Shader Framework implementation for the management of multiple effects

Manuel Iglesias*, Lucas Guaycochea*†, María Victoria Galan*, y Javier Luiso*
*Instituto de Investigaciones Científicas y Técnicas para la Defensa
†Universidad Nacional de General Sarmiento

***Resumen*—Nowadays a wide range of rendering techniques are used in computer graphics development, each of them running in a shader program. Techniques combinations force graphics engines to manage a great number of permutations that increase exponencially. In this article we introduce an effects framework to manage orderly, maintainable and extensively all the available shaders, which was developed and integrated with our graphics engine. This solution improves code modularity and reutilization, and consists in an algorithm implementation to generate entires shaders from a set of reusables functions.**

***Index Terms*—computer graphics, rendering, shader, framework, automatization, integration**

I. Introducción

Shading es el término utilizado para la operación de determinar el efecto de las luces sobre la superficie de los objetos de la escena para producir imágenes realistas. En un comienzo, estas operaciones, y las demás necesarias para generar una secuencia de imagenes a partir de un modelado 3D de una escena, se realizaban en un hardware gráfico especializado que definía un pipeline fijo de etapas. Este pipeline era configurable, pero las operaciones que se realizaban eran siempre las mismas y en el mismo orden. Con la llegada de las tarjetas gráficas programables (programmable graphics processor units or GPUs), algunas de estas etapas permitieron la ejecución de programas definidos por programadores a los que se les llamó *shaders*. Esta flexibilidad permitió la aparición de múltiples técnicas para generar nuevos tipos de efectos visuales, dando lugar a múltiples variantes en cuanto a rendimientos, combinaciones y mejoras de distintos detalles.

Los shaders se programan utilizando código del estilo C, en lenguajes denominados HLSL, Cg y GLSL y generalizados como *shading languages*. Estos programas son compilados a un lenguaje intermedio independiente de la arquitectura; luego, a partir de los drivers es convertido al lenguaje de ensamblador de la arquitectura que corresponda en la ejecución [1]. Estos programas son ejecutados eficientemente en los procesadores gráficos de las GPUs, por lo cual la programación presenta algunas restricciones y diferencias respecto de la programación que genera programas a ser ejecutados por unidades de procesamiento de propósitos generales.

La cantidad y flexibilidad del conjunto de instrucciones disponibles para la programación de shaders ha ido evolucionando desde su primer versión en código de ensamblador a principios de la década del 2000. A las distintas versiones de capacidades de programación tanto del lenguage utilizado (instrucciones disponibles) como de las etapas programables dentro del pipeline gráfico se la denominó *shader model*. Este modelo ha ido evolucionando junto con el desarrollo y las optimizaciones de hardware gráfico. Al comienzo los programas estaban limitados en instrucciones, acotados en cuanto a la disponibilidad de registros, imposibilitados de flujos de control. En la actualidad, en la reciente sexta versión de este modelo de shaders, todas esas limitaciones fueron superadas y hoy se cuenta con un modelo de programación uniforme para todas las etapas disponibles pudiendo incluso realizar distintas pasadas por el pipeline que se retroalimentan entre si.

La industria del software comenzó con máquinas gigantes y código ilegible que evolucionó en la programación estructurada y procedimental en los 60s y 70s [2], a cada evolución se la llamó un nuevo paradigma de programación y fue realizada en la dirección de lograr mayor abstracción y facilidad en la reutilización de código, habilitando mecanismos para el encapsulamiento y modularización de lo programado. Esto permitió, en los distintos paradigmas, generar bibliotecas de código que representan módulos que resuelven aspectos particulares de un problema. La ingeniería de software es la encargada de trabajar sobre la generación y mantenimiento de éstas para luego dar lugar a la generación de soluciones de software a partir de la integración de bibliotecas adecuadas según el problema que se esté enfrentando. Paradigmas actuales son la programación orientada a objetos [3] y la programación orientada a aspectos [2].

