Proyecto Fin de Carrera

OpenGL v4.3: Ejemplos de diseño de shaders e implementación mediante GPUs

Autor

Joaquín David Palomares García

Director

Dr. Francisco José Serón Arbeloa

Escuela de Ingeniería y Arquitectura

2015

Resumen

El objetivo de este Proyecto Fin de Carrera es el de mostrar la capacidad del proyectante para diseñar e implementar shaders para la libería de gráficos OpenGL en su versión 4.3.

Dichos shaders son pequeños programas que se ejecutan en la unidad de procesamiento gráfico (GPU) y aplican transformaciones y efectos especiales a la hora de renderizar una escena.

Para este PFC se ha creado una escena de muestra en tres dimensiones y se han utilizado distintas técnicas aplicadas en los shaders para modificarla en tiempo real. Concretamente, las técnicas aplicadas han sido:

- Mapas de desplazamiento: Una técnica que consiste en usar texturas con la información de la altura de la geometría para desplazar los vértices de la superficie texturizada.
- Luz ambiental: Una técnica que proporciona una iluminación global y homogénea a toda la escena.
- Luz difusa: Una técnica que permite reflejar una fuente de luz sobre una superficie en muchas direcciones.
- Mapas de sombras: Una técnica para reproducir las sombras creadas por las distintas fuentes de luz.
- Mapas normales: Una técnica que usa texturas para dar una iluminación y un relieve más detallados a la geometría de la escena.

Además del diseño e implementación de las shaders, se ha analizado su eficiencia en distintos tipos de GPU del fabricante Nvidia.

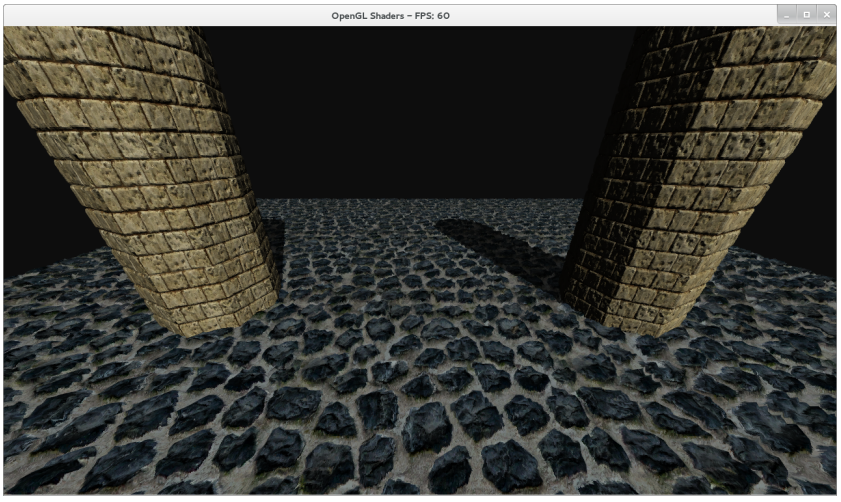


Figura 1: Escena final renderizada

Índice

Índice de figuras

Índice de códigos

Índice de cuadros

1. Introducción

1.1. Palabras clave que definen las herramientas en las que se basa este PFC

OpenGL (Open Graphics Library) es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D. La interfaz consiste en más de 250 funciones diferentes que pueden usarse para dibujar escenas tridimensionales complejas a partir de primitivas geométricas simples, tales como puntos, líneas y triángulos.

La tecnología basada en el uso de **shaders** es una tecnología reciente y que ha experimentado una gran evolución destinada a proporcionar al programador una interacción con la unidad de procesamiento gráfico (GPU) hasta ahora imposible. Los shaders son utilizados para realizar transformaciones y crear efectos especiales, como por ejemplo iluminación, fuego o niebla. Para su programación los shaders utilizan lenguajes específicos de alto nivel que permitan la independencia del hardware.

Unidad de procesamiento gráfico o **GPU** (Graphics Processing Unit) es un coprocesador dedicado al procesamiento de gráficos u operaciones de coma flotante que se utiliza para aligerar la carga de trabajo del procesador central en aplicaciones como los videojuegos o aplicaciones 3D interactivas. De esta forma, mientras gran parte de lo relacionado con los gráficos se procesa en la GPU, la unidad central de procesamiento (CPU) puede dedicarse a otro tipo de cálculos (como la inteligencia artificial o los cálculos mecánicos en el caso de los videojuegos).

1.2. Objetivos del proyecto

1. Diseño de una escena de muestra en 3D utilizando las librerías de OpenGL para C++.

2. Diseño de shaders utilizando el lenguaje GLSL que ofrece OpenGL en la versión 4.3. Dichos shaders modificarán distintos elementos de la escena (como pueda ser el suelo, paredes, columnas o cualquier otro elemento que pueda aparecer en ella). Para ello, según el elemento a modificar, se hará uso de diferentes técnicas gráficas, entre las que se pueden encontrar:

   - Mapas de desplazamiento
   - Mapas normales
   - Iluminación

3. Implementación de dichos shaders utilizando tarjetas Nvidia.

4. Análisis del comportamiento (eficiencia) de dichos shaders frente a diferentes tipos de tarjetas. Para ello se hará uso del programa Nvidia Nsight.

1.3. Resumen de las actividades para realizar el proyecto

1. Estudio de OpenGL v4.3, GLSL y Nvidia Nsight.

2. Diseño de la escena.

3. Diseño de los shaders.

4. Creación de los modelos, texturas y mapas que compondrán la escena y que usarán los shaders.

5. Implementación de la escena.

6. Implementación y validación de los shaders sobre GPUs Nvidia.

7. Análisis de eficiencia en diferentes tipos de GPUs.

8. Escritura de la memoria

1.4. Descripción de la estructura de la memoria

Diseño de los shaders (sección 2)
Diseño de los distintos shaders que ejecutará el programa para modificar la escena.

Diseño e implementación del programa (sección 3)
Diseño e implementación del programa que mostrará la escena.

Implementación y validación de los shaders (sección 4)
Implementación de los distintos shaders que ejecutará el programa para modificar la escena.

Análisis de eficiencia (sección 5)
Comparación de los distintos tipos de shaders elaborados en diferentes tarjetas gráficas.

Conclusiones (sección 6)
Valoración del trabajo realizado.

Horas de trabajo (sección 7)
Información detallada sobre el tiempo que ha llevado realizar este PFC.

Trabajo futuro (sección 8)
Posibilidades para continuar, ampliar y mejorar el trabajo desarrollado.

Referencias (sección 8)
Referencias y fuentes de información usadas para elaborar este PFC.

Anexo

- En la primera parte del anexo se da una visión de la pipeline de OpenGL v4.3 y de las diferentes etapas que la componen, incluyendo una explicación sobre cada una de las etapas programables usadas en este PFC (sección A del anexo).
- En la segunda parte del anexo se da información sobre las características de las diferentes tarjetas gráficas empleadas en los análisis de eficiencia. (sección B del anexo).

2. Diseño de los shaders

2.1. Conceptos básicos

Para entender mejor las técnicas aplicadas, a continuación se detallan algunos conceptos útiles que se emplean durante el desarrollo de shaders.

2.1.1. Espacios de coordenadas

Cuando se está trabajando en una aplicación en 3D, hay que tener en cuenta que la posición de un vértice puede ser relativa a distintos sistemas de coordenadas. Concretamente, la mayoría de las veces ese sistema pertenece a uno de estos cuatro:

- Coordenadas del modelo: los vértices de un modelo tienen una posición relativa al centro de dicho modelo.
- Coordenadas del mundo: los vértices de los objetos de la escena están definidos con relación al centro de la escena.
- Coordenadas de la cámara: la posición de los vértices es relativa a la cámara.
- Coordenadas homogéneas: la posición de los vértices es relativa a la pantalla donde se mostrarán.

Se hace pues necesario un sistema para pasar de un espacio de coordenadas a otro. Para ello, se emplean distintas matrices para cada uno de los cambios necesarios. Estas matrices se pueden ver en el diagrama de la figura 2.

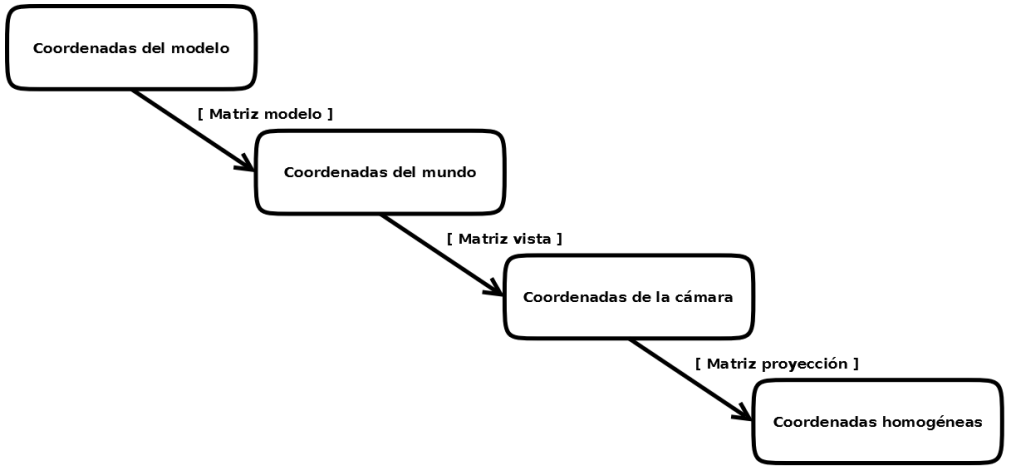


Figura 2: Diagrama de los espacios de coordenadas y sus matrices de cambio

La matriz de modelo permite situar un modelo en función de la escena. La matriz de vista contiene la posición donde se encuentra la cámara, así como la dirección y el sentido en que mira. La matriz de proyección pone la escena en función de los parámetros de la cámara, como el tipo de proyección (perspectiva, ortográfica...), el campo de visión, la anchura y altura de la visión o el rango de visión.

Todas estas matrices se pueden multiplicar para formar una sola, a la que llamaremos matriz MVP. La matriz MVP permite, al multiplicar el vértice de un modelo, ponerlo directamente en el espacio de coordenadas homogéneo.

Matemáticamente hablando, la matriz de modelo M se calcula como el resultado de todas las transformaciones que hay que hacer para pasar de las coordenadas del modelo a las coordenadas del mundo. Estas transformaciones pueden ser de traslación T, de rotación R o de escala S.

- La matriz de traslación moverá el punto en la dirección marcada por T_x, T_y y T_z:

$$T = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- La matrix de rotación R_x, R_y o R_z rotará sobre su respectivo eje α grados en sentido antihorario:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos(\alpha) & 0 & \sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- La matriz de escala escalará un vector en las direcciones de los ejes marcadas por S_x, S_y y S_z:

$$S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Combinando estas matrices de distintas formas podemos posicionar un modelo en nuestro mundo. La matriz resultante de dicha

combinación es la matriz de modelo M. El siguiente paso es convertir las coordenadas obtenidas a las coordenadas de la cámara usando la matriz de vista V.

La matriz de vista V es una variante de la matriz de modelo, ya que lo que haremos realmente es obtener la matriz que posiciona nuestra cámara en el mundo y luego obtener su inversa para poder convertir las coordenadas del mundo a las de la cámara. Los datos necesarios de la matriz de vista son la posición de la cámara (para poder situarla en el mundo), la dirección en que mira, la dirección que la cámara considera "hacia arriba" y la dirección que la cámara considera "hacia la derecha". Cambiando los valores de estas dos últimas direcciones se pueden conseguir efectos como la rotación de la cámara o simular un espejo. La matriz de vista V se define como:

$$V = \begin{bmatrix} derecha_x & arriba_x & mirada_x & 0 \\ derecha_y & arriba_y & mirada_y & 0 \\ derecha_z & arriba_z & mirada_z & 0 \\ -(posicion \cdot derecha) & -(posicion \cdot arriba) & -(posicion \cdot mirada) & 1 \end{bmatrix}$$

Por último, para pasar estas coordenadas de la cámara a las coordenadas homogéneas, multiplicaremos por la matriz de proyección P. Puesto que la matriz es diferente según si la proyección es en perspectiva u ortográfica, vamos a distinguir entre la matriz de proyección en perspectiva P_p y la matriz de proyección ortográfica P_o. En ambas usaremos la distancia más cercana y la distancia más lejana a la cámara (Z_n y Z_f respectivamente), pero puesto que la anchura y altura en la ortográfica es constante mientras que en la perspectiva cambia conforme nos alejamos de la cámara, como se puede comprobar en la figura 3, vamos a introducir el concepto de campo de visión o FoV (del inglés "field of view"), que indica el ángulo de la visión. Así pues, tenemos que:

$$P_p = \begin{bmatrix} \arctan(\frac{FoV_x}{2}) & 0 & 0 & 0 \\ 0 & \arctan(\frac{FoV_y}{2}) & 0 & 0 \\ 0 & 0 & -\frac{Z_f+Z_n}{Z_f-Z_n} & -\frac{2(Z_f Z_n)}{Z_f-Z_n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$P_o = \begin{bmatrix} \frac{1}{anchura} & 0 & 0 & 0 \\ 0 & \frac{1}{altura} & 0 & 0 \\ 0 & 0 & -\frac{2}{Z_f-Z_n} & -\frac{Z_f+Z_n}{Z_f-Z_n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

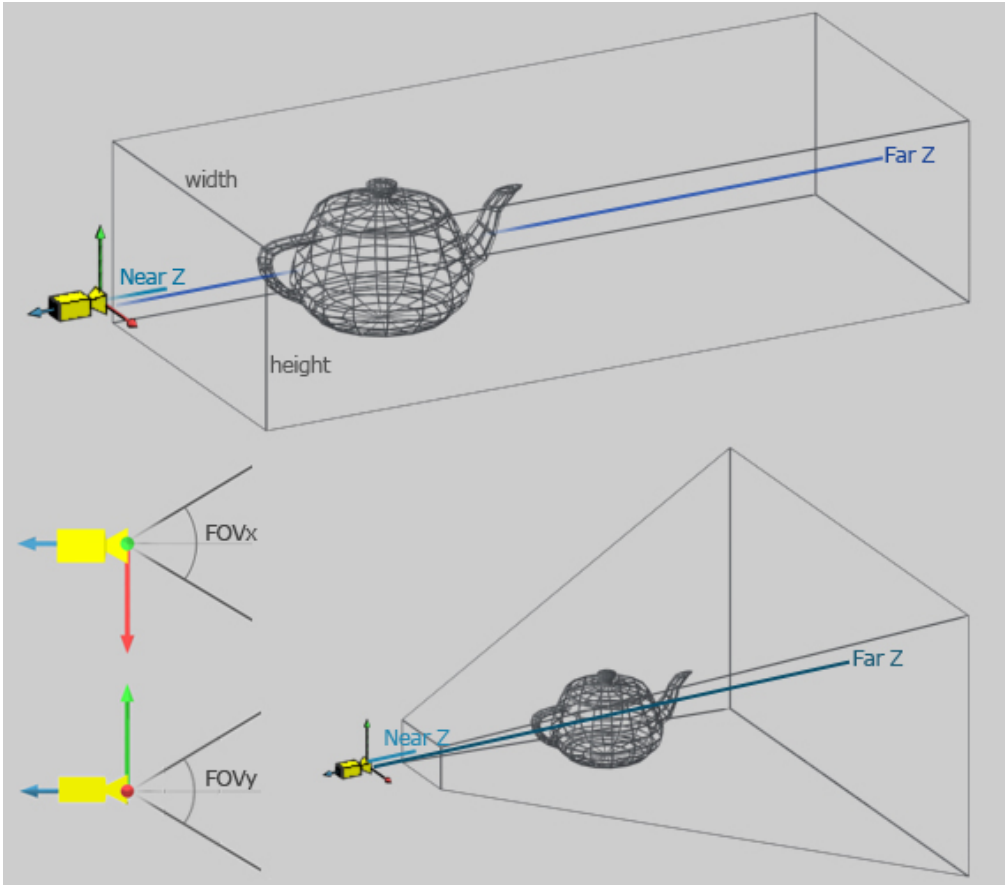


Figura 3: Proyección ortográfica y en perspectiva

Así pues, tenemos que multiplicando una posición por estas tres matrices M, V y P podemos pasar de las coordenadas del modelo p a las coordenadas homogéneas p' de la forma:

$$p' = P * V * M * p$$

Y agrupando estas tres matrices en una sola tenemos la matriz MVP:

$$MVP = P * V * M$$

En las figuras 4 y 5 se puede ver el cambio que sufren los objetos de la escena al multiplicar sus vértices por una matriz de proyección para pasar de las coordenadas de la cámara a las coordenadas homogéneas.

