APUNTES, MANUALES, PRESENTACIONES

Universidad de Jaén

Escuela Politécnica Superior de Jaén

Manual complementario de prácticas de programación de aplicaciones gráficas

Alfonso López Ruiz

2022-2023

Programación de aplicaciones gráficas

Universidad de Jaén

Departamento de Informática

Manual complementario de prácticas

Programación de aplicaciones gráficas

Grado en Ingeniería Informática

Alfonso López Ruiz

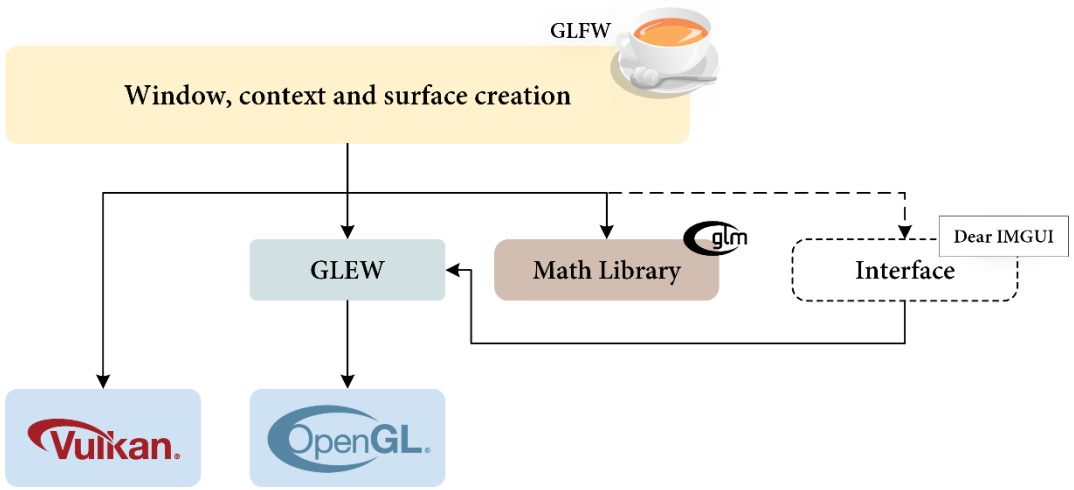
Elaborado en el curso académico 2022-2023

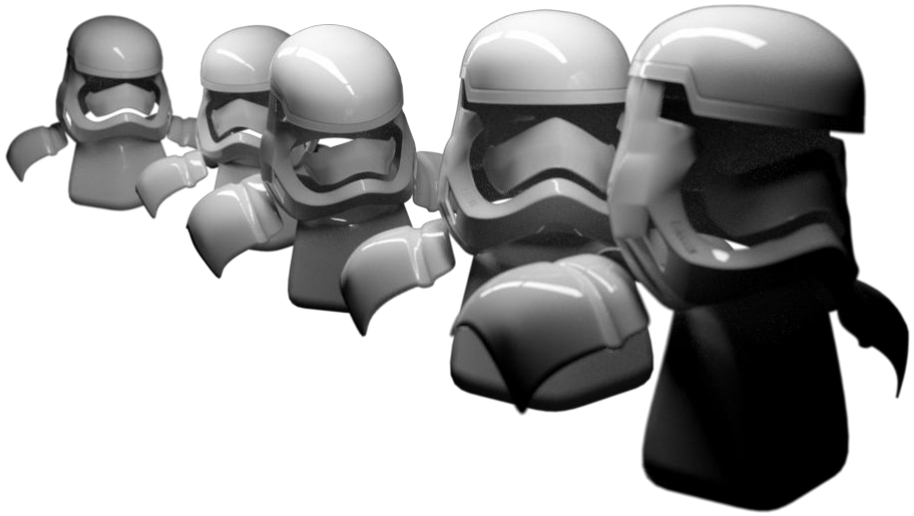
Objetivos de la práctica 1

- **Construir un proyecto de Visual Studio para comenzar las prácticas.**
- **Enlazar librerías.**
  - GLFW.
  - GLEW.
  - GLM.
- **Comprobar que nuestro proyecto funciona correctamente con dichas librerías.**
  - Creación de ventana y contexto gráfico de OpenGL.
  - Inicialización de GLEW.
  - Definición de *callbacks*.

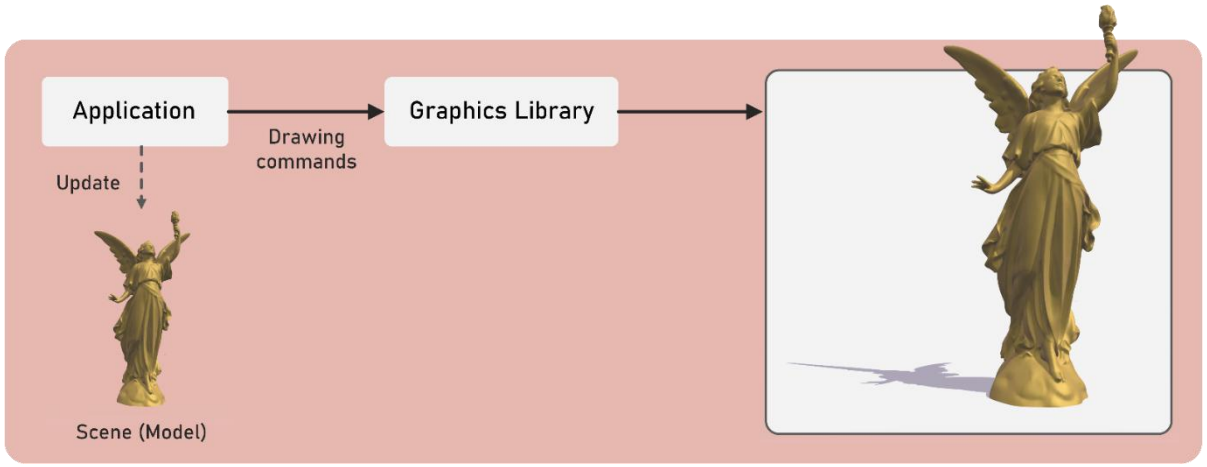
Librerías


GLSL PathTracer
Settings
Help
Samples: 598


GLFW
Window, context and surface creation
GLEW
Math Library
glm
Interface
Dear IMGUI
Vulkan
OpenGL

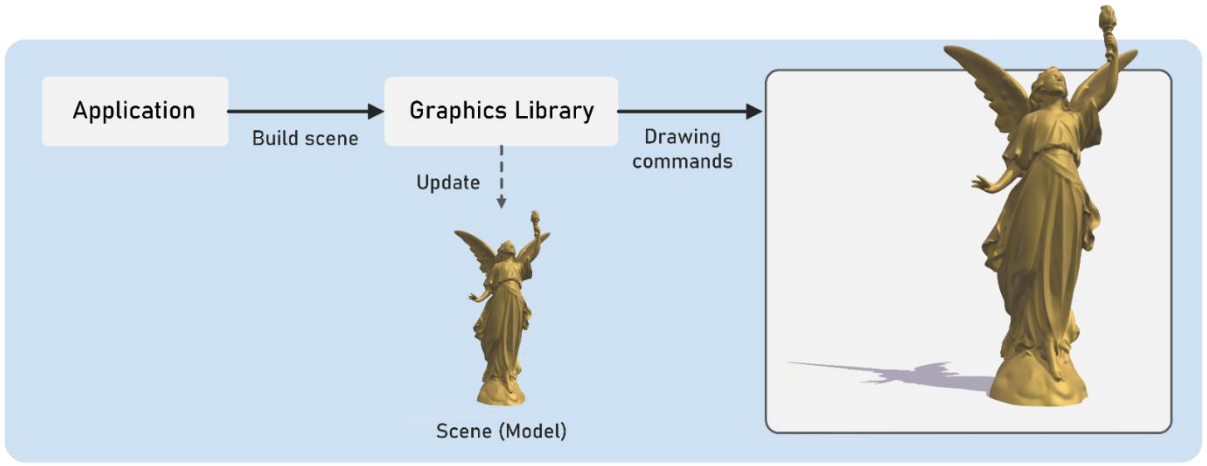


Tipos de aplicaciones gráficas



Immediate

- Informática Gráfica y Visualización.
  - glBegin, glEnd...
- Información de la escena en la aplicación.
- La información de la escena no se encuentra en la GPU, solo se envían comandos de dibujo.



