Práctica 03: Programación de *shaders*

En esta práctica trabajaremos lo siguiente:

- Programación de *vertex* y *fragment shaders*
- Modelo de iluminación local y una técnica de *shading* (Phong)
- Uso de texturas y su aplicación en técnicas como *environment mapping* o *normal mapping*

Utilizaremos WebGL, como en la práctica anterior. En este caso, como la práctica se centra en programación de *shaders*, partiremos ya de un código que provee de una serie de funcionalidades básicas (carga de modelos en formato OBJ, carga de texturas, movimiento de la cámara, etc.). Este código utiliza la librería three.js para muchas de las operaciones que realiza; aunque para nosotros esto será transparente, entre otras cosas esto permitirá que los mensajes de error al compilar los *shaders* sean más fáciles de interpretar. En cualquier caso, todo lo que necesitamos para la práctica se os provee en el .zip anexo a este guión, y la única parte del código que modificaremos es la de los *shaders* utilizados.

1. Conceptos básicos de *shaders*

Como ya hemos visto, los *shaders* son programas que se ejecutan en la GPU. Como se integran dentro de un *pipeline* que tiene ciertas etapas con funcionalidad fija, los *shaders* tienen importantes requisitos sobre sus entradas, y sobre todo sobre sus salidas.

Se programan en lenguajes de alto nivel; en nuestro caso, el lenguaje que utilizaremos es GLSL (OpenGL Shading Language). Os será útil la guía disponible en Moodle, accesible también en este enlace.

De los distintos tipos de *shaders*, en esta práctica vamos a trabajar sólo con el *vertex shader* y el *fragment shader*, cuya funcionalidad hemos visto en clase (recordadla en este punto si es necesario). Actualmente, ambos comparten el mismo conjunto de instrucciones, aunque pueden tener ciertas particularidades (por formar parte de ese *pipeline* con etapas de funcionalidad fija). Las instrucciones que tendremos que usar en esta práctica son sencillas, y en los casos en que se necesita alguna instrucción más específica, por regla general se os indica. Antes de comenzar, indicamos aquí aspectos generales que serán de relevancia:

- **Flujo de información y calificadores de almacenamiento**: De entre los muchos calificadores disponibles, tres van a ser de particular relevancia aquí: `uniform`, `attribute` y `varying`.
    - Las variables declaradas como `uniform` son de sólo lectura para el *shader*, designan información que entra al *shader* "desde el exterior", y tienen el mismo valor para todos los elementos (vértices, *fragments*) de una primitiva.
    - Las variables declaradas como `attribute` son entradas al *vertex shader*, y tienen un valor distinto por vértice. Ejemplos de atributos son: posición, normal, o coordenadas de textura del vértice.

    - Las variables declaradas como `varying` son aquellas que van a servir de interfaz entre el *vertex* y el *fragment shader*: son variables de salida del *vertex shader*, y de entrada al *fragment shader* (y se declaran en ambos como `varying`). Por tanto, los valores así declarados serán interpolados entre las etapas del *vertex* y el *fragment shader*, para dar valores por *fragment*.
- **Tipos y acceso a las componentes de un vector:** Los tipos básicos disponibles los podéis ver en la Sección 4.1 de la guía de GLSL proporcionada. Para esta práctica, fundamentalmente necesitaréis vectores y matrices de floats, además de los básicos int, float,… y los samplers que se usan para acceder a texturas (éstos se comentan en el siguiente punto). Cabe mencionar que se puede acceder a una o varias componentes de un vector con el operador ".", y que éstas se denotan, en orden, `xyzw` (generalmente usado para vectores de posición, normales…), `rgba` (generalmente usado para vectores que representan colores), o `stpq` (generalmente usado para coordenadas de textura).
- **Acceso a texturas:** El acceso a texturas se hace mediante variables denominadas *samplers*, que serán de un tipo u otro según el tipo de textura al que queramos acceder (e.g., `sampler2D` para texturas 2D, o `samplerCube` para texturas en formato *cube map*). En cuanto a las funciones para obtener el valor del texel correspondiente, sólo utilizaremos dos: `texture2D()` para acceder a texturas 2D, y `textureCube()` para acceder a texturas en formato *cube map*.

