Universidad Rey Juan Carlos

ETSII

Práctica 1: Introducción a *GLSL*

Grado en Ingeniería de Computadores
Grado en Ingeniería de Software

Gráficos por Computador

(Curso 2024/25)

Marcos García Lorenzo

Índice de contenidos

Introducción

El propósito de esta práctica es que el alumno desarrolle una comprensión profunda de los conceptos fundamentales de la programación de *shaders* presentados en la parte teórica de la asignatura. A través de una serie de pasos progresivos, los estudiantes tendrán la oportunidad de implementar y depurar sus primeros *shaders*.

Normativa

Este documento contiene el guion de una de las tres prácticas guiadas, además de incluir tareas que deben integrarse en el proyecto final y que pueden realizarse utilizando el contenido trabajado en el bloque guiado.

Bloque guiado

El bloque guiado se llevará a cabo en el aula de prácticas bajo la supervisión directa del profesor, con el objetivo principal de introducir a los alumnos en la aplicación práctica de los fundamentos teóricos de la asignatura. Esta etapa busca proporcionar a los estudiantes las herramientas necesarias para que posteriormente puedan trabajar de forma autónoma.

La asistencia del alumno al bloque guiado es obligatoria. Este bloque tiene un peso significativo en la calificación de la asignatura y no puede ser reevaluado en la convocatoria extraordinaria en caso de inasistencia.

Durante la sesión, el profesor proporcionará orientación en cada paso del proceso. **Se recomienda a los alumnos enfocarse en comprender y analizar el código proporcionado**, en lugar de centrarse exclusivamente en desarrollar un código funcional. Las soluciones completas estarán disponibles en el aula virtual para su consulta posterior.

Proyecto final

Los grupos de prácticas deberán realizar un proyecto final de forma no guiada fuera del horario de la asignatura. Cada una de las tres prácticas guiadas contiene tareas que deberán incluirse en el proyecto final. Estas tareas están relacionadas con el contenido tratado en la parte guiada. Se distinguen tareas de dos tipos: obligatorias y optativas. Las tareas obligatorias deben completarse para alcanzar la calificación mínima en la práctica final. No obstante, para obtener la máxima puntuación, es necesario realizar varias de las tareas optativas.

El código del proyecto deberá entregarse a través del *aula virtual* dentro del plazo establecido en la página web de la asignatura. Además del código, se deberán incluir todos los ficheros de configuración del proyecto, así como todos los modelos y librerías utilizados. **Si el proyecto entregado no contiene toda la información necesaria para ser compilado y ejecutado, la práctica será valorada como no**

apta. Por último, se deberá adjuntar una memoria explicativa breve que demuestre la realización y comprensión de todas las tareas obligatorias y opcionales implementadas en la práctica, **incluyendo capturas de pantalla cuando sea necesario**. Se deberá hacer especial hincapié en la descripción de las partes optativas.

Recomendaciones y normas para las partes obligatoria y opcional

- Todas las actividades realizadas en el bloque guiado, así como todas las tareas, formarán parte del contenido evaluable del examen final.
- *Formación de grupos*: Las prácticas podrán realizarse en grupos de un máximo de cinco personas. Aunque es posible realizarlas de forma individual, no se recomienda debido a la carga de trabajo que requieren.
- **Aunque la entrega de la práctica final se realiza una única vez, es fundamental completar las tareas propuestas tras cada bloque guiado. El tamaño de este proyecto es lo suficientemente grande como para no poder llevarse a cabo en las semanas previas a la fecha de entrega.**
- Se recomienda dividir el desarrollo de cada una de las tareas del proyecto final en pasos que puedan probarse de forma independiente. Esto facilita la depuración y garantiza la correcta implementación de la solución.

Parte guiada

Paso 1: Implementación de un *Shader* básico

1. El profesor explicará el entorno compuesto de:
    a. Un proyecto de Visual Studio que implementa la funcionalidad del cliente.
    b. La librería *IGlib* que esconde la funcionalidad de *OpenGL* utilizada.
    c. El modelo 3D de un cubo.
    d. El esqueleto de un *shader* de vértices (*shader.v0.vert*) y de un *shader* de fragmentos (*shader.v0.frag*), sobre los que se implementará la funcionalidad requerida.
2. Copia los ficheros *shader.v0.vert* y *shader.v0.frag* a *shader.v1.vert* y *shader.v1.frag*.
3. Modifica el fichero *main.cpp* para que utilice los nuevos *shaders* de vértices y fragmentos.
4. Modifica el *shader* de fragmentos de forma que todos los fragmentos se pinten del mismo color.
5. Diseña un *shader* de vértices de forma que los vértices del modelo se proyecten utilizando perspectiva simétrica con una apertura de 60º. Sitúa la cámara de forma que sea capaz de visualizar toda la escena.

