Ray Tracing en Vulkan

Joaquín Fontana
Computación gráfica.

31 de enero de 2024

Índice

1. Introducción

A lo largo de este informe explicaremos el *ray tracing* en GPU utilizando Vulkan, desglosando su pipeline y analizando la estructura del repositorio que contiene ejemplos prácticos. Este repositorio no solo sirve como punto de partida para comprender la implementación del *ray tracing* en Vulkan, sino que también ofrece la posibilidad de personalizar y construir aplicaciones propias. Además, examinaremos cada ejemplo proporcionado y demostraremos cómo modificar el código para adaptarlo a necesidades específicas.

Importante: Previo a adentrarnos en detalles, es crucial asegurarnos de que nuestra GPU cuente con soporte para *ray tracing* en Vulkan, o sea debe contar con las extensiones: `VK_KHR_ray_tracing_pipeline`, `VK_KHR_acceleration_structure`, `VK_KHR_deferred_host_operations`. Verificar la presencia de estas extensiones se puede realizar muy fácilmente al instalar el SDK de Vulkan o consultando la lista en la web proporcionada [1]. Es importante tener en cuenta que esta base de datos es mantenida por un usuario de la comunidad y podría estar desactualizada, por lo que se recomienda la primera opción.

2. NVpro core y NVVK helper

En los ejemplos presentados posteriormente, utilizaremos estos dos repositorios: **NVpro core**, para facilitar la inclusión de todos los recursos necesarios, y **NVVK** para facilitar la manipulación de Vulkan.

NVpro core de Nvidia es un repositorio que recopila código fuente y bibliotecas tanto de Nvidia como de terceros, útiles para el desarrollo de aplicaciones gráficas. Algunas de las más comunes y que utilizaremos son: nvvk helper, glm, imgui, OBJLoader, entre otras [2].

NVVK helper es una colección de funciones auxiliares para trabajar con la API de Vulkan. Esta biblioteca proporciona una serie de utilidades y clases para simplificar el trabajo con Vulkan, facilitando tareas comunes y reduciendo la cantidad de código repetitivo [3].

Es importante destacar que, si bien la biblioteca fue desarrollada por Nvidia, también es **compatible con dispositivos de AMD**, debido a que esta consiste en envolver código Vulkan, el cual es portable.

3. Pipeline de *ray tracing*

El pipeline de *ray tracing* consiste en múltiples etapas programables (shaders), que interactúan entre sí, de la forma que muestra el diagrama (ver figura 2), destinadas a el uso de los desarrolladores para controlar el renderizado. En el contexto del *ray tracing*, un shader puede ser responsable de una variedad de tareas, incluyendo la generación de rayos, la determinación de cómo los rayos interactúan con los objetos en la escena, y la combinación de estos resultados para producir la imagen final.

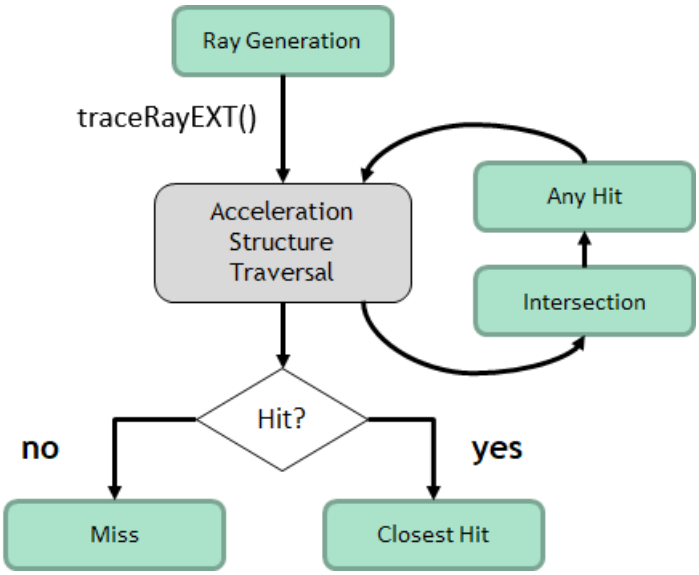

Figura 1: Orden de ejecución de shaders

Antes de comenzar con el trazado de rayos, deberemos construir una estructura de aceleración para nuestra escena. Esta es construida por el propio Vulkan y consta de una estructura de aceleración de nivel superior (TLAS) y múltiples estructuras de aceleración de nivel inferior (BLAS). Cada BLAS puede ser una malla de triángulos o una colección definida por el usuario de cajas delimitadoras (AABB).