2. Programa base proporcionado

En esta práctica, se os proporcionan cuatro ficheros HTML (`simpleTexture.html`, `phong.html`, `envMapping.html` y `normalMapping.html`). Todos ellos utilizan una serie de ficheros Javascript a los que ya enlazan. Sólo tendréis que editar los ficheros HTML, y, en concreto, el código de los *shaders* que contienen. De los cuatro, `simpleTexture.html` es una base proporcionada como referencia, y el resto tendréis que completarlos como parte de esta práctica, para que tengan la funcionalidad requerida (ver Secciones 3 a 5 de este guión).

En esta sección se describe brevemente `simpleTexture.html`, y en las siguientes se explican las tres tareas a realizar para la práctica.

Si no lo habéis hecho ya, abrid `simpleTexture.html` en el navegador para ver qué hace. Veréis que hay unos ejes de coordenadas, y varios botones, *checkboxes* y *sliders*. Además, se puede mover la cámara con el ratón.

- Los botones permiten cargar un modelo (una malla) en formato OBJ[1], y una textura a aplicar a dicho modelo. Junto con el código base se os proporcionan, en la carpeta "data", varias mallas y varias texturas para utilizar en la práctica.
- Los *checkboxes* permiten activar o desactivar ciertas funcionalidades que os pueden ayudar a visualizar, comprobar y entender vuestros resultados.
- Los *sliders* controlan ciertos parámetros relevantes para la visualización (aquí únicamente la longitud de las normales, si se muestran con el *checkbox* correspondiente). Algunos de ellos podrán ser *uniforms* pasados a los *shaders*, como el de exposición (*exposure*), que aquí no tiene ningún uso.

[1] El formato OBJ es un formato de archivo abierto diseñado para definir la geometría (la malla) de un modelo. No es objetivo de esta práctica o de la asignatura, pero si queréis saber más sobre él podéis consultar: https://www.cs.utah.edu/~boulos/cs3505/obj_spec.pdf

Probad a cargar un fichero .obj de los disponibles en "data/meshes", y una textura de las de "data/textures". Probad además el efecto de los distintos *checkboxes* (el de fijar las luces no tiene efecto aquí, porque no hay ningún cálculo de iluminación). Un interfaz similar a éste será el que tendréis en todas las tareas a realizar.

Mirando el código de `simpleTexture.html`, al comienzo del *vertex shader* veréis, comentados, todos los *attributes* y *uniforms* que están siendo pasados al *vertex shader*. Todos ellos pueden ser utilizados en vuestro código del *shader*, como sucede en el ejemplo con `position`, uv, `modelViewMatrix`, o `projectionMatrix`. En las distintas tareas a realizar, los *uniforms* van a estar ya declarados en el código del *shader* correspondiente.

El resto del código, tanto del *vertex* como del *fragment shader*, es sencillo y está comentado.

3. *Phong shader*

En esta parte tenéis que completar el código de `phong.html`. Se os proporciona una parte del código a modo de esqueleto o ayuda. La Figura 1 muestra un ejemplo de resultado.

El objetivo de esta parte es implementar un modelo de iluminación sencillo (usando una BRDF de Blinn-Phong, vista en clase, Tema 6) con *shading* de Phong (Tema 6.5).

Dicho modelo de iluminación calculará la intensidad resultante $\mathbf{I}$ como:

$$\boldsymbol{I} = \frac{1}{r^2}\left(\boldsymbol{c}_{diff} + \boldsymbol{c}_{spec}(\boldsymbol{n}\cdot\boldsymbol{h})^m\right)\boldsymbol{I}_{luz}(\boldsymbol{n}\cdot\boldsymbol{l})$$

donde: $\mathbf{I}_{\text{luz}}$ es la intensidad o color de la luz incidente; $\mathbf{n}$ es la normal en el punto; $\mathbf{l}$ es el vector que va del punto a la luz; $\mathbf{h}$ es el vector que marca la bisectriz del ángulo formado por $\mathbf{v}$ y $\mathbf{l}$, con $\mathbf{v}$ el vector que parte del punto y va hacia la cámara; $\mathbf{c}_{\text{diff}}$ es el coeficiente de reflexión difusa, proporcionado por el valor de la textura en ese punto[2]; $\mathbf{c}_{\text{spec}}$ es el coeficiente de reflexión especular, que fijaremos a 1.0 para los tres canales; $\mathbf{m}$ indica el valor de rugosidad o *roughness*[3]; y $\mathbf{r}$ es la distancia entre la luz y el punto.