    Recuerda que la matriz de proyección en *OpenGL* es:

$$\mathbf{P} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Paso 2: Cliente/Servidor

En este paso el cliente suministrará las transformaciones al *shader* de vértices:

1. Copia los ficheros *shader.v1.vert* y *shader.v1.frag* a *shader.v2.vert* y *shader.v2.frag*.
2. Modifica el fichero *main.cpp* para que utilice los nuevos *shaders* de vértices y fragmentos.

3. Modifica el fichero *main.cpp* de forma que la cámara se sitúe en el punto (0, 0, 6), mirando en la dirección de las *Zs* negativas y que la matriz de proyección defina una vista simétrica con una apertura de 60º.
4. Modifica el *shader* de vértices para que utilice las nuevas matrices de vista y proyección.
5. Ejecuta la aplicación.
6. Utiliza la función *Idle* para rotar el cubo sobre su eje.

Paso 3: Mejorando la eficiencia

La librería *IGL* calcula los productos de las matrices *model, view y projection*. Esto nos permite acelerar el procesado de los vértices.

1. Copia los ficheros *shader.v2.vert* y *shader.v2.frag* a *shader.v3.vert* y *shader.v3.frag*.
2. Modifica el fichero *main.cpp* para que utilice los nuevos *shaders* de vértices y fragmentos.
3. Mejora la eficiencia del *shader* de vértices.

Paso 4: Depurando primitivas

1. Copia los ficheros *shader.v3.vert* y *shader.v3.frag* a *shader.v4.vert* y *shader.v4.frag*.
2. Modifica el fichero *main.cpp* para que utilice los nuevos *shaders* de vértices y fragmentos.
3. Modifica la funcionalidad del *shader* de fragmentos para que pinte cada primitiva de un color distinto.

Paso 5: Depurando vértices

1. Copia los ficheros *shader.v4.vert* y *shader.v4.frag* a *shader.v5.vert* y *shader.v5.frag*.
2. Modifica el fichero *main.cpp* para que utilice los nuevos *shaders* de vértices y fragmentos.
3. Modifica la funcionalidad del *shader* de vértices y del *shader* de fragmentos para que pinte cada primitiva de un color distinto.

Paso 6: Color por vértice

1. Copia los ficheros *shader.v5.vert* y *shader.v5.frag* a *shader.v6.vert* y *shader.v6.frag*.
2. Modifica el fichero *main.cpp* para que utilice los nuevos *shaders* de vértices y fragmentos.
3. Modifica la funcionalidad del *shader* de vértices de forma que el color del fragmento venga determinado por el color de los vértices de la primitiva a la que pertenece.

Paso 7: Acceso a texturas

1. Copia los ficheros *shader.v6.vert* y *shader.v6.frag* a *shader.v7.vert* y *shader.v7.frag*.
2. Modifica el fichero *main.cpp* para que utilice los nuevos *shaders* de vértices y fragmentos.
3. Modifica el fichero *main.cpp* para que asigne una textura al objeto (`IGlib::addColorTex`).
4. Modifica la funcionalidad del *shader* de vértices y del *shader* de fragmentos de forma que el color final del fragmento venga determinado por una textura.

Paso 8: Depurando normales (I)

1. Copia los ficheros *shader.v7.vert* y *shader.v7.frag* a *shader.v8.vert* y *shader.v8.frag*.
2. Modifica el fichero *main.cpp* para que utilice los nuevos *shaders* de vértices y fragmentos.
3. Modifica la funcionalidad del *shader* de vértices y del *shader* de fragmentos de forma que el color final del fragmento venga por su normal.

Paso 9: Depurando normales (II)

1. Copia los ficheros *shader.v8.vert* y *shader.v8.frag* a *shader.v9.vert* y *shader.v9.frag*.
2. Modifica el fichero *main.cpp* para que utilice los nuevos *shaders* de vértices y fragmentos.
3. Modifica la funcionalidad del *shader* de vértices y del *shader* de fragmentos de forma que el color final del fragmento venga por su normal en coordenadas de la cámara.