Retained

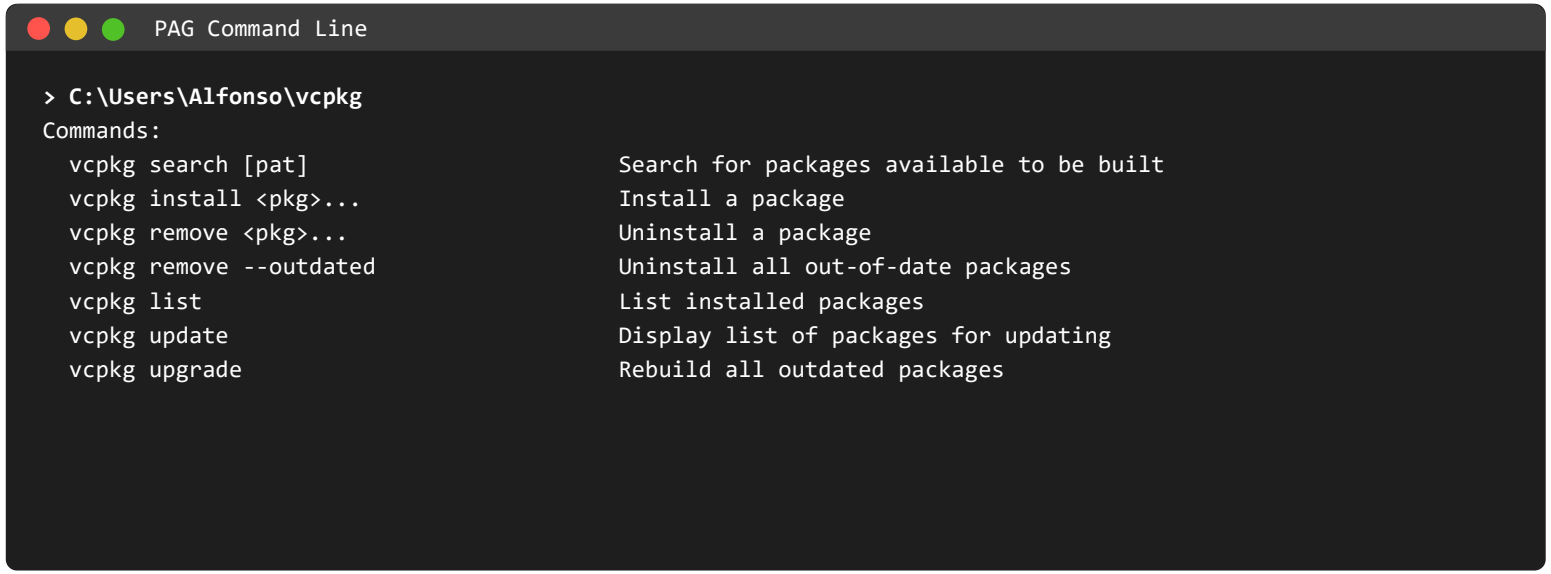
- La información de la escena se transfiere a la GPU.
- Desde la aplicación solo disponemos de *buffers* con información.

Librerías

- **Podemos enlazarlas...**
  - **Mediante gestor de paquetes: vcpkg.**
    - Mucho más sencillo.
    - Librerías globalmente accesibles para cualquier proyecto.
  - **Descargando dependencias, compilándolas (si fuera necesario) y definiendo localización de las mismas:**
    - C++ > General: Carpeta include/ (donde se encuentra el código fuente).
    - Vinculador > General: Carpeta con los ficheros .lib.
    - Vinculador > Entrada: Módulos concretos que necesitamos de cada librería.
    - Si necesitamos crear otra aplicación gráfica, será necesario volver a definir localizaciones, o modificar un proyecto de Visual Studio previo.
    - Es necesario añadir los ficheros .dll a nuestra carpeta Debug o Release.

vcpkg

- **Por comodidad podemos...**
  - Enlazar vcpkg al Path del sistema.
    - Una nueva línea en 'Path': C:/vcpkg
  - Definir la arquitectura del sistema (y de las librerías que buscamos).
    - x86, x64
    - Nueva variable del sistema. `VCPKG_DEFAULT_TRIPLET: x64-windows`



GLFW. Tarjeta gráfica

- **Al crear la ventana y el contexto, se seleccionará una tarjeta gráfica que disponga de los requisitos establecidos.**
  - Ordenadores portátiles: tarjeta gráfica integrada.

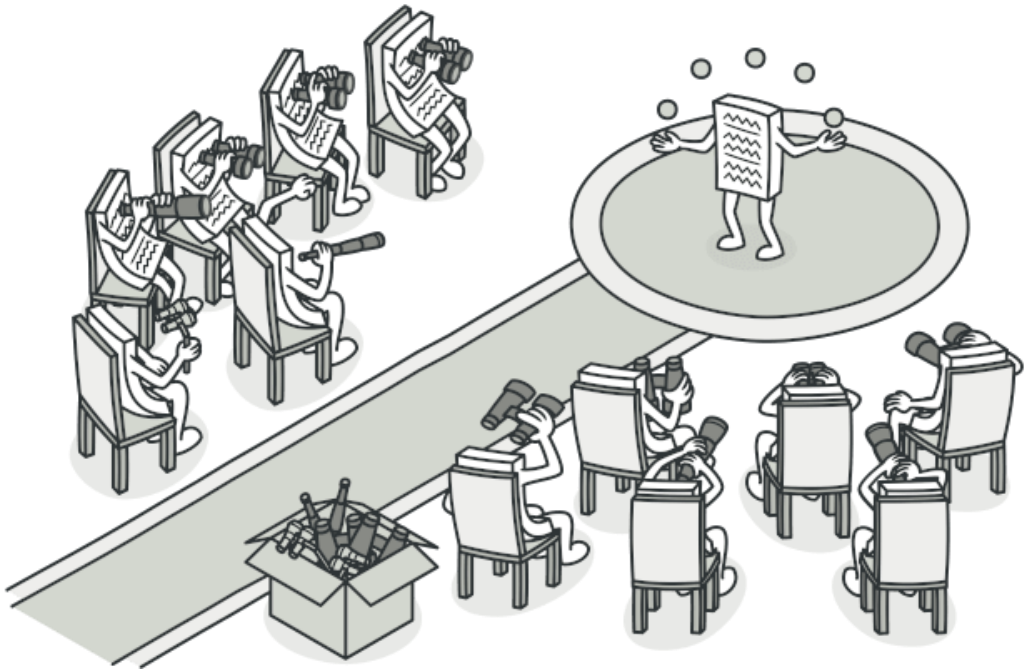
```
#include <windows.h>         // DWORD is undefined otherwise

// Laptop support. Use NVIDIA graphic card instead of Intel
extern "C" {
       _declspec(dllexport) DWORD NvOptimusEnablement = 0x00000001;
}
```

- **En Vulkan es posible gestionar dispositivos a muy bajo nivel y elegir el más adecuado.**

GLFW callbacks

- **Patrón “Observer”:**
  - Nos suscribimos a un evento, y nos notifican cuando se produzca dicha interacción.
  - La suscripción es para cada tipo de evento.
  - Limitado a un único observador (puede no haberlo).
  - El observador debe ser globalmente accesible.

```
while (!glfwWindowShouldClose(_window)) {
      glfwPollEvents();
      render();
}
```



Refactoring Guru©

GLFW callbacks

- **Patrón “Observer”:**
  - Nos suscribimos a un evento, y nos notifican cuando se produzca dicha interacción.
  - La suscripción es para cada tipo de evento.
  - Limitado a un único observador (puede no haberlo).
  - El observador debe ser globalmente accesible.

```
while (!glfwWindowShouldClose(_window)) {
      glfwPollEvents();
      render();
}
```

Refactoring Guru©

Práctica 2. Organizando el proceso de rendering

Curso académico 2022-2023

Programación de aplicaciones gráficas

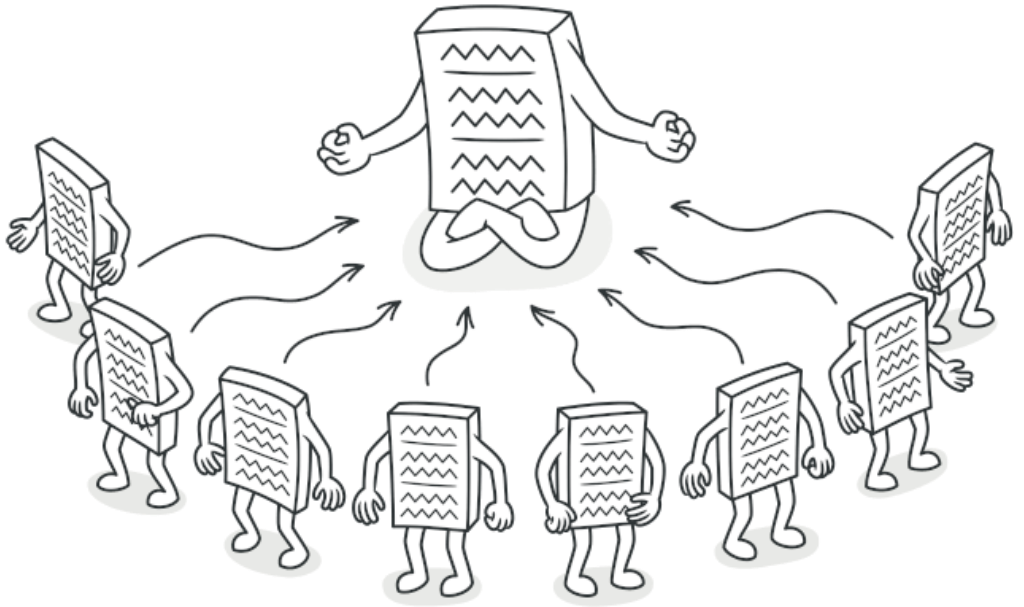
Grado en Ingeniería Informática

Objetivos de la práctica 2

- **Organización de clase `Renderer`.**
  - Encargado de organizar el dibujado de la escena.
  - Preparación de contexto de OpenGL.
  - Respuesta a eventos de usuario.
- **Definición de patrón Singleton.**
  - Acceso global a `Renderer`, representado por una instancia única.