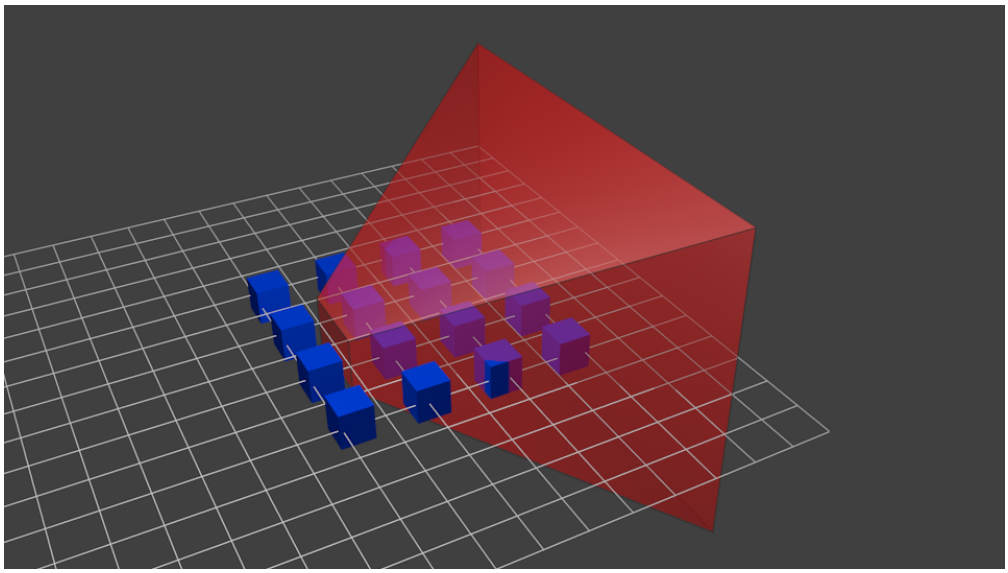


Figura 4: Una escena en el espacio de coordenadas de la cámara (en rojo)

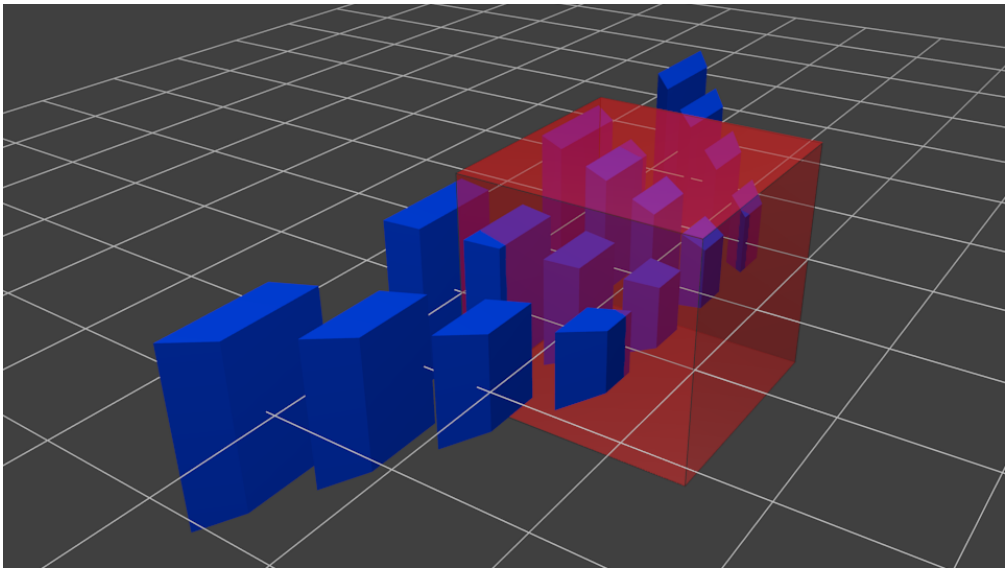


Figura 5: Una escena en el espacio de coordenadas homogéneas (de la pantalla, en rojo)

2.1.2. Composición de un modelo 3D

Los modelos 3D que usaremos están compuestos por triángulos. Cada triángulo del modelo se define como un conjunto de tres vértices, y cada vértice guarda la información de tres datos distintos: la posición del vértice en el espacio, la normal de la superficie del triángulo y el mapa UV.

La posición es un vector de cuatro elementos de la forma (X, Y, Z, W) donde "X", "Y" y "Z" indican la posición en dichos ejes y "W" es un valor constante 1 para indicar que es una posición en el espacio.

La normal de la superficie almacenada es la misma en los tres vértices de una cara y se guarda de la forma (X, Y, Z, W) donde "X", "Y" y "Z" indican la dirección en dichos ejes y "Wïndica que se trata de una dirección.

Los mapas UV sirven para proyectar una textura en 2D sobre un modelo en 3D. Para ello, a cada vértice se le asigna una posición (U, V) donde "U" denota el eje horizontal desde 0 hasta 1 de izquierda a derecha, y "V" indica el eje vertical desde 0 hasta 1 de abajo hacia arriba, tal como se puede observar en la figura 6

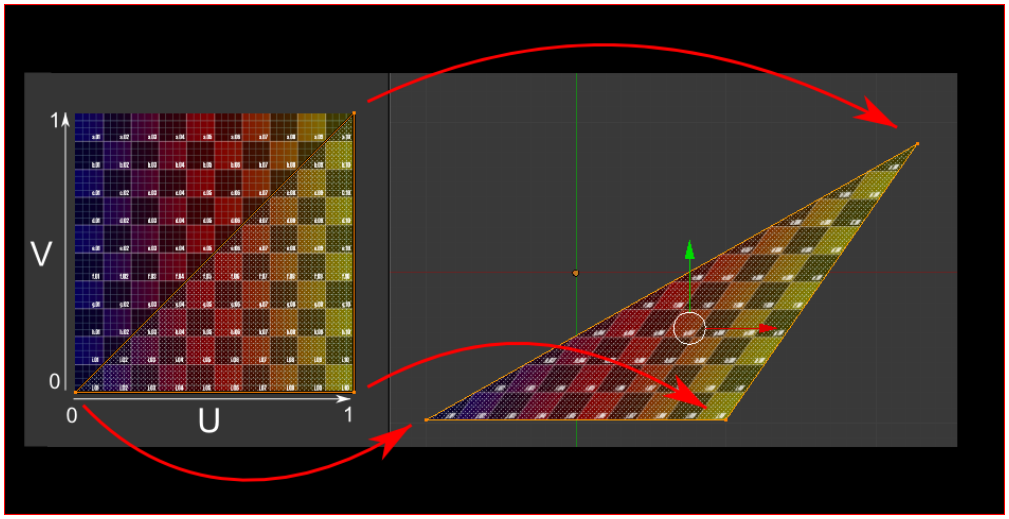


Figura 6: Proyección de una textura en un triángulo con los mapas UV

2.1.3. La pipeline de OpenGL

La pipeline de OpenGL marca el proceso que siguen los flujos de datos en su paso por la GPU. Estos datos atraviesan una serie de etapas, llamadas shaders, algunas de las cuales son programables. Estas shaders programables son:

- Vertex Shader: Procesa cada vértice individualmente.
- Tessellation Control Shader y Tessellation Evaluation Shader: Se encargan del proceso de dividir un parche (un conjunto de vértices) en un conjunto de triángulos más pequeños.
- Geometry Shader: Crea nueva geometría.

- Fragment Shader: Aplica el color a los píxeles que se mostrarán en pantalla.

Para una información más detallada de la pipeline y de las shaders programables, se puede consultar en el anexo A.

2.2. Mapas de desplazamiento

Los mapas de desplazamiento (displacement mapping) consisten en el uso de una textura que contiene en uno de sus canales RGB (normalmente en los tres) la información sobre el desplazamiento que tiene que seguir la superficie donde está aplicada. El desplazamiento se ejecuta a lo largo de la normal de la superficie.

Esta información se puede interpretar de varias maneras, por ejemplo, siendo el 0 negro y el 1 blanco, el negro puede significar que no hay desplazamiento y el blanco que hay un desplazamiento total, o puede sifnificar que el negro desplaza en dirección inversa a la normal y que un tono medio de 0.5 significaría no desplazar.

La figura 7 da una idea del aspecto de un mapa de desplazamiento comparado con la textura original.

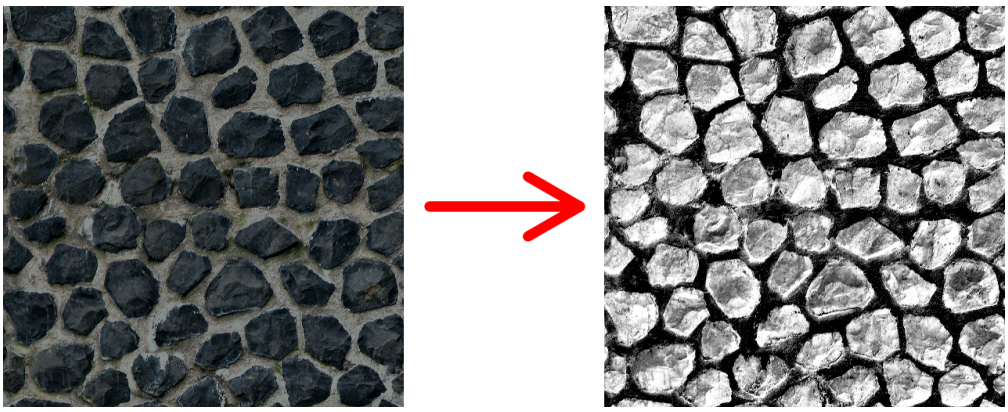


Figura 7: La textura original y su mapa de desplazamiento

La idea, por tanto, es desplazar los vértices de cada triángulo de la geometría según su posición en el mapa, pero surge un problema: con una geometría simple (por ejemplo, triángulos muy grandes) apenas hay puntos que desplazar. Es necesario entonces dividir dichos triángulos en triángulos más pequeños que seamos capaces de desplazar para poder conseguir un nivel de detalle mayor. Para ello podemos usar las etapas de teselación de la pipeline de OpenGL, Tesselation Control Shader (anexo A.2) y Tesselation Evaluation Shader (anexo A.3), que permiten dividir un triángulo en otros más pequeños según los valores que asignemos a cada arista y al interior.

OpenGL divide el triángulo en niveles interiores y exteriores y permite establecer sus valores individualmente, como se puede ver en la figura 8.

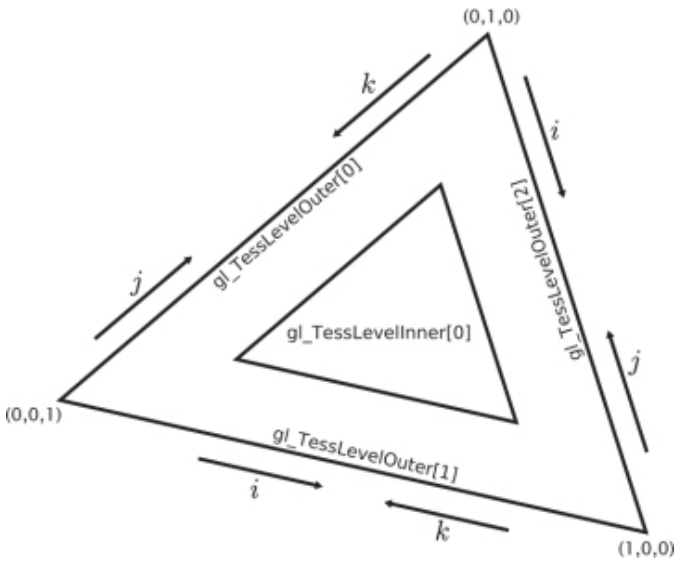


Figura 8: Niveles de teselación en OpenGL

Para establecer los valores de dichos niveles (o sea, la cantidad de triángulos en la que se dividirá el triángulo original) podemos darles un valor fijo, o, para mejorar el rendimiento, podemos darles un valor en función de la distancia de cada arista a la cámara.

Una vez tengamos dividido el triángulo ya podemos usar nuestro mapa de desplazamiento para desplazar los nuevos vértices creados. Para ello, simplemente cogeremos la posición de cada vértice y veremos en que parte del mapa de desplazamiento está situado, y lo desplazaremos a lo largo de la normal un porcentaje del factor total de desplazamiento dependiendo del tono de la textura.

Por último, hay que recordar que al desplazar los vértices hemos cambiado las normales de cada cara y es necesario recalcularlas usando las nuevas posiciones de los vértices.

En la figura 9 se puede ver el resultado de aplicar sobre una malla plana un mapa de desplazamiento, consiguiendo que los vértices de la malla se desplazen a lo largo de la normal según lo indicado en el mapa.

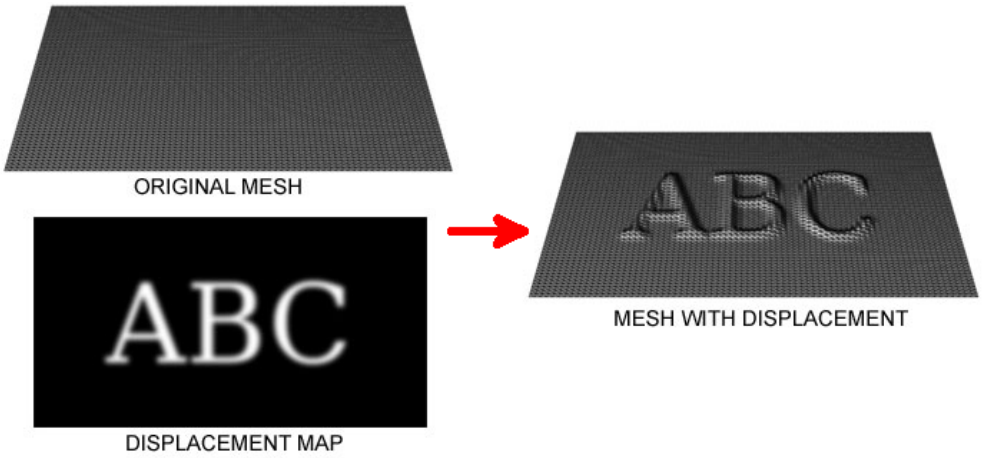


Figura 9: Mapa de desplazamiento aplicado sobre una malla plana

2.3. Iluminación

Para la iluminación se usarán dos técnicas diferentes en conjunto: la luz ambiental (ambient lighting) y la luz difusa (diffuse lighting).

2.3.1. Luz ambiental

La luz ambiental es el tipo de iluminación más simple y representa la luz que existe en el ambiente sin ningún otro tipo de iluminación directa. Es homogénea y de valor constante.

Para simular este tipo de luz, simplemente hay que ajustar el color en la etapa de Fragment Shader según la intensidad y color requeridos.

Para calcular la iluminación ambiental $I_{ambiental}$ en un punto dado esta depende del coeficiente de reflexión ambiental k_a (el color, en tres componentes RGB en el intervalo $[0, 1]$) y de la constante de intensidad ambiental I_a, de la forma:

$$I_{ambiental} = k_a I_a$$

2.3.2. Luz difusa

La luz difusa hace mención a la luz que es reflejada en una superficie en todas direcciones siguiendo la Ley de Lambert [6]. Dicha ley establece que la iluminación producida por una fuente luminosa sobre una superficie es directamente proporcional a la intensidad de la fuente y al coseno del ángulo que forma la normal a la superficie con la dirección de los rayos de luz y es inversamente proporcional al cuadrado de la distancia a dicha fuente.

En la figura 10 se puede observar la luz difusa reflejada en una superficie sobre la que incide una fuente de luz.

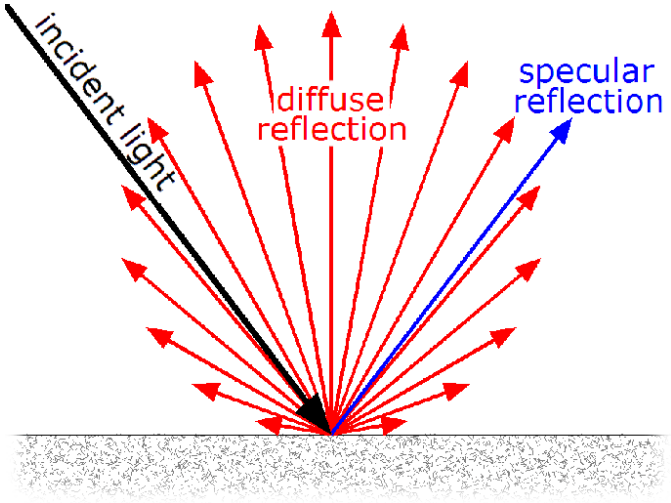


Figura 10: Reflexión de una fuente de luz

Para calcular la iluminación difusa de una fuente de luz en un punto tenemos que tener en cuenta pues el vector normal de la superficie N en el punto, el vector de incidencia de la luz L (el vector de la dirección de la fuente de luz al punto), la intensidad de la fuente de luz I_l y el coeficiente de reflexión difusa k_d. La iluminación difusa total será la suma de la iluminación difusa de cada fuente de luz:

$$I_{difusa} = \sum_i k_{di} I_{li} (N \cdot L_i) = \sum_i k_{di} I_{li} \cos \theta_i$$

Matemáticamente, podemos definir el modelo de iluminación como la suma de la iluminación ambiental y la iluminación difusa, tal que:

$$I = I_{ambiental} + I_{difusa} = k_a I_a + \sum_i k_{di} I_{li} (N \cdot L_i)$$

Por lo tanto para simular la luz solo tenemos que aplicar este modelo de iluminación Fragment Shader (anexo A.5).

