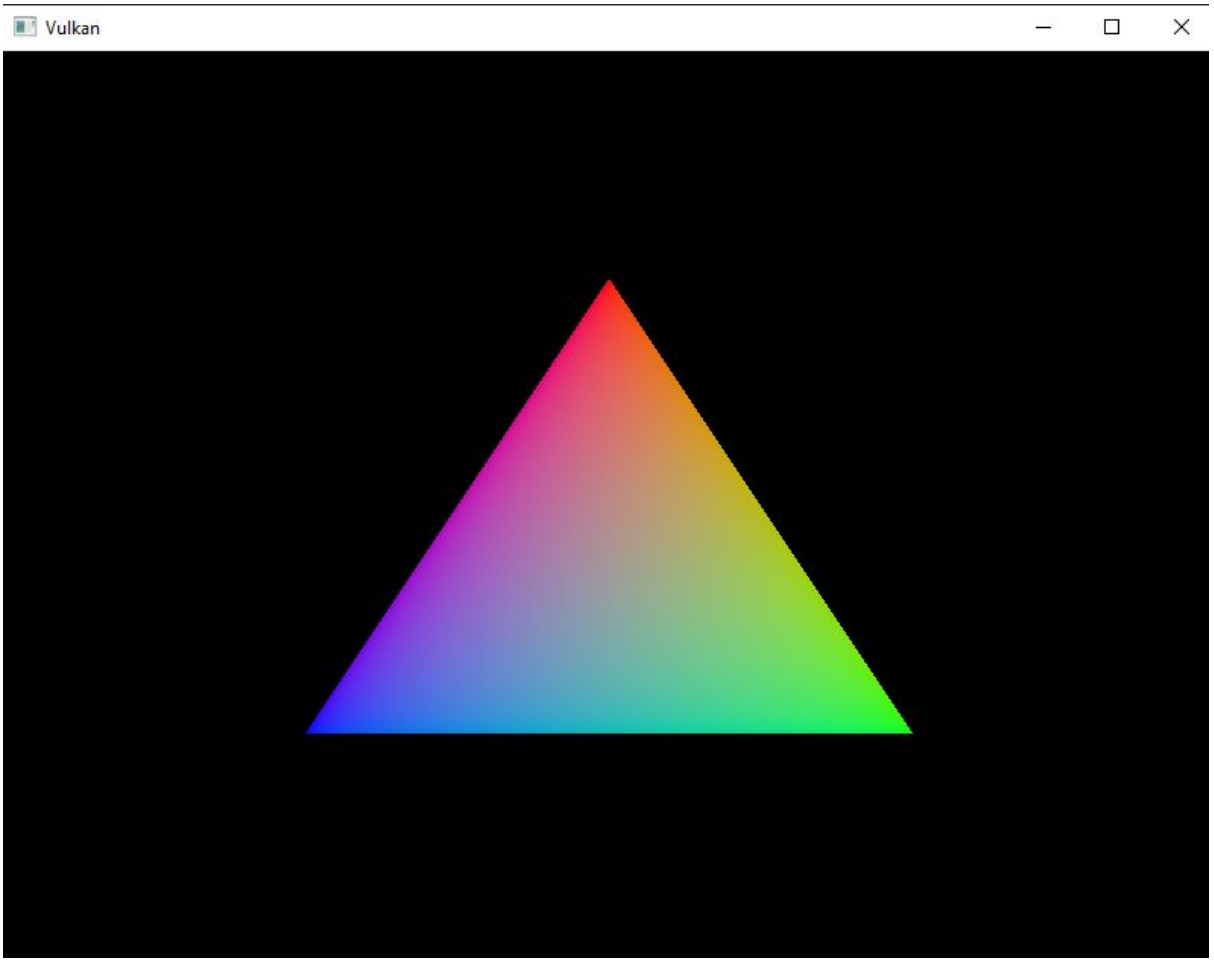


Figura 45. Triángulo rasterizado en Vulkan.

4. Vertex Buffers:

En esta sección se reemplazan los datos de los vértices en el shader de vértices con un búfer de vértices. Éste búfer es accesible por el CPU y se copian los datos de los vértices a él directamente. Luego se utiliza un búfer provisional para copiar los datos de los vértices a la memoria de alto rendimiento.

4.1. Descripción de entrada de vértices:

Empezando con el shader de vértices, se reemplazan los datos de los vértices por las variables "inPosition" e "inColor" (ambos atributos de variables). Estas variables son propiedades que se especifican por cada vértice en el búfer de vértices. Al igual que con "fragColor", "layout(location = x)" sirve para asignar índices a las entradas y para poder referenciarlas más tarde. Como nota, si se está utilizando un vector de 64 bits (flotante de doble precisión) como dvec3 utilizan múltiples "slots". Esto implica que el siguiente índice debe estar dos posiciones adelante.

Ahora en el programa principal, se puede incluir la librería de GLM y posteriormente utilizar sus tipos para crear una estructura llamada "Vertex" que contenga los atributos a enviar al shader. El arreglo especificado en esta estructura también se usa para combinar los valores de posición; a esto se le conoce como "interleaving vertex" vertices entrelazados.

El siguiente paso es especificar cómo transmitir estos datos al shader de vértices una vez que se carga a la memoria del GPU. Para esto se utilizan dos estructuras:

- VkVertexInputBindingDescription que describe la velocidad en la que se cargan datos de la memoria a través de los vértices. Especifica el número de bytes entre entradas de datos y si avanzar a la siguiente entrada de datos después de cada vértice o cada instancia.
- VkVertexInputAttributeDescription que describe cómo manejar los vértices entrantes. Indica cómo extraer los atributos de un vértice de un pedazo de los datos de vértices que vienen de la descripción de enlazamiento. Como hay dos atributos (posición y color) se necesitan dos estructuras.

Para ambas estructuras, se agregan funciones auxiliares a la estructura de vértices para poder llenar los datos correctos. Como los datos por vértice están en un solo arreglo, solo se necesita contar con un “binding” (enlazamiento). El parámetro binding especifica el índice del enlazamiento en el arreglo de enlazamientos. El parámetro “stride” indica el número de bytes de una entrada a la otra mientras que “inputRate” puede tener uno de los siguientes valores:

- VK_VERTEX_INPUT_RATE_VERTEX: Avanzar a la siguiente entrada después de cada vertice.
- VK_VERTEX_INPUT_RATE_INSTANCE: Avanza después de cada instancia.

En la segunda estructura, binding indica de dónde vienen los datos por vértice. El parámetro “location” es una referencia a la directiva del mismo nombre de la entrada en el shader de vértices. La entrada en este shader, de ubicación 0, es la posición cuyos componentes son de 32 bits. El miembro “format” describe el tipo de datos para el atributo. Estos formatos se especifican utilizando la misma enumeración que los formatos de color. Los siguientes tipos de shaders y formatos se utilizan comúnmente juntos:

- float: VK_FORMAT_R32_SFLOAT
- vec2: VK_FORMAT_R32G32_SFLOAT
- vec3: VK_FORMAT_R32G32B32_SFLOAT
- vec4: VK_FORMAT_R32G32B32A32_SFLOAT

Se recomienda usar el formato donde la cantidad de canales de color es igual coincide con el numero de componentes en el tipo de dato del shader. Si hay más canales que componentes, se descartan los que sobran; pero si los componentes sobrepasan los canales, los componentes BGA utilizan valores predeterminados (0,0,1). No obstante, el tipo de color (SFLOAT,

UINT, SINT) y anchura del bit tambien deberia coincidir con el el sipo de entrada del shader. Ej:

- ivec2: VK_FORMAT_R32G32_SINT, vector de dos componentes de enteros de 32 bits con firma (Signed).
- uvec4: VK_FORMAT_R32G32B32A32_UINT, vector de cuatro componentes enteros de 32 bits sin firma (Unsigned).
- Double: VK_FORMAT_R64_SFLOAT, flotante de doble precisión de 64 bits.

Este parámetro también define implícitamente el tamaño en bytes de los datos de los atributos. Por otro lado, “offset” especifica el número de bytes desde el inicio de los datos por vértice a leerse. El enlazamiento está cargando una estructura (Vertex) a la vez y el atributo de posición está en un offset de cero bytes desde el principio de la estructura. Esto se calcula con el macro “offsetof”. La estructura para el color. Se define similarmente.

El último paso es configurar el pipeline de gráficos para aceptar los datos de vértices en este formato haciendo referencias a estas nuevas estructuras durante su creación. Se modifica la estructura VertexInputInfo para que reciba como referencia estas descripciones. Los cambios hechos al pipeline, al vertex shader, y la creación de la estructura de vértices junto con la especificación del arreglo de vértices se muestra en la Figura 46.