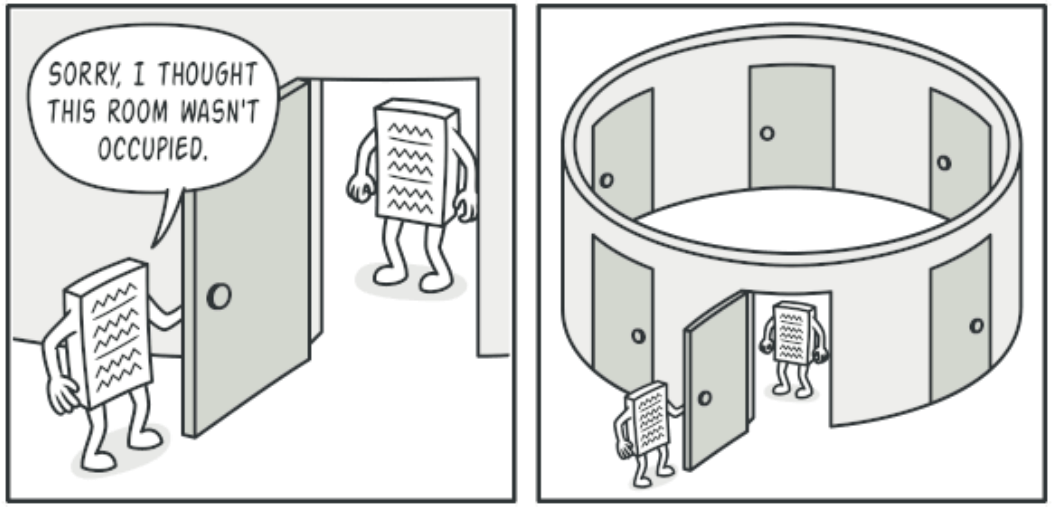
Patrón “Singleton”

- **Instancia globalmente accesible.**
  - Muy útil, pero no debemos abusar del uso de este patrón.
  - Características:
    - **Construcción dependiente de un método `getInstance`.**
      - Comprueba si ya existe una instancia creada.
      - Si no es así, se construye.
    - **Construcción y borrado no accesible de manera pública.**
      - De no ser así, no se podría garantizar la unicidad.
    - **Variable estática**: almacena la instancia única y es un puntero nulo por defecto.



Patrón “Singleton”

- **Construcción mediante `getInstance`.**
- **Todas las clases que implementen este patrón tienen una estructura en común:**
  - Variable estática.
  - Método *`getInstance`*.
- **¿Borrado?**
  - Punteros inteligentes de C++: `std::unique_ptr<>`, `std::shared_ptr<>()`.
  - Borrado automático al cerrar aplicación.



Práctica 3. Renderizando nuestro primer triángulo

Curso académico 2022-2023

Programación de aplicaciones gráficas

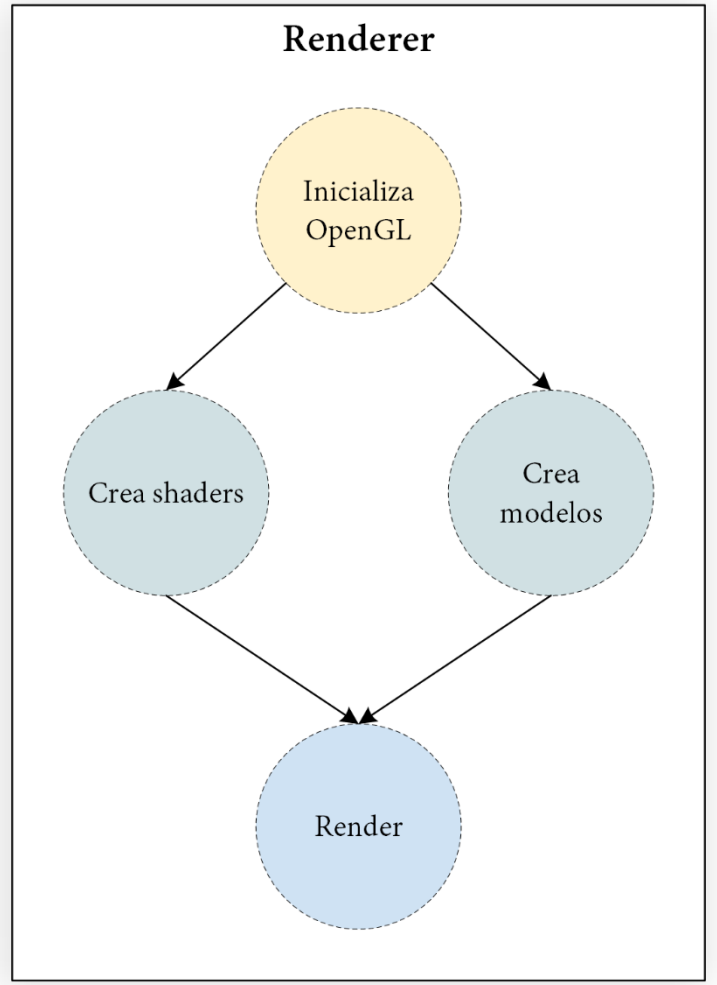
Grado en Ingeniería Informática

Objetivos de la práctica

- **Incluir nueva funcionalidad en nuestro Renderer.**
  - Envío de geometría a la GPU.
    - Triángulo.
    - Estructura enlazada y no entrelazada de VAOs.
  - Compilar y enlazar un *shader program* que indique a la GPU cómo procesar esa geometría.
    - Carga de *shaders* desde fichero.
    - Comunicación de información de *vertex shader* a *fragment shader*.
  - Configuración de parámetros globales de OpenGL.

Estructura básica

- **Inicializa OpenGL:**
  1. Activar profundidad.
  2. Activar *multisampling*.
- **Construir shaders:**
  - Desde variables string definidas dentro de nuestro Renderer.
    1. Crear *vertex shader*.
    2. Crear *fragment shader*.
    3. Unir *vertex* y *fragment shader* bajo un *shader program*.

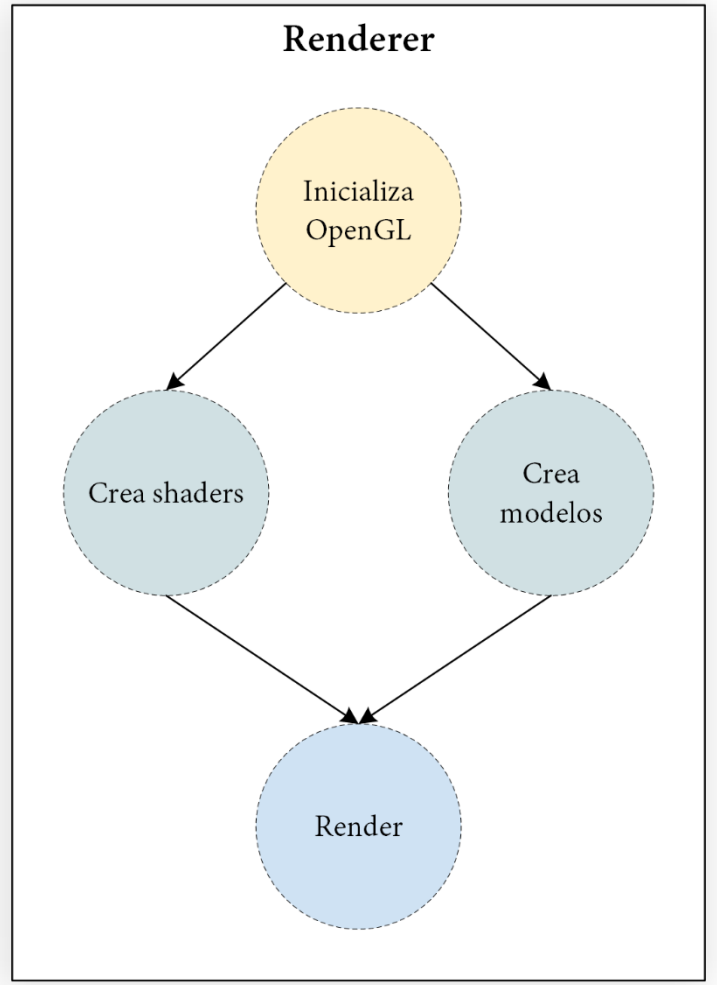


Estructura básica

- **Construir modelos:**
  - Triángulo: tres vértices y tres índices.
    - Recuerda: los índices deben definirse en ***counterclockwise order***.
  - Máquina de estados.
    1. Construye un VAO.
    2. Construye un VBO.
    3. Construye un IBO.