Para recapitular, introducir la posibilidad de programar distintas etapas del pipeline permitió la generación de distintos algoritmos y técnicas que combinan programas a nivel de vértices y a nivel de pixeles, los cuales producen distintos efectos visuales cada vez con mayor grado de realismo. Sin embargo, las casi dos décadas de desarrollo en programación de shaders aún no han hecho viable un mecanismo de programación de mayor nivel de abstracción que permita la modularización y la reutilización del código. Las posibilidades en cuanto a la programación de shaders apenas alcanzan caracterísitcas de la programación procedimental y estructurada, sin poder así generar y reutilizar código escrito como un conjunto de funciones en distintos archivos. A todo motor gráfico se le presenta el desafío de implementar y disponer de la mayor cantidad posible de todos de la amplia variedad de efectos existentes y por existir. Por este motivo, se han desarrollado distintos métodos *ad hoc* para administrar conjuntos de shaders, éstos

se repasarán en la segunda sección.

Nuestro equipo crea simuladores de entrenamiento que usan como motor de gráficos y de juegos, lo que llamamos Plataforma de Desarrollos 3D Distribuidos o P3D [4], desarrollo propio iniciado hace 10 años que es usado únicamente en nuestros simuladores. Hoy en día el motor se encuentra en una etapa de consolidación, con el objetivo de lanzar la versión 2.0 a mediados del 2019 y extender su utilización a terceros. Esto generó la necesidad de encontrar un mecanismo para facilitar la implementación y mantenimiento de nuevos efectos. Para ello desarrollamos un framework de efectos o shaders a incorporar en nuestra P3D. Dicha implementación está basada en lo propuesto por el artículo que expone el "Shader Shaker" [5] presentado por la empresa Fishing Cactus, e incluye algunas de las posibles mejoras sugeridas por el mismo. Este trabajo detalla la manera en que adaptamos y desarrollamos esta idea en nuestra plataforma y se muestran los detalles que conlleva toda implementación al momento de obtener una solución cerrada e utilizable en producción.

II. Trabajos previos

El artículo escrito por O'Rorke [6] expone el desafío que nos encontramos al querer desarrollar un motor gráfico que incluya distintos algoritmos de renderizado implementados a través de shaders. El desafío consiste en lidiar con el problema de la cantidad de permutaciones resultantes de las varias técnicas disponibles. El autor denota que abunda la documentación acerca de las APIs disponibles tanto para escribir shaders como para cargarlos [7] pero pocos trabajos discuten qué es necesario para que una aplicación administre de manera robusta y eficiente los diferentes shaders. Esto mismo implica dedicar al motor gráfico un esfuerzo considerable para facilitar un manejo flexible de los shaders. Sin embargo, algunos trabajos han sido publicados al respecto, y Shawn Kime publica en su blog [8] una propuesta como resultado de su experiencia al enfrentar este desafío. Dicha propuesta plantea tres categorías para clasificar las soluciones posibles.

- **Reutilizar código.** Esta categoría es especificada como la opción de escribir distintas funciones reutilizables y luego escribir la función principal de cada shader invocando a las mismas. Si bien la reutilización de código está presente, los desarrolladores deben ir proveyendo las distintas funciones principales según las permutaciones que quieran dejarse disponibles. Esto requiere un esfuerzo manual y además es poco transparente y mantenible.
- **Soluciones sustractivas.** Es un término elegante para referirse a la utilización de un super-shader [9] que filtra las funciones habilitadas o deshabilitadas a partir de instrucciones de precompilador. Esta alternativa implica tener un shader de cientos o más lineas de código intercalado con directivas de precompilador lo cual lo hace totalmente inmantenible. Esta solución es muy popular y fue la adoptada por el grupo de trabajo en una primera migración de la plataforma que pasó de trabajar con el pipeline gráfico fijo a trabajar con shaders.
- **Soluciones aditivas.** En contraposición al tipo de solución anterior, se han desarrollados trabajos [10] que utilizan un árbol de funciones de shaders, identificadas con sus parámetros de entrada y salidas, y según el efecto deseado se arma el código del shader adecuado a partir de recorrer el árbol de funciones. La principal desventaja de los primeros trabajos que propusieron esta alternativa es la dificultad en el control de la eficiencia del código generado, precisando muchas veces una intervención de un programador que optimice el mismo.