Un BLAS puede ser instanciado en el TLAS y recibe un *gl_InstanceID* único. Además, cada triángulo en una malla de triángulos y cada AABB en la colección de cajas de intersección recibe un *gl_PrimitiveID*.

Cada BLAS tiene su propia matriz de transformación a espacio global, que se asigna inicialmente cuando el BLAS se añade al TLAS. Esta transformación es accesible en el shader a través de las variables integradas *gl_ObjectToWorldEXT* y *gl_WorldToObjectEXT*. Cuando se produce un impacto en el recorrido del rayo y se llama al *intersection shader*, *any-hit* o *closest-hit*, estas variables mencionadas se establecen en consecuencia.

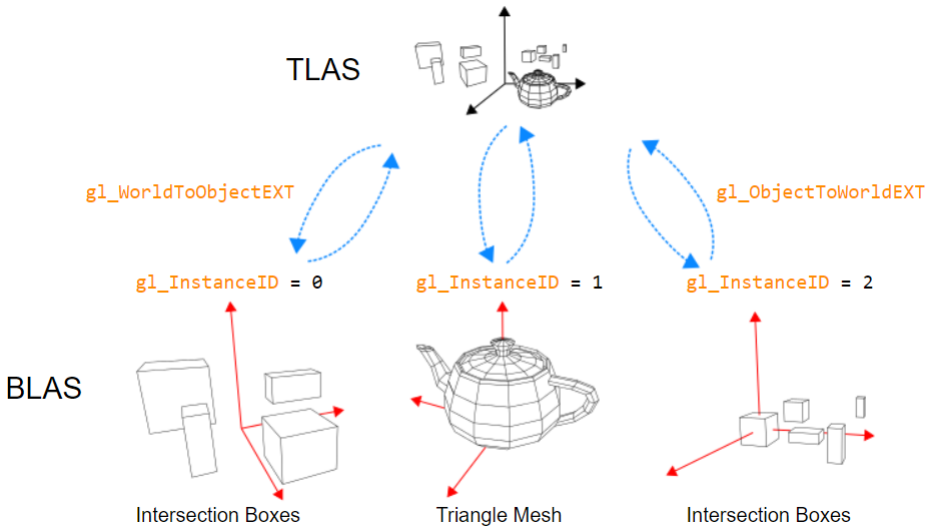

Figura 2: Diagrama de estructura de aceleración

3.1. Shaders de *ray tracing*

En esta sección daremos una descripción general de los diferentes tipos de shaders que podrías encontrar en un pipeline de *ray tracing*:

El *ray-generation shader* son el punto de partida para el pipeline de trazado de rayos. Especifica y lanza rayos a través de la "*Acceleration Structure*" llamando a la función $traceRayEXT(...)$. La función $traceRayEXT()$ lanza un solo rayo en la escena para probar intersecciones y, al hacerlo, puede activar otros shaders, que se presentarán en breve. El parámetro más importante de la función $traceRayEXT()$ es la variable *payload* la cual contiene información que se adjunta al rayo, como por ejemplo: color, origen, dirección, etc. Es el usuario quien define el tipo de esta variable y puede modificarse en las etapas de shaders que se llaman posteriormente en el recorrido del rayo. Una vez que el recorrido del rayo se completa, la función $traceRayEXT()$ vuelve al llamador. En general, luego de que esto ocurre, se evalúa el *payload* en el *ray-generation shader* para producir una imagen de salida.

El *closest-hit shader* se ejecuta cuando se determina que un rayo ha golpeado el objeto más cercano. Por lo que típicamente se utilizan para calcular o recopilar las propiedades del objeto (cálculos de iluminación, evaluación de materiales, etc) en el punto de intersección. Para lograr esto, el shader puede acceder a varias variables integradas, como $gl\_PrimitiveID$ o $gl\_InstanceID$, que se establecen en consecuencia para cada impacto y son un identificador tanto de la geometría, como del triángulo impactado por el rayo.