Ejercicio para trabajar con GLM y organizar mejor la creación de modelos:

- Sustituir *vector* de *floats* por *glm::vec3*.
- Generar VBO y VAO indicando tamaño de vector.
  - Podemos utilizar un vector de STL.



Estructura básica

- **Construir VBO:**
  - Inicio: **0**
  - Un vértice estará compuesto por **tres GL_FLOAT**, y la distancia entre el inicio de un vértice y el siguiente será **sizeof(GLfloat) * 3.**
  - Asiduidad de modificación de datos.
    - **GL_STATIC_DRAW.**
    - **GL_DYNAMIC_DRAW.**
    - **GL_STREAM_DRAW.**
    - Son solo *hints* de OpenGL (cualquier valor es válido, aunque pueda ser menos eficiente para nuestro caso de uso).

```
   GLfloat vertices[] = { -.5, -.5, 0,
                           .5, -.5, 0,
                           .0,  .5, 0 };
   GLuint indices[] = { 0, 1, 2 };

   glGenVertexArrays ( 1, &idVAO );
   glBindVertexArray ( idVAO );
   glGenBuffers ( 1, &idVBO );
   glBindBuffer ( GL_ARRAY_BUFFER, idVBO );
   glBufferData ( GL_ARRAY_BUFFER, 9*sizeof(GLfloat), vertices,
                  GL_STATIC_DRAW );
   glVertexAttribPointer ( 0, 3, GL_FLOAT, GL_FALSE, 3*sizeof(GLfloat),
                           nullptr );
   glEnableVertexAttribArray ( 0 );
   glGenBuffers ( 1, &idIBO );
   glBindBuffer ( GL_ELEMENT_ARRAY_BUFFER, idIBO );
   glBufferData ( GL_ELEMENT_ARRAY_BUFFER, 3*sizeof(GLuint), indices,
                  GL_STATIC_DRAW );
}
```

Estructura básica

- **Construir IBO:**
  - Tamaño total del buffer de índices.
  - La GPU se encargará de distinguir cada primitiva.
    - **GL_TRIANGLES, GL_POINTS, GL_LINES**, etc.
    - Podemos distinguir primitivas con un índice nulo.
      - GL_RESTART_PRIMITIVE_INDEX.

```
   GLfloat vertices[] = { -.5, -.5, 0,
                           .5, -.5, 0,
                           .0,  .5, 0 };
   GLuint indices[] = { 0, 1, 2 };

   glGenVertexArrays ( 1, &idVAO );
   glBindVertexArray ( idVAO );
   glGenBuffers ( 1, &idVBO );
   glBindBuffer ( GL_ARRAY_BUFFER, idVBO );
   glBufferData ( GL_ARRAY_BUFFER, 9*sizeof(GLfloat), vertices,
                  GL_STATIC_DRAW );
   glVertexAttribPointer ( 0, 3, GL_FLOAT, GL_FALSE, 3*sizeof(GLfloat),
                           nullptr );
   glEnableVertexAttribArray ( 0 );
   glGenBuffers ( 1, &idIBO );
   glBindBuffer ( GL_ELEMENT_ARRAY_BUFFER, idIBO );
   glBufferData ( GL_ELEMENT_ARRAY_BUFFER, 3*sizeof(GLuint), indices,
                  GL_STATIC_DRAW );
}
```

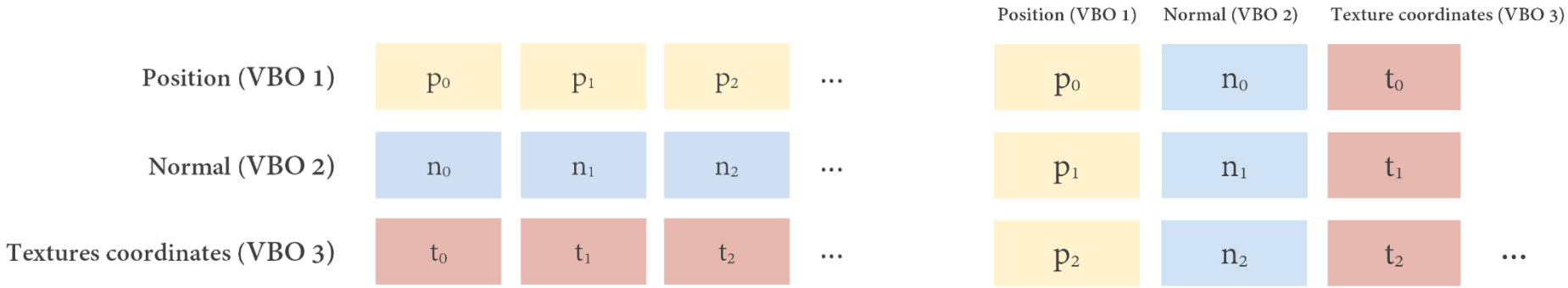
Ejercicios

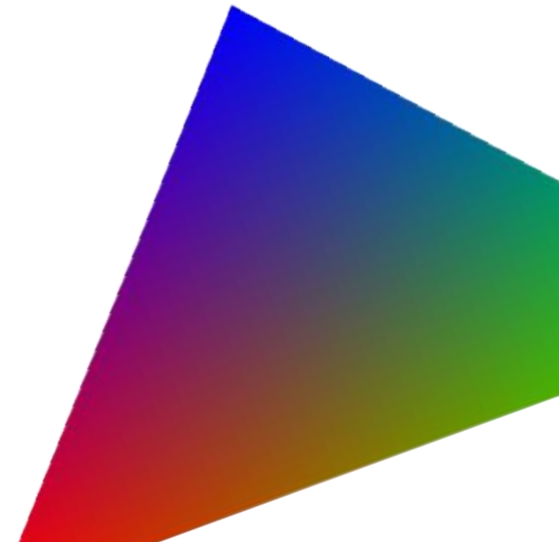
1. Organiza tu código tal y como se muestra en la práctica.

- Comprueba que eres capaz de dibujar un triángulo sin color.

2. A partir de aquí, guarda una copia de seguridad y comienza a modificar tus estructuras en GPU.

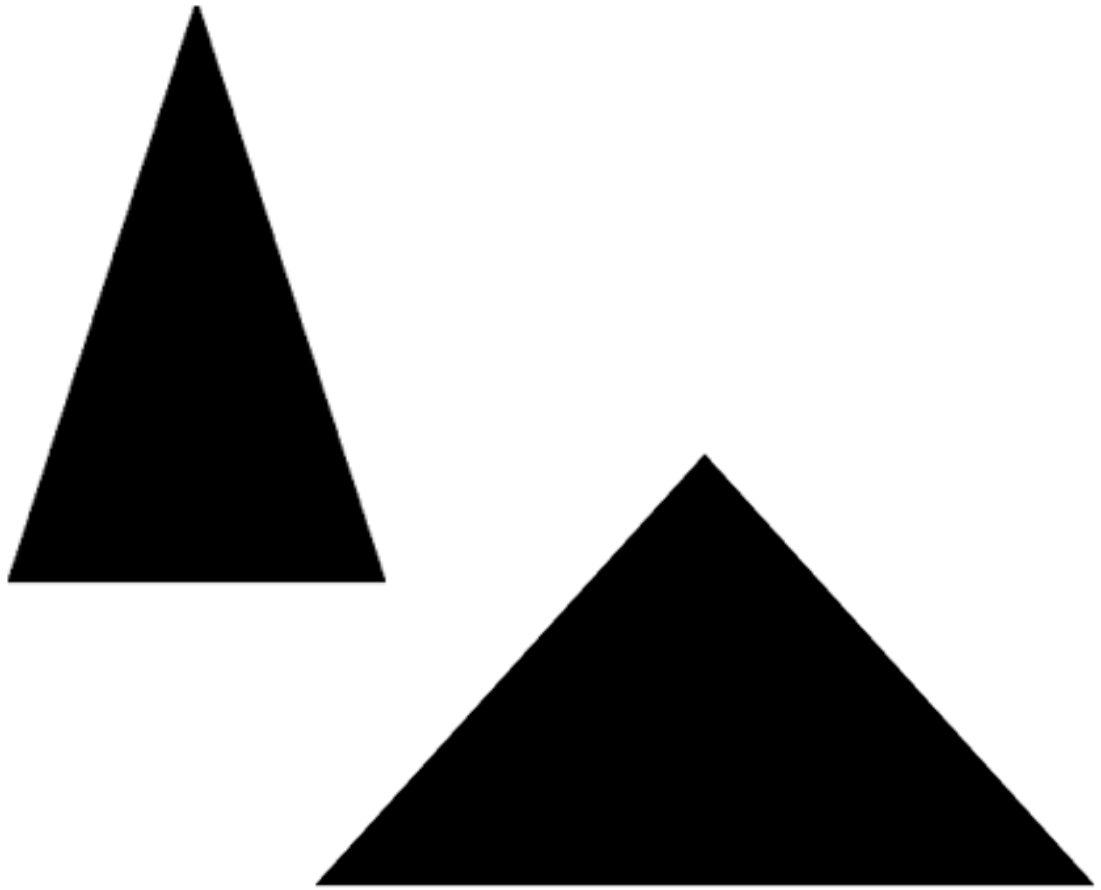
1. Añade un atributo color a cada vértice. Haz los cambios correspondientes en *shader* y comprueba que eres capaz de obtener un gradiente.
2. Modifica tu VAO. Prueba a especificar una estructura entrelazada.
    - Guarda tu código anterior. Debes subir ambas versiones.

