Universidad de Huelva

Tema 6

Vulkan - Shaders

Índice

- A la hora de desarrollar un programa en Vulkan es necesario tener en cuenta que debemos programar en varios planos:
  - Por un lado utilizaremos las funciones de Vulkan como las de cualquier otra librería incluida en un programa escrito en C/C++.
  - Por otro lado incluiremos fragmentos de código a insertar en los shaders. Este código utiliza el lenguaje GLSL. De momento solo vamos a utilizar el Vertex_Shader y el Fragment_Shader.
  - Para distinguir los diferentes tipos de código vamos a utilizar distintos colores de fondo.

Código incluido en el programa fuente. A ejecutar en la CPU.	Código incluido en el Vertex Shader. A ejecutar en la GPU (tarjeta gráfica).	Código incluido en el Fragment Shader. A ejecutar en la GPU (tarjeta gráfica).

- Shaders
  - Los shaders utilizados en los programas escritos en Vulkan deben estar descritos en un lenguaje objeto denominado SPIR-V. Esto es una diferencia importante respecto a otras APIS como OpenGL donde los shaders se cargan directamente en código fuente y deben ser compilados en tiempo de ejecución. El uso de SPIR-V permite evitar problemas de versiones del compilador del lenguaje de shaders de alto nivel y acelera el proceso de carga. Además, se pueden desarrollar compiladores de varios lenguajes (GLSL, HLSL, …).

- Shaders
  - La distribución del SDK de Vulkan incluye un compilador de GLSL a SPIR-V llamado *glslangValidator*.
  - Este compilador utiliza extensiones distintas para cada tipo de shader:
    - .vert - vertex shader
    - .tesc - tessellation control shader
    - .tese - tessellation evaluation shader
    - .geom - geometry shader
    - .frag - fragment shader
    - .comp - compute shader

```
glslangValidator.exe -V shader.vert
glslangValidator.exe -V shader.frag
```

- Shaders
  - El resultado de la compilación son ficheros en código SPIR-V con un nombre prefijado en función del tipo de shader:
    - vert.spv
    - tesc.spv
    - tese.spv
    - geom.spv
    - frag.spv
    - comp.spv

- Ejemplo de Vertex_Shader

```
#version 400

in vec3 VertexPosition;
in vec3 VertexColor;

out vec3 Color;

void main()
{
    Color = VertexColor;
    gl_Position = vec4( VertexPosition, 1.0);
}
```

- Ejemplo de Fragment_Shader

```
#version 400

in vec3 Color;

out vec4 FragmentColor;

void main()
{
    FragmentColor = vec4( Color, 1.0);
}
```

Índice

- La estructura básica de un shader sigue el siguiente esquema:

```
#version VERSION_DE_GLSL_UTILIZADA

DECLARACIONES

void main()
{
    CÓDIGO
}
```

Las declaraciones pueden ser:

- Declaraciones de variables de entrada (***in***):
  - Definen las entradas del shader.
  - Cada shader tiene además una serie de entradas predefinidas que no hay que declarar.
  - En el Vertex_Shader las entradas corresponden a los atributos de cada vértice.
  - En el Fragment_Shader las entradas corresponden a los atributos de cada pixel, calculados mediante interpolación de las salidas del Vertex_Shader.

Las declaraciones pueden ser:

- Declaraciones de variables de salida (***out***).
  - Definen las salidas del shader.
  - Cada shader tiene además una serie de salidas predefinidas que no hay que declarar.
  - Las salidas del Vertex_Shader deben corresponder a las entradas del Fragment_Shader. El proceso de linkado del programa verifica esta condición.
  - El Fragment_Shader debe tener una salida de tipo vec4 que corresponde al color del pixel.

Las declaraciones pueden ser:

- Declaraciones de variables uniformes (***uniform***):
  - Corresponden a variables que tienen un valor común en todas las ejecuciones del shader.
  - Por ejemplo, el Vertex_Shader se ejecuta sobre cada vértice de manera que en cada ejecución los valores de las variables de entrada serán diferentes (los valores de los atributos de cada vértice). Sin embargo las variables uniformes tendrán el mismo valor en todas las ejecuciones.
  - Se suelen utilizar, por ejemplo, para almacenar matrices de transformación de coordenadas.