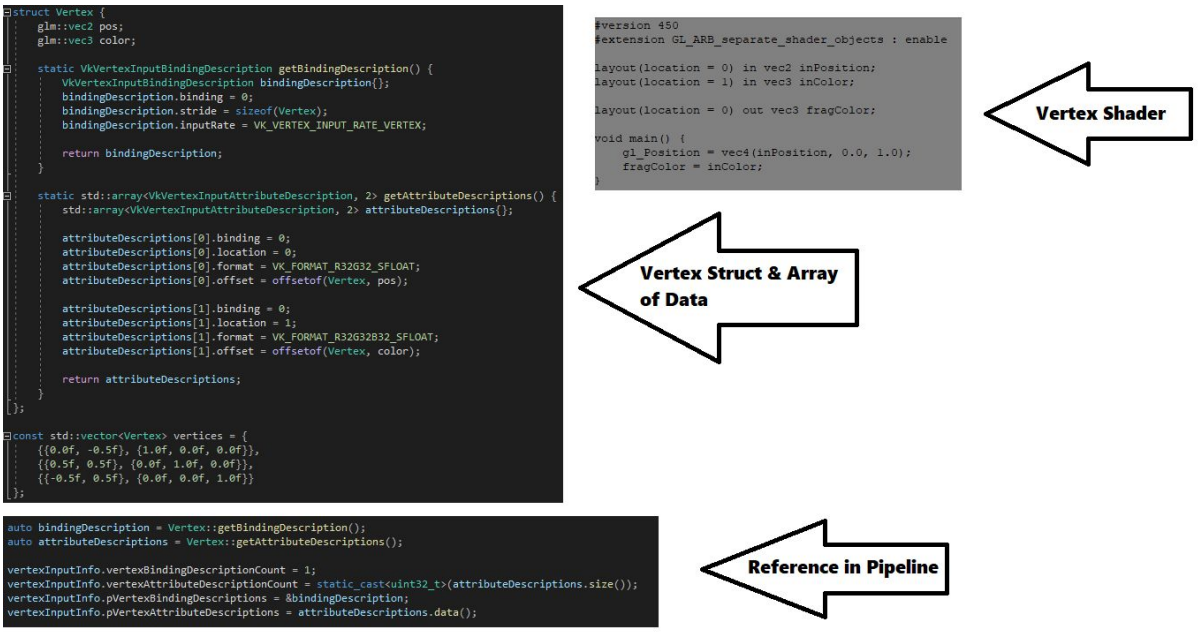


Figura 46. Shader de vértices con variables de atributos de vértices, estructura de vértices con descripciones de atributos y enlaces, y sus referencias en el pipeline gráfico

4.2. Creación de búfer de vértices:

Se explica que en Vulkan, los búferes son regiones de memoria que se usan para almacenar datos arbitrarios que se pueden leer desde el GPU. Para este punto, se van a almacenar los datos de los vértices. Parte de este proceso será asignar memoria para los búferes. Se empieza entonces agregando una función para la creación de estos búferes en la inicialización, justo antes de crear los búferes de comandos. Como siempre, se agrega una estructura, en este caso de tipo "VkBufferCreateInfo". Su campo "size" indica el tamaño del búfer en bytes el cual se calcula con "sizeof". Su segundo campo, "usage", define los propósitos para los que se van a usar los datos en el búfer. Como nota, se pueden indicar múltiples propósitos con un operador bitwise "or". El siguiente miembro, "sharingMode", define si el búfer formará parte de una famila de colas especifica o si se comparte entre varias al mismo tiempo. En este caso sólo se usará por la cola de gráficos por lo que el acceso es

exclusivo. Finalmente, el parámetro “flags” se usa para configurar la memoria de búfer escasa pero se define como 0 ya que no es de relevancia para los propósitos de la aplicación actual. Haciendo lo usual, se crea un miembro “VkBuffer” y su función para llamarlo, “vkCreateBuffer”; y se limpia con “vkDestroyBuffer” en la función de limpieza.

Ahora que se creó el búfer, se le puede asignar memoria. Primero se obtienen los requisitos de memoria con “vkGetBufferMemoryRequirements” que recibe como argumentos el dispositivo lógico, el búfer de vértices y la estructura “VkMemoryRequirements”. Esta estructura tiene tres campos:

- “size”: tamaño de memoria requerida en bytes.
- “alignment”: offset en bytes donde el búfer comienza en la región de memoria asignada.
- “memoryTypeBits”: Campo de bits de los tipos de memoria que son adecuados para el búfer.

Las tarjetas gráficas ofrecen distintos tipos de memoria de dónde asignar. Estas varían en operaciones permitidas y características de rendimiento. Para encontrar los requerimientos adecuados para el búfer y la aplicación se usa entonces la función “findMemoryType”. Dentro de esta función se buscan los tipos de memoria disponibles mediante ‘vkGetPhysicalDeviceMemoryProperties” que recibe como argumento el dispositivo físico y una nueva estructura: “VkPhysicalDeviceMemoryProperties”. Esta estructura cuenta con dos arreglos:

- “memoryTypes”. Arreglo que consiste de estructuras VkMemoryType que especifican el “heap” y sus propiedades para cada tipo de memoria. Estas propiedades definen características especiales, como mapear la memoria para

escribirle desde el CPU (propiedad indicada por "VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT").

- "memoryHeaps": recursos de memoria dedicados, como VRAM, que intercambian espacio en la memoria RAM cuando se agota la memoria VRAM.

Se crea entonces un ciclo "for" para encontrar la memoria adecuada para el búfer. Aquí es donde entra el parámetro "typeFilter" para especificar el campo de bits de tipos de memoria. Se encuentra entonces el índice de un tipo de memoria adecuado iterando sobre estos y revisando si el bit correspondiente es 1. También se confirma que la propiedad VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT sea compatible. También se revisa que el resultado del bitwise AND sea igual al campo de bits de propiedades deseadas.

Ahora que se tienen los requerimientos de memoria, se puede determinar el tipo de memoria para poder asignarla. Esto se logra con la estructura "VkMemoryAllocateInfo". Sus miembros se obtienen de los requerimientos de memoria. Nótese que se incluyen dos propiedades, la segunda siendo VK_MEMORY_PROPERTY_HOST_COHERENT_BIT. Luego se crea un miembro de la clase para almacenar el "handle" a la memoria y asignarla con la función vkAllocateMemory. Finalmente se asocia esta memoria al búfer utilizando "vkBindBufferMemory". El cuarto parámetro de esta función se refiere al offset dentro de la región de memoria. Como la memoria se asignó específicamente para el búfer de vértices, entonces este parámetro se define como 0. Si fuera un valor distinto de cero, tendría que ser divisible entre "memRequirements.alignment". La memoria enlazada al búfer se limpia una vez que no es utilizada, por lo que se debe destruir después del búfer en la función de limpieza.

Ulteriormente, se deben copiar los datos de vértices a los búferes. Se logra mapeando la memoria del búfer de vértices a la memoria accesible por el GPU utilizando "vkMapMemory". Esta función permite acceder una región de memoria específica que se define por un offset y un tamaño. En este caso el offset es 0 y el tamaño se deriva del mismo búfer. Para mapear toda la memoria se puede utilizar "VK_WHOLE_SIZE". El penúltimo parámetro sirve para especificar banderas y el último parámetro indica la salida para el apuntador a la memoria mapeada. Luego se utiliza "memcpy" para copiar los datos de los vértices a la memoria mapeada y desmapear usando "vkUnmapMemory". Es posible que el driver no copie inmediatamente los datos a la memoria del búfer de vértices por "caching". Asimismo, es posible que lo que se escribe al búfer no sea visible a la memoria mapeada. Para resolver estos inconvenientes hay dos soluciones:

- Usar un heap de memoria que es coherente al anfitrión. Esta opción afecta el rendimiento.
- Llamar "vkFlushMappedMemoryRanges" después de escribir a la memoria mapeada, y llamar "vkInvalidateMappedMemoryRanges" antes de leer de la memoria mapeada.

Se usó la primera opción en este caso con VK_MEMORY_PROPERTY_HOST_COHERENT_BIT. Esto sirvió para asegurarse de que la memoria mapeada siempre coincida con el contenido de la memoria asignada. Cualquier opción hace que el driver esté consciente de las escrituras al búfer, pero no implica que sean visibles para el GPU. Sin embargo, esto sucede en el fondo y la especificación de Vulkan promete que la transferencia de datos al GPU se completa en la siguiente llamada a vkQueueSubmit (comando de envío de colas, llamado en la función de dibujo).

Finalmente se deben enlazar los búferes de vértices durante las operaciones de renderización. Entonces se extiende la función createCommandBuffers, agregando primero la función vkCmdBindVertexBuffers para enlazar los búferes de vértices a los enlaces. El primer y tercer parámetro especifica el offset y el número de enlaces para los que se van a especificar los búferes de vértices. Los últimos dos parámetros especifican el arreglo de búferes de vértices para enlazar y los offsets en bytes de donde empezar a leer datos de vértices. Finalmente se modifica la llamada a vkCmdDraw para tomar como argumento el número de vértices en el búfer.

4.3. Staging Búfer

Son recursos intermedios o temporales que se utilizan para transferir los datos de una aplicación. El tipo de memoria utilizado por el CPU para acceder al búfer de vértices no es el más óptimo para leer desde el GPU. El tipo de memoria más óptima tiene la bandera "VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT", pero normalmente no es accesible por el CPU en tarjetas gráficas dedicadas. Para esto, es necesario crear dos búfer de vértices. El primero, denominado como "staging buffer", en memoria accesible por el CPU para cargar los datos del arreglo de vértices, y el búfer de vértices final en la memoria local del dispositivo. Luego se utilizará un comando de copia de búferes para mover los datos del staging buffer al búfer de vértices.

Se comienza haciendo una cola de transferencia. El comando de copia de búferes requiere una familia de colas compatible con operaciones de transferencia, indicado por la bandera "VK_QUEUE_TRANFER_BIT". Asimismo, cualquier familia de colas con "VK_QUEUE_GRAPHICS_BIT" o "VK_QUEUE_COMPUTE_BIT" es compatible con operaciones de transferencia.

Como se van a crear varios búferes, se mueve el código de creación del búfer de vértices (excepto la parte del mapeo) a una nueva función general de creación de búfer. Esta debe contar con parámetros para el tamaño del búfer, las propiedades de la memoria y su uso para poder crear distintos tipos de búferes. Los últimos dos parámetros son variables de salida a las que se les van a escribir los handles. En la función de creación de búfer de vértices se puede simplemente llamar esta función. Ambas funciones deberían asemejarse al código de la Figura 47.

```
void createVertexBuffer() {
    VkDeviceSize bufferSize = sizeof(vertices[0]) * vertices.size();
    createBuffer(bufferSize, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, vertexBuffer, vertexBufferMemory);

    void* data;
    vkMapMemory(device, vertexBufferMemory, 0, bufferSize, 0, &data);
    memcpy(data, vertices.data(), (size_t)bufferSize);
    vkUnmapMemory(device, vertexBufferMemory);
}

void createBuffer(VkDeviceSize size, VkBufferUsageFlags usage, VkMemoryPropertyFlags properties, VkBuffer& buffer, VkDeviceMemory& bufferMemory) {
    VkBufferCreateInfo bufferInfo{};
    bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    bufferInfo.size = size;
    bufferInfo.usage = usage;
    bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

    if (vkCreateBuffer(device, &bufferInfo, nullptr, &buffer) != VK_SUCCESS) {
        throw std::runtime_error("failed to create buffer!");
    }

    VkMemoryRequirements memRequirements;
    vkGetBufferMemoryRequirements(device, buffer, &memRequirements);

    VkMemoryAllocateInfo allocInfo{};
    allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    allocInfo.allocationSize = memRequirements.size;
    allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, properties);

    if (vkAllocateMemory(device, &allocInfo, nullptr, &bufferMemory) != VK_SUCCESS) {
        throw std::runtime_error("failed to allocate buffer memory!");
    }

    vkBindBufferMemory(device, buffer, bufferMemory, 0);
}
```