Finalmente, el trabajo presentado en [5] toma la idea propuesta por Wolfgang Engel [11] de mantener un workflow de generación de shaders a partir de una biblioteca de archivos que proveen funciones aisladas que resuelven un aspecto particular (por ejemplo, cálculo de iluminación, transformación de normales, etc.) y le incopora la resolución de la interconexión de las funciones de manera automática a partir del uso de semánticas, como las utilizan Trapp y Döllner [12] pero para la generación de un super shader.

III. Fundamentos de la solución

El mantenimiento de shaders en un ambiente de producción suele ser complejo debido a la gran cantidad de técnicas de renderizado que se gestionan. Esto hace que el número de combinaciones de shaders crezca exponencialmente. En el artículo [5] se describe una solución para desarrollar y mantener eficientemente las posibles permutaciones de fragmentos. La técnica propuesta produce shaders automáticamente a partir de un conjunto de fragmentos independientes, cada uno encargado de una sola característica.

Nuestra solución toma [5] como base y aporta la posibilidad de armar automáticamente no solo uno sino ambos shaders, vertex y pixel, como también la configuración completa de las etapas fijas del pipeline gráfico. La implementación agrega algunas optimizaciones (sugeridas en [5]) que permiten un mayor control del shader y una mejor performance: soporta semánticas para variables uniformes (cbuffer, sampler, etc) y permite agrupar variables, optimizando el uso de memoria. Los detalles de nuestra solución serán desarrollados en las secciones siguientes. Por último, el diseño realizado es también extensible para soportar las nuevas etapas que admiten shaders a partir del Shader Model 4 y 5 (tesselation, geometry y compute shaders).

Nuestra plataforma P3D ha sido desarrollada sobre la biblioteca gráfica DirectX. La solución adoptada trabaja con el lenguaje **`HLSL`** el cual es el lenguaje de shaders nativo de DirectX. **`HLSL`** soporta la característica *semánticas definidas por usuario* que son etiquetas que sirven para nombrar los parámetros de una función. Dicha etiqueta es una palabra (`'string'`) usada para etiquetar entradas, salidas y variables de un fragmento. Esta opción es la pieza clave del generador, esta etiqueta se usa para conectar la entrada de un fragmento con la salida de otro.

Por otra parte, llamamos fragmento, no a la estructura de datos utilizada para generar la información de un pixel, sino a un archivo escrito en **`HLSL`** con una única función que

es responsable de implementar una característica, y contiene toda la información requerida para su ejecución incluyendo declaraciones de variables y lógica del código. Un ejemplo de fragmento en este contexto es el siguiente:

```
matrix matriz_normal : OBJETO_MATRIZ_NORMAL;

float3 modelViewNormal(in float3 normal :
    NORMAL) : ModelViewNormal
{
    return normalize(mul(normal, (float3x3)
        matriz_normal));
}
```

Tomando lo propuesto por [5], el algoritmo, a partir de la lista de fragmentos disponibles, utiliza las semánticas para generar la lista de llamadas a funciones necesarias para que el shader calcule la información requerida como salida. Las funciones son ordenadas en un grafo, que va desde los atributos de entrada del vertex (nodos hoja) hasta las semánticas de salida (nodos hijos de la raiz). El algoritmo entonces decide cuáles fragmentos serán requeridos para generar el shader completo que calcule la información deseada. Los fragmentos son completamente desacoplados, es decir que pueden ser escritos sin tener en cuenta de dónde viene y hacia dónde va la información que manejan. Por ejemplo: para un fragmento que declara una función que necesita como **entrada** un parámetro con semántica *"MatrizMundoVistaProyeccion"*, la herramienta buscará otro fragmento que declare que su función genera como **salida** la semántica *"MatrizMundoVistaProyeccion"* para conectarla con el primer fragmento. En cualquier caso, el fragmento que usa la semántica desconoce de donde viene su valor. El generador adopta una estructura de **compilador**, pasando por las siguientes fases:

- Los fragmentos **`HLSL`** son analizados para generar una *estructura* que guarde su información.
- Las *estructuras* son procesadas para generar un único *Árbol de Sintaxis Abstracta (ASA)* final que contenga todo el código necesario (funciones/uniformes/samplers). Para generar el *ASA* final, el algoritmo empieza por las semánticas de salida requeridas, y arma el árbol conectado los fragmentos cuyas semánticas de salidas y semanticas de entrada coinciden.
- El *ASA* final se convierte a lenguaje **`HLSL`**.