Las declaraciones pueden ser:

- Declaraciones de funciones auxiliares:
  - Son funciones que pueden ser llamadas desde el bloque principal del shader.
  - Se declaran de la misma forma que las funciones normales de C.
- Declaraciones de subrutinas (***subrutine***):
  - Permiten declarar tipos de funciones auxiliares (interfaces).
  - Se utilizan para trabajar con "punteros a funciones".

- El bloque main() describe el código a ejecutar en el shader.
  - Debe asignar los valores de las variables de salida.
  - Puede utilizar los valores de las variables de entrada, de las variables uniformes o de variables locales definidas en el propio bloque.
  - Puede incluir llamadas a las funciones auxiliares declaradas en el shader.
  - Puede incluir llamadas a funciones predefinidas de GLSL.
  - El código es parecido al lenguaje C, aunque maneja tipos de datos optimizados para vectores y matrices y operadores específicos para estos tipos de datos.
  - Las instrucciones incluidas son **if**, **if-else, switch-case-default, for, while, do-while, return, break, continue** y **discard.**
  - La instrucción **discard** solo se puede utilizar en el Fragment Shader para descartar el fragmento.

- El cualificador **layout**
  - El cualificador *layout* puede colocarse delante de una definición para asignar propiedades a tener en cuenta por el compilador.
  - La sintaxis es

    layout(*propiedad*, *propiedad* = *valor*, …) *declaración*
  - En las variables de entrada del Vertex Shader se puede asignar la propiedad *location* que indica el índice del atributo declarado.
  - En los bloques de variables uniformes se puede asignar la propiedad *binding* que el índice usado para identificar el descriptor set asociado al bloque.

- Tipos de datos básicos en GLSL:
  - **void**: tipo de dato de las funciones que no devuelven nada.
  - **bool**: tipo de dato booleano. Puede tomar los valores **true** y **false**.
  - **int**: tipo de dato entero con signo de 32 bits.
  - **uint**: tipo de dato entero sin signo de 32 bits. Los literales de tipo uint se marcan con el sufijo 'u', por ejemplo, "255u".
  - **float**: tipo de datos en coma flotante de 32 bits. Los literales de tipo float no llevan sufijo. Por ejemplo, "0.5" es un valor de tipo float.
  - **double**: tipo de datos en coma flotante de 64 bits. Los literales de tipo double utilizan el sufijo 'LF'. Por ejemplo, "0.5LF" es un valor de tipo double.

- Tipos de datos vectoriales en GLSL:
  - **vec2, vec3, vec4**: tipos de datos que describen vectores de 2, 3 o 4 componentes de tipo *float*.
  - **dvec2, dvec3, dvec4**: tipos de datos que describen vectores de 2, 3 o 4 componentes de tipo *double*.
  - **ivec2, ivec3, ivec4**: tipos de datos que describen vectores de 2, 3 o 4 componentes de tipo *int*.
  - **uvec2, uvec3, uvec4**: tipos de datos que describen vectores de 2, 3 o 4 componentes de tipo *uint*.
  - **bvec2, bvec3, bvec4**: tipos de datos que describen vectores de 2, 3 o 4 componentes de tipo *bool*.
- Se puede acceder a los componentes del vector mediante los campos ***x***, ***y***, ***z*** y ***w***. Estos campos también se pueden nombrar como ***r,g,b,a*** o ***s,t,p,q***. Puede accederse a trozos del vector mediante los campos ***xy*** o ***xyz***.

- Tipos de datos matriciales en GLSL:
  - **mat2, mat3, mat4**: tipos de datos que describen matrices cuadradas 2x2, 3x3 y4x4 de tipo *float*.
  - **mat2x2, mat2x3, mat2x4, mat3x2, mat3x3, mat3x4, mat4x2, mat4x3, mat4x4**: tipos de datos que describen matrices de tipo float de distintos tamaños.
  - **dmat2, dmat3, dmat4**: tipos de datos que describen matrices cuadradas 2x2, 3x3 y4x4 de tipo *double*.
  - **dmat2x2, dmat2x3, dmat2x4, dmat3x2, dmat3x3, dmat3x4, dmat4x2, dmat4x3, dmat4x4**: tipos de datos que describen matrices de tipo *double* de distintos tamaños.
- Se puede acceder a los campos de una matriz como si fuera un array. Por ejemplo, si *m* es una variable de tipo *mat4*, *m[0]* es un valor de tipo *vec4* y *m[0][0]* es un valor de tipo *float*.

