Ray Tracing en Vulkan

Joaquín Fontana
Computación gráfica.

20 de enero de 2024

Índice

1. Introducción

Antes de comenzar se deben tener las extensiones: rt pipeline, etc, se puede ver si se tiene estas extensiones instalando vulkan, o mirando la lista proporcinada: tener en cuenta que la lista es hecha por un usuraio, por lo que puede estar desactualizada.

2. NVpro core y NVVK helper

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, etc.

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.

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.

En los ejemplos presentados posteriormente utilizaremos estos dos repositorios, NVpro core, para la facilitar inclusión de todos los recursos necesarios, y NVVK para facilitar la manipulación de Vulkan.
link a documentacion: https://github.com/nvpro-samples/nvpro$_c ore/blob/master/nvvk/README.md$

3. Pipeline de ray tracing

El pipeline de ray tracing consiste en múltiples etapas programables (shaders) separadas, que interactúan entre sí, de la forma que muestra el diagrama (ver figura 1), 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.

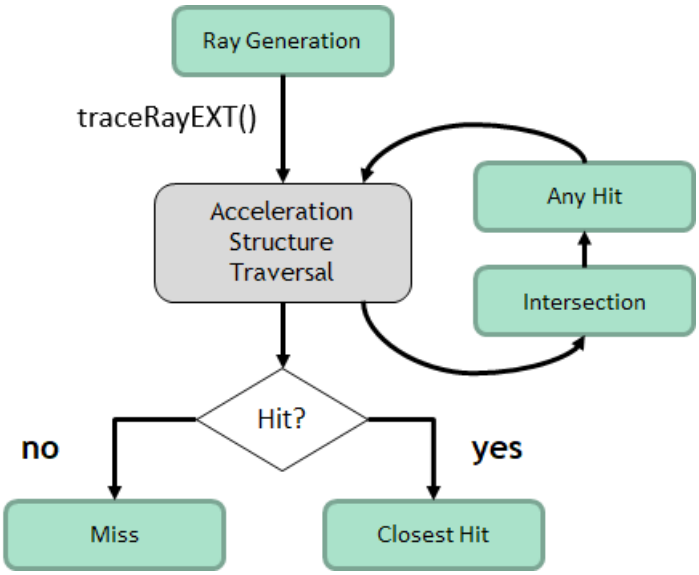

Figura 1: Orden de ejecución de shaders

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

El shader "Ray Generation"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. Esta variable es de tipo definido por el usuario 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 shader de generación de rayos para producir una imagen de salida.

El shader "closest-hit"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 de el tringulo impactados por el rayo.

De lo contrario, si no se produjo ningún impacto, se llama al shader "miss". Este usualmente muestrea un mapa de entorno, por ejemplo, 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 shader de "intersección"se utilizan para intersecar rayos con geometría definida por el usuario. Las intersecciones de rayo/triángulo tienen soporte incorporado y, por lo tanto, no requieren un shader de intersección. 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 shader de intersección. Si el shader determina que ha ocurrido una intersección rayo-primitiva dentro de la caja delimitadora, se notifica con la función reportIntersectionEXT(...). Además, el shader de intersección puede llenar una variable hitAttributeEXT (que puede ser de tipo definido por el usuario). En el caso de triángulos, ya hay un shader de intersección incorporado. El shader de intersección 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.), debes proporcionar tu shader de intersección personalizado.

El shader "any-hit"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 shader any-hit por defecto devuelve información sobre las intersecciones para que se pueda determinar la intersección más cercana.

En resumen: Programaremos 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. Donde cada uno, en general, se encargará de una tarea especifica.
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.

3.1. Variables y funciones integradas

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 2) 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 2: 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 shaders de generación de rayos, closest-hit y miss. Las aplicaciones típicas son disparar los rayos primarios desde la posición de la cámara en el shader de generación de rayos, 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 shader closest-hit. 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: La recursión no es soportada por todo hardware, en general, dependiendo la GPU que utilicemos puede variar el numero 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 llamada recursiva, 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 shader de generación de rayos. gl_LaunchIDEXT es un vector donde sus primeras dos entradas indican las coordenadas del pixel 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 al procesar una intersección en un shader de closest-hit, any-hit o intersección.
- gl_HitTEXT contiene el valor t de una intersección (distancia desde el origen del rayo al punto de intersección) a lo largo del rayo (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 gloabales, 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 shaders de intersección personalizados. Las variables hitAttributeEXT es de solo lectura para shders any-hit y closest-hit y de lectura-escritura en shaders de intersección.

3.2. Uniform buffer y push constants

3.3. Tone maping

4. Repositorio

Se recomienda leer y entender las secciones anteriores mientras se comprara con el contendio de los ejemplos.

explicar que si bien explicamos lo de nvvk, casi ni se va atocar el codigo vulkan, ni el codigo de la aplicación en general.

4.1. Estructura

4.2. Ejemplos

Referencias

[1] Languardia, R. IMERL (2023) Métodos Numéricos. Facultad de Ingeniería, Universidad de la República.