IV. Generación de shaders

La implementación del algoritmo generador de shaders sigue en lineas generales el diseño descripto en [5]. En particular, se decidió modificar la estructura del *ASA* para simplificar el acceso a la información. También se decidió utilizar archivos auxiliares generados offline que contienen datos necesarios para generar el *ASA*.

Los pasos descriptos en los puntos, IV-A IV-B, IV-C y IV-D (Figura 1) se ejecutan en dos veces: primero para generar el pixel-shader, y luego para generar el vertex-shader. Esta división en fases brinda mayor control sobre el proceso de generación.

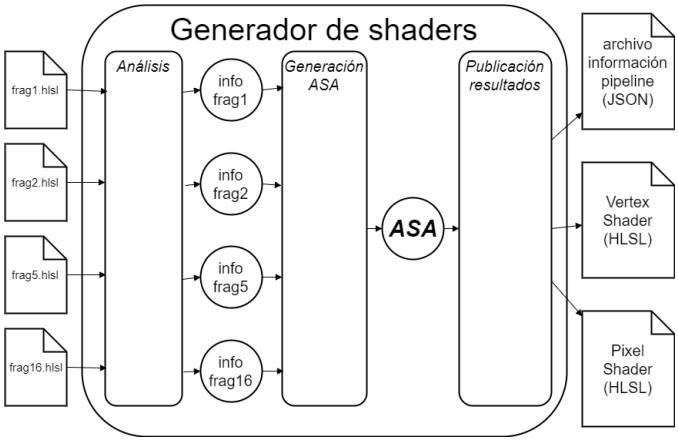

Figura 1. Proceso completo de generación de shaders e información de pipeline.

IV-A. *Parámetros de entrada*

El generador necesita como entrada: una lista de fragmentos a usar, las semánticas de salida esperadas del shader, y semánticas de entrada disponibles para el shader.

Para la generación del pixel-shader, el sistema usa como entrada:

- **lista de fragmentos a usar**: fragmentos de pixel-shader.
- **semánticas de salidas esperadas del shader**: salida deseada por el usuario, usualmente, en un rendering directo, el color final del fragmento o pixel.
- **semánticas de entrada disponibles para el shader**: todas las posibles semánticas que puede brindar el vertex-shader, en la práctica es igual a todos los fragmentos del vertex-shader (cada uno da información de una semántica).

Para la generación del vertex-shader, el sistema usa como entrada:

- **lista de fragmentos a usar**: fragmentos de vertex-shader.
- **semánticas de salidas esperadas del shader**: semánticas de entrada requeridas por el pixel-shader.
- **semánticas de entrada disponibles para el shader**: las semánticas de sistema que corresponden a los parámetros del vertex-shader.

La salida del vertex shader depende únicamente de la entrada requerida por el pixel shader, por lo que es necesario que se cumpla el orden de ejecución: primero se genera el pixel shader, y luego el vertex shader.

IV-B. *Análisis de fragmentos*

La primer etapa del algoritmo es el análisis de los fragmentos para identificar su información y guardarla en una estructura de datos, que luego se usará tanto para la generación del *ASA* como para la publicación del shader resultante. La información recolectada de cada fragmento es:

- **definición de función**: estructura con toda la información de la definición de función del fragmento: *nombre de*

la función, tipo de variable de retorno y semántica de variable de retorno.

- **parámetros de la función**: para cada parámetro de la función se obtiene: *referencia ('in', 'out' o 'inout'), nombre, tipo, semántica definida por usuario, semántica sistema e index*. Los parámetros se agrupan según su referencia en cuanto sean de entrada, de salida o de entrada/salida.
- **código completo:** string con todo el contenido del fragmento
- **bloque de código de función:** string con el código de la función del fragmento
- **declaraciones**: Las declaraciones de las variables uniformes que utiliza la función en su código, relevadas de las siguientes estructuras y tipos:
  - **cbuffers**: agrupación de variables uniformes
  - **struct**: estructura de variables uniformes.
  - **variables textura:** diferenciando entre sus tipos posible: 2D o 2DArray.
  - **variables sampler:** estructuras utilizadas para la toma de muestra de valores de una textura.
  - **variables uniformes:** sueltas o no agrupadas.