- Arrays:
  - Se pueden definir arrays sobre cualquier tipo de datos.
  - La declaración es de la forma "*tipo nombre [tamaño]*".
  - Por ejemplo, "*vec4 points[10];*" define un array de 10 vectores.
  - Tambien se puede dejar sin declarar el tamaño. "*vec4 points[];*"
- Estructuras:
  - Se pueden definir estructuras formadas por campos.

```
struct light {
  float intensity;
  vec3 position;
} lightVar;
```

- Tipos de datos opacos:
  - Se refieren a tipos de datos que almacenan objetos que no pueden ser modificados. Sólo se puede acceder a la información que contienen.
  - Las texturas se declaran como tipos de datos opacos denominados sampler. Existen diferentes tipos de datos para los diferentes tipos de texturas: ***sampler1D***, ***sampler2D***, ***sampler3D***, ***samplerCube***, ***sampler1DShadow***, ***sampler2DShadow***.
  - Las imágenes tambien se almacenan como tipos opacos. Se declaran como ***image1D*** o ***image2D***.

- Constructores:
  - Para inicializar un vector se puede utilizar una lista de valores de sus componentes. Por ejemplo, "*vec4 v = { 0.1, 0.2, 0.3, 0.4 }; *"
  - También se puede utilizar un constructor con los valores de sus componentes. Por ejemplo, "*vec4 v = vec4(0.1, 0.2, 0.3, 0.4);*".
  - Si se utiliza un único número en el constructor, todos los componentes se asignan a ese valor. Por ejemplo, "*vec4 v = vec4(1.0);*".
  - Los constructores de vectores tambien pueden utilizar como argumento a otros vectores de mayor tamaño. En ese caso se eliminan las componentes sobrantes. Por ejemplo, "*vec4 v = vec4(0.1,0.2,0.3,0.4); vec3 w = vec3(v); *". El vector *w* solo copia los tres primeros componentes.
  - Hay versiones de constructores que toman como argumento vectores mas pequeños y los componentes que falta. Por ejemplo, "*vec3 w=vec3(0.1,0.2,0.3); vec4 v = vec4(w, 0.4);*"

- Constructores:
  - Para inicializar matrices se pueden utilizar listas de listas. En ese caso cada lista corresponde a una columna de la matriz. Por ejemplo, "*mat2 m = {{0.1,0.2},{0.3,0.4}};*".
  - Se puede inicializar una matriz diagonal indicando un único valor en el constructor. Por ejemplo, "*mat3 m = mat3(1.0);*".
  - También se pueden utilizar constructores basados en vectores. Cada vector corresponde al contenido de una columna. Por ejemplo, "*mat3(vec3,vec3,vec3)*".
  - Se pueden utilizar constructores basados en una única lista de componentes. En ese caso se asume que la matriz se rellena por columnas. Por ejemplo, "*mat2 m = mat2(0.1,0.2,0.3,0.4);*" es lo mismo que "*mat2 m = {{0.1,0.2},{0.3,0.4}};*".

- Operadores
  - Los operadores aritméticos incluidos son la suma (+), la resta (-), la multipliación (*), la división (/) y el módulo (%). El módulo solo se aplica a datos *int* y *uint*.
  - Estos operadores se pueden aplicar a dos escalares, con el significado habitual.
  - Si se aplican a un escalar y un vector, el resultado es un vector y el operador se aplica de manera independiente a cada componente. Por ejemplo, "*2*vec3(1.0,2.0,3.0)*".
  - Si se aplican a un escalar y una matriz, el resultado es una matriz y el operador se aplica de manera independiente a cada componente. Por ejemplo, "*mat3(1.0) / 2*".
  - Si se aplican a dos vectores (del mismo tamaño) el operador se aplica componente a componente. Por ejemplo, "*vec3(1.0,2.0,3.0) + vec3(0.1,0.2,0.3)*".
  - Si se aplica entre dos matrices, los operadores suma (+), resta (-) y división (/) se aplican componente a componente. Si embargo, el producto (*) aplicado a matrices o a un vector y una matriz se trata como un producto matricial.
  - Para calcular el producto escalar de dos vectores se utiliza la función *dot()*. Para calcular el producto vectorial se utiliza la función *cross()*.