Tareas para el proyecto final

Tareas obligatorias

Las tareas descritas a continuación deberán realizarse de forma obligatoria:

1. **Implementa una matriz de proyección que mantenga el *aspect ratio* al cambiar el tamaño de la ventana.** En la parte guiada, la matriz de proyección está diseñada de forma que el ángulo de apertura horizontal es igual al vertical. Para garantizar que el *aspect ratio* se conserve, debe fijarse la apertura vertical (60º) y calcular la apertura horizontal teniendo en cuenta la relación de aspecto de la ventana. Es importante que la apertura vertical permanezca constante mientras que la otra se ajuste dinámicamente para mantener la proporción adecuada. *No puede utilizarse ninguna función de GLM para calcular la matriz de proyección*. Recuerda que:

$$\mathbf{P}_{0,0} = \frac{2n}{r-l} = \frac{2n}{w}$$
$$\mathbf{P}_{1,1} = \frac{2n}{t-b} = \frac{2n}{h}$$

   Aunque w y h están definidos en coordenadas de la cámara en el plano cercano ($near$), la función que maneja el evento de cambio de tamaño de la ventana nos proporciona el número de píxeles (W, H). Sin embargo, el ratio $\frac{W}{H}$ (relación de aspecto en píxeles) y el ratio $\frac{w}{h}$ (relación de aspecto en coordenadas de la cámara) deben ser equivalentes para garantizar que la proyección no se distorsione.

2. **Control de la cámara con el teclado (First Person Shooter)**. Se deberán implementar controles básicos para la cámara, que incluyan: movimiento hacia adelante, retroceso, desplazamientos laterales (izquierda y derecha) y giros (izquierda y derecha). No está permitido utilizar funciones de *glm*, únicamente aquellas que permitan definir transformaciones básicas como rotaciones, traslaciones y escalados. Será necesario almacenar y gestionar los vectores $\mathbf{lookAt}$, $\mathbf{up}$ y $\mathbf{CoP}$, modificándolos según los eventos de teclado que se produzcan. A partir de estos vectores, se deberá construir una nueva matriz de vista ($\mathbf{view}$) desde cero después de cada acción del usuario, sin reutilizar la matriz $\mathbf{view}$ previamente existente. Las siguientes implementaciones *NO* son válidas:

$$\mathbf{view} = \mathbf{T}_{z,\text{inc}} \cdot \mathbf{view}$$
o
$$\mathbf{view} = \mathbf{view} \cdot \mathbf{Rot}_{y,\alpha}$$

3. **Carga la geometría desde un fichero.** Se recomienda el uso de la librería *ASSIMP* (http://www.assimp.org/). La manera más sencilla es instalarla a través de *NuGet*.

4. **Añade al menos 2 objetos a la escena**. Al menos uno de estos objetos debe describir una trayectoria cíclica. Lo más sencillo es hacerlo orbitar alrededor de un punto.

Tareas optativas

Las tareas que se describen a continuación podrán realizarse de forma opcional:

1. Controlar el movimiento de alguno de los objetos móviles de la escena utilizando curvas de *Bézier*, *splines* cúbicos o polinomios de interpolación de *Catmull-Rom*. (*relevancia media*)
2. Utilizar una función matemática para dar color a las caras de los objetos. Se pueden usar las coordenadas de textura como parámetros de la función. Ejemplos: (*relevancia media*)
    - The Book of Shaders by Patricio Gonzalez Vivo & Jen Lowe[1] - Cap 10 y Cap 11
    - ShaderToy Perlin
    - Puedes buscar o inventarte tu propia función.
    - Ejemplo - dibujar un círculo utilizando la sentencia `discard`:

$$(r - 0.5)^2 + (s - 0.5)^2 < 0.2$$

```
**Nota: $r$ y $s$ son las coordenadas de textura del
fragmento.**
```

    - Ejemplo - función inspirada en YT

```
vec3 F0 = vec3(0.0);
// Ajustar el valor para controlar el efecto.
float FPower = 2.0; // 5.0 valor usado en PBR
vec3 fresnelSchlick(in vec3 N, in vec3 V, in vec3 F0, in float
FPower)
{
    return F0 + (vec3(1.0) - F0) * pow(1.0 - max(dot(V, N), 0),
FPower);
}
```

Bibliografía

[1] P. G. Vivo y J. Lowe, *The Book of Shaders*. 2023. Disponible en: https://thebookofshaders.com/