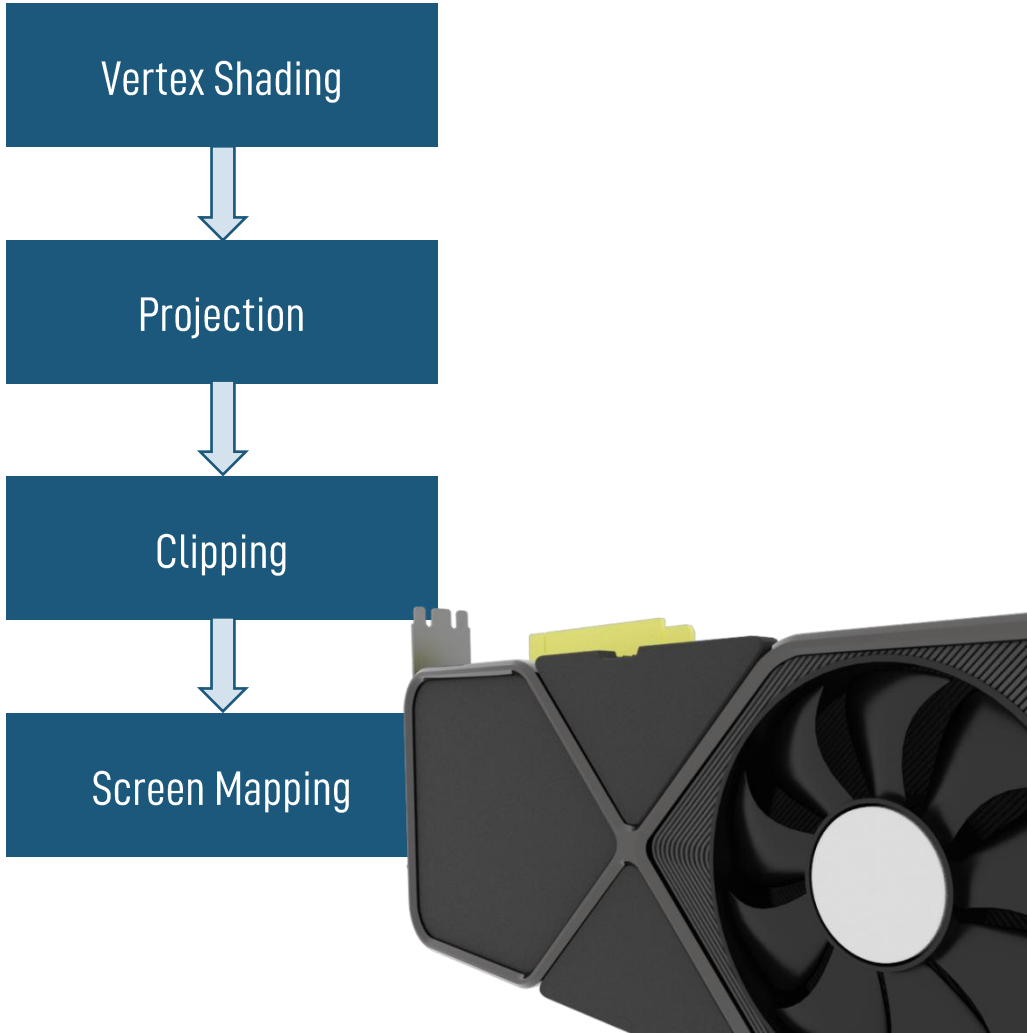




Ejercicios

¿Por qué escala el triángulo con la ventana?





Práctica 4. Gestionando shader programs y modelos

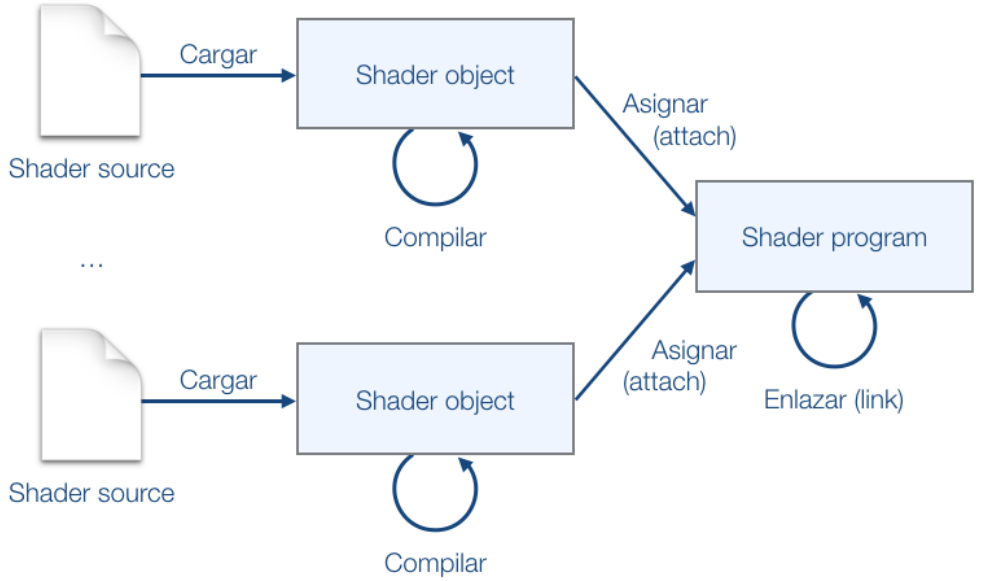
Curso académico 2022-2023

Programación de aplicaciones gráficas

Grado en Ingeniería Informática

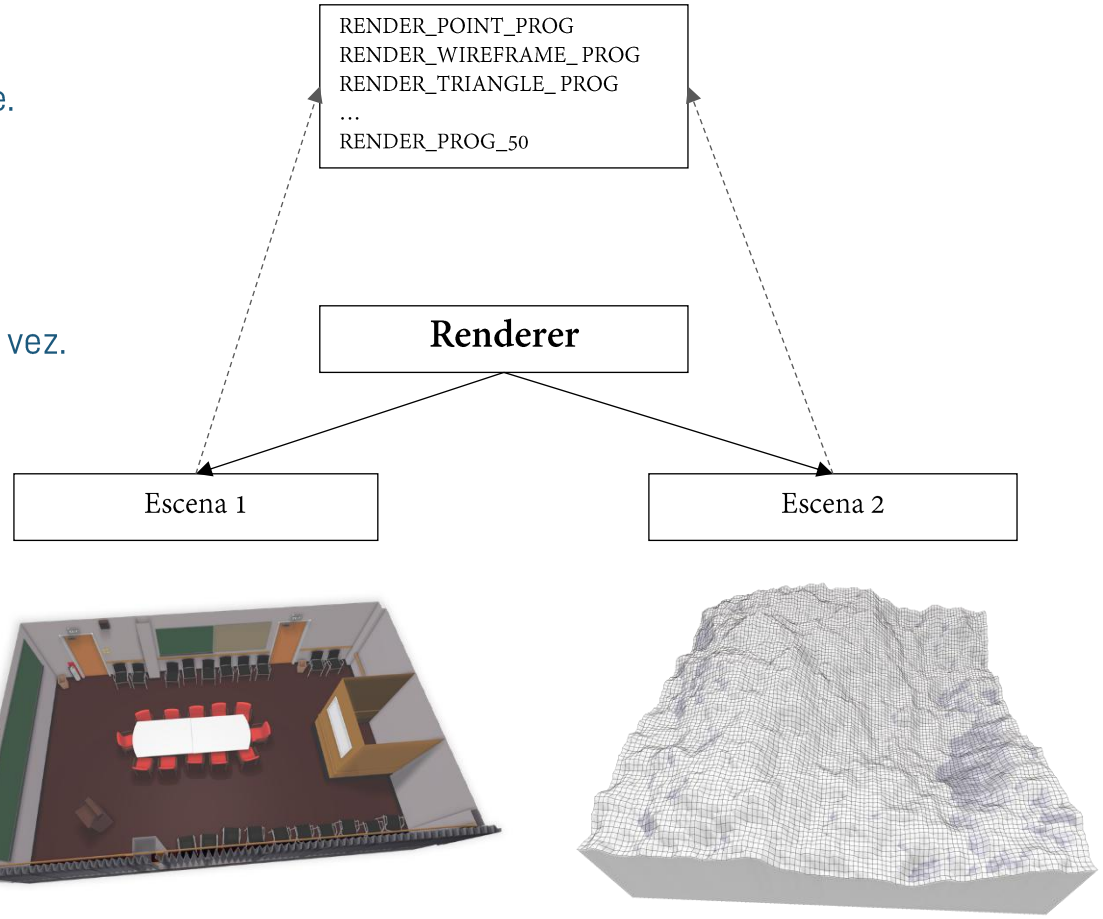
Objetivos de la práctica

- **Carga, compilación y enlazado de SHADER OBJECTS.**
  - *Vertex*
  - *Fragment*
  - *Geometry*
  - *Tessellation*
  - *Compute*
- **Creación de SHADER PROGRAMS.**
  - Similar a la Práctica 3.
  - Encapsular funciones de OpenGL en un objeto es mucho más intuitivo que repetir bloques de código.