De lo contrario, si no se produjo ningún impacto, se llama al *miss shader*. Este usualmente muestrea un mapa de entorno, o simplemente devuelve un color. Se le puede dar otro uso junto a un rayo de sombra, indicando a través de un *payload* que un objeto no está ocluido

si se ejecuta este shader.

Los dos últimos shaders, que se presentan a continuación, son opcionales.
El *intersection shader* se utilizan para intersecar rayos con geometría definida por el usuario. Las intersecciones de rayo/triángulo tienen soporte incorporado, por lo tanto, no requieren un *intersection shader*. Si se detecta una intersección del rayo con una AABB (axis-aligned bounding boxe) definida por el usuario o un triángulo de una malla de triángulos, se llama al *intersection shader*. Si el shader determina que ha ocurrido una intersección rayo/primitiva dentro de la caja delimitadora, se notifica con la función $reportIntersectionEXT(...)$. Además, el *intersection shader* puede llenar una variable $hitAttributeEXT$ (que puede ser el usuario quien define su tipo). En el caso de triángulos, ya hay un *intersection shader* incorporado. El *intersection shader* incorporado proporciona coordenadas baricéntricas de la ubicación del impacto dentro del triángulo con la variable "*hitAttributeEXT vec2 baryCoord*". Para primitivas geométricas que no son triángulos (como cubos, cilindros, esferas, superficies paramétricas, etc.), se debe proporcionar tu *intersection shader* personalizado.

El *any-hit shader* se ejecuta en cada intersección con una primitiva (no solo en la más cercana). Se puede utilizar para descartar intersecciones, por ejemplo, para realizar alfa testing, mediante una búsqueda de textura e ignorando la intersección si el valor obtenido no cumple con un criterio específico. El *any-hit shader* por defecto devuelve información sobre las intersecciones para que se pueda determinar la intersección más cercana.

En resumen: Implementaremos nuestro algoritmo de *ray tracing* mediante la programación de los diferentes shaders, los cuales se ejecutan en el orden especificado en el diagrama de arriba (ver figura 2). Donde cada uno, en general, se encargará de una tarea específica.
Gracias a esta metodología de trabajo, aprovecharemos la capacidad de paralelización entre etapas del pipeline, así como la creación de la estructura de aceleración y el trazado eficiente de rayos sobre ella, ambas provistas por la GPU. Esto resultará en un notable incremento en la performance de nuestra aplicación.

3.2. Variables y funciones integradas (built-in functions)

Al habilitar la extensión *GLSL_EXT_ray_tracing* en nuestros shaders se definirán nuevas variables, constantes y funciones ya integradas en la GPU para el uso del pipeline de *ray tracing*. En esta sección daremos una lista de todas estas variables (ver figura 3) y explicaremos las más importantes.

	Ray generation	Closest-hit	Miss	Intersection	Any-hit
uvec3 gl_LaunchIDEXT	✓	✓	✓	✓	✓
uvec3 gl_LaunchSizeEXT	✓	✓	✓	✓	✓
int gl_PrimitiveID		✓		✓	✓
int gl_InstanceID		✓		✓	✓
int gl_InstanceCustomIndexEXT		✓		✓	✓
int gl_GeometryIndexEXT		✓		✓	✓
vec3 gl_WorldRayOriginEXT		✓	✓	✓	✓
vec3 gl_WorldRayDirectionEXT		✓	✓	✓	✓
vec3 gl_ObjectRayOriginEXT		✓		✓	✓
vec3 gl_ObjectRayDirectionEXT		✓		✓	✓
float gl_RayTminEXT		✓	✓	✓	✓
float gl_RayTmaxEXT		✓	✓	✓	✓
uint gl_IncomingRayFlagsEXT		✓	✓	✓	✓
float gl_HitTEXT		✓			✓
uint gl_HitKindEXT		✓			✓
mat4x3 gl_ObjectToWorldEXT		✓		✓	✓
mat4x3 gl_WorldToObjectEXT		✓		✓	✓