2.4. Mapas de sombras

Los mapas de sombras (shadow mapping) son una de las diferentes técnicas que nos permiten simular las sombras que producen todas las luces en una escena.

La técnica consiste en renderizar cada fotograma de la escena en varias pasadas. Primero, haremos una pasada por cada fuente de luz existente en la escena, situando la cámara en dicha fuente de luz. En estas pasadas no nos interesa la salida de color (GL_COLOR_BUFFER_BIT) sino el buffer de profundidad (GL_DEPTH_BUFFER_BIT), también conocido como z-buffer, obtenido para cada fuente de luz. Seleccionando un buffer tenemos almacenada la distancia desde la cámara a cada punto renderizado de la escena visto desde la fuente de luz seleccionada. Al no utilizar el buffer de color nos ahorramos los cálculos del Fragment Shader. Este buffer lo podemos almacenar en el formato de una textura que servirá como dato al algoritmo de cálculo de sombras.

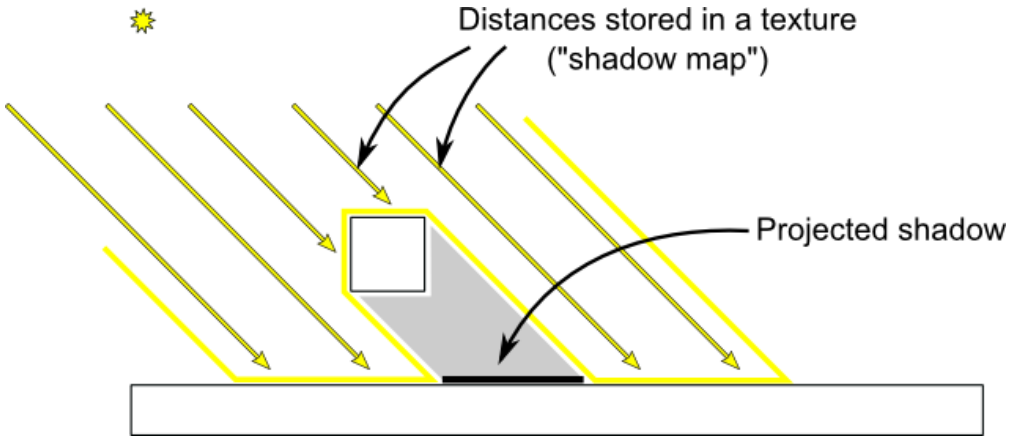


Figura 11: Ejemplo de cálculo del buffer de profundidad

Una vez calculados los z-buffer de la escena desde el punto de vista de cada fuente de luz, pasamos a renderizar la escena en sí como lo haríamos normalmente, con la cámara situada en la parte de la escena

que deseemos y suministrándole al Fragment Shader (anexo A.5) los z-buffer generados. A la hora de calcular la iluminación de una fuente de luz dada lo que haremos será relacionar el fragmento que estamos calculando con el z-buffer generado para decidir si hay sombra o no en dicho fragmento. Para ello, mediremos otra vez la distancia de la fuente de luz al fragmento dado (d_f) y la compararemos con la distancia de la luz almacenada en el z-buffer de dicha luz (d_b). Si la distancia $d_f > d_b$, el fragmento que estamos analizando está a la sombra.

En la figura 12 se puede observar el proceso de decidir si un fragmento P está a la sombra o no al comparar la profundidad almacenada en el z-buffer desde el punto de vista de la luz (Z_A) con el valor de profundidad del fragmento que estamos analizando visto desde la luz (Z_B).

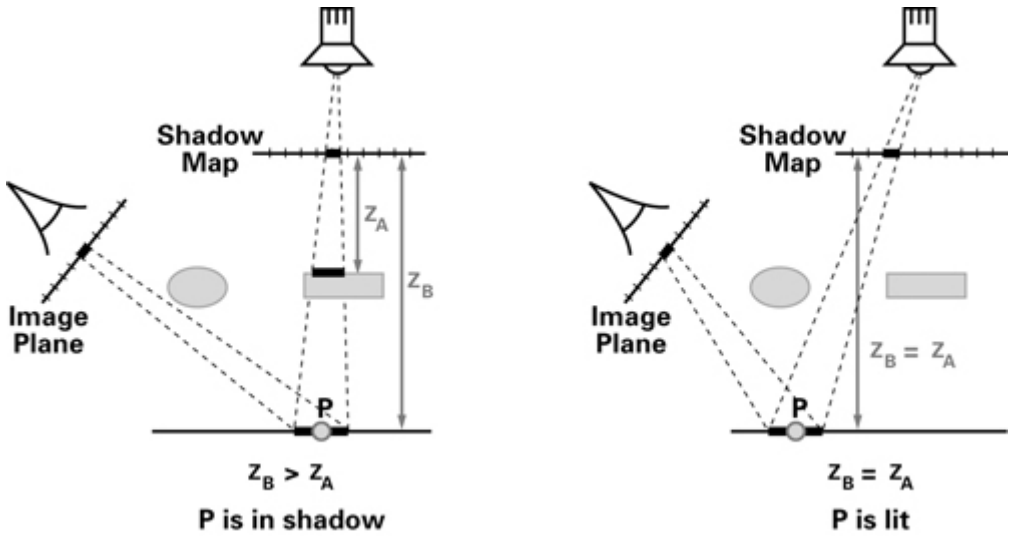


Figura 12: Ejemplo de análisis de la sombra de un fragmento

En esta técnica las sombras obtenidas son dinámicas, lo que quiere decir que se calculan para cada fotograma renderizado, al contario que en otras técnicas en las que se calcula de antemano las sombras

y luego se aplican a la escena sin posibilidad de moverlas o quitarlas (lo que se conoce como baked lighting o iluminación "cocinada").

Al tener que calcularse para cada fotograma, su rendimiento es mucho menor, pero como ventaja, nos permiten proyectar las sombras de modelos animados o las sombras producidas por luces móviles.

2.5. Mapas normales

Hemos hablado de que la luz difusa se refleja según el ángulo que forma con la normal de la superficie, pero existe un problema: cuanto mayor es la superficie, menor es el detalle conseguido, ya que toda la superficie comparte la misma normal.

Con la técnica de los mapas normales (normal mapping), se consigue dar más detalle y relieve a la iluminación de las superficies. La idea consiste en codificar la información de la normal en cada punto de una superficie dentro de los canales RGB de una textura, de tal manera que el canal rojo contiene la componente X de la normal, el verde tiene la componente Y, y el azul la componente Z. Puesto que los colores RGB van de 0 a 1 (nada de color a todo el color), el valor de 0 significa que la normal en dicho punto es -1, el valor de 1 indica una normal de +1, y el valor intermedio de 0.5 indica una normal en esa componente de valor 0.

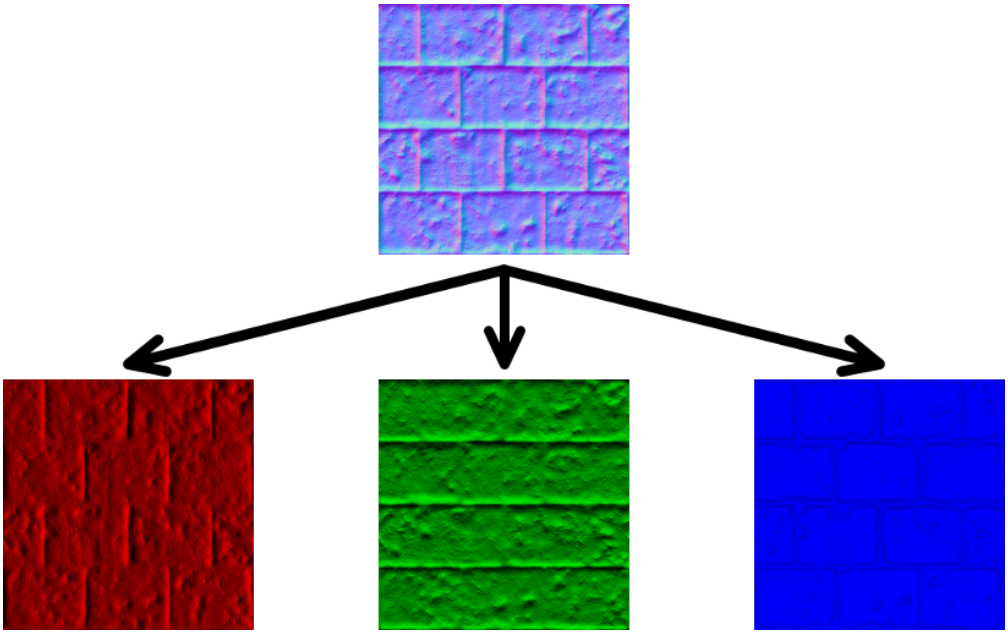


Figura 13: Descomposición de un mapa normal en los canales RGB

A la hora de renderizar un fragmento de la escena, simplemente obtenemos la información de la normal del mapa normal en vez de usar la normal de la superficie y aplicamos la luz difusa.

En la figura 14 podemos ver el efecto resultante de la iluminación en un cubo antes y después de aplicar un mapa normal.

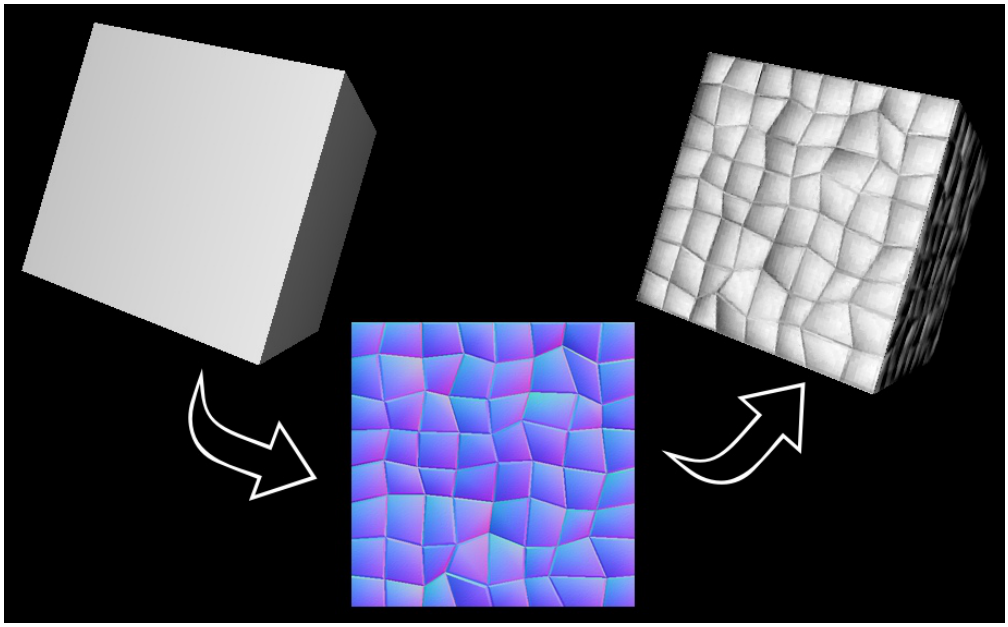


Figura 14: Un cubo iluminado antes y después de aplicarle un mapa normal

2.6. Texto

Por último, para mostrar texto por pantalla OpenGL no dispone de ninguna instrucción nativa, así que tenemos que renderizar nosotros mismos las letras encima de la escena renderizada seleccionándolas de un mapa de carácteres con transparencia y dibujándolas una a una.

Figura 15: Mapa de carácteres

3. Diseño e implementación del programa

Para poder hacer uso de los shaders, hace falta crear un programa que suministre los datos, las configuraciones y otros parámetros necesarios. Dicho programa se detalla en esta sección.

El programa está implementado en C++, mientras que los shaders están implementados en el lenguaje GLSL. Para tratar con OpenGL nos ayudaremos de un conjunto de librerías que nos facilitarán la tarea:

- GLEW: La librería GLEW (The OpenGL Extension Wrangler Library) [7] es una librería de carga de extensiones de código abierto y multiplataforma que provee de mecanismos para determinar que extensiones de OpenGL admite la plataforma en la que se ejecuta.

- GLFW: La librería GLFW (OpenGL FrameWork) [8] es una librería de código abierto y multiplataforma que ayuda en la creación de ventanas de OpenGL así como con su contexto, eventos y entrada de teclado y ratón.

- GLM: La librería GLM (OpenGL Mathematics) [9] proporciona clases y funciones matemáticas que siguen las convenciones de nombres y funcionalidades usadas en GLSL de tal manera que se puedan relacionar fácilmente en C++.

En la figura 16 se puede ver el diagrama de clases que componen el programa.

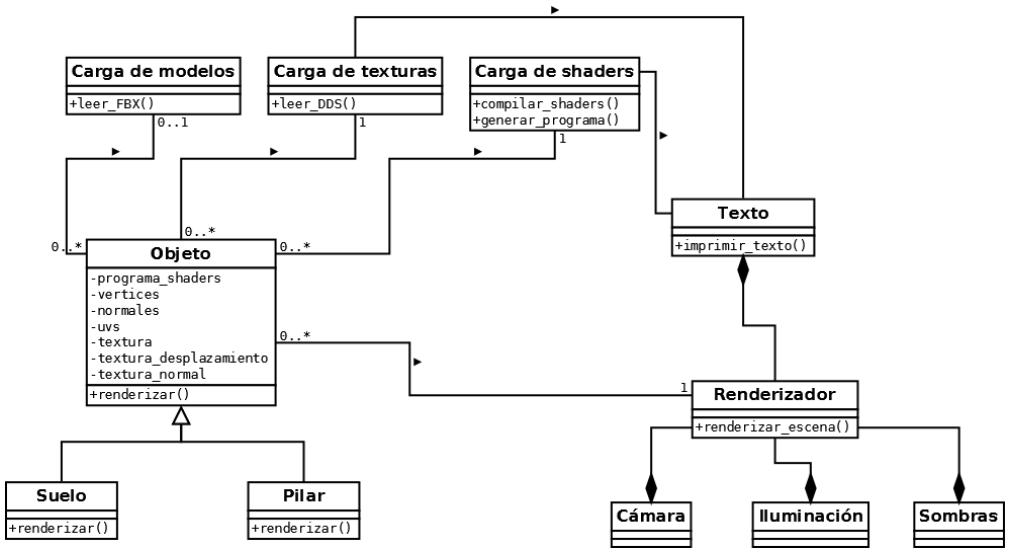


Figura 16: Diagrama de clases del programa

3.1. Programa principal

El programa principal será el encargado de inicializar los datos de OpenGL y de controlar los eventos de teclado. Creará la ventana y delegará la creación, composición y renderizado de la escena en una clase creada a tal fin.

La ventana se crea con una serie de parámetros gracias a GLFW (usar versión 4.3 de OpenGL, antialiasing 4x, sincronización vertical, ventana redimensionable). Después pasa al bucle principal, en el que llama continuamente a la clase renderizadora para que le suministre fotogramas.

3.2. Composición de la escena

3.2.1. Renderizador

La clase del renderizador será la encargada de crear y posicionar los objetos en la escena, así como de indicarles el momento y el modo en el que deben de dibujarse en pantalla (ver figura 16).

También será la encargada de controlar la cámara, la iluminación y las sombras de la escena.

Según el modo de renderizado, controlado por una variable, decide si tiene que renderizar la escena una sola vez o varias para incluir los mapas de sombras. A continuación, llama a la función de renderizado de cada uno de los objetos que componen la escena, suministrándoles la matriz MVP, la iluminación y el modo de renderizado.

3.2.2. Cámara

La clase de la cámara será la encargada de posicionar y mover la cámara a lo largo de la ejecución (ver figura 16).

Esta clase contiene también la matriz MVP de la cámara. La matriz está compuesta por tres mastrices de 4x4: la proyección (contiene que la vista está perspectiva, el campo de visión, la anchura y altura de la visión y el rango de cerca y de lejos), la vista (contiene la posición donde se encuentra la cámara, la posición a la que mira y el sentido en el que mira) y el modelo (que en este caso es una matriz identidad).

La posición de la cámara se puede cambiar modificando la matriz de vista y rehaciendo la matriz MVP.

3.2.3. Iluminación

La clase de iluminación contiene los datos de posición, intensidad y color de los dos tipos de luces que vamos a usar (ver figura 16).

Además contiene la matriz MVP desde el punto de vista de la luz difusa, lo cual nos servirá para calcular las sombras. Como la luz difusa que emplearemos pretende simular una luz muy lejana, como la solar, donde los rayos inciden paralelamente, la matriz de proyección es ortográfica en vez de en perspectiva.