IV-C. Algoritmo A y generación del ASA*

El objetivo es encontrar un camino que vaya desde las semánticas de salida deseadas hasta las semánticas de entrada requeridas. Para ello se utiliza una estrategia basada en el conocido algoritmo A* [13] donde el camino se calcula usando dos conjuntos: semánticas abiertas (SA) y semánticas cerradas (SC). Los nodos del **ASA** contienen cada uno a un solo fragmento. Hay tres tipos de nodos:

- **Nodo Principal (NP)**: es el nodo raíz del **ASA**. Contiene la información de las semánticas de entrada requeridas. No contiene ningún fragmento.
- **Nodo Semántica salida (NSS)**: son nodos hijos del Nodo Principal que contienen cada uno un fragmento correspondiente a una semántica de salida.
- **Nodo Fragmento (NF)**: son nodos hijos de los NSS, y se extienden hasta las hojas del árbol.

Inicialmente se cumplen las siguientes condiciones:

1. el conjunto de **SA** es igual a las semánticas de salida esperadas
2. conjunto **SC** esta vacío
3. la estructura del **ASA** es un **NP** que tiene como hijos tantos **NSS** como semánticas de salida haya. La definición de los fragmentos de los **NSS** esta vacía.

En cada ciclo del algoritmo A* se siguen los pasos:

1. se obtiene un fragmento de la lista de fragmentos.
2. se itera en la lista de SA hasta encontrar alguna que coincida con la salida del fragmento seleccionado.
3. se actualiza SA y SC: se eliminan de SA las semánticas de salida del fragmento seleccionado y se agregan a SC, y al mismo tiempo se agregan a SA las semánticas de entrada del fragmento seleccionado.
4. se relacionan los nodos del ASA:
   a) se crea un nuevo NF con el fragmento seleccionado.
   b) se buscan los NF de las semánticas de salida del fragmento.
   c) se les asigna como hijo el nuevo NF.
   d) se agrega el NF nuevo al mapa de nodos para que relacionen sus semánticas de entrada (que ahora están en SA).
5. se vuelva a iniciar la iteración de fragmentos (paso 1).

Un ciclo se completa cuando se analizan todos los fragmentos contra todas las semánticas abiertas iniciales. Si en el conjunto SA queda una o mas semánticas iguales a las que había antes de iniciar el ciclo, significa que quedaron semánticas sin definir. Esto implica una falla en el algoritmo que no puede continuar.

El algoritmo A* termina cuando el conjunto de SA queda vacío.

Como resultado final de una ejecución exitosa del algoritmo, se obtiene un ASA estructurado de forma tal que los fragmentos de los NF padres dependen de los fragmentos de los NF hijos, siendo el NP el nodo raiz que representa las semánticas de salida del shader.

IV-C1. Archivos auxiliares: Para la generación del **ASA**, se usan dos archivos auxiliares que contienen información necesaria:

1. *archivo con información de cada formato de vertex*: Detalla la etiqueta del **input layout**, los elementos de entrada y la semántica de salida del vertex-shader.
2. *archivo con información de tipo de cada semántica definida por usuario*: Detalla el tipo de dato, valor de indice máximo y su respectiva semántica de sistema (en caso que la tenga).

IV-D. Publicación de resultados

Se recorre de forma pre-order el **ASA** desde el nodo raíz para completar la información del shader. La estructura del shader a completar es:

- **nombre**: etiqueta del shader
- **fragmento shader**: contiene la información de definición del shader y las variables que serán usadas por los fragmentos.
- **fragmentos componentes**: lista de fragmentos necesarios para cumplir con la función del shader. Están ordenados según su orden de aparición dentro del shader. Es decir, el primer fragmento de todos es independiente de los demás fragmentos porque no necesita información de ningún otro, mientras que el último seguramente necesite variables que son calculadas en otros fragmentos, por lo tanto depende de ellos.