Figura 3: Lista de variables integradas

La extensión define varias funciones integradas, siendo la más utilizada $traceRayEXT()$, que inicia una operación de trazado de rayo. Esta puede ser llamada desde un *ray-generation*, *closest-hit* o *miss shader*. Las aplicaciones típicas son disparar los rayos primarios desde la posición de la cámara en el *ray-generation shader*, y disparar un rayo "de sombra" en dirección a la fuente de luz para determinar si la superficie está ocluida por otros objetos en el *closest-hit shader*. Debemos tener especial cuidado con llamar a esta función en shaders que no sean el de generación de rayos, puesto que una llamada en otro shader se considera **recursión**. Esto no es deseable por dos motivos: Dependiendo la GPU que utilicemos puede variar el número de llamadas recursivas soportado, por ejemplo, una Nvidia Quadro RTX 6000 soporta 31 llamadas recursivas, mientras que una AMD Radeon RX 7600 soporta una sola, por lo que si realizamos llamadas recursivas, nuestra aplicación, podría no funcionar sobre cierto hardware. El otro motivo es que los algoritmos recursivos si bien, son fáciles de programar, suelen ser menos eficientes que su contraparte iterativa, por lo que no utilizar recursión impactará positivamente en el rendimiento de la aplicación.

También define las variables:

- *gl_LaunchIDEXT* y *gl_LaunchSizeEXT* identifican un hilo en la cuadrícula de lanzamiento para un *ray-generation shader*. *gl_LaunchIDEXT* es un vector donde sus primeras dos entradas indican las coordenadas del píxel para el cual se está ejecutando el shader. Mientras que *gl_LaunchSizeEXT* contiene en sus primeras dos coordenadas el ancho y alto de la imagen de salida.
- *gl_PrimitiveIDEXT* y *gl_InstanceIDEXT* identifican una primitiva (en general un triángulo) y una instancia (en general una malla de triángulos) respectivamente, al procesar una intersección con un *closest-hit shader*, *any-hit* o *intersection*.
- *gl_HitTEXT* contiene la distancia desde el origen del rayo al punto de intersección (accesible en shaders *closest-hit* y *any-hit*).

Se definen nuevos calificadores de almacenamiento para datos que necesitan compartirse entre los shaders. *rayPayloadEXT* declara una variable con almacenamiento para un *payload*. Esto suele ser un struct que almacena propiedades del rayo, las cuales se necesita transmitir entre las distintas etapas. Se permite utilizar en cualquier shader que pueda llamar a *TraceRayEXT*().

rayPayloadInEXT declara una variable *payload* de entrada, sin almacenamiento (se espera que se haya definido en una etapa diferente mediante el calificador *rayPayloadEXT*). Se permite utilizar en cualquier etapa que pueda ser invocada durante la ejecución de *TraceRayEXT*(). El tipo de variable utilizado en *rayPayloadEXT* y *rayPayloadInEXT* debe coincidir entre el llamador y el receptor. Básicamente estos dos calificadores se utilizan para definir variables globales, que se acceden desde distintos shaders para su comunicación.

hitAttributeEXT declara una variable con almacenamiento para datos de intersección rayo/primitiva. La declaración explícita de un *hitAttributeEXT* solo es necesaria cuando el pipeline incluye *intersection shaders* personalizados. Las variables *hitAttributeEXT* es de solo lectura para shders *any-hit* y *closest-hit* y de lectura-escritura en *intersection shaders*.

4. Repositorio

Si bien se dio un pantallazo general de las librerías utilizadas, todos los ejemplos tienen una estructura ya definida para lo que es la aplicación en *C++*. Por esta razón los ejemplos se enfocarán en la modificación y programación de los shaders.

Por supuesto, la implementación de algunas funcionalidades implicarán la modificación de la aplicación, pero esto se mantendrá acotado.