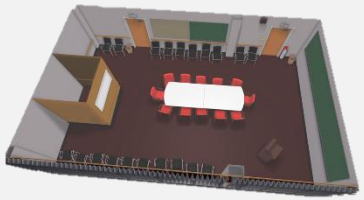
Trabajo autónomo

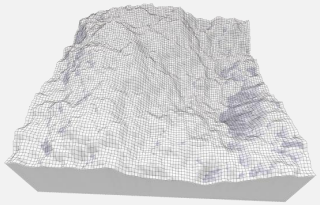
- **Desacoplar RENDERER y SHADER PROGRAMS.**
  - Utiliza una clase para encapsular los SHADER PROGRAMS.
  - Documenta cómo lo has hecho y qué métodos tiene la nueva clase.
- **Podemos ir un poco más allá…**
  - Podríamos pedir shaders desde múltiples puntos de la aplicación.
  - Inicialización perezosa: solo se crean cuando se piden por primera vez.
  - **¿Cómo gestionamos esta interacción?**



Trabajo opcional

- **Puede existir comportamiento común entre varios shaders.**
  - **Un único shader con subrutinas para elegir comportamiento.**
    - Se complica la legibilidad del shader a medida que introducimos variaciones.
  - **Varios shaders más legibles.**
    - **Problema**: código duplicado.
    - Podemos introducir librerías personalizadas.
      - Preprocesamiento personalizado de shaders.





RENDER_TRIANGLE_01

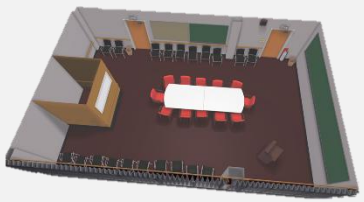
RENDER_TRIANGLE_02

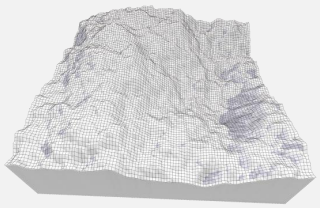
Leer propiedades de material
Cálculo de color en función de luz y material
Comportamiento personalizado

#include <module/triangRendering.glsl>
Comportamiento personalizado

Trabajo opcional

- **Puede existir comportamiento en común entre varios shaders.**
  - **Detección de sintaxis específica para nuestros módulos.**
    - Podemos emplear una sintaxis similar a la de C++:
      - #include "Path de Módulo" o #include <Path de Módulo>.
  - **¡Ojo! Un módulo se plantea como tal porque se empleará en varios SHADER PROGRAMS.**
    - No hace falta leerlo desde fichero cada vez que se pida.
    - Podemos almacenarlos una vez se hayan cargado.





RENDER_TRIANGLE_01

RENDER_TRIANGLE_02

Leer propiedades de material
Cálculo de color en función de luz y material
Comportamiento personalizado

#include <module/triangRendering.glsl>
Comportamiento personalizado

Práctica 5. Gestionando modelos 3D

Curso académico 2022-2023

Programación de aplicaciones gráficas

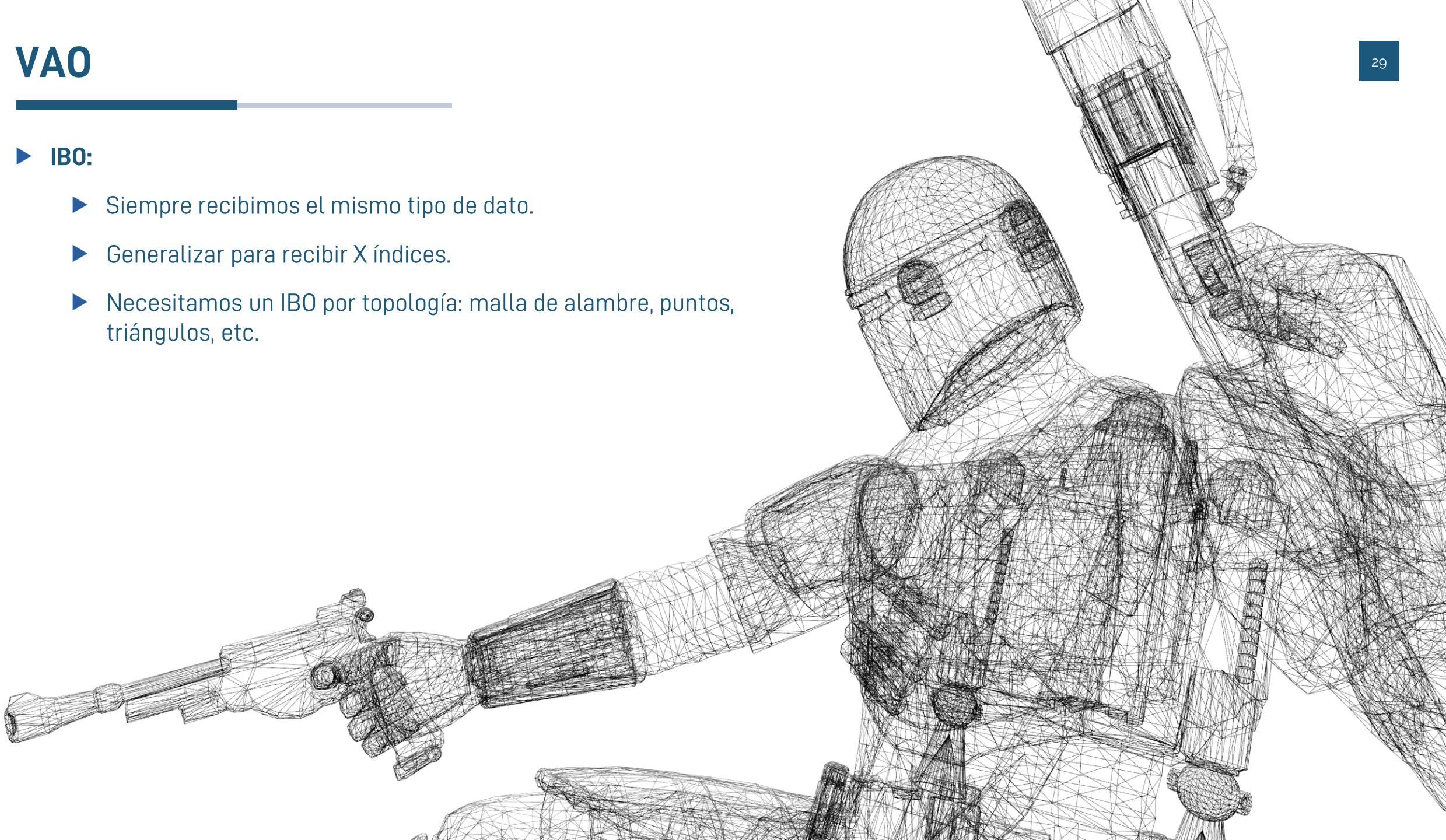
Grado en Ingeniería Informática

Objetivos de la práctica

- **Encapsular el concepto de VAO para generalizarlo a un modelo cualquiera.**
  - Una nueva clase VAO.
    - Definición de estructura.
    - Transferencia de datos hacia GPU.
      - VBO(s), IBO(s).
- **Carga de modelos.**
  - Encapsulamiento de modelos en una nueva clase.
  - Por defecto, ocupará una posición en el mundo.
  - Cada modelo podrá sufrir una transformación diferente, definida mediante M_i.