3.2.4. Sombras

La clase de sombras se encargará de guardar en un buffer un fotograma renderizado desde el punto de vista de una luz concreta, que será su mapa de sombras, y lo suministrará cuando sea llamada con otra función (ver figura 16).

3.3. Herramientas

3.3.1. Carga de shaders

La clase de carga de shaders lee archivos GLSL y los almacena para compilarlos en la etapa que se le indique. Una vez tiene todas las etapas a usar con los archivos GLSL cargados, compila el programa notificando si ha habido algún error (ver figura 16).

3.3.2. Carga de texturas

La clase de carga de texturas debe ser capaz de leer una textura en formato DDS (DirectDraw Surface) [10], un formato muy usado para almacenar texturas, y convertirlas al formato de texturas usado por OpenGL (ver figura 16).

Estas texturas permiten el uso de mipmaps (la misma textura contiene versiones de distintos tamaños en potencias de dos desde 1x1 píxeles hasta el tamaño mayor, de manera que según el tamaño que ocupa en pantalla la textura, carga una u otra para optimizar).

3.3.3. Carga de modelos

La clase de carga de modelos permitirá cargar modelos en el formato FBX (Filmbox) [11], un formato que permite almacenar los vértices, las normales y los mapas UV de un modelo 3D, de manera que al leer el archivo, pueda cargar estos datos en los buffers de OpenGL (ver figura 16).

3.3.4. Texto

Una clase para poder imprimir texto por pantalla, ya que OpenGL no dispone de ninguna instrucción nativa (ver figura 16).

La clase de texto relaciona el número ASCII de cada carácter con la posición de dicho carácter en la textura del mapa de carácteres.

3.4. Objetos

Cada objeto de la escena heredará de una clase padre para poder compartir la función de renderizado (ver figura 16). Además, cada objeto ha de ser capaz de guardar la información de:

- Programa de shaders: Un programa compilado con los shaders de las distintas etapas que usará el objeto para renderizarse.
- Posición de vértices: La posición de cada vértice en el espacio de coordenadas de la escena, ya sea cargada desde un archivo FBX o codificada a mano. Dichos vértices deben componer siempre las caras de un triángulo, ya que solo vamos a trabajar con triángulos.
- Mapas de UV: Los mapas UV de los vértices, ya sea cargado desde un archivo FBX o codificado a mano.
- Normales de vértices: La normal de la superficie en cada uno de los tres vértices que intervienen, ya sea cargada desde un archivo FBX o codificada a mano.
- Textura principal: La textura que le da color al objeto.
- Textura de mapa de desplazamiento: La textura necesaria para poder usar mapas de desplazamiento.
- Textura de mapa normal: La textura necesaria para poder usar mapas normales.

Los distintos objetos usan las herramientas para cargar el programa de shaders, las texturas y los modelos, guardando todo en los distintos buffers de OpenGL habilitados para ello. A la hora de renderizar, según el modo de renderizado suministrado, indican a los shaders de distintas etapas que opciones quieren que usen, y les suministra las variables de entrada.

4. Implementación y validación de los shaders

A la hora de implementar los shaders, vamos a usar las ventajas de OpenGL v4.3 para programar varias funciones intercambiables dentro de cada shader, de manera que podamos elegir una u otra en función del modo de renderizado que deseemos. Así por ejemplo, una misma función de calcular la iluminación se comportará de forma diferente según los modos de renderizado que queramos.

Para comprobar la correcta ejecución de los shaders, se han implementado dos modos extra de visualización, uno que ofrece la escena vista en modo de trazado solo de aristas (wireframe) y otra que ofrece una visión de las normales de cada superficie según los códigos de colores explicados en la sección 2.5 y en la figura 13.

En la figura 17 se puede comprobar la renderización por defecto de la escena sin ninguna técnica aplicada, mientras que en las figuras 18 y 19 se puede observar la misma escena vista en modo de trazado solo de aristas y en modo de visualización de normales respectivamente.

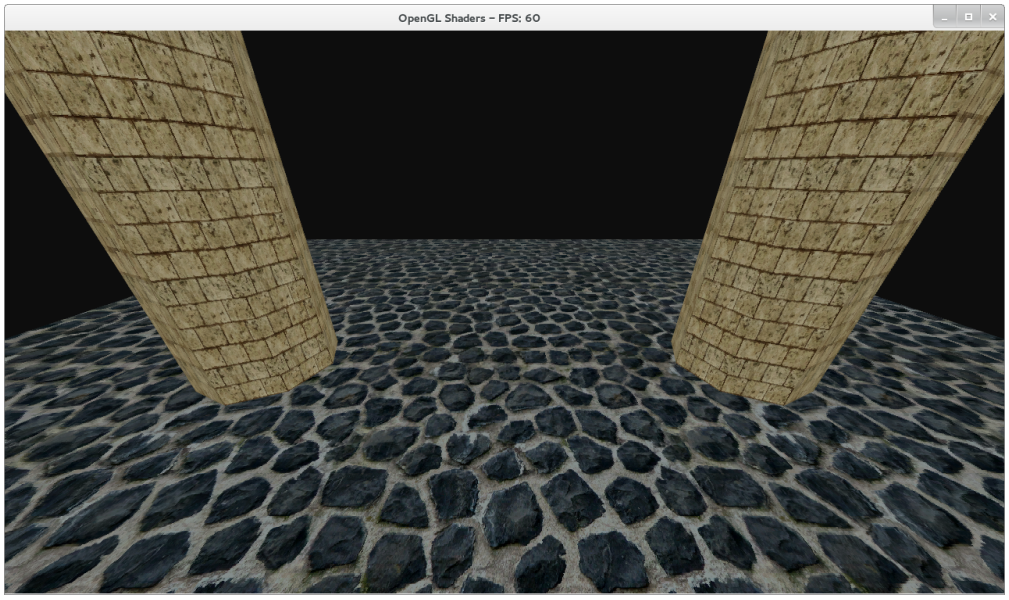


Figura 17: La escena original

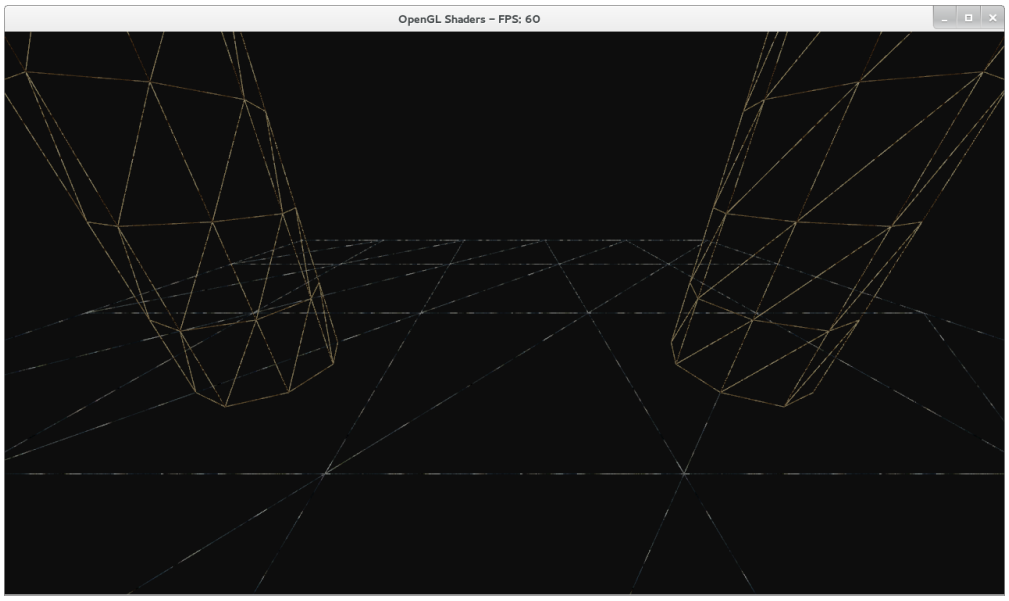


Figura 18: La escena original en modo wireframe

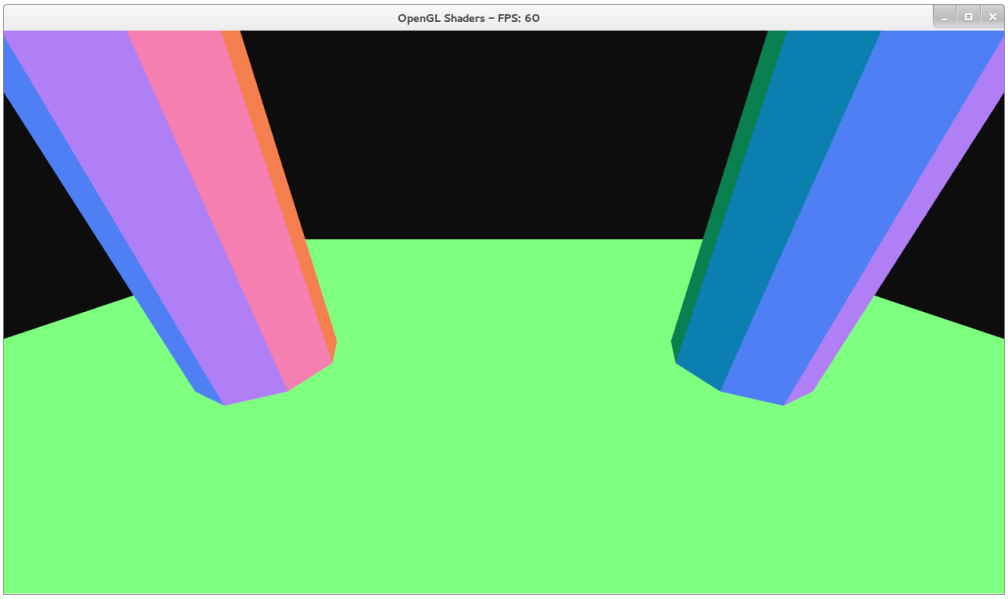


Figura 19: La escena original en modo de mostrar normales

Hay que recalcar antes de explicar la implementación de cada técnica que aunque puedan parecer escasos en cuanto a código, los shaders son muy complejos y requieren de muchas líneas de código en el programa que los crea para hacerlos funcionar. Puesto que los shaders se tienen que ejecutar multitud de veces, precisamente el conseguir que funcionen con el mínimo código posible es una de las dificultades de su implementación.

Junto a la descripción de la implementación de cada técnica, se incluyen fragmentos de código de los shaders en el lenguaje GLSL que ayuden a entenderlas. Estos fragmentos representan el núcleo funcional de la técnica en cuestión e indican con un comentario al inicio en que shader están implementados.

4.1. Mapas de desplazamiento

Para implementar los mapas de desplazamiento, vamos a trabajar principalmente en tres etapas: Tesselation Control Shader (TCS),

Tesselation Evaluation Shader (TEV) y Geometry Shader. Los parámetros de entrada que necesitaremos serán la textura con el mapa de desplazamiento, el factor de desplazamiento que queramos aplicar al objeto, y la posición de la cámara en las coordenadas de la escena, así como los parámetros básicos (vértices, UVs y normales del objeto y la matrix MVP de la cámara).

La primera parte es definir como queremos dividir los triángulos, lo cual se realiza en el TCS. Esta etapa recibe los tres vértices de un triángulo de golpe y opera con ellos (esta estructura recibe el nombre de parche). Lo que vamos a hacer es calcular la distancia desde la cámara al centro de cada arista del triángulo, y dependiendo de ella, aumentar o disminuir el nivel de teselación. Este proceso lo podemos comprobar en el código 1. Una vez hayamos definido los niveles de teselación, pasaremos los tres vértices al TEV.

```
/* Tessellation Control Shader */

subroutine(renderMode) void renderDisplacement () {
   // Calcular la distancia de la camara a cada vertice
   float cameraDistance0 = distance(CameraPos, gl_in[0].
      gl_Position.xyz);
   float cameraDistance1 = distance(CameraPos, gl_in[1].
      gl_Position.xyz);
   float cameraDistance2 = distance(CameraPos, gl_in[2].
      gl_Position.xyz);
   // Calcular los niveles de teselado
   gl_TessLevelOuter[0] = getTessLevel(cameraDistance1,
      cameraDistance2);
   gl_TessLevelOuter[1] = getTessLevel(cameraDistance2,
      cameraDistance0);
   gl_TessLevelOuter[2] = getTessLevel(cameraDistance0,
      cameraDistance1);
   gl_TessLevelInner[0] = (gl_TessLevelOuter[0] +
      gl_TessLevelOuter[1] + gl_TessLevelOuter[2]) / 3.0;
}

float getTessLevel (float distanceA, float distanceB) {
   float avgDistance = (distanceA + distanceB) / 2.0;
   if (avgDistance <= 2.5) {
      return 50.0;
   }
```

```
    if (avgDistance <= 5.0) {
        return 30.0;
    }
    return 15.0;
}
```

Código 1: Tessellation Control Shader del mapa de desplazamiento

En el TEV, lo primero que haremos será calcular la posición, UVs y normales de los nuevos vértices creados ponderándolos con los originales. Una vez los tengamos, podemos usar los UVs para acceder a la posición de la textura que tiene el mapa de desplazamiento y usar su valor multiplicándolo por el factor de desplazamiento. Multiplicando este valor por la normal y sumándoselo a la posición original conseguimos desplazar los vértices. Todo este proceso se puede ver en el código 2.

```
/* Tessellation Evaluation Shader */

// Posicionar
vec4 position = (gl_TessCoord.x * gl_in[0].gl_Position) +
                (gl_TessCoord.y * gl_in[1].gl_Position) +
                (gl_TessCoord.z * gl_in[2].gl_Position);
vec2 uvs = (vec2(gl_TessCoord.x) * tes_in[0].uvs) +
           (vec2(gl_TessCoord.y) * tes_in[1].uvs) +
           (vec2(gl_TessCoord.z) * tes_in[2].uvs);
vec4 normals = (gl_TessCoord.x * tes_in[0].normals) +
               (gl_TessCoord.y * tes_in[1].normals) +
               (gl_TessCoord.z * tes_in[2].normals);
normals = normalize(normals);

// Desplazar
float displacement = texture(DisplacementMap, uvs).x;
position = vec4((position + (normals * displacement *
    DispFactor)).xyz, 1.0f);

// Salida
gl_Position = position;
tes_out.uvs = uvs;
tes_out.normals = normals;
```

Código 2: Tessellation Evaluation Shader del mapa de

desplazamiento

El resultado lo podemos comprobar en la figura 20.

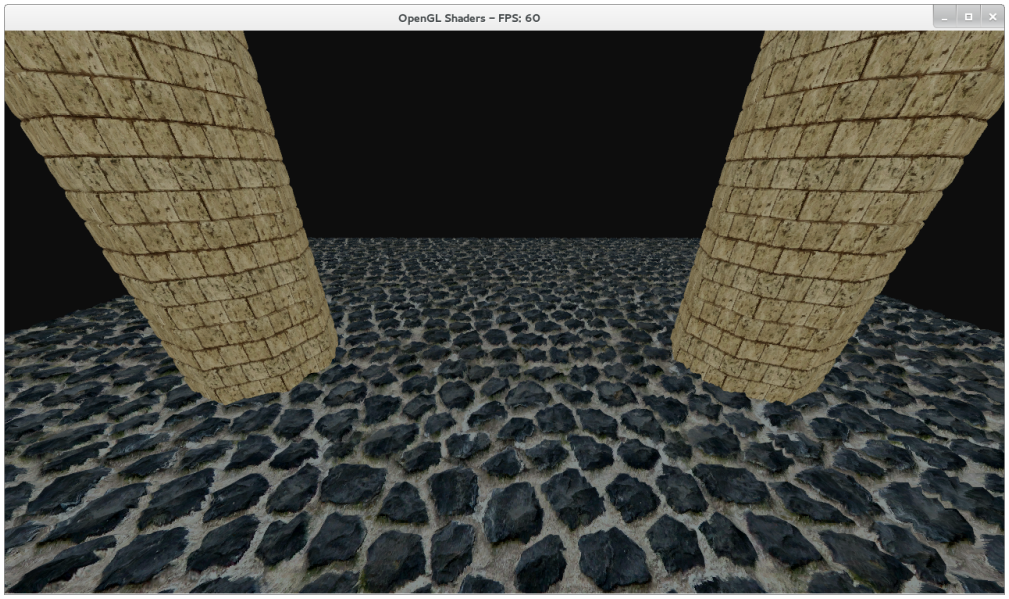


Figura 20: La escena original con el mapa de desplazamiento aplicado

Al renderizar en modo wireframe podemos comprobar los nuevos vértices creados y su desplazamiento (ver figura 21).