4.1. Estructura

Para cada proyecto encontraremos una serie de archivos con un objetivo concreto, los cuales describiremos a continuación:

- **Aplicación:** Los archivos *main.cpp*, *hello_vulkan.h* y *hello_vulkan.cpp*, son los correspondientes al código de la aplicación. Dentro de *main.cpp* se inicializan los pipelines, se modifica la interfaz gráfica de usuario, se construye la escena a partir de archivos .obj y sus matrices de transformación 4, etc. En los archivos restantes simplemente se definen y declaran las funciones relacionadas al *ray tracing* en Vulkan.

```
// Creation of the example

//helloVk.loadModel(nvh::findFile("media/scenes/CornellBox-Sphere.obj", defaultSearchPaths, true));

{  //cornell dragon
    helloVk.loadModel(nvh::findFile("media/scenes/CornellBox-Empty-CO.obj", defaultSearchPaths, true));
    helloVk.loadModel(nvh::findFile("media/scenes/dragon.obj", defaultSearchPaths, true),
                      glm::translate(
                      glm::rotate(
                      glm::scale(glm::mat4(1.0f), vec3(1.5,1.5,1.5)), (float)1.5, vec3(0, 1, 0)),vec3(0, 0.5, 0)));
}
```

Figura 4: Ejemplo de creación de escena, cargando archivos .obj, con sus matrices de transformación

En la carpeta *shaders* encontraremos lo siguiente:

- **Host_device:** Este archivo está incluido tanto por los shaders como por la aplicación, por lo que en él se establecen definiciones que deben ser conocidas por ambas partes, la aplicación que corre en la PC (host) y los shaders ejecutados por la GPU (device). Se utiliza por ejemplo, para la transferencia de estructuras de datos, en la que tanto la aplicación y los shaders deben ser conscientes del formato de estas.
- **Post shader:** Este shader se ejecuta con la imagen obtenida del pipeline de *ray trcaing* como input. Por defecto, no hace nada, asigna a un píxel el color que ya tiene, pero su propósito es ser utilizado para la implementación de un tone mapping, por ejemplo, una corrección gamma.
- **shaders de *ray tracing* y rasterizado:** Obviamente en cada ejemplo se encontrarán los shaders correspondientes al pipeline de *ray tracing*, además siempre tendremos disponibles shaders de rasterizado con sombreado de *Phong* para renderizar nuestra escena con este algoritmo si lo deseamos.
- **Ray common:** En este archivo se define la estructura del *payload* del rayo, este debe ser incluido en todos los shaders.

También tenemos la posibilidad de utilizar la librería **ImGui** (ya integrada en la aplicación), esta nos permitirá cambiar parámetros de la escena en tiempo real, desde una interfaz gráfica (ver figura 5) y es muy fácil de usar. Se recomienda ver los ejemplos en su documentación [4] e indagar en el código de los ejemplos.

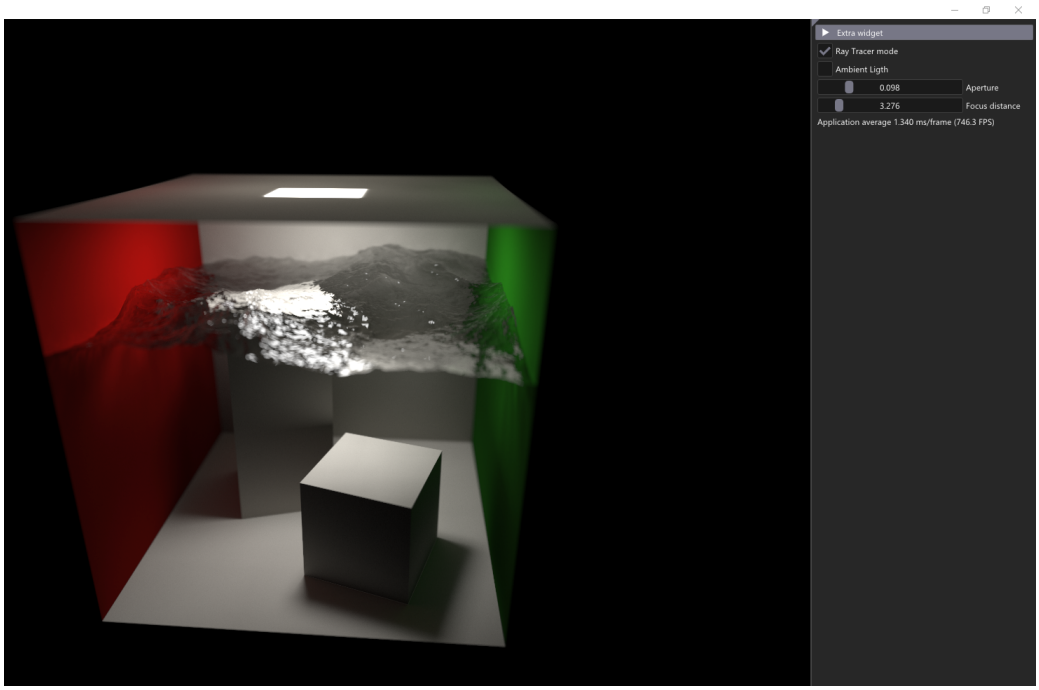

Figura 5: Ejemplo de interfaz gráfica que permite cambiar la distancia focal, la apertura de la cámara y el modelo de iluminación