Recordad que $\mathbf{n}$, $\mathbf{l}$ y $\mathbf{h}$ son vectores unitarios (la función `normalize()` de GLSL os ayudará con esto); y que tanto el producto $\mathbf{n}\cdot\mathbf{l}$ como el producto $\mathbf{n}\cdot\mathbf{h}$ no pueden ser negativos.

Si hay varias luces, la intensidad o color resultante es la suma de la contribución de cada una de ellas.

Como sabéis, dado un cierto modelo de iluminación, hay distintas maneras de implementar el *shading* (Tema 6.5). Aquí utilizaremos Phong *shading*, por lo que los cálculos de iluminación se harán en el *fragment shader*, y en espacio de la cámara (denominado *camera space*, *view space* o *eye space*).

Nota adicional:

- `to_sRGB()` y `from_sRGB()` son dos funciones sencillas que hacen el paso hacia y de sRGB (*standard RGB*, un espacio de color), respectivamente. No tenéis que tocar su

[2] Para acceder a la textura podéis utilizar la función `texture2D()`, y el `sampler2D diffuseTexture` ya declarado en el código proporcionado.
[3] Como veréis, en el código se utiliza su inverso, esto es, la variable `roughness` contiene $1/m$.

implementación, simplemente debéis utilizar `from_sRGB()` para convertir el valor que os devuelve el acceso a la textura que se usa para la componente difusa.

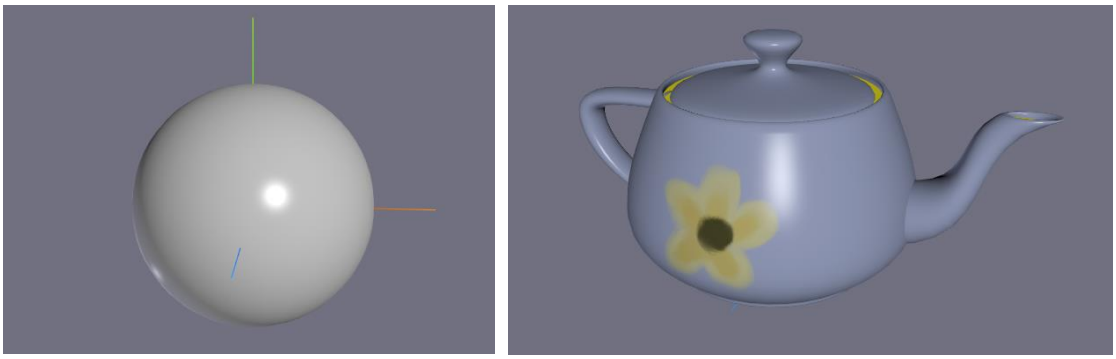

Figura 1. Ejemplos del resultado, sin (*izquierda*) o con (*derecha*) la selección de una textura difusa.

4. *Environment mapping*

En esta parte tenéis que completar el código de `envMapping.html`. Se os proporciona una parte del código a modo de esqueleto o ayuda. La Figura 2 muestra un ejemplo de resultado.

El objetivo de esta parte es implementar *environment mapping* (Tema 6), técnica en la que se usa una textura para simular el entorno en el que está situado un objeto. Dicha textura se denomina mapa de entorno, y la idea es que sirve para iluminar la escena. En el caso más sencillo (el nuestro), simplemente cada punto del objeto refleja de forma especular ideal el punto correspondiente del entorno, produciendo un efecto como el mostrado en la Figura 2. Para saber qué punto del mapa de entorno contribuye a cada punto del objeto, se calcula y utiliza el vector de reflexión en ese punto.

Consideraciones para la implementación:

- El mapa de entorno será una textura en formato *cube map*. Podéis verla en la carpeta "data/textures/LancellottiChapel". Para acceder a la misma utilizaréis el *uniform* de tipo `samplerCube` ya declarado, y la función `textureCube()`.
- En las diapositivas del Tema 6 tenéis explicado el concepto y la información (los vectores) necesarios para implementarlo. Como veréis ahí, podéis calcular el vector de reflexión $\mathbf{r}$ a partir de la normal $\mathbf{n}$ y el vector $\mathbf{v}$, utilizando la función `reflect()` de GLSL; la función `reflect()` requiere que $\mathbf{v}$ vaya de la cámara al punto.
- Haced los cálculos en espacio de la cámara.