Figura 47. Función de creación de búferes y búferes de vértices

Posteriormente se tiene que modificar el búfer de creación de vértices para que solo utilice un búfer visible por el anfitrión como búfer temporal y también usar búfer del dispositivo local como el búfer de vértices real. Entonces se agrega un VkBuffer (llamado en el tutorial "stagingBuffer") y un VkDeviceMemory (llamado "stagingBufferMemory") para mapear y copiar los datos de los vértices. La función createVertexBuffer ahora tiene dos llamadas a la función de creación de búfer (la primera para el staging buffer y la segunda para el de vértices) que utilizan dos nuevas banderas separadamente:

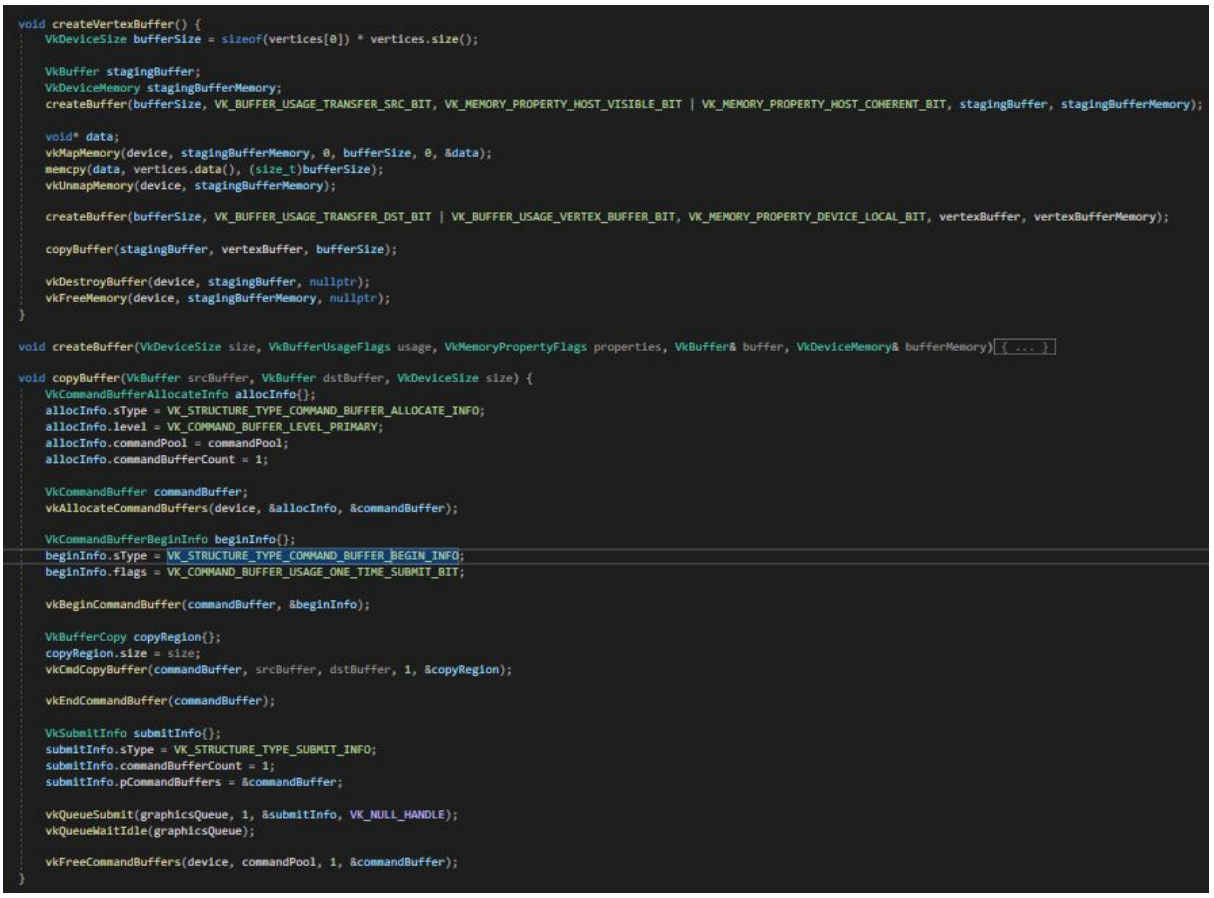
- VK_BUFFER_USAGE_TRANSFER_SRC_BIT: Se puede usar el búfer como una fuente en las operaciones de transferencia de memoria.
- VK_BUFFER_USAGE_TRANSFER_DST_BIT: Se puede usar el búfer como un destino en las operaciones de transferencia de memoria.

Ahora el búfer de vértices se asigna desde un tipo de memoria local al dispositivo por lo que no se puede usar vkMapMemory. Entonces, se copian los datos del staging búfer al búfer de vértices. Esto se hace especificando las banderas de fuente en la función de creación del staging buffer y de destino en la creación de búfer de vértices.

Se escribe entonces una función para copiar el contenido de un búfer al otro ("copyBuffer"). Como las operaciones de transferencia utilizan búferes de comandos, primero se debe asignar y comenzar a registrar un búfer de comandos temporal. Se puede usar "VK_COMMAND_BUFFER_USAGE_ONE_TIME_BIT" para usar el búfer de comandos una vez y esperar a que se terminen las operaciones de copiado antes de regresar de la función. Luego se hace una estructura de tipo "VkBufferCopy" para poder utilizar "vkCmdCopyBuffer". Este comando toma el búfer fuente y destino como argumentos y arreglo de regiones que se definen por la estructura citada cuyos miembros son el offset del búfer fuente y destino, y el tamaño. Como este búfer de comandos solo tiene el comando de copia, puede dejar de registrarse inmediatamente. Luego se ejecuta este búfer de comandos para completar la transferencia usando "VkSubmitInfo" y "vkQueueSubmit". Para esperar a que se complete la transferencia se puede usar "vkQueueWaitIdle". Luego se limpia el búfer de comandos para esta operación.

Por último, se llama la función de copia de búfer desde la creación del búfer de vértices para transferir los datos de vértices al búfer del dispositivo local. Luego copiar el búfer se debe

limpiar. La función de creación del búfer de vertices y copia de búferes de muestra en la Figura 48.



```
void createVertexBuffer() {
    VkDeviceSize bufferSize = sizeof(vertices[0]) * vertices.size();

    VkBuffer stagingBuffer;
    VkDeviceMemory stagingBufferMemory;
    createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT, VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT, stagingBuffer, stagingBufferMemory);

    void* data;
    vkMapMemory(device, stagingBufferMemory, 0, bufferSize, 0, &data);
    memcpy(data, vertices.data(), (size_t)bufferSize);
    vkUnmapMemory(device, stagingBufferMemory);

    createBuffer(bufferSize, VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, vertexBuffer, vertexBufferMemory);

    copyBuffer(stagingBuffer, vertexBuffer, bufferSize);

    vkDestroyBuffer(device, stagingBuffer, nullptr);
    vkFreeMemory(device, stagingBufferMemory, nullptr);
}

void createBuffer(VkDeviceSize size, VkBufferUsageFlags usage, VkMemoryPropertyFlags properties, VkBuffer& buffer, VkDeviceMemory& bufferMemory){ ... }

void copyBuffer(VkBuffer srcBuffer, VkBuffer dstBuffer, VkDeviceSize size) {
    VkCommandBufferAllocateInfo allocInfo{};
    allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
    allocInfo.commandPool = commandPool;
    allocInfo.commandBufferCount = 1;

    VkCommandBuffer commandBuffer;
    vkAllocateCommandBuffers(device, &allocInfo, &commandBuffer);

    VkCommandBufferBeginInfo beginInfo{};
    beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    beginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;

    vkBeginCommandBuffer(commandBuffer, &beginInfo);

    VkBufferCopy copyRegion{};
    copyRegion.size = size;
    vkCmdCopyBuffer(commandBuffer, srcBuffer, dstBuffer, 1, &copyRegion);

    vkEndCommandBuffer(commandBuffer);

    VkSubmitInfo submitInfo{};
    submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
    submitInfo.commandBufferCount = 1;
    submitInfo.pCommandBuffers = &commandBuffer;

    vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE);
    vkQueueWaitIdle(graphicsQueue);

    vkFreeCommandBuffers(device, commandPool, 1, &commandBuffer);
}
```

Figura 48. Creación de búfer de vértices y copia de búferes

4.4. Búfer de índices:

Este búfer es una matriz de apuntadores al búfer de vértices que permite reordenar los datos de los vértices, y reutilizar datos existentes para múltiples vértices. Una manera simple de visualizar esto es considerando que para formar un rectángulo, se necesitan dos triángulos rectángulos idénticos en donde dos de sus vértices coinciden. Los datos de estos vértices se ven duplicados y por lo que la mitad son redundantes.

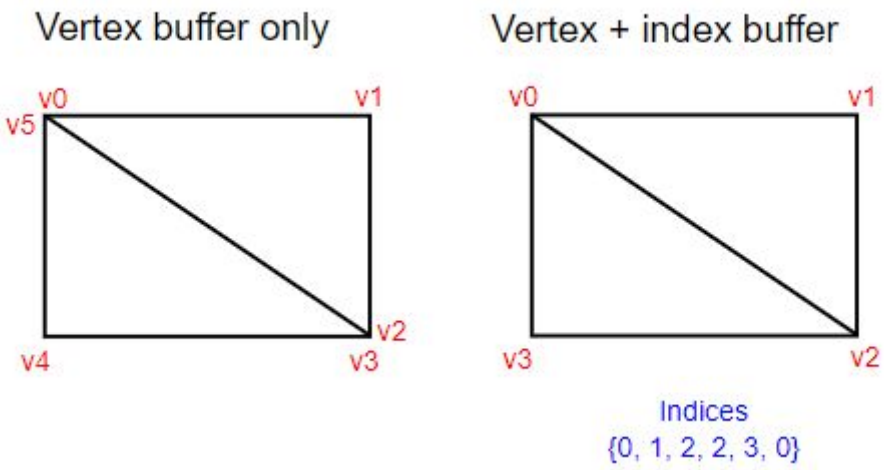


Figura 49. Distinción visual entre búfer de vértices y búfer de índices (Overvoorde, A., s.f.)