Los ejemplos ya vienen con una GUI básica, que permite cambiar parámetros de la cámara, algoritmo de renderizado, entre otras cosas. Para activarla se debe descomentar el código correspondiente del archivo *main.cpp*, similar al de la imagen (figura 6)

```
// Show UI window.
if(helloVk.showGui()) {
  ImGuiH::Panel::Begin();
  renderUI(helloVk);

  ImGui::Checkbox("Ray Tracer mode", &useRaytracer);  // Switch between raster and ray tracing
  ImGui::Checkbox("Ambient Ligth", &helloVk.m_pcRay.ambientLigth); //enable ambient ligth
  ImGui::SliderFloat("Aperture", &helloVk.m_pcRay.camAperture, 0.001f, 0.5f); //camera parameters
  ImGui::SliderFloat("Focus distance", &helloVk.m_pcRay.focusDist, 1.f, 20.f);

  ImGui::Text("Application average %.3f ms/frame (%.1f FPS)", 1000.0f / ImGui::GetIO().Framerate, ImGui::GetIO().Framerate);
  ImGuiH::Panel::End();
}
```

Figura 6: Código que genera el panel de la figura 5

4.2. Ejemplos

Se recomienda leer y entender las secciones anteriores mientras se contrasta con el código de los ejemplos a continuación.

4.2.1. Visualización LaunchID

Este ejemplo es el más simple de todos. Solo utiliza el *ray-generation shader*. Este se ejecuta en paralelo para cada píxel de la imagen final, con distintos valores para la variable $gl\_LaunchIDEXT$, por lo que este ejemplo nos ayuda a visualizar esto, modulando los canales rojo y verde del píxel, con el valor de la variable $gl\_LaunchIDEXT$.

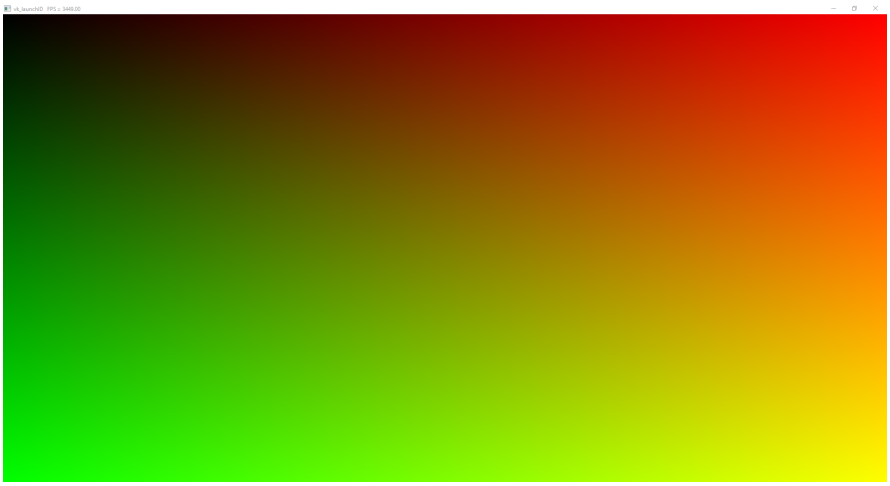

Figura 7: Captura de pantalla del ejemplo vk_launchID

4.2.2. Visualización PrimitiveID

A continuación, se introducen distintos elementos, el *colsest-hit shader*, el *miss shader* y el uso del *payload* del rayo para transferir el color desde estos shaders al *ray-generation shader*. el *miss shader* se encarga exclusivamente de asignar el color negro al fondo, mientras que el *closest-hit* aisgna un color correspondiente al valor de primitiveID.