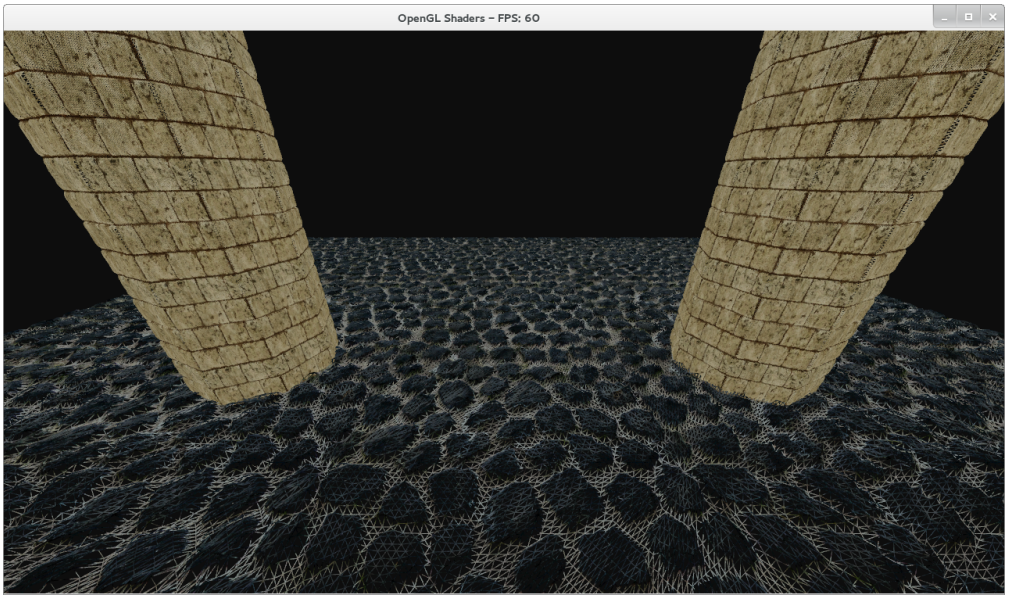


Figura 21: La escena con el mapa de desplazamiento en modo wireframe

Por último tenemos que recalcular las normales, ya que al crear nuevos vértices y moverlos, éstas han cambiado. Como la información de los vértices la tenemos parche a parche, solo podemos calcular la normal de la superficie (no podemos calcular la normal en un vértice ponderando todas las superficies en las que interviene). Esta informacoón la recibimos en el Geometry Shader, donde podremos operar con ella. Para calcular la nueva normal simplemente calculamos el producto vectorial de los vectores que van desde el primer vértice hasta el segundo y el tercero. Además, calculamos la tangente y la bitangente para formar una matriz TBN con la que poder cambiar los mapas normales del espacio de cordenadas de la textura al de la Tangente-Bitangente-Normal. Como en teoría podemos elegir infinitas tangentes y bitangentes a la normal, lo que haremos será orientarlas en la misma dirección que las coordenadas de la textura, como podemos ver en la figura 22.

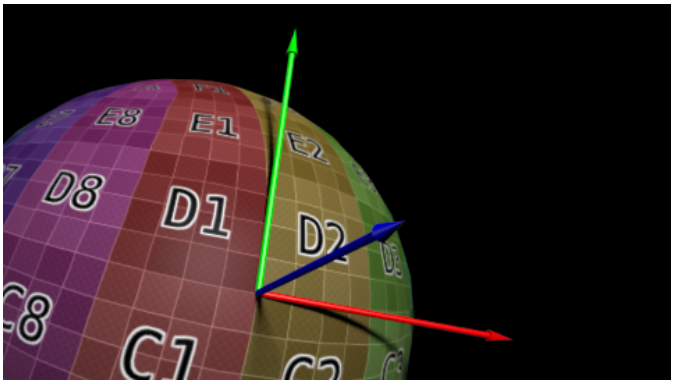


Figura 22: Los vectores T, B, y N según las coordenadas de la textura

El cálculo de la matriz TBN se puede ver en el código 3.

```
/* Geometry Shader */

subroutine(renderMode) normalData_t renderFaceNormals (int i) {
   normalData_t nData;
   // Calculamos la normal
   vec4 edge1 = gl_in[1].gl_Position - gl_in[0].gl_Position;
   vec4 edge2 = gl_in[2].gl_Position - gl_in[0].gl_Position;
   nData.normals = normalize(vec4(cross(edge1.xyz, edge2.xyz),
      0.0f));
   // Calculamos la tangente y bitangente en funcion de las
      coordenadas de la textura
   vec2 deltaUV1 = gs_in[1].uvs - gs_in[0].uvs;
   vec2 deltaUV2 = gs_in[2].uvs - gs_in[0].uvs;
   float r = 1.0f / (deltaUV1.x * deltaUV2.y - deltaUV2.x *
      deltaUV1.y);
   vec4 tangent = (deltaUV2.y * edge1 - deltaUV1.y * edge2) * r
      ;
   tangent = normalize(vec4(tangent.xyz, 0.0f));
   vec4 bitangent = (-deltaUV2.x * edge1 - deltaUV1.x * edge2)
      * r;
   bitangent = normalize(vec4(bitangent.xyz, 0.0f));
   // Creamos la matriz TBN
   nData.TBN = mat4(tangent, bitangent, nData.normals, vec4(0.0
      f, 0.0f, 0.0f, 0.0f));
   return nData;
}
```

Código 3: Geometry Shader del mapa de desplazamiento

Una vez calculadas las normales, no tenemos que olvidarnos de modificar la posición de los vértices del espacio de coordenadas del mundo al espacio de coordenadas de la cámara usando la matriz MVP. El resultado del cálculo de las normales lo podemos ver en la figura 23.

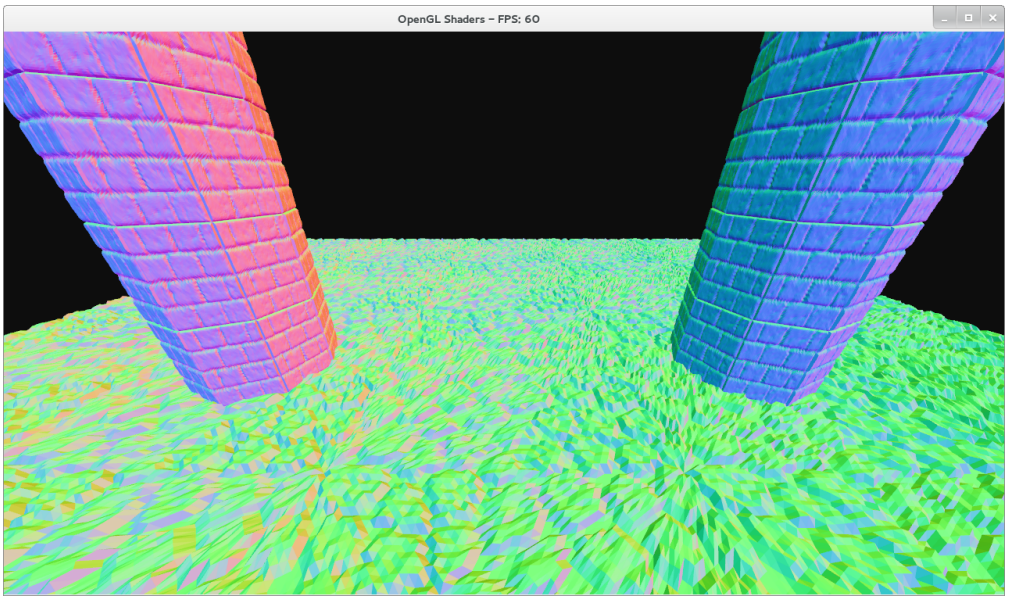


Figura 23: La escena con el mapa de desplazamiento en modo de mostrar normales

4.2. Iluminación

4.2.1. Luz ambiental

Para la luz ambiental solo hace falta modificar el color de salida de un fragmento en el Fragment Shader multiplicando el color original por la intensidad y el color de la luz, que serán los parámetros de entrada.

4.2.2. Luz difusa

En cuanto a la luz difusa, además de la intensidad y el color, necesitaremos su posición y dirección. Con ellas podemos calcular la

iluminación en cada punto dependiendo de la distancia y de el ángulo que forma la dirección con la normal de la superficie en ese punto. Una vez calculada, hay que sumarle la luz ambiental para obtener la iluminación total. La función que calcula la iluminación se puede ver en el código 4 y el resultado se puede ver en la figura 24.

```
/* Fragment Shader */

vec3 getLighting (vec4 normals) {
   vec3 ambLight = AmbientLight.color * AmbientLight.intensity;
   // Calculamos el producto escalar de la direccion de la luz
      con la normal
   float difFactor = dot(normalize(normals), vec4(-DiffuseLight
      .direction, 0.0f));
   vec3 difLight;
   if (difFactor > 0) {
      difLight = vec3(DiffuseLight.color * DiffuseLight.
         intensity * difFactor);
   } else {
      difLight = vec3(0, 0, 0);
   }
   return (ambLight + difLight);
}
```

Código 4: Fragment Shader de la iluminación

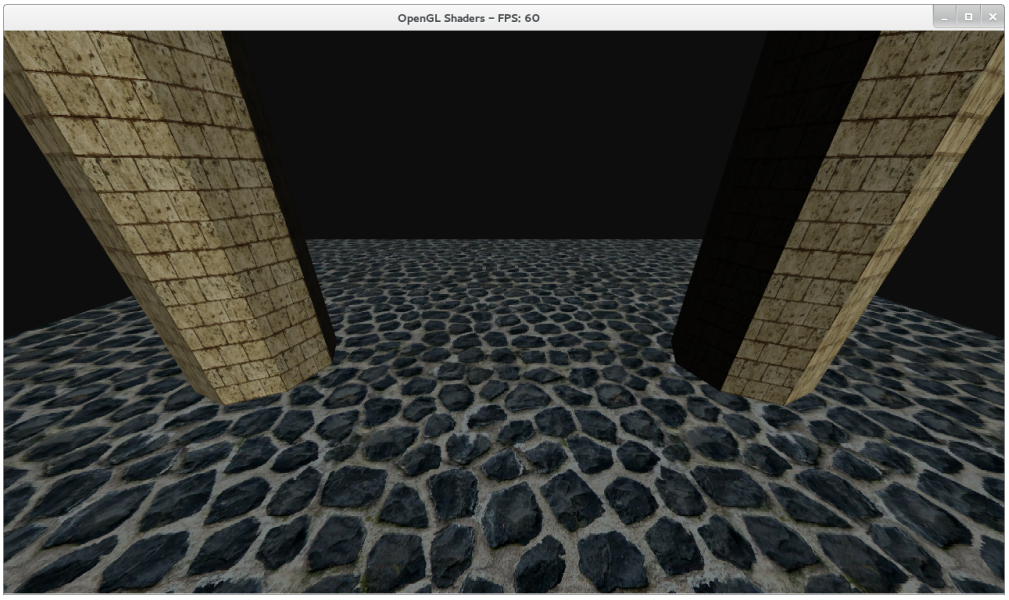


Figura 24: La escena original con la iluminación activada

Hay que tener en cuenta que si iluminamos la escena cuando está aplicado el mapa de desplazamiento, puesto que la normal es la misma para toda la superficie de un triángulo, se notarán diferencias en la iluminación entre triángulos adyacentes con normales muy diferentes como se puede comprobar en la figura 25. Esto lo arreglaremos luego con la técnica de los mapas normales.

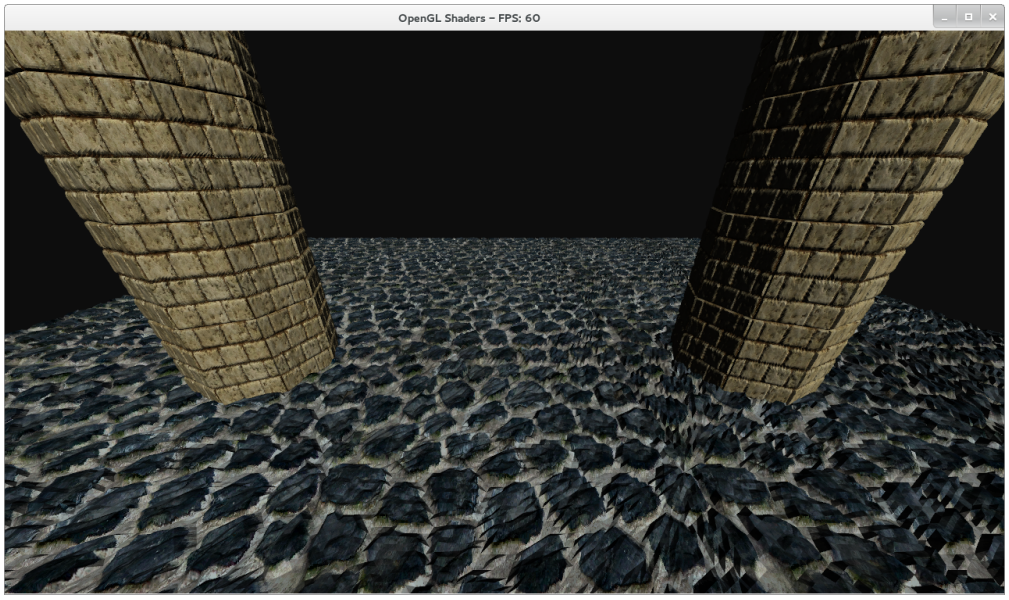


Figura 25: La escena con el mapa de desplazamiento y la iluminación activadas

4.3. Mapas de sombras

Las sombras las calcularemos como una manera alternativa de iluminación del punto anterior. Con los z-buffers generados, podemos calcular en el Fragment Shader la posición que le corresponde en el mapa al fragmento que estamos analizando. Necesitaremos como parámetro de entrada la matriz MVP de la luz que generó el z-buffer.

En la figura 26 podemos ver el z-buffer que queda almacenado en la textura tras ser calculado con la vista desde la posición de una luz. Esta textura será lo se suministre luego al Fragment Shader.

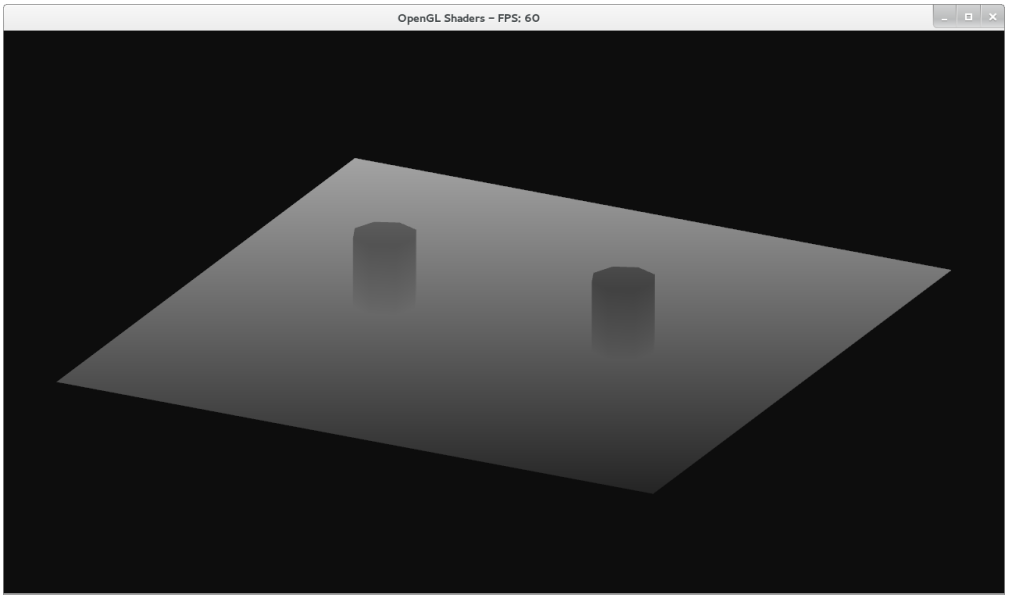


Figura 26: Textura con el z-buffer generado desde la vista de la luz difusa

Con estos datos, podemos comparar la distancia de la luz al fragmento actual con la distancia guardada en nuestro textura con el z-buffer en el mismo punto, lo que recibe el nombre de test de profundidad, y multiplicar la luz difusa calculada por el factor de la sombra en caso de que el fragmento falle dicho test. El resultado final lo podemos ver en la figura 27.