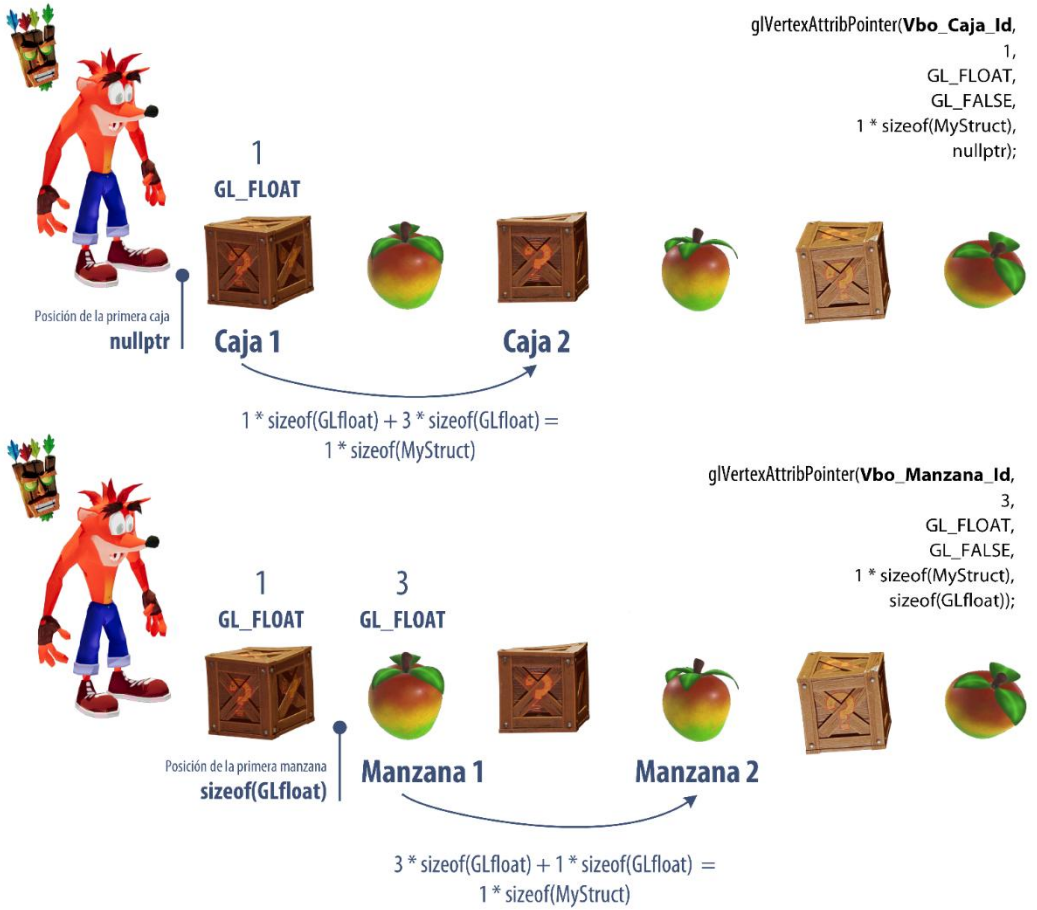
VAO

- **IBO:**
  - Siempre recibimos el mismo tipo de dato.
  - Generalizar para recibir X índices.
  - Necesitamos un IBO por topología: malla de alambre, puntos, triángulos, etc.



VAO

- **Un poco más difícil:**
  - **Constructor de un VAO:**
    - Selección de *layout* (entrelazado, no entrelazado).
    - Por lo general, disponemos de un conjunto de atributos estáticos: **Posición, Normal, Coordenadas de textura**, etc.
      - Ídem para entrelazado, la estructura de un objeto es conocida de antemano y es fija.
  - Transferencia de datos de VBO.
    - Puede ser un vector de posiciones, normales, un objeto personalizado…
      - **¿Cómo lo haríais?**

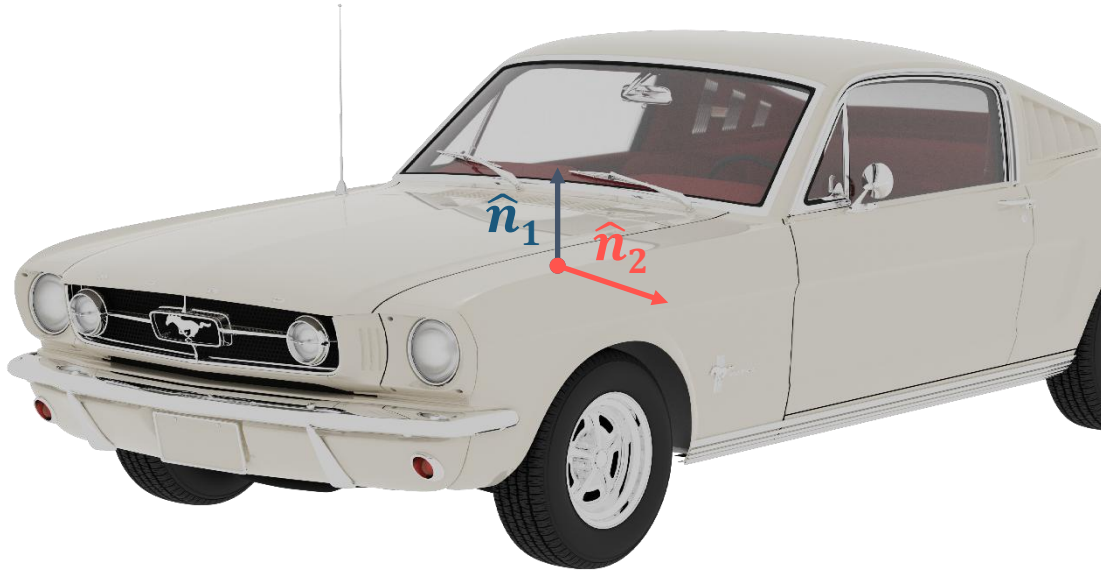


Carga de modelos

- **Utilización de modelos de una fuente externa.**
  - **Formatos: OBJ, GLTF** (GL Transmission Format)**, COLLADA** (.DAE, Digital Asset Exchange), **FBX** (Autodesk), **PLY** (Polygon File Format), etc.
  - Proyectos para cargar modelos OBJ: OBJ-Loader, tinyobjloader, obj-loader, etc.
  - **ASSIMP**: *Open Asset Import Library*.
    - Librería para cargar múltiples formatos, incluyendo todos los anteriores.
    - Modelos con animaciones.
    - Podemos instalarlo con **vcpkg**, como hicimos en la práctica 1 con el resto de librerías.

Carga de modelos

- **ASSIMP**
  - **Utilizad como referencia la guía de LearnOpenGL.**
  - Para abrir un modelo se deben especificar algunos flags:
    - *Join Identical Vertices*: menos espacio en memoria.
    - Triangulación: algunos modelos contienen otros tipos de polígonos.
      - *Quads*.
      - Trabajamos con GL_TRIANGLES.
    - Tipos de normales: smooth o duras (*GenSmoothNormals* vs *GenNormals*).

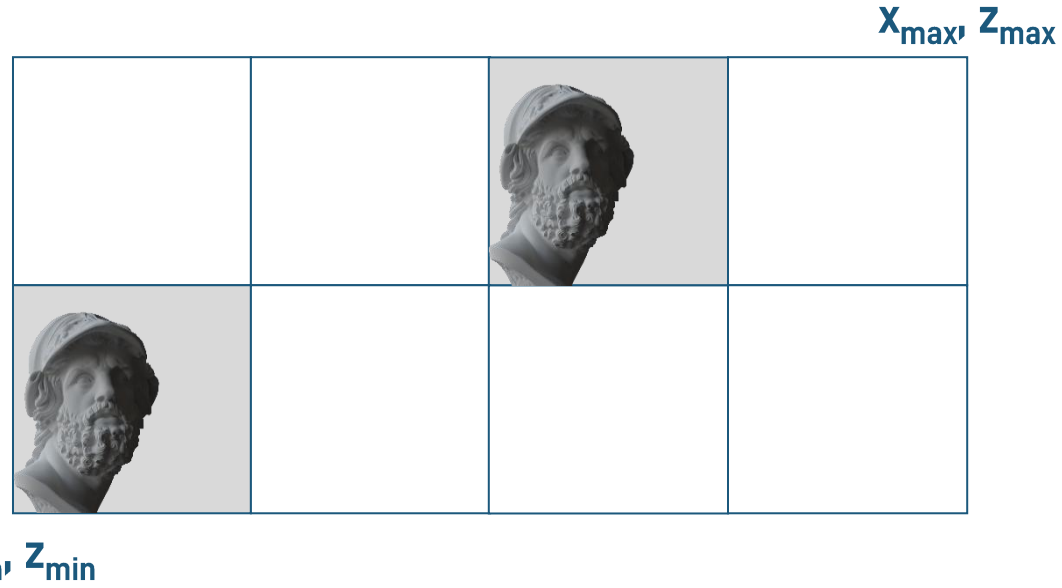


Flujo de la aplicación

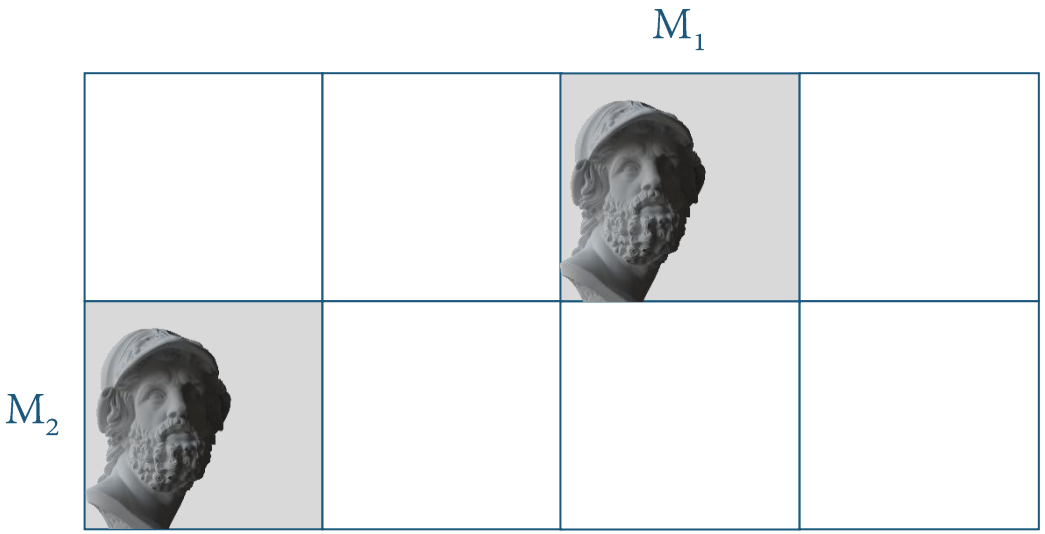
1. **Genera el contenido de la escena.**
   - Crea modelo(s).
   - Crea shader(s).
2. **Renderizado.**
   - Activa shader.
   - Itera por los objetos de la escena.
     - Itera por componentes de un objeto.
       - Dibuja utilizando el VAO.
3. **Comprobar callbacks.**
   1. Añadir modelo si el usuario pulsa la tecla ‘a’.
   2. Eliminar modelo si el usuario pulsa la tecla ‘d’.