Notas adicionales:

- Podéis asumir que el mapa de entorno está fijo en el espacio de la cámara, esto es, el ratón rota el objeto respecto al mapa de entorno, pero el mapa de entorno permanece fijo respecto a la cámara.
- Podéis ignorar el parámetro *exposure* para la resolución, no tenéis que hacer nada con él. De la misma forma, no necesitáis las funciones `to_sRGB()` o `from_sRGB()`.
- Por la forma en que se cargan las texturas para el *cube map*, por restricciones de seguridad, tendréis que utilizar un servidor para poder utilizarlas. Podéis hacerlo de

forma sencilla con Python, simplemente ejecutando, en la carpeta de `envMapping.html`, el comando[4]:

```
python -m http.server 8000
```

Hecho esto, ya podéis acceder a `http://localhost:8000/envMapping.html` y debería funcionar la carga de texturas.

Figura 2. Ejemplo del resultado de *environment mapping*.

5. *Normal mapping*

En esta parte tenéis que completar el código de `normalMapping.html`. Se os proporciona una parte del código a modo de esqueleto o ayuda. La Figura 3 muestra un ejemplo de resultado.

El objetivo de esta parte es implementar *normal mapping* (Tema 6), técnica en la que se usa una textura para almacenar las normales que serán utilizadas en el cálculo de iluminación. Estas normales almacenadas en la textura ofrecen un nivel de detalle mucho mayor que las normales geométricas, las de la malla, por lo que es una forma eficiente de simular detalle de un objeto.

Para esta parte calcularéis la iluminación como habéis hecho en la Sección 3, esto es, tenéis que tener el Phong *shader* hecho y funcionando para hacer esta parte. El cambio que supone esta parte respecto a la Sección 3 es que las normales usadas para calcular la iluminación no serán las de los vértices, sino que se leerán de una textura (*normal map*).

La idea en esta práctica es que utilicéis la esfera como geometría, `earthNormalMap_1k.png` como mapa de normales, y `earthmap1k.jpg` como textura difusa, tal y como muestra el ejemplo de la Figura 3.

Consideraciones para la implementación:

- La textura contiene las normales en espacio tangente (*tangent space*). Éste es un espacio definido en cada punto por: (i) la normal en dicho punto, (ii) la tangente (podría ser cualquiera, se toma la tangente en la dirección de la coordenada *u* creciente de las coordenadas de textura), y (iii) la bitangente (perpendicular a los dos vectores anteriores).
- La tangente y la bitangente en cada punto se proporcionan ya como variables `attribute`, que simplemente hay que transformar en el *vertex shader* a espacio de la cámara y pasarlas al *fragment shader* (es decir, habrá dos `varying` más para ello).
- Será el *fragment shader* el que haga los cálculos de iluminación. Para hacerlos, todos los vectores tienen que estar en el mismo espacio de coordenadas. Si recordáis, en la

[4] Éste es el comando para Python 3; para Python 2: `Python -m SimpleHTTPServer 8000`

Sección 3 hemos dicho que los cálculos de iluminación se harían (y se harán) en el espacio de la cámara. Sin embargo, la normal de la textura está en espacio tangente, así pues, hay que transformar dicha normal a espacio de la cámara con la matriz de transformación adecuada.

- La matriz de transformación del espacio tangente a la cámara, $M_{t\text{-}c}$, se puede construir en el *fragment shader* a partir de los tres vectores que forman dicho espacio, expresados en coordenadas de la cámara (la normal (geométrica), la tangente, y la bitangente, provenientes del *vertex shader*). Se trata de una matriz ortogonal (y por tanto, su inversa es igual a su traspuesta).