El último paso para la generación del shader es escribirlo en formato **`HLSL`** en un archivo de texto. El archivo del shader final se puede dividir en cuatro secciones

1. **variables**: se escriben todas las variables con sus respectivas semánticas. La variables se dividen en sub-secciones
   a) **texturas**: variables de tipo `Texture2D` y `Texture2DArray`. Cada tipo de textura

lleva una semántica identificadora, *TEXTURA* y *TEXTURA_ARRAY* respectivamente.

b) **samplers**: variable de tipo '`SamplerState`'. Lleva la semántica *SAMPLER*.

c) **structs**: estructuras de datos requeridas por los fragmentos.

d) **cbuffers**: conjuntos de variables que utilizan los fragmentos.

2. **fragmentos**: funciones que contienen el código de cada uno de los fragmentos.
3. **input**: struct con los parámetros de entrada del shader.
4. **función shader**: función principal del shader. La definición contiene un parámetro de entrada y los parámetros de salida del shader. El código de la función son llamadas a las funciones de la sección 2.

La información para escribir las secciones 1, 2 y 3 se obtienen del **fragmento shader**, mientras que la información para escribir 4 se obtiene de **fragmentos componentes**. Cada sección del código se escribe por separado porque son independientes, todas en un mismo y único archivo.

Una vez generado el shader, se compila usando `DirectX` para comprobar que sea válido.

V. Integración con la plataforma: comunicación **P3D** - pipeline gráfico

Una aplicación que utilice los shaders debe compilarlos, cargarlos en el dispositivo `DirectX` y crear las *texturas*, *samplers* y *cbuffers*. Ademas, debe gestionar el uso de distintos shaders y la actualización de sus respectivas variables (Figura 2). Para simplificar esta tarea, el **generador de shaders** crea un archivo de texto formato `JSON` para cada shader, que detalla información correspondiente al pipeline gráfico de cada efecto. Esto separa la `generación` de la `utilización` de shaders. El **`P3D`** contiene un módulo que representa al pipeline gráfico para gestionar el renderizado. Este módulo usa la información del `JSON` para levantar en memoria la representación del pipeline.

V-A. Estructura archivo 'información de pipeline'

El `JSON` exportado por el **generador de shaders** contiene información de las etapas **Input Assembler**, **Vertex Shader** y **Pixel Shader** del pipeline.

V-A1. Input Assembler:

- nombre
- **layout**
  - nombre
  - **elementos**

Los elementos tienen la información:

- semántica
- index
- tipo

V-A2. Vertex/Pixel Shader:

- nombre
- path hlsl
- **texturas_2D**

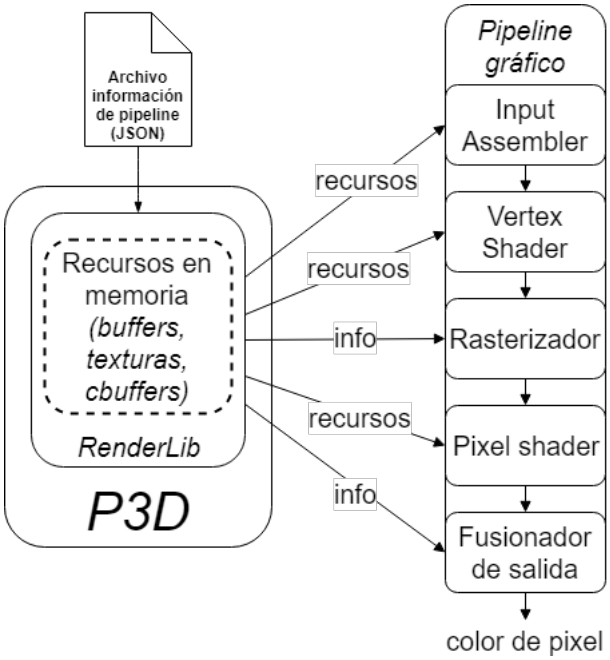

Figura 2. Comunicacion entre P3D y pipeline.

- **texturas_2D_array**
- **samplers**
- **cbuffers**
  - **variables**

Las variables de tipo *textura_2D*, *textura_2D_array*, *samplers* tiene la siguiente información:

- tipo
- nombre
- tamanio_array
- semántica
- tamanio
- tamanio_adaptado
- **register**
  - tipo
  - numero
  - sub-componente

los **cbuffers** tienen la misma información pero se agrega dos campos que indican si son pertenecientes al *objeto renderizado*, o a la *escena*.