En esta sección se modifican los datos de vértices y se agregan datos de índices para dibujar un rectángulo. En el vector de vértices, se agrega un renglón más con la posición y color del cuarto vértice. Luego se agrega un nuevo vector llamado "indices" para representar el contenido del búfer de índices de forma que coincida con la Figura 49. El vector puede consistir de "uint16_t" o "uint32_t" pero como se están usando menos de 65535 vértices unidos, la primera opción funciona.

Al igual que los datos de vértices, los índices necesitan ser cargados a un búfer para que los pueda acceder el GPU. Entonces se agregan dos nuevos miembros de la clase para sostener los recursos del búfer de índices y se crea una función para su creación (muy similar a la función de creación de búfer de vértices). La diferencia entre este búfer y el búfer de vértices está en el tamaño, ya que este es igual al número de índices por el tamaño del tipo de índices (uint16_t). Asimismo, se utiliza la bandera "VK_BUFFER_USAGE_INDEX_BUFFER_BIT" en lugar de

VK_BUFFER_USAGE_VERTEX_BUFFER_BIT. Este búfer también se debe limpiar al final del programa.

Para usar este búfer, se debe modificar la función de creación de búferes de comandos. Primero se debe enlazar el búfer pero a diferencia del búfer de vértices, es que solo se puede tener uno solo búfer de índices. Como no se pueden usar diferentes índices para cada atributo de vértices, se tienen que duplicar los datos de los vértices aunque solo un atributo varíe. El búfer se enlaza con "vkCmdBindIndexBuffers" que recibe el búfer de índices, un offset en bytes y el tipo de datos de índices. Finalmente se cambia el comando de dibujo por "vkCmdDrawIndexed". El segundo y tercer parámetro de la función especifican el número de índices y el número de instancias, respectivamente. Como no se está usando instanciación, el valor del tercer parámetro solo es uno. El número de índices representan el número de vértices que se envían al búfer de vértices. El cuarto parámetro especifica un offset en el búfer de índices, el quinto parámetro indica un offset para agregar a los índices en el búfer y el último un offset para instanciación.

5. Buffers Uniformes:

Ahora ya es posible avanzar a implementar las matrices de proyección, modelo y vista para poder graficar objetos tridimensionales.

Para hacer esto posible se implementan *descriptores de recursos* los cuales permiten que los shaders puedan acceder a recursos como buffers o imágenes libremente. El primer paso es preparar un búfer que contenga la información de estas matrices y luego hacer que el shader acceda a estas matrices por medio del descriptor. El uso del descriptor consiste en las siguientes partes:

- Especificar el diseño del descriptor durante la creación del pipeline.
- Asignar un conjunto de descriptores desde un grupo de descriptores.
- Enlazar el conjunto de descriptores durante la renderización.

El diseño del descriptor especifica los tipos de recursos que van a ser accedidos por el pipeline. El conjunto de descriptores especifica los recursos de imagen o búfer a vincular con los descriptores. El conjunto de descriptores está vinculado a los comandos de dibujo al igual que los búferes de vértices y el búfer de frames.

5.1. Uniform Buffer Objects:

Uniform Buffer Object (UBO) es el tipo de descriptor básico que se utilizará inicialmente y el cual se definirá como una estructura la cual tendrá las matrices de modelo, vista y proyección. Los datos se copian a un búfer de Vulkan y se acceden mediante el descriptor UBO desde el shader de vértices. Asimismo, se actualizarán, dentro del vertex shader, cada una de las matrices para rotar el rectángulo y comprobar que las transformaciones están funcionando correctamente. La siguiente Figura demuestra como utilizar los miembros del UBO dentro del shader de vértices:

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(binding = 0) uniform UniformBufferObject {
    mat4 model;
    mat4 view;
    mat4 proj;
} ubo;

layout(location = 0) in vec2 inPosition;
layout(location = 1) in vec3 inColor;

layout(location = 0) out vec3 fragColor;