Notas adicionales:

- La textura de normales (*normal map*) almacena las tres coordenadas de las normales (x, y, z) en los tres canales de color de la imagen (r, g, b). Las coordenadas de las normales pueden tener valores negativos, mientras que en una imagen los valores de (r, g, b) están siempre en el rango [0, 1]. Por tanto, tendremos que transformar los valores (r, g, b) de la textura restando 0.5 a cada componente, para recuperar los valores negativos.
- La textura que tenéis para el normal map (`earthNormalMap_1k.png`) tiene las normales codificadas en formato de DirectX, esto es, la componente y (el canal verde) está invertida (es positiva si va "hacia abajo", y negativa si va "hacia arriba"). Así pues, al usarla, vuestro código deberá invertir la componente y de la normal de la textura.
- Se ha añadido una variable de tipo *uniform* denominada `bumpiness`, que tiene un *slider* asociado en el interfaz, y que es una forma un tanto cruda de regular la fuerza o intensidad del *normal map*. Para utilizarla en el *shader*, basta con multiplicar las coordenadas x e y de la normal de la textura por el valor de `bumpiness`.

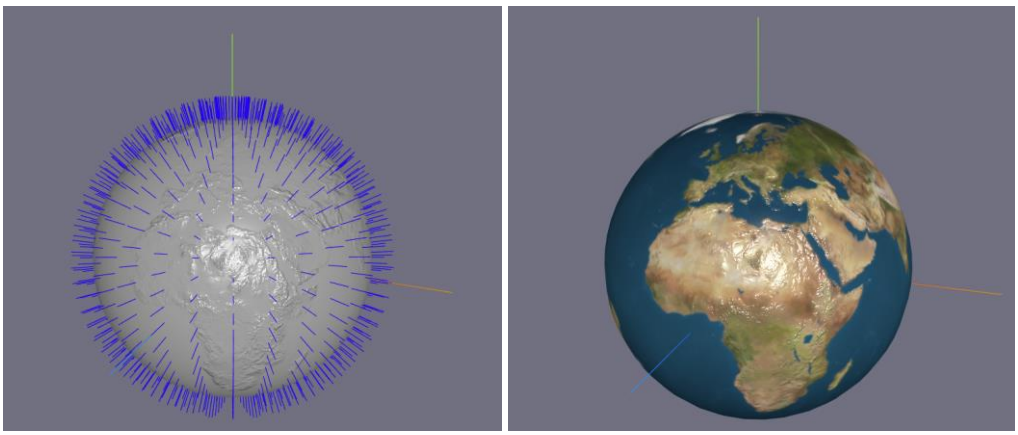

Figura 3. *Normal mapping*. *Izquierda*: Sin textura difusa, y con las normales geométricas añadidas, para ilustrar mejor el efecto. *Derecha*: Resultado incluyendo *normal mapping* y la textura difusa.

6. Entrega

Se entregarán únicamente los ficheros .html que contienen la resolución de la práctica (*no* hay que entregar los ficheros Javascript proporcionados). El código añadido a la base proporcionada debe estar claramente indicado y apropiadamente comentado. Se entregará también un *readme* que contenga los nombres de los autores, y el tiempo invertido por cada autor. Si habéis tomado alguna decisión de diseño ante algún aspecto no especificado en el guión, indicadla también en

el *readme*, de forma razonada. Todo ello se entregará comprimido en un único fichero .zip de nombre: `VID_practica03_GrupoXX.zip`, con XX el número de vuestro grupo.

No se utilizarán librerías adicionales, ni código de terceros adicional al proporcionado, para la resolución. En el caso de consultar alguna fuente para la resolución (más allá de la documentación de la API), se debe incluir en el *readme*.

La entrega se realizará vía Moodle, a través del enlace habilitado al efecto. En dicho enlace de Moodle se indica la fecha límite de entrega.

Consideraciones adicionales: *Debugging*

Depurar código de *shaders* no es trivial, ya que en principio no disponemos de las posibilidades habituales. Para esta práctica, como las tareas a realizar son sencillas, bastará con los mensajes que proporcionará three.js en la consola. Si eso no es suficiente, se pueden mostrar los datos a inspeccionar como color, renderizar esos valores (e.g., las normales). Sed además especialmente cuidadosos con los nombres de variable, para tener claro qué almacena cada una y en qué espacio de coordenadas.

Referencias

- The OpenGL Shading Language: GLSL 4.3.0 Reference. John Kessenich, Dave Baldwin, and Randi Rost.
- Esta práctica está adaptada de una práctica del curso "*Introduction to Computer Graphics*" de Cornell University, impartido por Steve Marschner.