- tipo
- nombre
- tamanio_array
- semántica
- tamanio
- tamanio_adaptado
- *de_escena*
- *de_objeto*
- **register**
  - tipo
  - numero
  - subcomponente

Las variables de los **cbuffers** tienen información similar, cambia la información de *register* por *packoffset*:

- tipo
- nombre
- tamanio_array
- semantica
- tamanio
- tamanio_adaptado
- ***packoffset***
  - numero
  - componente

*V-B. Representación del pipeline gráfico en la **P3D***

La clase *Efecto* contiene toda la información del pipeline gráfico necesaria para renderizar un determinado efecto. Cada efecto esta asociado a un archivo de información de pipeline. Los efectos se pueden ver como un conjunto de etapas de pipeline, donde cada etapa contiene **recursos en memoria** a través de los cuales la **P3D** le pasa información a los shaders.

La carga/descarga de efectos en el dispositivo `DX11` y su activación/desactivación (bindeo) es manejada por el *Gestor de Efectos*, que además se encarga de reiniciar los recursos luego de ser usados en cada ciclo de renderizado.

Los **layouts** (formatos de vertex) del **Input Assembler** están predefinidos y se crean cuando se inicia la aplicación. La creación, asignación y eliminación de estos **layouts** está a cargo del *Gestor de Layouts*. Cuando se crea un objeto, se le pide al *Gestor de Layouts* que le asigne su **layout** correspondiente, que va a depender del formato de vertex que tenga el mallado del objeto.

V-B1. Etapas del pipeline:

1. **Input Assembler**: contiene el *Layout*, *IndexBuffer* y *VertexBuffer* del mallado a renderizar. Cada objeto que se quiere renderizar debe primero pasarle al **Input Assembler** esta información.
2. **Vertex Shader**: contiene la información de los recursos del shader y los mapea en el `ID3D11VertexShader` creado.
3. **Rasterizador**: contiene el tipo de rasterizado (`ID3D11RasterizerState`) a utilizar.
4. **Pixel Shader**: contiene la información de los recursos del shader y los mapea en el `ID3D11PixelShader` creado.
5. **Fusionador de Salida**: contiene información del estado de mezcla (`ID3D11BlendState`) y del depth stencil (`ID3D11DepthStencilState`) del efecto.

V-B2. Recursos en memoria: La semántica definida por el usuario de cada variable es la clave para el mapeo de su información. De esta forma, para asignar el valor de una variable solamente hay que indicar su semántica y la tira de bytes a mapear.

- **CBuffer**: sirven para pasarle información al **Vertex Shader** y al **Pixel Shader**. Al tener semánticas tanto el cbuffer como sus variables, es posible mapear los datos de dos formas:
  1. directamente toda la información del **cbuffer**, es decir todas las variables como si fuese una sola porción de memoria.
  2. mapear cada una de las variables por separado.

  Para optimizar el rendimiento del renderizado, solo pueden haber dos tipo de **cbuffers** por cada shader (**Vertex** y **Pixel**):
  1. *de objeto*: su información se reinicia luego de renderizar un objeto. Por ejemplo, un **cbuffer** *de objeto* contiene las variables: *matriz_mundo*, *matriz_normales*, etc.
  2. *de escena*: su información se reinicia luego de renderizar la escena completa. Por ejemplo, un **cbuffer** *de escena* contiene las variables: *matriz_vista*, *luces*, etc.
- **IndexBuffer**: clásico buffer que contiene el índice de vértices a renderizar.
- **VertexBuffer**: clásico buffer que contiene la información de cada uno los vértices.
- **Sampler**: contiene información del *sampler* (`ID3D11SamplerState`) utilizado para mapear las texturas del objeto.
- **Textura2D**: contiene información de la vista del recurso (`ID3D11ShaderResourceView`) y de la textura (`ID3D11Texture2D`) creada en el dispositivo `DX`.
- **Textura2DArray**: contiene información de la vista del recurso (`ID3D11ShaderResourceView`) y de la textura `ID3D11Texture2D`) creada en el dispositivo `DX`.
- **VariableCBuffer**: contiene información de una variable del **cbuffer**. Este recurso *no se mapea directamente* en el shader, sino que sirve para mapear un **cbuffer** entero.