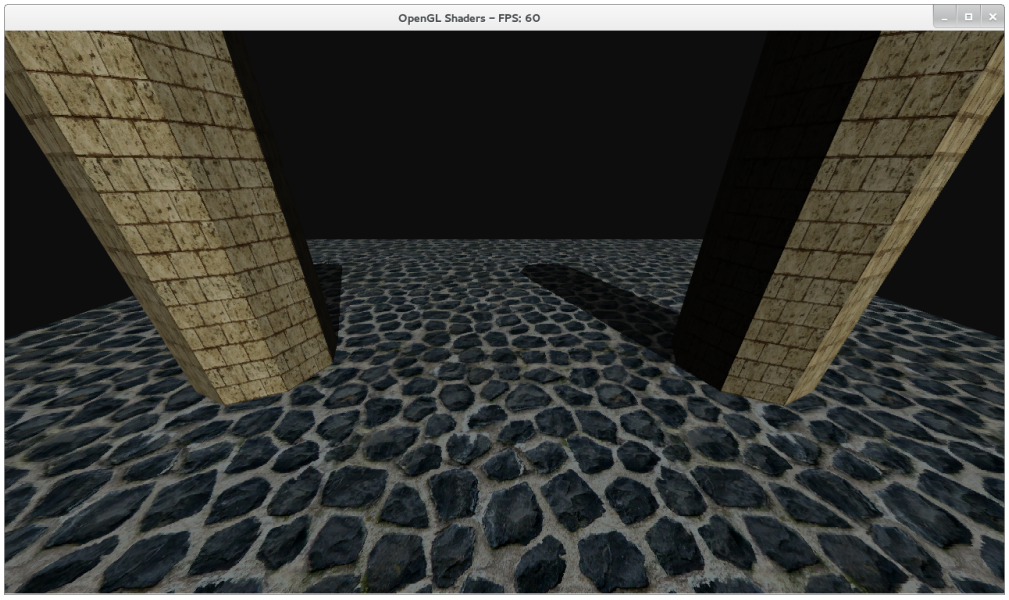


Figura 27: La escena original con las sombras activadas

Para evitar que al activar el mapa de desplazamiento se creen pequeñas sombras triangulares correspondientes a los triángulos desplazados, se puede añadir un margen de error de modo que para considerar un fragmento a la sombra no se compare la profundidad del mapa con la distancia de la luz al fragmento sino con dicha distancia menos el ajuste que queramos darle. Este margen de error ayuda también a evitar el problema del "acné de sombras", que se produce cuando algunos de los píxeles fallan el test de profundidad cuando no deberían. La función que calcula la iluminación con las sombras se puede ver en el código 5 y el resultado se puede ver en la figura 28.

```
/* Fragment Shader */

vec3 getLightingWithShadows (vec4 normals) {
   // Multiplicamos la matrix MVP de la luz por la posicion
      actual (este calculo se realiza en una etapa anterior
      pero se indica aqui para clarificar)
   // vec4 lightSpacePosition = LightMVP * gl_in.gl_Position;
   // Como ahora tenemos la posicion en funcion de la ``camara
      '' de la luz
   vec3 projCoords = lightSpacePosition.xyz /
```

```
        lightSpacePosition.w;
    vec2 uvCoords;
    uvCoords.x = 0.5 * projCoords.x + 0.5;
    uvCoords.y = 0.5 * projCoords.y + 0.5;
    float z = 0.5 * projCoords.z + 0.5;
    // Usamos un sesgo para evitar el problema de ''acne de
        sombras''
    float bias = 0.005;
    float depth = texture(ShadowMap, uvCoords).x;
    float shadowFactor = 1.0;
    // Comparamos la profundidad actual con la almacenada
    if (depth < (z - bias)) {
        shadowFactor = 0.25;
    }
    vec3 ambLight = AmbientLight.color * AmbientLight.intensity;
    // Calculamos el producto escalar de la direccion de la luz
        con la normal
    float difFactor = dot(normalize(normals), vec4(-DiffuseLight
        .direction, 0.0f));
    vec3 difLight;
    if (difFactor > 0) {
        difLight = vec3(DiffuseLight.color * DiffuseLight.
            intensity * difFactor);
    } else {
        difLight = vec3(0, 0, 0);
    }
    return (ambLight + (shadowFactor * difLight));
}
```

Código 5: Fragment Shader del mapa de sombras

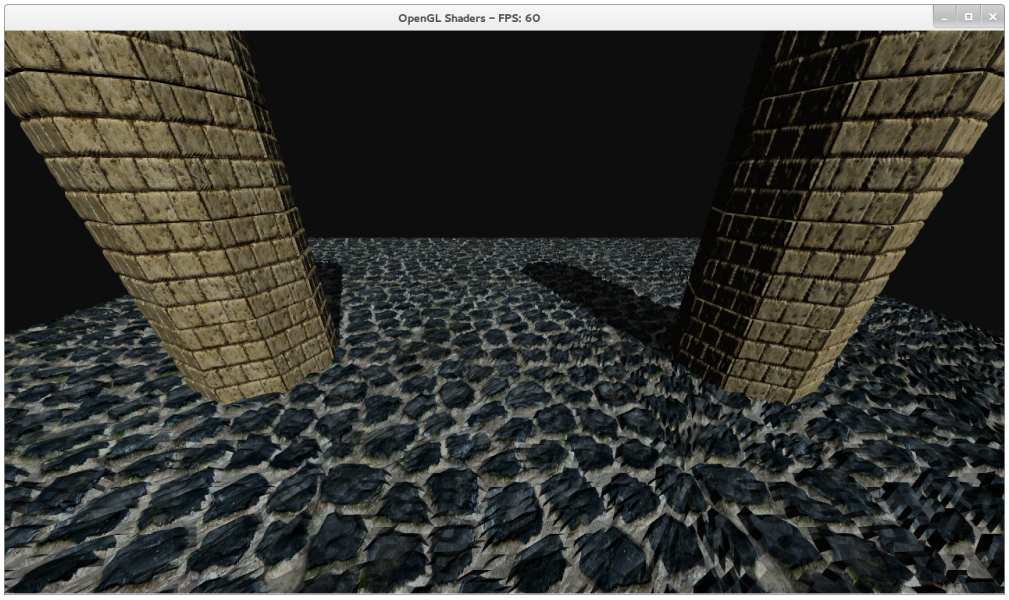


Figura 28: La escena con el mapa de desplazamiento y las sombras activadas

4.4. Mapas normales

Los mapas normales se pueden aplicar simplemente leyendo el valor de la textura del mapa correspondiente al fragmento en el Fragment Shader gracias a su UV y convirtiendo el valor de la textura, que está entre 0 y 1, al valor de la normal, que está entre -1 y 1. Una vez hecho esto, tenemos que multiplicar esta normal por la matrix TBN que habíamos calculado previamente en el Geometry Shader para poder transformar la normal del espacio de coordenadas de la textura al espacio de coordenadas del mundo. Ahora podemos usar esta nueva normal calculada en vez de la original para aplicar la iluminación. El cálculo se puede ver en el código 6 y el resultado se puede ver en la figura 29.

```
/* Fragment Shader */

subroutine(renderMode) vec3 renderShadowNormal () {
   vec4 normals = fs_in.TBN * vec4(texture(NormalMap, fs_in.uvs
      ).rgb * 2.0f - vec3(1.0f, 1.0f, 1.0f), 0.0f);
```

```
    normals = normalize(normals);
    vec3 lighting = getLightingWithShadows(normals);
    return texture(TextureSampler, fs_in.uvs).rgb * lighting;
}
```

Código 6: Fragment Shader del mapa normal

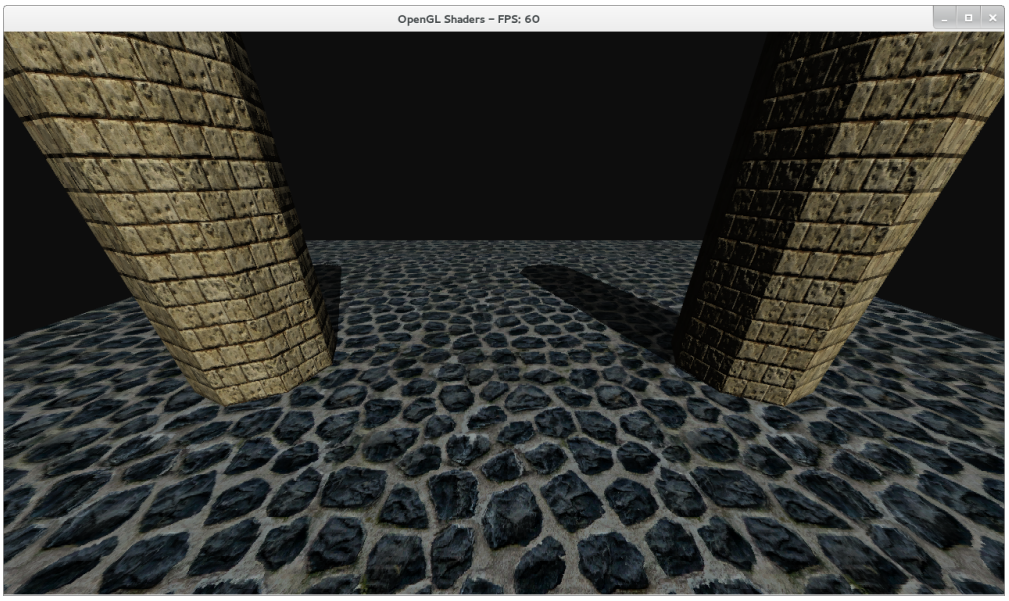


Figura 29: La escena original con el mapa normal

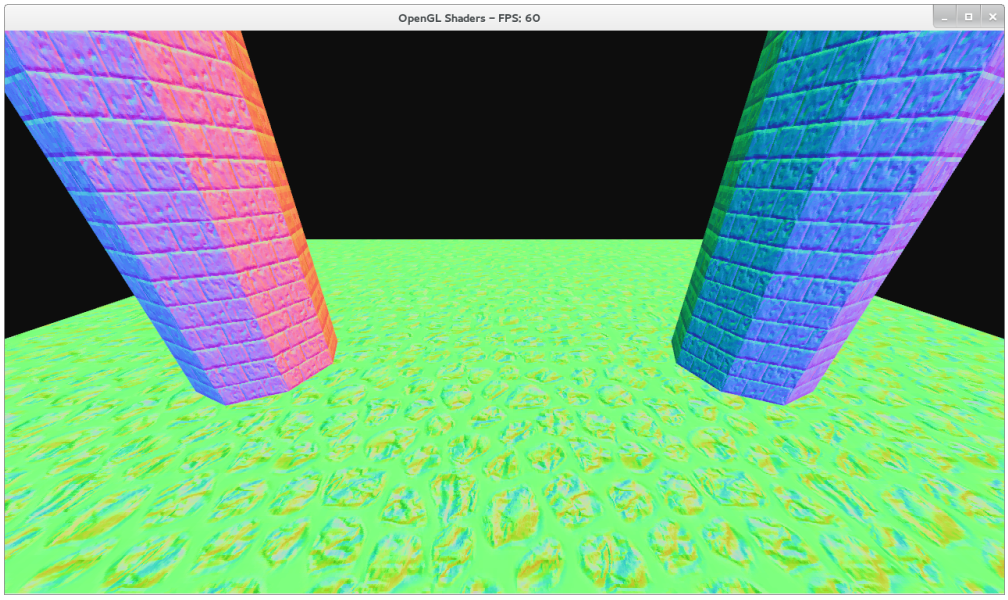


Figura 30: La escena original con el mapa normal en modo de mostrar normales

Como se puede apreciar en la figura 30 ahora la superficie no tiene una única normal, sino diferentes según el punto de la textura. Como ahora la normal se calcula con la textura y no con la superficie, hemos corregido los problemas de iluminación del mapa de desplazamiento, como podemos comprobar en las figuras 31 y 32.

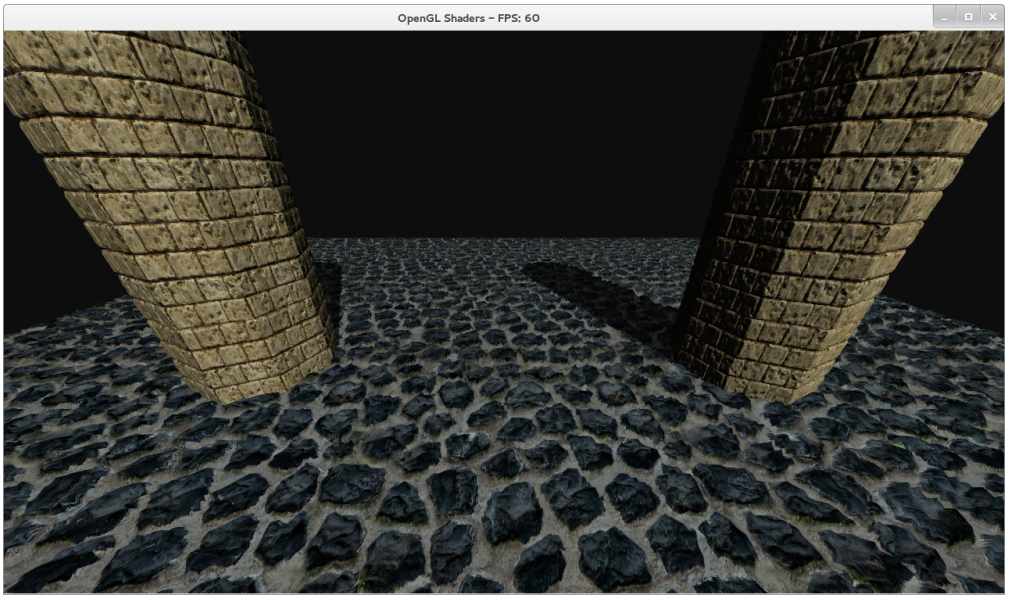


Figura 31: La escena con el mapa de desplazamiento y el mapa normal

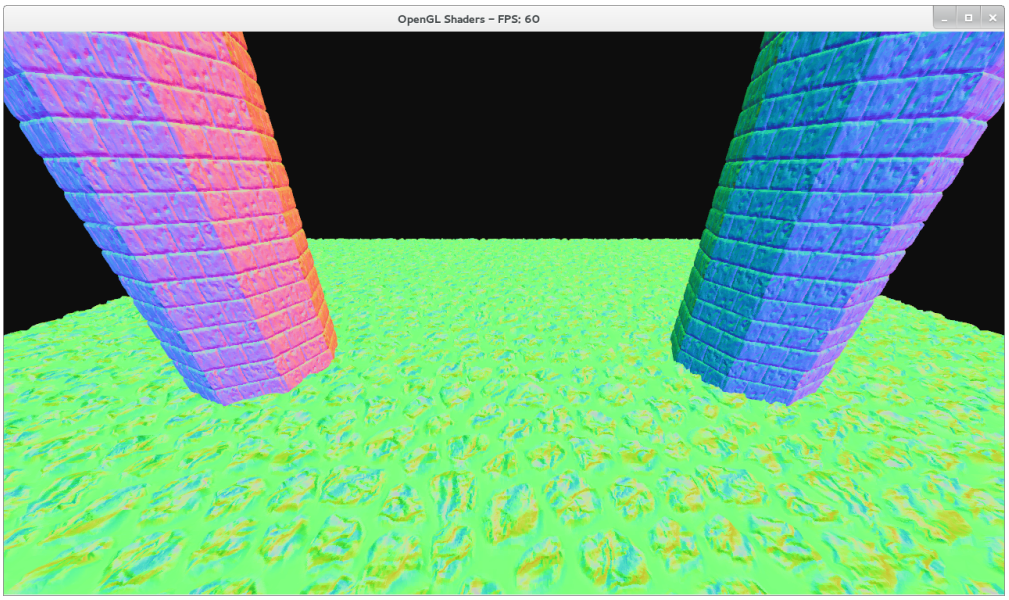


Figura 32: La escena con el mapa de desplazamiento y el mapa normal en modo de mostrar normales

4.5. Texto

El texto solo necesita hacer uso del Vertex Shader y el Fragment Shader. En el Vertex Shader se le pasa como parámetros de entrada la altura y anchura de la ventana para mapear la posición de las letras, que va dada en una posición entre 0 y la anchura y entre 0 y la altura, a la posición de la ventana de OpenGL, que va entre -1 y 1 para el ancho y para el alto. Una vez se tiene la posición, se aplica el color dado como parámetro de entrada en el Fragment Shader. Se puede ver la escena con el texto informativo en la figura 33 y las instrucciones que lo hacen posible en el código 7.

```
/* Vertex Shader */

// Map [0..WindowWidth][0..WindowHeight] to [-1..1][-1..1]
vec2 pos = vertex_position.xy * 2.0f / vec2(WindowWidth,
   WindowHeight);
pos -= vec2(1.0f, 1.0f);
gl_Position =  vec4(pos, -1, 1);
uvs = vertex_uv;

/* Fragment Shader */

color = texture2D(TextureSampler, uvs) * vec4(TextColor, 1.0f);
```

Código 7: Vertex Shader y Fragment Shader del texto

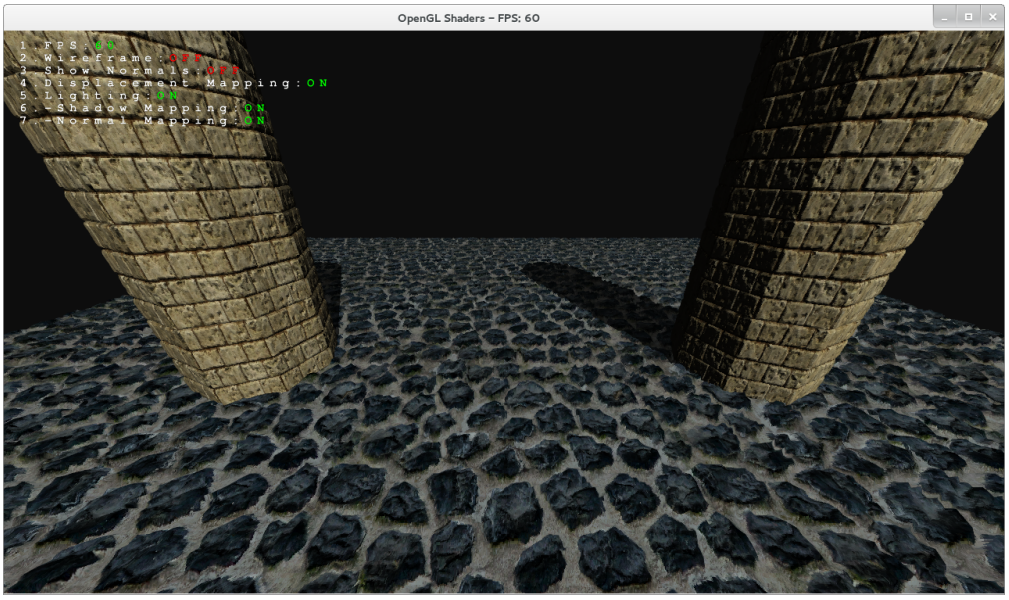


Figura 33: La escena final renderizada con el texto informativo

5. Análisis de eficiencia

5.1. Nvidia Nsight

Nvidia Nsight es una plataforma para depurar y analizar el código de aplicacines de gráficos de distintas tecnologías (entre ellas OpenGL). La versión utilizada se integra en el entorno de desarrollo Visual Studio 2013 de Microsoft.