void main() {
    gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 0.0, 1.0);
    fragColor = inColor;
}
```

Figura 50. Integrando UBO en Vertex Shader

El orden en el que se declara *in, out y uniform* no impacta la funcionalidad del shader y la directiva *binding* (vinculación) es similar a la directiva *location* para atributos. Esta directiva de vinculación es la que se va a referenciar en el diseño del descriptor. Como se puede observar se utilizan las transformaciones del UBO para asignar la posición final de los vértices. A diferencia de los triángulos en 2D, el último componente de las coordenadas del clip puede no ser 1, lo que resultará en una división cuando se convierta en las coordenadas finales normalizadas del dispositivo en la pantalla. Esto se usa en la proyección en perspectiva como la *división en perspectiva* y sirve para que los objetos más cercanos se vean más grandes que los más lejanos.

Podemos coincidir exactamente con la definición en el shader utilizando tipos de datos en GLM. Los datos en las matrices son compatibles binariamente con la forma en que el shader lo espera, por lo se puede copiar el UBO a VkBuffer. Al igual que con los atributos de los vértices y su índice de ubicación, se tiene que detallar cada vinculación del descriptor utilizado por los shaders para la creación del pipeline. Entonces, se crean una función para

definir la información necesaria del descriptor. Esta función se debe llamar antes de la creación del pipeline ya que va ahora va a ser un requisito de este. Dentro de esta función para crear el descriptor, se debe incluir una estructura de Vulkan para describir conjuntos de descriptores que especifican la vinculación usada en el shader y el tipo de descriptor (en este caso UBO). Otro miembro de esta estructura especifica el número de valores en el arreglo y que funciona para especificar las transformaciones de uno o más objetos, en este caso solo se tiene un objeto por lo que se le asigna el valor de 1. Khronos pone como ejemplo que este miembro se puede utilizar para animar cada hueso de un esqueleto. Luego de esto, se especifica en qué etapas del shader se va a referenciar el descriptor. Se define un nuevo miembro en la clase principal utilizando un tipo nativo de Vulkan que contiene todos las vinculaciones de descriptor y otro crearlo. Finalmente, se modifica el objeto de creación del pipeline para especificar durante su creación que descriptores se van a utilizar por los shaders y luego se destruye en la función de limpieza.

5.2. Creación del búfer uniforme:

Este búfer es el que contendrá los datos del UBO para el shader. Se van a copiar nuevos datos a este búfer cada frame por lo que sería redundante tener un búfer provisional ya que probablemente perjudica el rendimiento del programa. Se deben tener múltiples buffers para evitar actualizar un búfer en preparación del siguiente frame mientras uno previo sigue leyendo de el. Lo más conveniente es tener un búfer uniforme por imagen de cadena de intercambio. Se crean entonces dos nuevos miembros de tipo VkBuffer y VkDeviceMemory, y una función subsecuente a la creación del búfer de índices que asigna estos nuevos buffers. Luego, se crea una nueva función que actualiza cada cuadro con una nueva transformación. Los datos uniformes se usarán para cada llamada de dibujo, por lo que el búfer que los contiene solo se destruirá en cuanto se termine de renderizar. Ya que depende del número de

imágenes en la cadena de intercambio, la cual cambia después de una recreación, se debe limpiar el búfer uniforme en la función de limpieza de la cadena de intercambio. Consecuentemente, se debe recrear (colocar) la función de creación del búfer uniforme en la cadena de intercambio.

5.3. Actualizar datos uniformes:

Se crea una función para actualizar los datos del búfer uniforme y se llama desde la función para dibujar cuadros. Esta función servirá para generar nuevas transformaciones cada marco y utilizar los encabezados de modelo, vista y proyección de GLM que se descargaron para implementar su funcionalidad. No obstante se utiliza la librería estándar *chrono* para controlar en segundos en lugar de cuadros, la velocidad en la que se aplican las transformaciones. Finalmente se definen las transformaciones y se copian los datos desde el objeto búfer uniforme al búfer uniforme actual.

5.4. Grupo descriptor:

Esto se usa debido a que los conjuntos de descriptores no se pueden crear de manera directa, sino que tienen que asignarse desde un grupo, como los buffers de comando. Entonces, se crea una función de creación de grupo descriptor en donde se describen cuantos y que tipos de de descriptores que los conjuntos contendrán. Cada uno de los descriptores se debe asignar por cuadro y se debe especificar el máximo número de conjuntos de descriptores y así como descriptores originales que se pueden asignar. El resto de la implementación es similar a las funciones anteriores en donde se debe declarar un miembro cuyo tipo es existente y crear, destruir y recrear las funciones que contienen como argumento este miembro. Al hacer esto, se pueden asignar y utilizar los grupos de descriptores

6. Texture Mapping:

Esto permite agregar una textura a la aplicación lo cual involucra crear un objeto de imagen respaldado por la memoria del dispositivo, llenar este objeto de pixeles desde un archivo de imagen, crear una muestra de imagen, y agregar un descriptor combinado de muestra de imagen para muestrear colores de la textura.

6.1. Imagenes:

El primer paso, para facilitar el proceso de cargar una imagen, en lugar de escribir código para realizar esta tarea, se puede simplemente importar una librería que ya hace esto posible. Específicamente, la librería *stb_image la* cual forma parte de la colección stb se puede descargar para realizar este proceso. Se sugiere agregar esta carpeta en la misma sección en la que se han agregado las demás librerías. Esta se agrega a la sección de inclusión de directorios adicionales en la parte general de C/C++ en la configuración del proyecto y se incluye el encabezado de la librería. Posteriormente, se crea una función de para crear texturas y se llama en la función de inicializacion. Asimismo, se debe descargar una imagen para importar. Lo sugerido por Khronos es crear una carpeta llamada texturas en el mismo directorio en donde se encuentra la carpeta de shaders y almacenar la textura en esta. Se crea un búfer en la memoria visible para asignar un objeto de memoria al espacio de direcciones de la aplicación y así poder copiar píxeles a este buffers. Luego se crea un objeto de imagen para acceder a los valores (colores) de los pixeles en los buffers usando coordenadas bidimensionales.

6.2. Vista de imagen y muestreo:

Se crean dos recursos necesarios para el pipeline gráfico para poder muestrear una imagen. La primera es la vista de imagen de textura la cual requiere su función de creación para llamarse en la inicialización del programa y un manejador del objeto de tipo *VkImageView.*

El muestreo de textura es el segundo recurso necesario. Esto permitirá aplicar filtros para cuando se está leyendo el color de una textura para enfrentar problemas como sobremuestreo, submuestreo (resuelto por medio de muestreo anisótropo), y la aplicación de transformaciones.

6.3. Combinación de imágenes de muestreo:

La combinación de imágenes de muestreo es un descriptor que hace posible que los shaders accedan a la imagen por medio de un objeto de muestreo. Se debe entonces actualizar la función de diseño de asignación de conjuntos de descriptores. Seguido de esto, se modifica la estructura de vértices para incluir un vector bidimensional para las coordenadas de la textura y poder acceder las coordenadas como una entrada del shader de vértices. Finalmente se modifican los shaders de fragmentación y vértices para poder muestrear los colores de la textura.

7. Buffering de Profundidad:

En esta sección se explora cómo agregar una tercera coordenada (Z) para poder comenzar a trabajar con objetos tridimensionales. Los pasos son los siguientes:

- Cambiar la estructura Vertex para que utilice un vector 3D para la posición.
- Actualizar el miembro *format* de la estructura que especifica la descripción de cada atributo de entrada de vértice (*VkVertexInputAttributeDescription*).
- Actualizar el shader de vértices para que acepte y transforme coordenadas tridimensionales como entrada.
- Actualizar el contenedor (vector) de vértices para incluir las coordenadas en el eje z.

Si se duplican los vértices en su contenedor y se ajustan las coordenadas en Z a un valor negativo, los fragmentos del cuadro inferior se terminan dibujando sobre los del cuadro superior. Esto sucede debido a que el cuadro inferior se lee posteriormente al superior en el arreglo de índices. La manera tradicional de resolver esto es utilizando un buffers de profundidad (*depth buffer*). Este búfer almacena la profundidad para cada posición. Cuando el rasterizador produce un nuevo fragmento, se realiza una prueba de profundidad la cual para comprobar el nuevo fragmento se encuentra más cerca o más lejos que el anterior. Si el nuevo fragmento está más lejos, entonces se descarta, de lo contrario, escribe su profundidad en el buffers. La siguiente secuencia de pasos es la que Khronos explica para cumplir esto:

- Utilizar la definición *GLM_FORCE_DEPTH_ZERO_TO_ONE*. Se requiere gracias a que la proyección en perspectiva de GLM utiliza el rango de profundidad -1.0 a 1.0 de OpenGL, mientras que Vulkan utiliza un rango de 0.0 a 1.0.
- Para implementar la imagen de profundidad, se necesitan declarar los recursos de tipo imagen, vista y memoria junto con una función para su creación y configuración (llamada en la función de inicialización de Vulkan).
- Se crea una función para encontrar el formato compatible con la imagen de profundidad.Esta función tendrá un ciclo for para iterar sobre los formatos candidatos utilizando la función *vkGetPhysicalDeviceFormatProperties* y una estructura con las posibles propiedades de cada formato que son:
    - *linearTilingFeatures:* casos que son compatibles con un embaldosado lineal
    - *optimalTilingFeatures:* casos que son compatibles con un embaldosado óptimo
    - *bufferFeatures:* casos que son compatibles con buffers. (esta no es relevante para este objetivo).

- Posteriormente se utiliza una sentencia condicional *if* para checar el embaldosado y determinar un formato. Si no se encuentra ningún formato deseado, entonces se arroja un error.
- Esta función se utiliza para crear una función auxiliar para seleccionar un formato con un componente de profundidad compatible con el uso de la profundidad adjunta.
- Se agrega a esta función auxiliar otra función auxiliar que devuelve un booleano para determinar si el formato de profundidad elegido contiene un componente de estencil.
- La función utilizada para encontrar un formato de profundidad se llama desde la función de creación de recursos de profundidad.
- Se invocan las funciones auxiliares para crear imagen y crear vista de imagen.
- Se actualiza la función de creación de vista de imagen para convertir el sub recurso *VK_IMAGE_ASPECT_COLOR_BIT* en un parámetro y se actualizan todas las llamadas a esta función para utilizar el aspecto correcto.
- Se actualiza la estructura *VkRenderPassCreateInfo* al agregar una referencias para dos nuevas estructuras, una de descripción (*VkAttachmentDescription*) y una de referencia (VkAttachmentReference) para la profundidad. Asimismo, se agrega una referencia a la estructura de referencia de profundidad desde la estructura *VkSubpassDescription*.
- Luego se modifica la creación del framebuffer para enlazar la imagen de profundidad con la adjunción de profundidad desde la función de *createFramebuffers.* Para esto se especifica el miembro de vista de imagen de profundidad como un segundo elemento adjunto. Luego se mueve la llamada a esta función después de la función de creación de recursos de profundidad.
- Se especifican valores de limpieza creando un arreglo de estructuras *VkClearValues* en la función de creación de buffers de comando en donde el orden es idéntico al de

las adjunciones y el rango de profundidades en el buffers de profundidad está entre 1.0 (plano lejano) y 0.0 (plano cercano)

- Se configura el adjunto de profundidad dentro del pipeline gráfico mediante la estructura de especificación del estado de stencil *VkPipelineDepthStencilStateCreateInfo.*
- Se actualiza la estructura *VkGraphicsPipelineCreateInfo* para que haga referencia al estado del stencil.

Finalmente para emparejar la resolución del buffers de profundidad con los adjuntos de de color nuevos al efectuar cambios en las dimensiones de la ventana, se incluye la función de creación de recursos de profundidad en la función de recreación del swap chain y consecuentemente se agregan funciones de limpieza para la vista de profundidad de la imagen, la profundidad de la imagen y la memoria de la profundidad de la imagen mediante *vkDestroyImageView, vkDestroyImage* y *vkFreeMemory*. .

8. Cargar Modelos:

Debido a la facilidad de integrar, se utiliza la librería *tinyobjloader* (descargable desde https://github.com/tinyobjloader/tinyobjloader) para descargar los vértices y caras de archivos OBJ. Los pasos para integrar esta librería al proyecto de visual studio son los mismos que para stb_image.h. Debido a que todavía no se tiene un sistema de iluminación integrado al proyecto, es necesario importar modelos con la iluminación cocinada en ellos los cuales se pueden encontrar en Sketchfab (https://sketchfab.com/). Luego se crea un nuevo directorio llamado *models* dentro del mismo archivo donde se encuentran los archivos de shaders y texturas. El obj se incluye dentro de este nuevo directorio y su imagen correspondiente dentro del directorio de texturas. Subsecuentemente:

- Se agregan las nuevas configuraciones de variables en el programa para definir las rutas de acceso al modelo y a su textura.
- Actualizar la función *createTextureImage* para utilizar estas variables.
- Eliminar los arreglos de vértices e índices globales y reemplazarlos con contenedores no constantes como miembros de la clase.
- Debido al incremento en vértices, cambiar el tipo de índices *u_int32_t* y, consecuentemente, cambiar el parámetro correspondiente en *vkCmdBindIndexBuffer*.
- Incluir la nueva librería definiendo primero *TINYOBJLOADER_IMPLEMENTATION*.
- Crear una función de carga de modelos en la función de inicialización de Vulkan y llamarla antes de las funciones de creación de buffers de vértices e índices.
- Cargar modelos a las estructuras de datos de la librería mediante *tinyobj::LoadObj*.
- Utilizar un ciclo *for* para iterar sobre todas las figuras y llenar el vector de vértices e índices.
- Utilizar los índices para obtener los atributos de los vértices en los arreglos de atributos.
- Invertir el componente vertical de las coordenadas de texturas.
- Utilizar <*map*> o <*unordered_map*> para rastrear y evitar la duplicación de vértices utilizando el buffers de índices.
- Sobrecargar el operador de comparación de igualdad en la estructura de vértices para incluir la estructura de vértices como llave en la tabla hash.
- Implementar una función hash para la estructura de Vértices especificando una plantilla especializada para *std::hash<T>*. Esta función se debe posicionar fuera de la

estructura de vértices y se debe incluir el encabezado de GLM *<glm/gtx/hash.hpp>* definiendo previamente *GLM_ENABLE_EXPERIMENTAL*.

9. Generar Mipmaps:

El Grupo Khronos describe este concepto sucintamente como: versiones precalculadas y reducidas de una imagen. Cada nueva imagen es la mitad del ancho y alto de la anterior. Mipmaps se utilizan como una forma de Nivel de detalle,o *Level of Detail (LOD)*. Los objetos que están lejos de la cámara tomarán muestras de sus texturas de las imágenes Mip más pequeñas. El uso de imágenes más pequeñas aumenta la velocidad de representación y evita artefactos como los patrones de Moiré.

9.1. Creación de Imagen:

Las imágenes mip se almacenan en diferentes niveles Mip del manejador de objetos de imagen de Vulkan (*VkImage*). El nivel de mip 0 es la imagen original, y los niveles de mip después del nivel 0 se conocen comúnmente como la cadena de mip. En Vulkan, se tiene que especificar en la creación de VkImage. El valor de los niveles mips se calcula por medio de las dimensiones de la imágen:

1. Se agrega un miembro de la clase de tipo *uint32_t*.
2. En la creación de textura de la imagen se le asigna a este miembro el número de los de niveles en la cadena mip igualandolo a la siguiente línea: *static_cast<uint32_t>(std::floor(std::log2(std::max(texWidth, texHeight)))) + 1*;

La función *max* selecciona la dimensión más grande. La función l*og2* calcula cuántas veces se puede dividir esa dimensión entre 2. La función *floor* maneja casos donde la dimensión más grande no es una potencia de 2. Se agrega 1 para que la imagen original tenga un nivel de mip. Para usar este valor, se tienen que agregar un parámetro que lo reciba dentro de las

funciones de creación de imagen, creación de vista de imágen y actualizar sus llamadas paras que se usen los valores correctos.

9.2. Generación de Mipmap:

El *staging buffer* solo se puede usar para llenar el nivel 0 de mip. Los otros niveles aún no están definidos. Para completar estos niveles se deben generar los datos desde el nivel único que se tiene mediante el comando *vkCmdBlitImage,* el cual realiza operaciones para copiar, escalar y filtrar una imagen. En este caso, se va a llamar esta función múltiples veces para enviar datos a cada nivel de nuestra imagen de textura. Este proceso se conoce como *blit,* que es copiar bits de una parte de la memoria gráfica de una computadora a otra parte. Sin embargo, como este comando se considera como una operación de transferencia por lo que se le tiene que especificar que la textura de la imagen se va a utilizar como la fuente y el destino de la transferencia. Se debe entonces, hacer lo siguiente:

- En los indicadores de uso de la imagen de textura en la creación de imagen de textura, agregar *VK_IMAGE_USAGE_TRANSFER_SRC_BIT*
- Dentro de la creación de textura de imagen, eliminar la transición existente a VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL

Posteriormente se crea la función de generación de mipmaps en la cual se utilizará nuevamente *VkImageMemoryBarrier* para poder realizar varias transiciones. También, se crea un loop que vaya de 1 a la cantidad de niveles en el mipmap para registrar cada comando *VkCmdBlitImage* y llenar los mapas mip de la imagen de textura.

9.3. Soporte de filtrado lineal:

En esta sección, se utiliza *vkGetPhysicalDeviceFormatProperties* para garantizar la función de construcción de mipmaps en todas las plataformas. Entonces se agrega un nuevo parámetro a la función de generación de mipmaps que especifique el formato de la imagen. Y

utilizando la función mencionada anteriormente (obtener propiedades de formato del dispositivo físico) y comprobar que el formato de la imagen sea compatible con blitting lineal.

9.4. Sampler:

En esta sección, se usa *VkSampler* para controlar cómo se leen los datos del mimap en *VkImage* mientras se procesan. Vulkan permite especificar el nivel mínimo de detalle, el nivel máximo de detalle, el sesgo en el nivel de detalle del mip y el modo del mipmap meditante *minLod, maxLod, mipLodBias* y *mipmapMode* respectivamente.

10. Multisampling:

Esta sección cubre cómo implementar el método de *Multisample anti-aliasing.* Esta es una de las varias técnicas existentes para combatir el problema de *aliasing* el cual surge cuando hay un número limitado de píxeles para renderizar, causando un patrón dentado. Para el proceso se hace lo siguiente:

- Determinar cuántas muestras puede usar el hardware.
- Configurar el objetivo de render

Al compilar el proyecto, el resultado en pantalla debería ser similar a lo que se demuestra en la siguiente Figura:

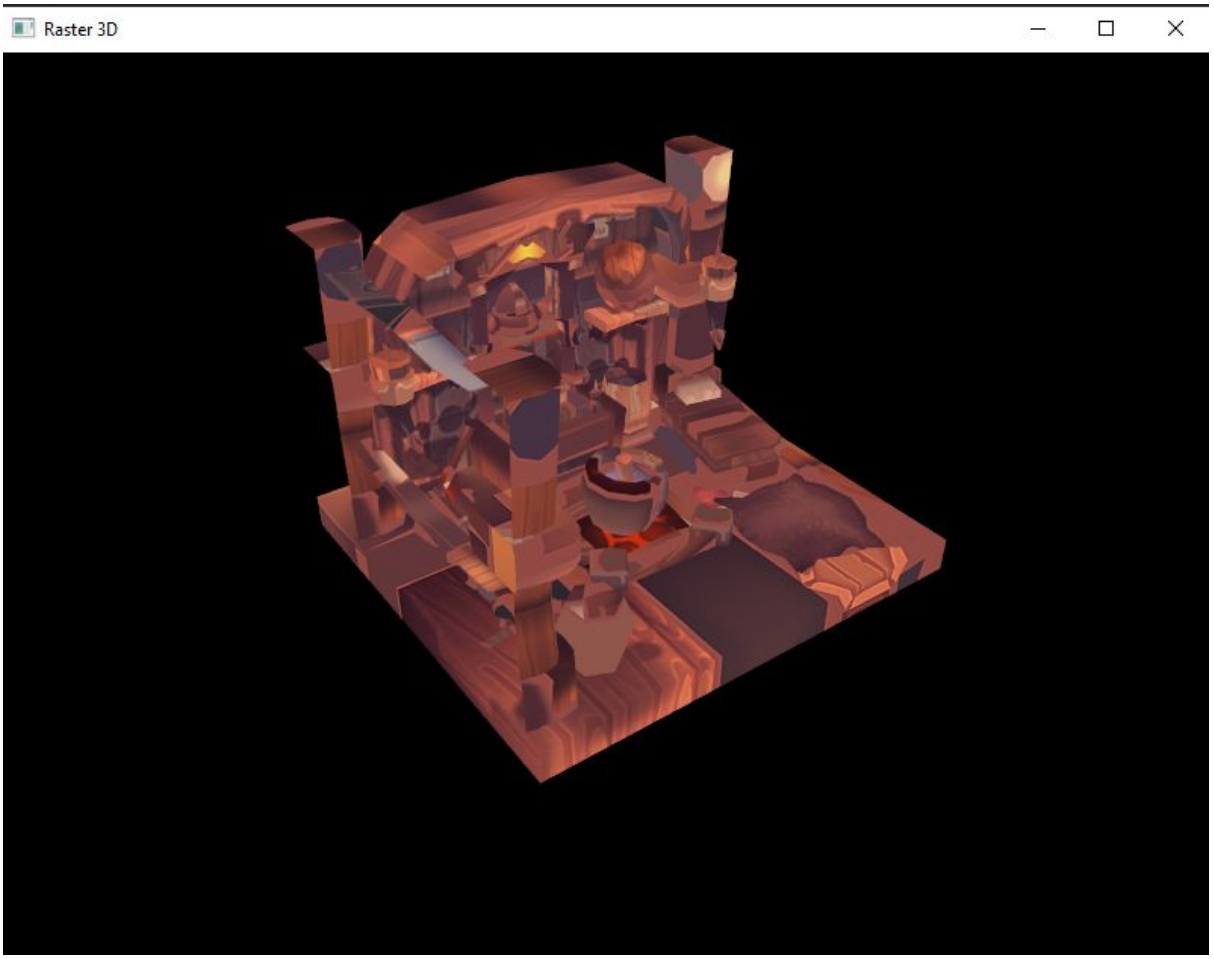


Figura 51. Objeto 3D con mipmaps y multisampling

Contexto de Ray Tracing en Vulkan

Como lo describe Nuno Subtil (2018), desde el lanzamiento de la arquitectura Turing, ha habido un enfoque en ray tracing con relación a DirectX. No obstante, muchos desarrolladores prefieren una aproximación inclusiva de otras plataformas que no sean Windows. Asimismo, con la evolución de los GPUs y la integración de la programabilidad permitieron computar problemas complejos como los algoritmos basados en la rasterización complejos y el lanzamiento de CUDA (plataformas de cómputo del GPU). La aceleración del

ray tracing por medio del GPU comenzó a través de CUDA y generó Optix (API específicamente diseñado para ray tracing) lo cual a su vez, culminó en RTX.

Subtil explica que la extensión de Vulkan para ray tracing aprovecha el trabajo realizado en APIs pasados, capaces de realizar estas operaciones. Se debe resaltar que esta extensión encaja con conceptos existentes de Vulkan como: asignación de memoria, manejo de recursos, lenguaje de shader y bytecode.

1. **Ray tracing Pipeline**

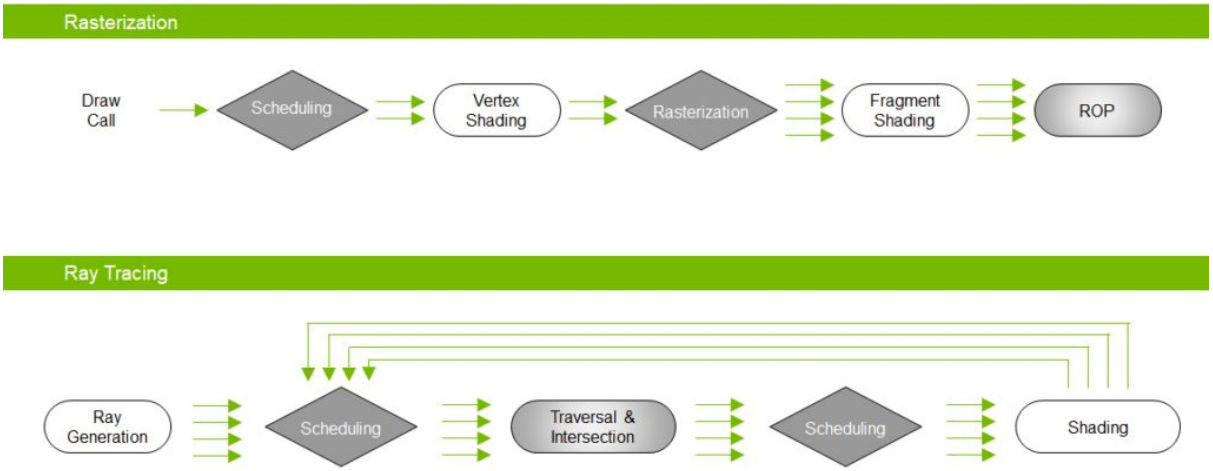


Figura 52. Ray Tracing Pipeline vs Rasterization Pipeline

Dentro del diagrama de la Figura anterior, los bloques grises representan componentes no-programables (funciones fijas y/o hardware) mientras que los blancos representan aquellos que sí son programables. Los diamantes representan las etapas en donde se agenda el trabajo.

En el pipeline de ray tracing, el número de unidades (rayos) de trabajo realizadas depende del resultado de unidades de trabajo previas. Esto significa que se pueden generar nuevos trabajos mediante etapas programables y alimentarlos directamente al pipeline.

Subtil detalla cuatro componentes clave componen el API de trazado de rayos:

2. Estructuras de aceleración:

En la rasterización, cada primitiva geométrica se procesa de forma independiente. En ray tracing, se debe probar cada rayo contra todas las primitivas en la escena. Para optimizar esto, se utilizan las estructuras de aceleración. Una estructura de aceleración es un objeto que contiene la información geométrica de las primitivas en la escena. Esta información se preprocesa de tal manera que se de tal manera que permita el rechazo trivial de posibles intersecciones rayo-primitivo. Esta es la primitiva del API con la que se pueden trazar los rayos.

La estructura de aceleración se presenta como una estructura de datos opaca, definida por la implementación, sin dependencia de ningún algoritmo subyacente o método de selección. Se modela como una estructura de dos niveles: bottom level (nivel inferior) el cual contiene los datos de la geometría y top-level (nivel superior) cuyos nodos contienen una lista de referencias a los nodos del nivel inferior junto con información de transformaciones y de shading.

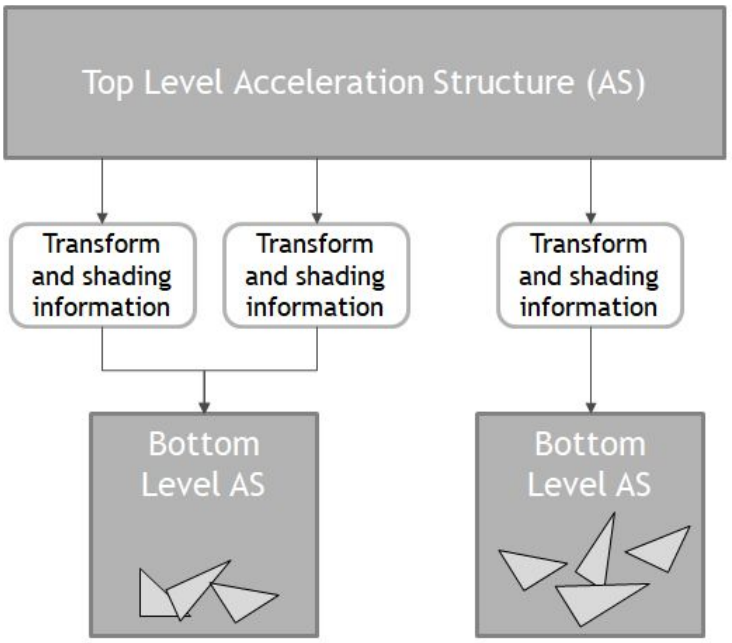


Figura 53. Estructuras de aceleración de nivel superior e inferior (TLAS, BLAS)

Estas estructuras pueden construirse de geometría existente en un objeto de búfer; esto conlleva dos procesos: crear los nodos del nivel inferior y sucesivamente generar los del nivel superior. Los procesos de compilación (crear un nuevo objeto desde cero, e.j., la configuración inicial de una escena) y actualización (actualizar un objeto existente con datos nuevos, e.j., cuando un personaje se mueve en una escena) ocurren en el GPU.

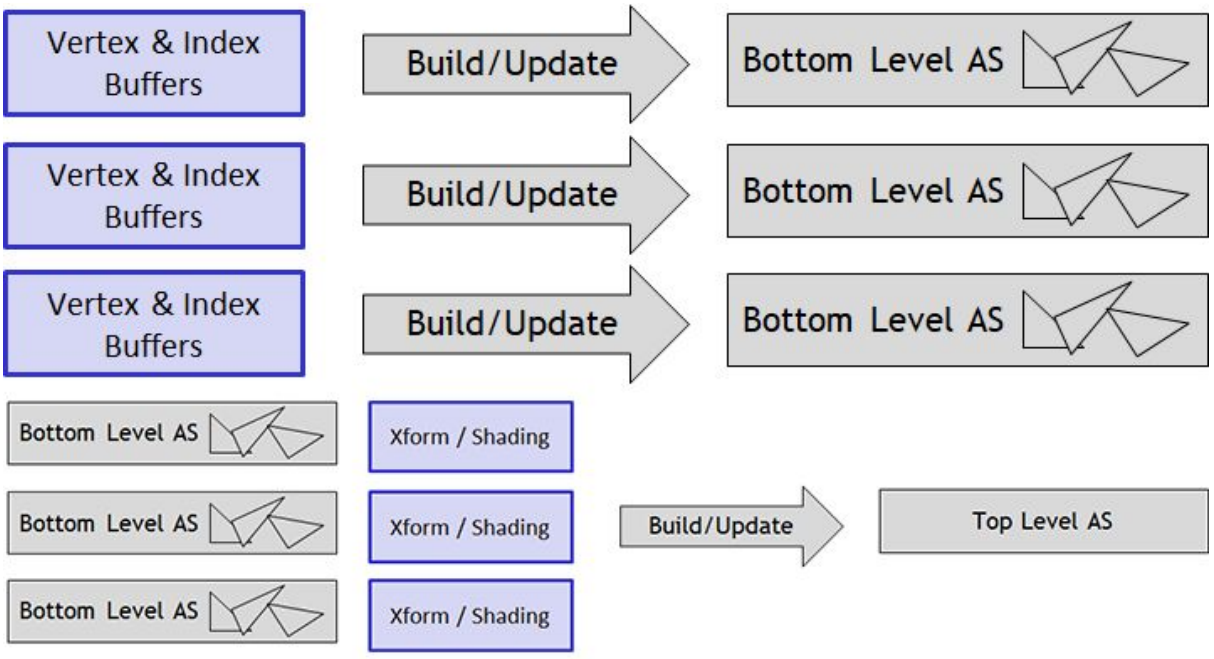


Figura 54. Flujo de estructura de aceleración

En el API, se define objeto Vulkan (VkAccelerationStructureNV) cuyas instancias encapsulan un nodo de una estructura superior o inferior. Asimismo, se define un nuevo tipo de descriptor de recurso (VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_NV) para enlazar estos objetos a los shaders. La creación y destrucción de objetos sigue el paradigma común de Vulkan:

A. Se crea un nodo de la estructura de aceleración con una llamada a vkCreateAccelerationStructureNV, lo cual regresa un handle opaco.
B. Este handle se puede utilizar con vkCreateAccelerationStructureNV para obtener información de que tipo de memoria y cuanta se necesitará para esta estructura de aceleración.

C. Se puede asignar memoria con vkAllocateMemory y enlazarla a el objeto llamando vkBindAccelerationStructureMemoryInfoNV.

Las operaciones de actualización y creación de las estructuras de aceleración requieren de memoria temporal "scratch" la cual consultada llamando vkGetAccelerationStructureScratchMemoryRequirementsNV. Esta memoria toma la forma de un objeto de búfer (VkBuffer) que se asigna basándose en los requerimientos de memoria regresados por la implementación. Esto se pasa como argumento a los comandos de compilación y actualización.

La cantidad de memoria requerida depende de los datos de la geometría. Los datos devueltos por vkGetAccelerationStructureMemoryRequirementsNV representan el límite superior para la cantidad de memoria que será requerida por un objeto en particular. Una vez construida la estructura, se puede utilizar vkCmdWriteAccelerationStructurePropertiesNV para que el GPU escriba el tamaño comprimido de un objeto de estructura de aceleración dado en un objeto de consulta Vulkan, el cual se puede leer en el CPU. Esto se puede utilizar posteriormente para asignar un objeto de estructura de aceleración separado con la cantidad de memoria exacta que se requiere, y vkCmdCopyAccelerationStructureNV se puede usar para comprimir el objeto original en uno nuevo.

Los comandos de compilación y actualización de la estructura de aceleración se ejecutan en el GPU y se pueden enviar a las colas de gráficos o cómputo. El API permite permite que la implementación paralelice los sucesivos comandos de compilación y actualización para maximizar la utilización del GPU. Se debe tener cuidado al reutilizar scratch buffers ya que su ejecución puede superponerse. Para evitarse, se necesita emplear el método de sincronización de barrera. Los bits de la bandera de acceso a la memoria

K_ACCESS_ACCELERATION_STRUCTURE_READ_BIT_NV y VK_ACCESS_ACCELERATION_STRUCTURE_WRITE_BIT_NV, pueden utilizarse en barreras de búferes de memoria en el scratch búfer antes de reutilizar la misma del búfer para otra operación y en las barreras de memoria global para asegurar que la complicación y actualización de las estructuras se complete antes de que el objeto de estas se utilice para rastrear rayos.

Para maximizar superposición, se recomienda asignar suficiente memoria de scratch búfer para realizar múltiples operaciones de complicación y actualización, y asignar cada operación consecutiva a distintas regiones de memoria. Cuanta memoria asignar depende de qué tan sensible sea la aplicación para el rendimiento de la compilación de la estructura de aceleración y de cuantas estructuras de aceleración se utilicen.

3. Nuevos dominios de shaders para ray tracings

Para exponer ray tracing a Vulkan, se definen nuevos dominios de shaders junto con primitivas para la comunicación entre shaders.

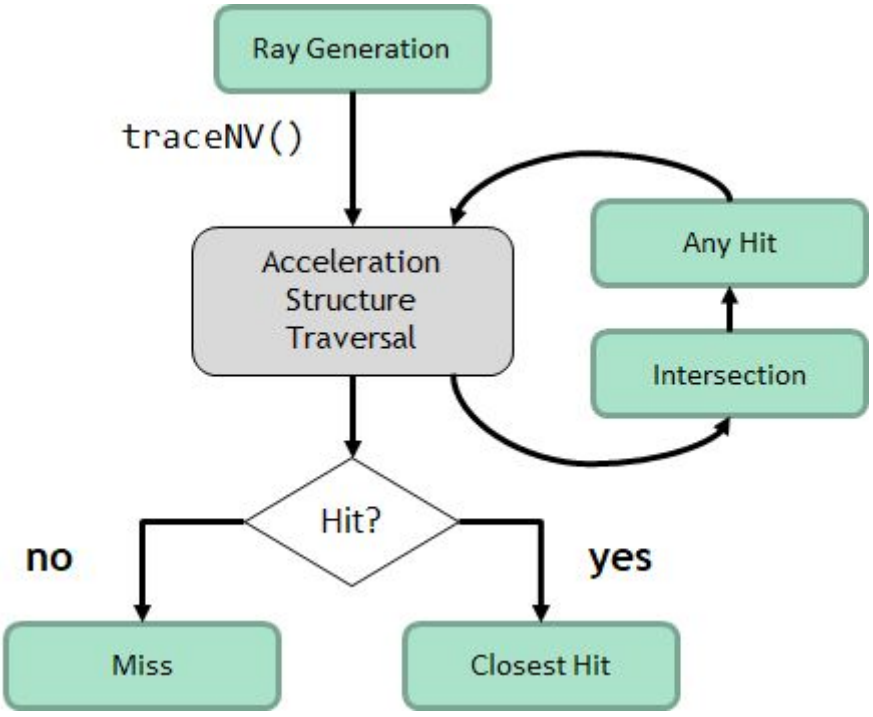


Figura 55. Dominios de shaders y sus relaciones

El primer dominio de la Figura anterior (*Ray Generation* o *raygen*) se ejecuta en una cuadrícula bidimensional de *threads* y es el punto de inicio para trazar rayos en la escena. Este shader también es responsable de escribir la salida final del algoritmo de ray tracing a la memoria.

El segundo shader a notar es el de intersección. Este implementa algoritmos de intersección entre rayos y primitivas arbitrarios. Sirven para interceptar rayos con diversas primitivas que cuyo soporte no está incorporado. Primitivas triangulares no requieren de este shader ya que están incorporadas.