VI. Conclusiones y Trabajos futuros

Teniendo en cuenta la existencia de múltiples efectos que pueden utilizarse en el momento del renderizado de una escena en un motor gráfico, surge la necesidad de implementar una solución que permita administrar de manera eficiente el desarrollo y el uso de estos efectos. A partir de las limitaciones que presentan las APIs y los lenguajes para trabajar con programas de shaders, un motor gráfico necesita, en virtud de la mantenibilidad y escalabilidad, contar con un módulo o framework que facilite el trabajo con los shaders. En este artículo presentamos el trabajo desarrollado por nuestro equipo para implementar dicho framework.

Nuestro trabajo se basa en lo propuesto en ShaderShaker [5], lo adapta a nuestra plataforma **P3D** y lo extiende en algunos aspectos como los descriptos a lo largo del artículo, como ser, extensión del uso de semánticas a las variables uniformes o conjunto de ellas, y la utilización de la información disponible en los vertex-buffer junto al tipo de objeto que se desea renderizar. Vale aclarar que al basarnos en un diseño ya armado, lo más complejo fue adaptar ese diseño a nuestras necesidades para lograr integrarlo con la **P3D**.

El generador de shaders aumenta la productividad del nuestros desarrollos: reduciendo los tiempos de codificación, minimizando errores de compilación y estandarizando el estilo de escritura de shaders para una mejor lectura.

A continuación seguiremos trabajando en esta línea, un aspecto que nos interesa es vincular los shaders desarrollados a aplicaciones de diseño para que sean utilizados por los diseñadores, como ser el 3D Studio Max y el Blender, entre otros. Un trabajo interesante en este sentido es [14].

REFERENCIAS

[1] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering 3rd Edition*. Natick, MA, USA: A. K. Peters, Ltd., 2008.

[2] D. M. Simmonds, "The programming paradigm evolution," *Computer*, vol. 45, no. 6, pp. 93–95, June 2012.

[3] B. Stroustrup, "What is object-oriented programming?" *IEEE Software*, vol. 5, no. 3, pp. 10–20, May 1988.

[4] L. E. Guaycochea, J. E. Luiso, M. V. Galán, and H. A. Abbate, "Plataforma de desarrollos 3d distribuidos (p3d)," in *X Simposio de Informática en el Estado (SIE 2016)-JAIIO 45 (Tres de Febrero, 2016)*, 2016.

[5] M. Delva, J. Hamaide, and R. Ladlani, "Semantic-based shader generation using shader shaker," in *GPU Pro 6*, W. Engel, Ed. CRC Press, 2016, pp. 505–519.

[6] J. O'Rorke, "Integrating shaders into applications," in *GPU Gems*, R. Fernando, Ed. Addison-Wesley, 2004, pp. 601–615.

[7] "Tips and tricks for d3dx effects based renderer," in *Shader X4*, W. Engel, Ed., 2005.

[8] S. Kime. (2008) Shader permutations. [Online]. Available: https://shaunkime.wordpress.com/2008/06/25/shader-permutation/

[9] "The supershader," in *Shader X4*, W. Engel, Ed., 2005, pp. 485–486.

[10] M. McGuire, G. Stathis, H. Pfister, and S. Krishnamurthi, "Abstract shade trees (preprint)," in *Symposium on Interactive 3D Graphics and Games*, March 2006. [Online]. Available: http://www.cs.brown.edu/research/graphics/games/AbstractShadeTrees/index.html

[11] W. Engel. (2008) Shader workflow. [Online]. Available: http://diaryofagraphicsprogrammer.blogspot.com.ar/2008/09/shader-workflow.html

[12] M. Trapp and J. Döllner, "Automated combination of realtime shader programs," in *Proceedings of Eurographics 2007*, 2007, pp. 53–56.

[13] N. J. N. P. E. Hart and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems, Science, and Cybernetics*, vol. SSC-4, no. 2, pp. 100–107, 1968.

[14] "Case study: Designing a shader-subsystem for a next-gen graphics engine," in *Shader X4*, W. Engel, Ed., 2005, pp. 362–363.