Con esta herramienta podemos analizar los shaders que hemos implementado y ver la carga de trabajo de la GPU, las operaciones más costosas, y otros elementos útiles, como muestran la figura 34 y la figura 35.

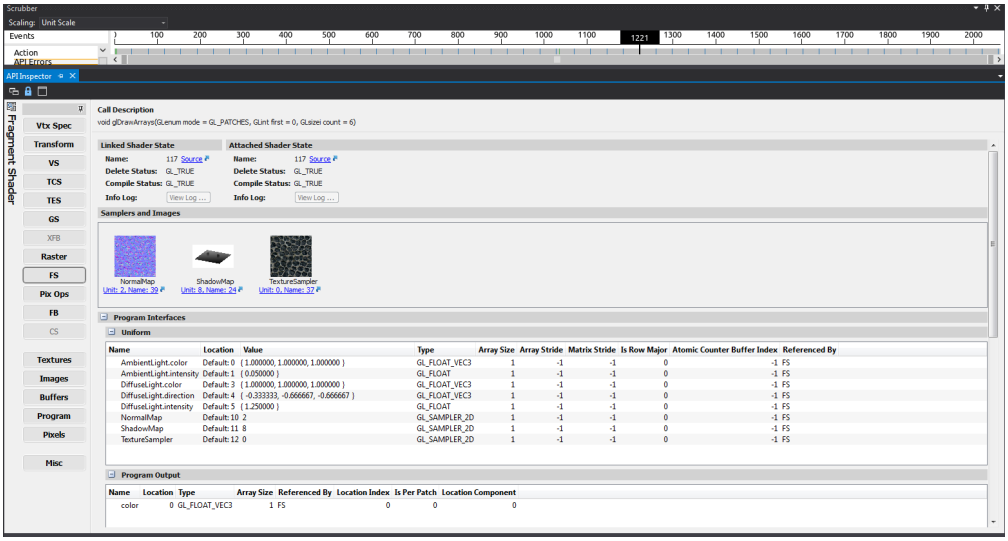


Figura 34: Nsight analizando el Fragment Shader en Visual Studio

La prueba se ha realizado con la sincronización vertical activada (es decir, se generan tantos fotogramas por segundo como la tasa de refresco de la pantalla, que en este caso es de 60Hz).

Si nos fijamos en los resultados obtenidos en una ejecución continua del programa, veremos que la GPU está ociosa entre el 85 % y el

95 % del tiempo con la escena original. Tras aplicarle distintas combinaciones de las técnicas implementadas mediante shaders (mapas de desplazamiento, iluminación, mapas de sombras y mapas normales) vemos que el trabajo de la GPU apenas asciende tras activar cualquier combinación de ellas menos los mapas de desplazamiento.

Si los mapas de desplazamiento están activados, y cualquier otra técnica menos los mapas de sombras está activada, la GPU está ociosa el 50 % del tiempo.

Por último, combinando mapas de desplazamiento y mapas de sombras (independientemente de la iluminación y de los mapas normales) la GPU se encuentra ociosa entre el 20 % y el 25 % del tiempo.

La inmensa mayoría del trabajo de la GPU en todos los casos se debe al cómputo de los shaders (siendo las otras dos posibilidades la carga de geometría o de texturas, que juntas apenas suponen un 2 % de la carga de trabajo).

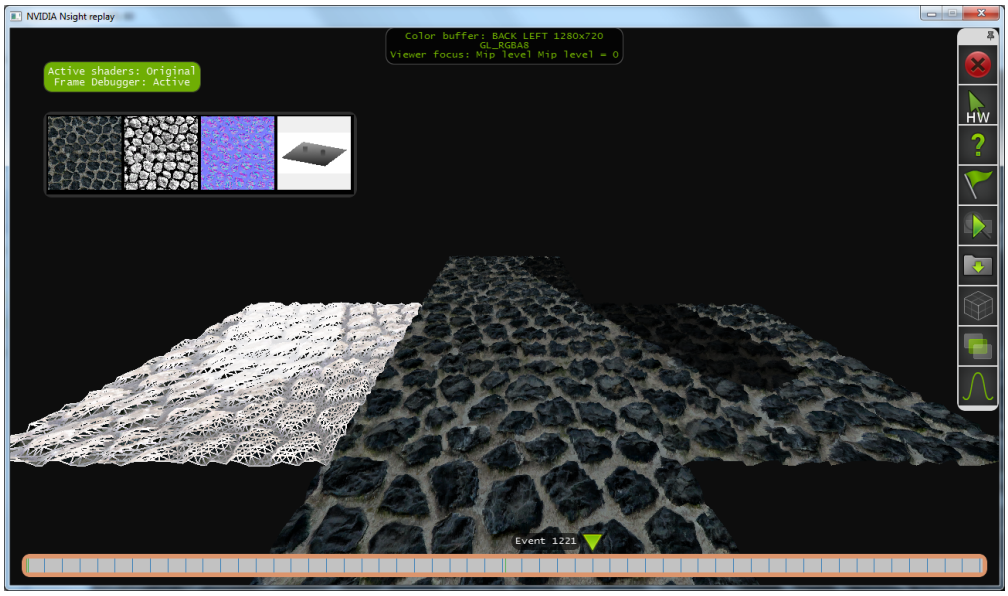


Figura 35: Nsight analizando un fotograma paso a paso

Por consiguiente, podemos deducir que la iluminación no necesita optimización puesto que apenas añade carga, mientras que se deberían optimizar los mapas de desplazamiento, que suponen la mayoría del trabajo. Los mapas de sombras obviamente también suponen un esfuerzo adicional, pero es porque tienen que renderizar la escena una vez extra por cada punto de luz.

Llegados a este punto, cabe preguntarse si realmente es necesario renderizar las sombras con la calidad que ofrecen los mapas de desplazamiento, en vez de sacrificar dicha calidad para ganar mucho más rendimiento, de manera que las sombras proyectadas sobre la escena desplazada correspondan a las sombras de la escena original.

Además, es un buen punto para pensar en el nivel de detalle que queremos crear en la fase de teselado, ya que el mapa de desplazamiento de este PFC se ha diseñado e implementado con la opción de sacrificar detalle según la distancia a la cámara, y aunque para las pruebas se ha usado la versión de detalle más alto, se puede adaptar para mejorar el rendimiento en las tarjetas gráficas que lo necesiten.

5.2. Comparativa con tarjetas gráficas Nvidia

A continuación se presenta una comparativa del programa ejecutándose en diferentes tarjetas gráficas cuyas características se pueden consultar en el cuadro 1. Una información más detallada sobre estas tarjetas se puede encontrar en el anexo B.

		GTX 660	GTX 750	GTX 750 Ti	GTX 760	GT 650M
GPU	Núcleos CUDA	960	512	640	1152	384
	Frecuencia de reloj normal (MHz)	980	1020	1020	980	900
	Frecuencia acelerada (MHz)	1033	1085	1085	1033	900
	Tasa de relleno de texturas (GTexel/s)	78.4	32.6	40.8	94.1	27.2
Mem	Frecuencia de la memoria (Gbps)	6.0	5.0	5.4	6.0	2.2
	Cantidad de memoria (MB)	2048	2048	2048	2048	1024
	Interfaz de memoria (GDDR5)	192-bit	128-bit	128-bit	256-bit	128-bit
	Ancho de banda máx. (GB/s)	144.2	80.0	86.4	192.2	80.0

Cuadro 1: Comparativa de las tarjetas gráficas usadas

Para evaluar el rendimiento, se ha desactivado la sincronización vertical, de manera que cada GPU genere tantos fotogramas como pueda. El programa se ejecuta durante 10 segundos en los que la cámara se mueve rotando alrededor de la escena.

Los resultados de todos los análisis realizados se pueden ver en las figuras 36, 37, 38, 39 y 40.

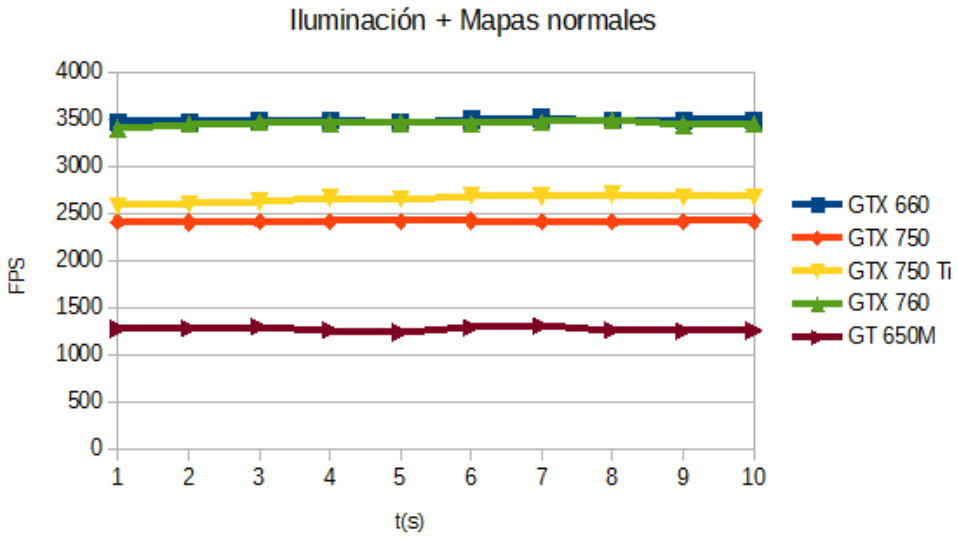


Figura 36: Fotogramas por segundo (FPS): Análisis 1

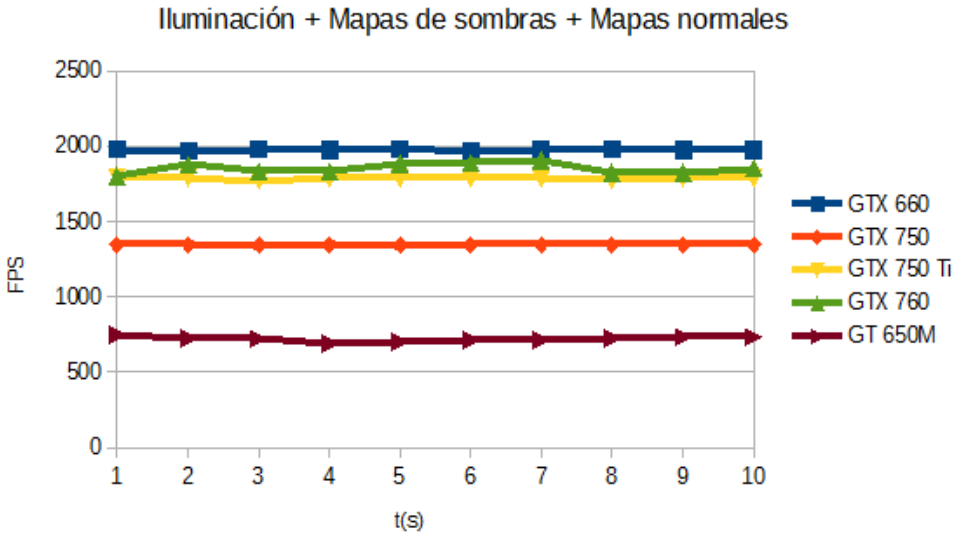


Figura 37: Fotogramas por segundo (FPS): Análisis 2

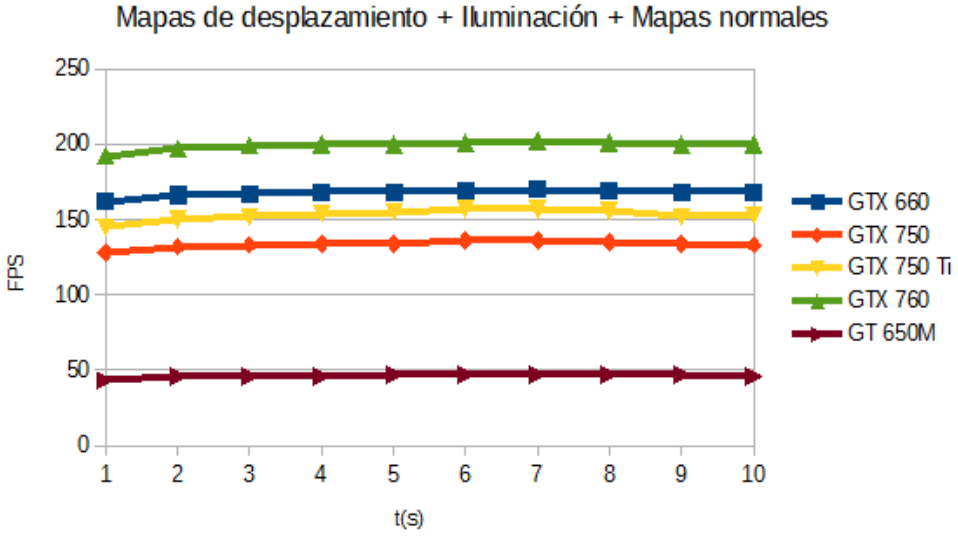


Figura 38: Fotogramas por segundo (FPS): Análisis 3

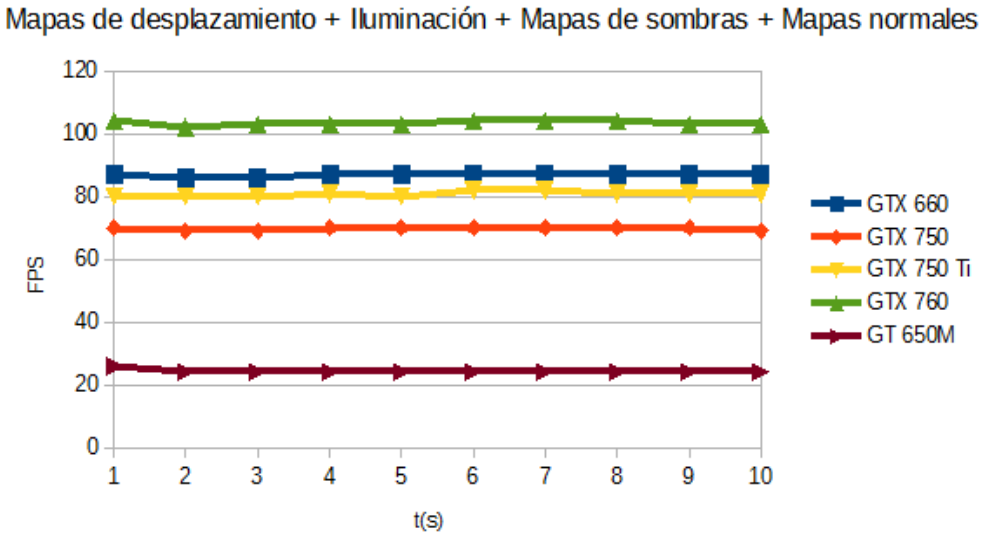


Figura 39: Fotogramas por segundo (FPS): Análisis 4

Podemos observar como los modelos más potentes (GTX 760 y GTX 660) son los que mejor resultado dan, seguidos de la siguiente gama de modelos (GTX 750 y su revisión mejorada GTX 750 Ti). Por último, la versión móvil para portátiles (GT 650M) es obviamente la que peor resultado ofrece.