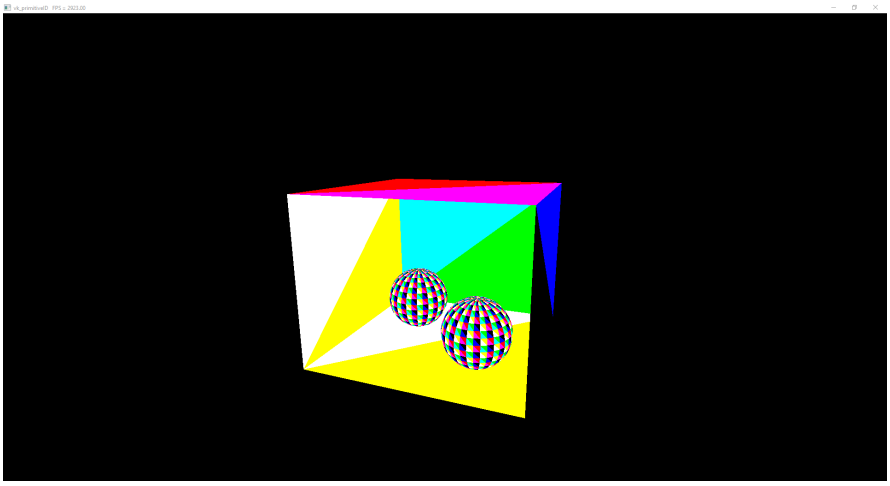

Figura 8: Captura de pantalla del ejemplo vk_primitiveID

4.2.3. Evaluar materiales

Por otro lado mostraremos como procesar una intersección, accediendo a los buffers de materiales, vértices, normales, y otros datos transferidos al *closest-hit* shader, como texturas, posición de la luz, etc. Estos datos están disponibles para su uso en el cuerpo del shader. En este caso se utilizan para realizar un sombreado difuso.

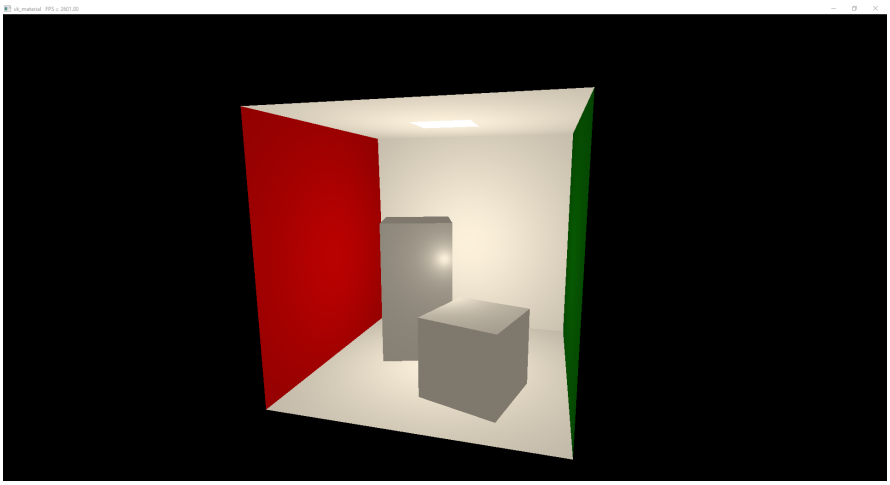

Figura 9: Captura de pantalla del ejemplo vk_material

4.2.4. Rebotes

Pasemos ahora a agregar reflejos en ciertos materiales. En este ejemplo se expandió el *payload* del rayo para que pueda guardar el lugar de impacto, y la dirección con que se va a reflejar el rayo, esta información es comunicada al *ray-generation shader* para trazar un segundo rayo a partir estos parámetros.