Los siguientes shaders, (*Hit Shaders*) se invocan cuando se detecta una intersección. Estos shaders son responsables de computar la interacción que ocurre en el punto de intersección y, de ser necesario,invocar nuevos rayos. *Any Hit*, se invoca en todas las intersecciones de un rayo con primitivas de la escena en un orden arbitrario, y pueden rechazar intersecciones. *Closest Hit,* se invoca en el punto de intersección más cercano a lo largo del rayo. Naturalmente, *Miss Shader* se invoca cuando no se detecta una intersección para un rayo dado.

Durante la ejecución de una llamada a ray tracing, estos shaders tienen que transmitir información entre ellos. Los shaders de intersección tienen que comunicar datos de intersección, los de colisión tienen que consumir los datos de intersección y modificar datos de resultados arbitrarios. El shader de generación de rayos tiene que consumir los datos del resultado final y transmitirlos a la memoria.

Subtil introduce dos conceptos importantes con respecto a este sistema de comunicación entre los shaders. El primero siendo, *ray payload.* Esta es una una estructura de datos arbitraria,

definida por la aplicación, que almacena la información que se acumula a lo largo de la trayectoria del rayo. Normalmente se inicializa en el shader de generación de rayos pero se puede leer y modificar por los shaders de colisión. Se utiliza normalmente para regresar información sobre las propiedades de los materiales que se acomulo a lo largo de la trayectoria del rayo, lo cual el raygen escribe en la memoria.

El segundo concepto es *Ray attributes.* Este está compuesto de un conjunto de valores que regresan del shader de intersección al shader de colisión y que contienen cualquier dato que la aplicación necesite emitir desde la prueba de intersección. Estos valores se encapsulan en una estructura definida por la aplicación la cual se escribe por el shader de intersección y se lee por los shaders de colisión invocados en una intersección. Para el shader de intersección entre rayo y triángulo ya incorporado, los atributos del rayo son un vector bidimensional que contiene las coordenadas baricéntricas del punto de intersección a lo largo del triángulo.

4. Mapeos de Lenguaje GLSL:

Las cinco etapas de shaders anteriores cuentan con definiciones dentro de Vulkan para construir shaders GLSL. Asimismo, se incorporaron nuevas variables, dependiendo de la etapa de shader, que contienen información sobre varios parámetros. Estas incluyen varias cosas como "handles" del raygen thread, el tamaño de la cuadrícula de lanzamiento del raygen, primitivas y valores de identificación de instancia, origen del rayo actual y dirección en el espacio del mundo y del objeto, etcétera.

No obstante, se define un nuevo tipo de recurso opaco para los enlaces de recursos de estructura de aceleración accelerationStructureNV. Las estructuras de aceleración están atadas al ray tracing pipeline de manera que otro tipos de recursos. Subtil resalta que varias

estructuras de aceleración se pueden utilizar dentro de un mismo shader, permitiendo el recorrido jerárquico de distintos conjuntos de datos geométricos dentro del shader.

También se definieron nuevos nuevos calificadores de almacenamiento para tipos definidos por el usuario: rayPayloadNV, rayPayloadInNV, and hitAttributeNV que declaran la carga de datos útiles del rayo y definen qué etapa del shader posee el almacenamiento para dicha carga; y también los tipos de atributos de colisión a utilizarse. También se agregó un calificador que declara un *Shader Storage Buffer Object* (SSBO-utilizado para almacenar y recuperar datos) como vinculado al *shader binding table* (tabla de vinculación de shaders). Por último, se agregaron las funciones traceNV(), que lanza rayos a la escena, ignoreIntersectionNV() que descarta un punto de intersección dado, y terminateRayNV() para detener el procesamiento de un rayo al detectar una colisión.

5. **Emparejamiento de Estructura de Ray Payload:**

El shader API de ray tracing permite que cualquier estructura definida por la aplicación se use como el ray payload. El *payload* o carga de datos, se declara en una etapa de shader que puede invocar rayos (comúnmente raygen), y se pasa como referencia a los shaders de colisión para su modificación. Subtil explica que como GLSL no cuenta con polimorfismo de tipos, no se puede definir la llamada a traceNV() como tipo polimórfico; lo cual implica que no se pueden que no se pueden pasar tipos arbitrarios como argumentos. Para contrarrestar esto, se emparejan tipos de datos basándose en el calificador de diseño de ubicación:

6. Cada ray payload se declara con el calificador rayPayloadNV y un calificador de diseño de ubicación que lo asocie con un valor entero que sirva como un "handle" numérico para la variable del payload

7. Este "handle" se pasa a la llamada traceNV() en lugar de una referencia a la variable real.
8. Los shaders de colisión, deben declarar el mismo tipo de dato como una variable con el calificador rayPayloadInNV, permitiendo que la infraestructura de ray tracing trate esa variable como una referencia a los datos del payload entrantes.

Se hace notar que dentro de los shaders de colisión y fallo, sólo puede existir una variable rayPayloadInNV y solo estos shaders pueden declarar esta variable. Sin embargo, no existe validación estática para emparejar tipos de payload; por lo que un emparejamiento erróneo conduce a comportamiento indefinido. Finalmente, el valor del payload en traceNV() debe ser un valor inmediato en tiempo de compilación .

6. Shader Binding Table

Shade Binding Table (SBT) es un objeto VkBuffer que contiene un conjunto de registros de tamaño uniforme y que consiste de shader "handle" seguido de datos definidos por la aplicación. Estos "handles", determinan que shaders ejecutar para un registro de SBT dado mientras que los datos definidos por la aplicación en el registro se ponen a disposición de los shaders como un SSBO. Además, cada instancia de SBT tiene un tamaño de registro fijo. Dentro de cada registro, una serie de reglas de indexación determinan como la infraestructura del ray tracing obtendrá los "handles" y los datos del SSBO para que el siguiente shader los ejecute.

La intención es que los datos del SSBO en el SBT especifiquen qué recursos (texturas, búfer uniformes, etc.) se deben usar por cada shader. El patrón de uso previsto es que todos los

recursos potencialmente accesibles se vinculen al pipeline a través de conjuntos de descriptores, utilizando matrices de recursos. Entonces, el SBT contendrá índices en estas matrices que indiquen los recursos específicos a usar por un conjunto de shaders dado.

7. Objetos del pipeline de Ray Tracing

Finalmente, el último requerimiento es el objeto de estado del ray tracing pipeline (*Pipeline State Object* o *PSO*). Los pipelines de ray tracing consisten de una colección de los cinco shaders descritos y algunos parametros especificos de ray tracing como profundidad máxima de recursión de rayos. Un PSO de ray tracing puede contener muchos shader lo cual se traduce a mayores tiempos de compilación. Este costo se disminuye por medio de un interfaz que permite controlar la compilación de shaders individuales. En el tiempo de creación del PSO, la bandera VK_PIPELINE_CREATE_DEFER_COMPILE_BIT_NV, sirve para dar la instrucción al driver de omitir compilar shaders de manera inmediata. Posteriormente, la aplicación debe llamar vkCompileDeferredNV para cada shader para poder detonar el trabajo de compilación. Para minimizar tiempos de compilación, esto se puede paralelizar a través de threads. Una vez creado, objetos de pipeline de ray tracing pueden enlazarse a colas de graficación o cómputo usando las llamadas estandarizadas de Vulkan con un nuevo punto de enlace de pipeline: VK_PIPELINE_BIND_POINT_RAYTRACING_NV.

En suma, Subtil explica que al haber construido y enlazado las estructuras de aceleración, la tabla de enlace de shaders y el PSO de ray tracing, se utiliza el comando vkCmdTraceRaysNV para comenzar a trazar los rayos. Los argumentos que recibe este

comando especifican las dimensiones de la cuadrícula de threads 2D y el objeto VkBuffer que contiene el SBT a usarse y los *offsets* dentro de ese SBT para los

para los diversos elementos de datos requeridos (offsets entre cada etapa de shader y sus datos SSBO correspondientes).

Referencias

iOrange. (2018). Vulkan Raytracing Tutorials. Abril 24, 2020, de iOrange Sitio web: https://iorange.github.io/

Overvoorde, A. (s.f.). Vulkan Tutorial. Enero 15, 2020, de Alexander Overvoorde Sitio web: https://vulkan-tutorial.com/

Pharr .M, Jakob. W. & Humphreys G. (2004). Physically Based Rendering: From Theory To Implementation. Noviembre 7, 2019, de Physically Based Rendering Sitio web: http://www.pbr-book.org/3ed-2018/contents.html

Subtil, N. (2018). Introduction to Real-Time Ray Tracing with Vulkan. Abril 24, 2020, de NVIDIA Sitio web: https://developer.nvidia.com/blog/vulkan-raytracing/

Vries, J. (s.f.). Coordinate Systems. Junio 20, 2020, de Joey de Vries Sitio web: https://learnopengl.com/Getting-started/Coordinate-Systems