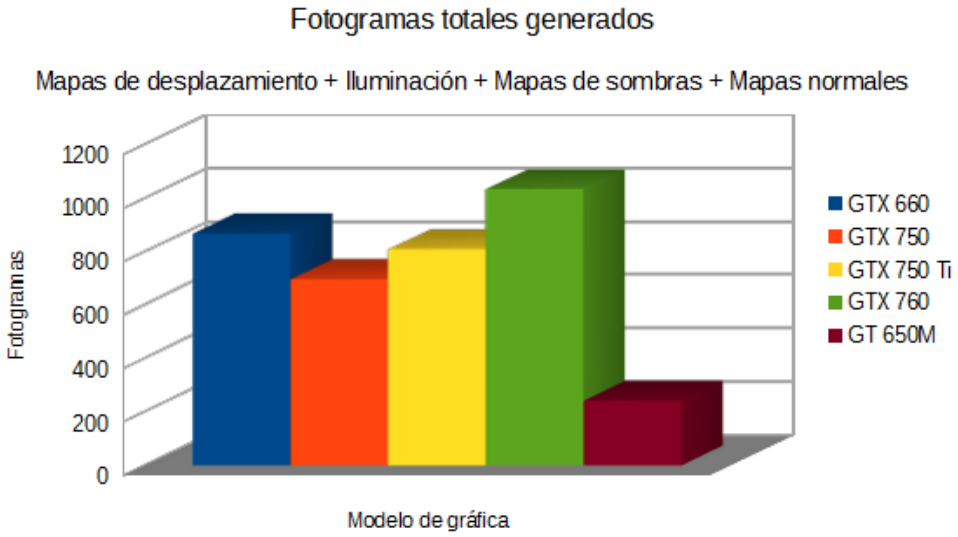


Figura 40: Fotogramas totales generados renderizando la escena durante diez segundos con todas las técnicas activadas.

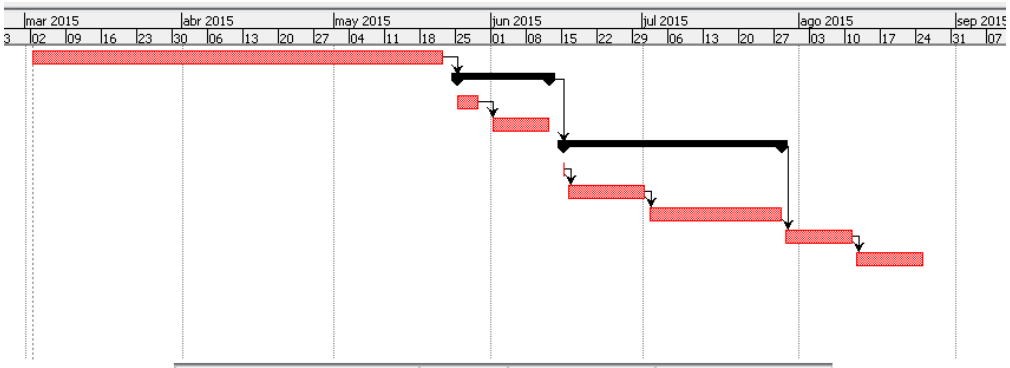
6. Conclusiones

A continuación se enumeran los objetivos del proyecto y su grado de cumplimiento:

1. Diseño de una escena a modo de ejemplo en 3D utilizando las librerías de OpenGL para C++: **realizado**.
2. Diseño de shaders utilizando el lenguaje GLSL que ofrece OpenGL en la versión 4.3:
   - Mapas de desplazamiento: **realizado**.
   - Mapas normales: **realizado**.
   - Iluminación: **realizado con iluminación ambiental, iluminación difusa y mapas de sombras**.
3. Implementación de dichos shaders utilizando tarjetas Nvidia: **realizado**.
4. Análisis del comportamiento de dichos shaders frente a diferentes tipos de tarjetas y uso del programa Nvidia Nsight: **realizado usando cinco tarjetas gráficas diferentes**.
5. El código implementado (sin contar los shaders) está formado por unas 2000 líneas de código.
6. Los tamaños aproximados del código de cada shader para cada tipo de figura diferente en la escena (suelo, columnas y texto) son:
   - Vertex Shader: 20 líneas de código.
   - Tessellation Control Shader: 60 líneas de código.
   - Tessellation Evaluation Shader: 70 líneas de código.
   - Geometry Shader: 70 líneas de código.
   - Fragment Shader: 120 líneas de código.

7. Horas de trabajo

En la figura 41 se puede ver el diagrama de Gantt del tiempo de dedicación de este PFC.



Nombre	Duracion	Inicio	Terminado
Estudio y aprendizaje de OpenGL 4.3	60 days	2/03/15 10:00	22/05/15 14:00
Diseño	15 days	**25/05/15 10:00**	**12/06/15 14:00**
Diseño del programa	5 days	25/05/15 10:00	29/05/15 14:00
Diseño de los shaders	10 days	1/06/15 10:00	12/06/15 14:00
Implementación	32 days	**15/06/15 10:00**	**28/07/15 14:00**
Creación de modelos y texturas	1 day	15/06/15 10:00	15/06/15 14:00
Implementación y validación del progra	12 days	16/06/15 10:00	1/07/15 14:00
Implementación y validación de los sha	19 days	2/07/15 10:00	28/07/15 14:00
Análisis de eficiencia	10 days	29/07/15 10:00	11/08/15 14:00
Escritura de la memoria	10 days	12/08/15 10:00	25/08/15 14:00

Figura 41: Diagrama de Gantt del trabajo realizado

El desglose de las horas de trabajo realizadas es:

- Estudio y aprendizaje de OpenGL v4.3: 240 horas.
- Diseño del programa: 20 horas.
- Diseño de los shaders: 40 horas.
- Creación de modelos y texturas: 4 horas.
- Implementación y validación del programa: 48 horas.
- Implementación y validación de los shaders: 76 horas.
- Análisis de eficiencia: 40 horas.

- Escritura de la memoria: 40 horas.

El tiempo total dedicado a la realización de este PFC ha sido de 508 horas.

8. Trabajo futuro

Los shaders desarrollados en este PFC se pueden aplicar a otras geometrías para elaborar escenas con más elementos, siempre que se suministren los modelos en el formato adecuado y los mapas de texturas necesarios para las técnicas que se quieran aplicar.

Con respecto a las mejoras que podrían recibir los shaders, el apartado más indicado es el de la iluminación y sombreado. Por un lado, el programa actual solo ilumina con una luz de ambiente y una luz direccional, así que sería interesante implementar los otros dos tipos de luces que se suelen utilizar en la iluminación de gráficos 3D: el punto de luz (similar a una bombilla, que ilumina alrededor de un punto dado) y la luz de foco (una mezcla de la luz direccional y el punto de luz, ya que ilumina desde un punto dado pero solo en una dirección concreta).

Por otro lado, además de la técnica usada de luz difusa, se puede emplear otra ténica de iluminación, la luz especular (specular lighting), que consiste en reflejar la luz en un único ángulo en vez de en varios (como hace la luz difusa), de manera que se crea un efecto de espejo que se puede regular dependiendo del material al que se quiera aplicar (por ejemplo un metal pulido reflejaría más la luz que una pared de hormigón).

Por último, el aspecto de la luz ambiental se puede mejorar usando la técnica de oclusión ambiental (ambient occlusion), que consiste en calcular cómo de expuesto está cada punto a la luz ambiental (por ejemplo, el interior de una cueva estaría menos expuesto a la luz ambiental conforme más nos adentrásemos en ella).

Referencias

[1] Graham Sellers and Richard S. Wright, Jr. and Nicholas Haemel, *OpenGL SuperBible Sixth Edition*, Addison Wesley, 2013.

[2] opengl-tutorial.org, *Tutorials for modern OpenGL*, http://www.opengl-tutorial.org/, 2012-2015.

[3] Etay Meiri, *Modern OpenGL Tutorials*, http://ogldev.atspace.co.uk/, 2014.

[4] Anton Gerdelan, *Anton's OpenGL 4 Tutorials*, http://antongerdelan.net/opengl/, 2014.

[5] Lighthouse3d.com, *GLSL Tutorial – Core*, http://www.lighthouse3d.com/tutorials/glsl-tutorial/, 2015.

[6] Wikipedia, *Ley de Lambert*, https://es.wikipedia.org/wiki/Ley_de_Lambert

[7] GLEW, *The OpenGL Extension Wrangler Library*, https://github.com/nigels-com/glew

[8] GLFW, OpenGL FrameWork, http://www.glfw.org/

[9] GLM, OpenGL Mathematics, http://glm.g-truc.net/

[10] Microsoft, *Formato DDS*, https://msdn.microsoft.com/en-us/library/windows/desktop/bb943990%28v=vs.85%29.aspx

[11] Autodesk, *Formato FBX*, http://www.autodesk.com/products/fbx/overview

Anexo

Anexo

A. La pipeline de OpenGL 4.3

Una pipeline consiste en ir transformando un flujo de datos en un proceso comprendido por varias fases secuenciales, siendo la entrada de cada una la salida de la anterior. En el caso de OpenGL, esta pipeline tiene una serie de fases o etapas, algunas de las cuales son programables y reciben el nombre de shaders. Cada una de estas etapas se ejecutan en paralelo tantas veces como sea necesario según los datos de entrada.

En este anexo se da una visión de la nueva pipeline empleada por OpenGL en su versión 4.3 y del recorrido que siguen los datos, así como una explicación más detallada de las etapas programables. En la figura 42 se puede ver un diagrama simplificado de dicha pipeline y sus etapas.

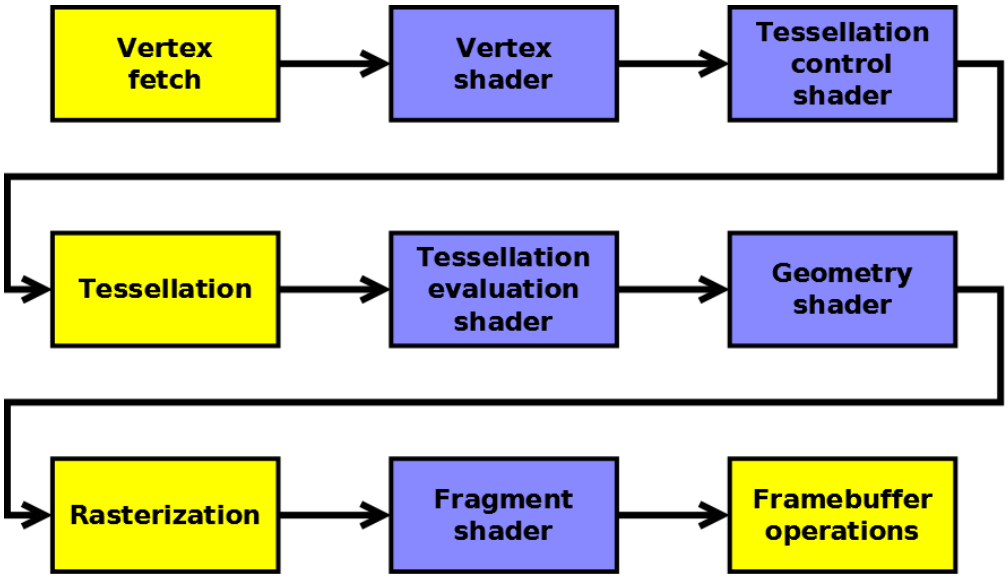


Figura 42: La pipeline simplificada de OpenGL 4.3, con las etapas no programables en amarillo y las programables en azul

La pipeline empieza con un proceso que dicta como serán introducidos los vértices de una figura en la pipeline y los prepara para la primera etapa programable. Por ejemplo, los vértices pueden ser puntos, triángulos, parches (caras de múltiples vértices) y se pueden especificar de multiples formas, como en listas, o todos seguidos.

Esta primera etapa programable (Vertex Shader) es la única que tiene que estar siempre presente, siendo las demás etapas programables opcionales (aunque sin la última etapa, el Fragment Shader, no se verá nada en la pantalla). Si una de las etapas opcionales no está presente (programada) se saltará a la siguiente).

Una vez se han preparado los vértices, existe la opción de teselar, esto es, partir las primitivas recibidas (como parches) en primitivas más pequeñas (como triángulos). Esto se controla con tres etapas seguidas, dos de las cuales son programables y controlan el proceso antes y después, y otra no programable que es la que realmente ejecuta la teselación.

Despúes viene la etapa de Geometry Shader, la única etapa que deja crear geometría nueva.

Lo siguiente que se realiza es el proceso de rasterización, un proceso automático que consiste en convertir las primitivas recibidas en un conjunto de píxeles o puntos. Concretamente OpenGL convierte las primitivas en fragmentos, que son la representación de un trozo de las primitivas. Los fragmentos producidos están relacionados con los píxeles disponibles, de manera que siempre hay al menos un fragmento por píxel, pero se pueden llegar a producir más fragmentos para un mismo píxel, dependiendo de los parámetros de OpenGL.

Estos fragmentos pasan al Fragment Shader, que es la etapa programable capaz de tratarlos.

Por último, se realizan una serie de operaciones con la salida final, de forma que esta salida se pueda mostar por pantalla.

A.1. Vertex Shader

El Vertex Shader se encarga del procesamiento de vértices individuales: recibe siempre un vértice de entrada y genera uno de salida. Es el sitio donde se suelen aplicar las transformaciones para cambiar el espacio de coordenadas.

A.2. Tessellation Control Shader

El Tessellation Control Shader recibe como entrada un parche (un conjunto de varios vértices) y controla la forma en la que será teselado. También define el número de vértices que componen el parche.

A.3. Tessellation Evaluation Shader

El Tessellation Evaluation Shader opera con las posiciones interpoladas de los vértices resultantes del proceso de teselación. Su labor es la de coger el parche abstracto generado durante la teselación así como los vértices originales del parche y generar con ellos nuevos vértices.

El TES se ejecuta al menos una vez por cada vértice generado en el parche abstracto y cada invocacion genera un solo vértice.

A.4. Geometry Shader

El Geometry Shader recibe una sola primitiva como entrada (por ejemplo, un triángulo) y genera cero o más primitivas. Es la única etapa que puede crear nueva geometría.

A.5. Fragment Shader

El Fragment Shader es la etapa que recibe un fragmento del proceso de rasterización y le asigna un conjunto de colores.

Al poder cambiar el color de los fragmentos, es la etapa utilizada para las tareas de iluminación.

B. Tarjetas gráficas utilizadas

Para analizar el rendimiento del programa se han usado una serie de tarjetas gráficas, todas diseñadas por Nvidia. Estas tarjetas gráficas son fabricadas por distintas empresas (como Gigabyte o Asus) que las fabrican según las especificaciones de Nvidia pero otorgándoles ligeras personalizaciones, como por ejemplo la cantidad y tipo de memoria.

Las tarjetas gráficas usadas han sido:

- Gigabyte GeForce GTX 660 OC 2GB GDDR5
- Gigabyte GeForce GTX 750 OC 2GB GDDR5
- Gigabyte GeForce GTX 750 Ti OC Windforce 2GB GDDR5
- Asus GeForce GTX 760 DirectCU II OC 2GB GDDR5
- Nvidia GeForce GT 650M

Para ponerlas en contexto, Nvidia nombra las tarjetas gráficas según la serie (serie 600, serie 700...) siendo cada serie superior la siguiente generación, y dentro de cada serie, se distinguen por la gama (50, 60, 80...). Además, si realizan una revisión de un modelo para mejorarlo, le añaden el sufijo Ti, y si se trata de una versión móvil (tarjetas gráficas para portátiles) el sufijo M.

Siguiendo estas indicaciones, podemos avanzar que el modelo más potente será la GTX 760, seguido del modelo de la misma gama pero de la serie anterior, la GTX 660. Después viene la gama inferior, primero la revisión GTX 750 Ti, luego la GTX 750, y por último, muy inferior a todas las demás, el modelo móvil GT 650M.

En el cuadro 2 se exponen las características de las tarjetas gráficas de Nvidia utilizadas para los análisis de rendimiento.

		GTX 660	GTX 750	GTX 750 Ti	GTX 760	GT 650M
GPU	Núcleos CUDA	960	512	640	1152	384
	Frecuencia de reloj normal (MHz)	980	1020	1020	980	900
	Frecuencia acelerada (MHz)	1033	1085	1085	1033	900
	Tasa de relleno de texturas (GTexel/s)	78.4	32.6	40.8	94.1	27.2
Mem	Frecuencia de la memoria (Gbps)	6.0	5.0	5.4	6.0	2.2
	Cantidad de memoria (MB)	2048	2048	2048	2048	1024
	Interfaz de memoria (GDDR5)	192-bit	128-bit	128-bit	256-bit	128-bit
	Ancho de banda máx. (GB/s)	144.2	80.0	86.4	192.2	80.0

Cuadro 2: Comparativa de las tarjetas gráficas usadas