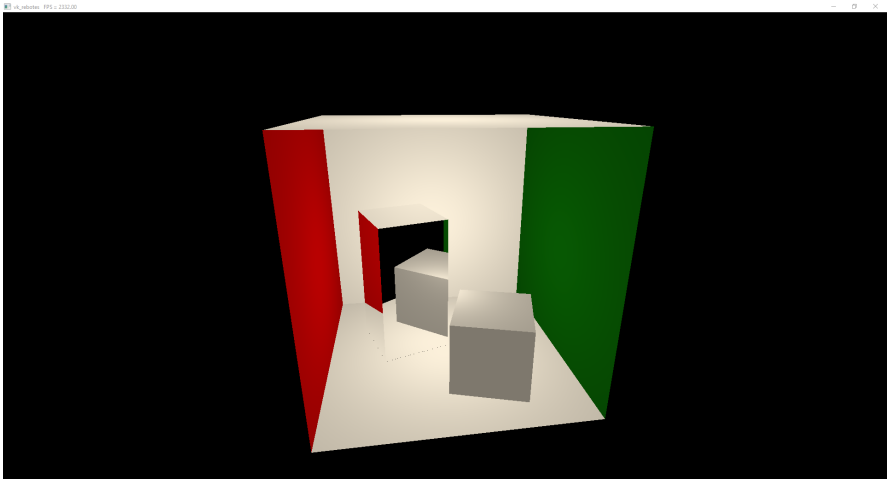

Figura 10: Captura de pantalla del ejemplo vk_rebotes

4.2.5. Números Aleatorios

Este ejemplo es el más complejo. En él se agrega un archivo *random.glsl* con funciones para la generación de un número semilla y de números aleatorios de punto flotante.
A demás se agrega el número de frame que se está procesando, en el archivo *host_device.h*, esto nos permitirá transferirlo desde la aplicación, y así utilizarlo en combinación con la variable $LaunchID$ para generar una semilla distinta por cada píxel y cada frame (figura 11).

```
uint seed = InitRandomSeed(gl_LaunchIDEXT.y * gl_LaunchSizeEXT.x + gl_LaunchIDEXT.x, pcRay.frame);
```

Figura 11: Generación de semilla

En el ejemplo, en lugar de trazar cada rayo primario en dirección al centro del píxel, se utiliza la semilla para trazar el rayo en una dirección aleatoria en un entorno del centro del píxel. Como ya mencionamos, esto cambia píxel a píxel y frame a frame, lo que genera este efecto de ruido en la imagen (figura 12).

Figura 12: Zoom en la escena de vk_random

Para terminar, dejamos tutoriales de Nvidia que pueden ser de utilidad para implementar funcionalidades complementarias o los shaders que no fueron utilizados en los ejemplos (*any-hit* e *intersection shader*) [5] [6] [7] [8]. Los ejemplos vistos anteriormente son una extensión del código base del que parten estos tutoriales, por lo que es muy fácil seguirlos e incorporarlos a nuestro proyecto si fuera necesario.

Referencias

[1] Vulkan gpuinfo: https://vulkan.gpuinfo.org/listdevicescoverage.php?extension=VK_KHR_ray_tracing_pipeline.

[2] NVIDIA. Nvpro-core. NVIDIA DesignWorks Samples. https://github.com/nvpro-samples/nvpro_core/tree/master.

[3] Documentación nvvk helper: https://github.com/nvpro-samples/nvpro_core/blob/master/nvvk/README.md.

[4] Documentación ImGui: https://github.com/ocornut/imgui?tab=readme-ov-file.

[5] Intersection Shader - Tutorial: https://github.com/nvpro-samples/vk_raytracing_tutorial_KHR/tree/master/ray_tracing_intersection.

[6] Any Hit Shaders - Tutorial: https://github.com/nvpro-samples/vk_raytracing_tutorial_KHR/tree/master/ray_tracing_anyhit.

[7] Callable Shaders - Tutorial: https://github.com/nvpro-samples/vk_raytracing_tutorial_KHR/tree/master/ray_tracing_callable.

[8] Specialization Constants: https://github.com/nvpro-samples/vk_raytracing_tutorial_KHR/tree/master/ray_tracing_specialization.

[9] NVIDIA Vulkan Ray Tracing Tutorial: https://nvpro-samples.github.io/vk_raytracing_tutorial_KHR/vkrt_tutorial.md.html.

[10] NVIDIA vk_mini_path_tracer Tutorial: https://nvpro-samples.github.io/vk_mini_path_tracer/index.html.

[11] Ray Tracing Gems II (2021): https://link.springer.com/book/10.1007/978-1-4842-7185-8.