- El código de los shaders no permite incluir librerías externas. Sin embargo, el estándar incluye un gran número de funciones predefinidas que pueden incluirse en el código. (Ver la descripción de las funciones en el manual)

Angle and Trigonometry Functions					
radians	degrees	sin	cos	tan	asin
acos	atan	sinh	cosh	tanh	asinh
acosh	atanh				

Exponential Functions					
pow	exp	log	exp2	log2	sqrt
inversesqrt					

Common Functions					
abs	sign	floor	trunc	round	roundEven
ceil	fract	mod	modf	min	max
clamp	mix	step	smoothstep	isnan	isinf
floatBitsToInt	intBitsToFloat	fma	frexp	ldexp	

Floating-Point Pack and Unpack Functions			
packUnorm2x16	packSnorm2x16	packUnorm4x8	packSnorm4x8
unpackUnorm2x16	unpackSnorm2x16	unpackUnorm4x8	unpackSnorm4x8
packDouble2x32	unpackDouble2x32	packHalf2x16	unpackHalf2x16

Geometric Functions				
length	distance	dot	cross	normalize
faceforward	reflect	refract	ftransform	

Matrix Functions				
matrixCompMult	outerProduct	transpose	determinant	inverse

Vector Relational Functions				
lessThan	greaterThan	equal	all	any
lessThanEqual	greaterThanEqual	notEqual	not	

Integer Functions				
uaddCarry	umulExtended	bitfieldInsert	bitfieldReverse	findLSB
usubBorrow	imulExtended	bitfieldExtract	bitCount	findMSB

Texture Functions			
textureSize	textureSamples	textureQueryLod	textureQueryLevels
texture	textureProj	textureLod	textureProjLod
textureOffset	textureProjOffset	textureLodOffset	textureProjLodOffset
textureGrad	textureProjGrad	textureGather	texelFetch
textureGradOffset	textureProjGradOffset	textureGatherOffset	texelFetchOffset
texture1D	texture1DProj	texture1DLod	texture1DProjLod
texture2D	texture2DProj	texture2DLod	texture2DProjLod
texture3D	texture3DProj	texture3DLod	texture3DProjLod
textureCube	textureCubeLod		

Atomic Counter Functions		
atomicCounterIncrement	atomicCounterDecrement	atomicCounter

Atomic Memory Functions				
atomicAdd	atomicMin	atomicMax	atomicAnd	atomicOr
atomicXor	atomicExchange	atomicCompSwap		

Image Functions			
imageSize	imageSamples	imageLoad	imageStore
imageAtomicAdd	imageAtomicMin	imageAtomicMax	imageAtomicAnd
imageAtomicOr	imageAtomicXor	imageAtomicExchange	imageAtomicCompSwap

Fragment Processing Functions		
dFdx	dFdy	fwidth
dFdxFine	dFdyFine	fwidthFine
dFdxCoarse	dFdyCoarse	fwidthCoarse
interpolateAtCentroid	interpolateAtSample	interpolateAtOffset

Geometry Shader Functions			
EmitStreamVertex	EndStreamPrimitive	EmitVertex	EndPrimitive

Shader Control Functions			
barrier	memoryBarrier	memoryBarrierAtomicCounter	memoryBarrierBuffer
memoryBarrierShared		memoryBarrierImage	groupMemoryBarrier

- OpenGL Mathematics (GLM)
  - Es una biblioteca escrita en C++ basada en la especificación de GLSL, que implementa los tipos de datos y las funciones de GLSL para su uso en los programas de OpenGL o Vulkan.
  - Por ejemplo, define los tipos *glm::mat3* o *glm::vec4* para tratar con matrices o vectores.
  - La biblioteca contiene además muchas funciones para tratar con vectores y matrices. Por ejemplo, *glm::frustum()* o *glm::rotate()*.
  - Para usar la biblioteca tan solo es necesario copiarla en el directorio *include* del compilador.