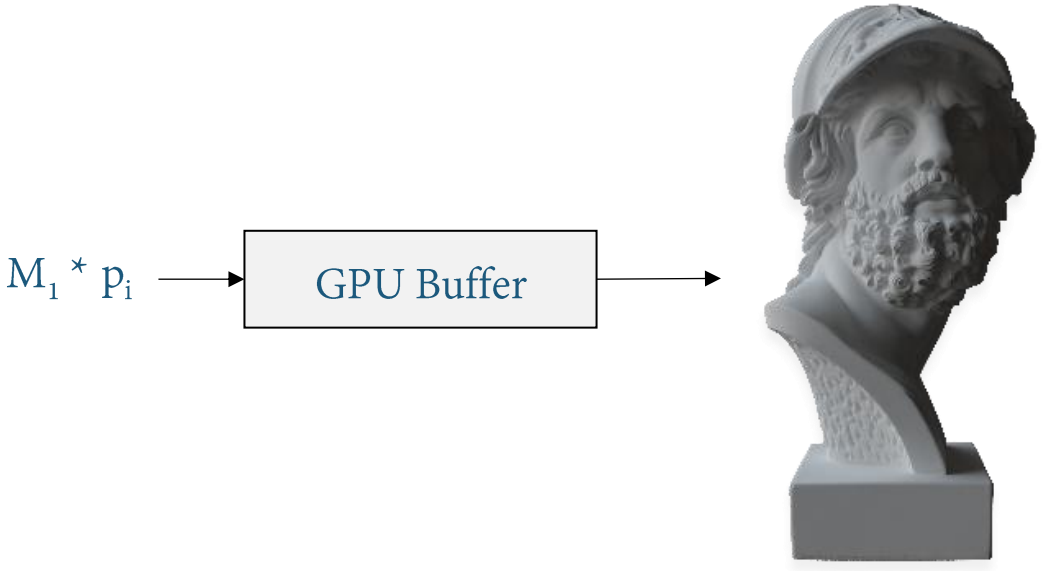
Flujo de la aplicación

1. **Genera el contenido de la escena.**
   - Crea modelo(s).
   - Crea shader(s).
2. **Renderizado.**
   - Activa shader.
   - Itera por los objetos de la escena.
     - Itera por componentes de un objeto.
       - Dibuja utilizando el VAO.
3. **Comprobar callbacks.**
   1. Añadir modelo si el usuario pulsa la tecla ‘a’.
   2. Eliminar modelo si el usuario pulsa la tecla ‘d’.

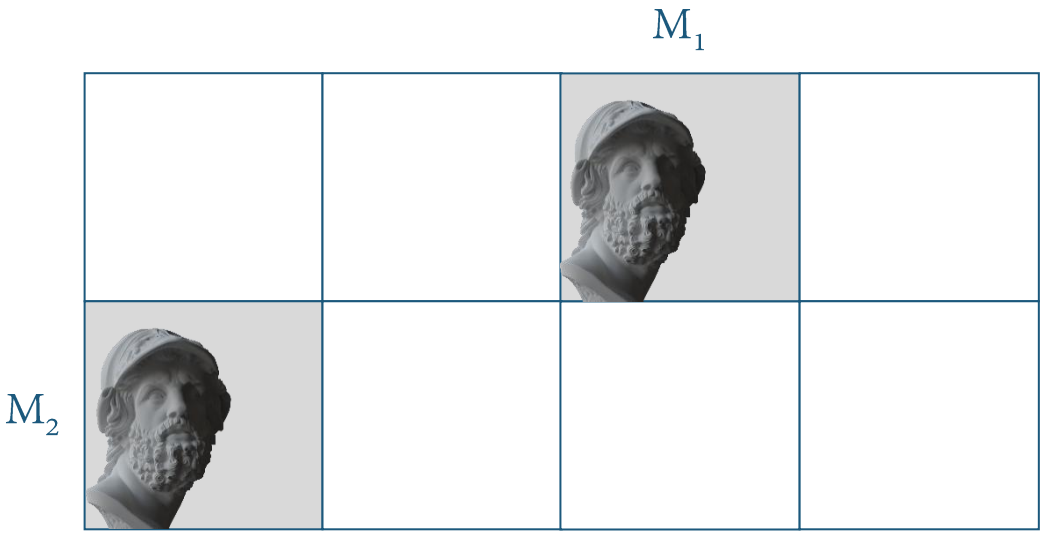


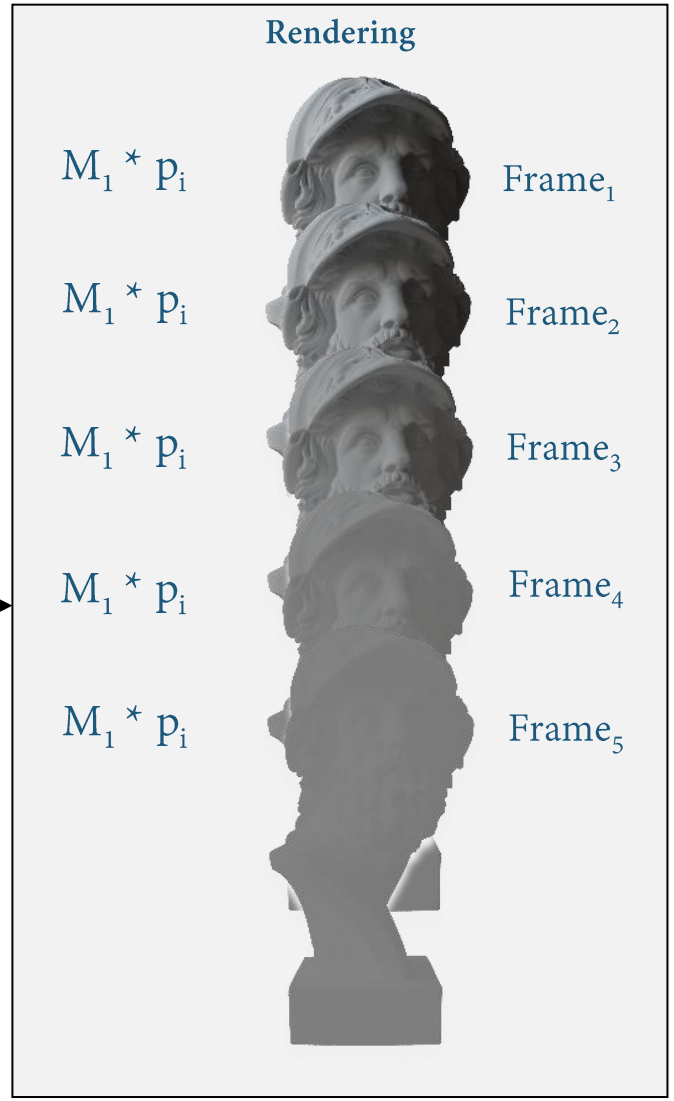
Matrices de transformación





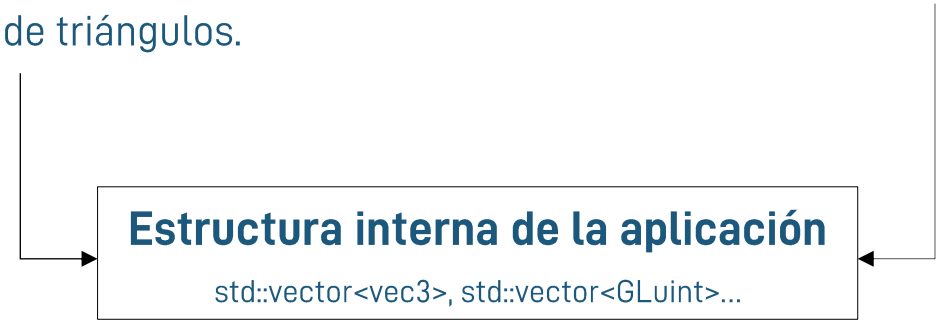
Matrices de transformación





Ejercicio opcional

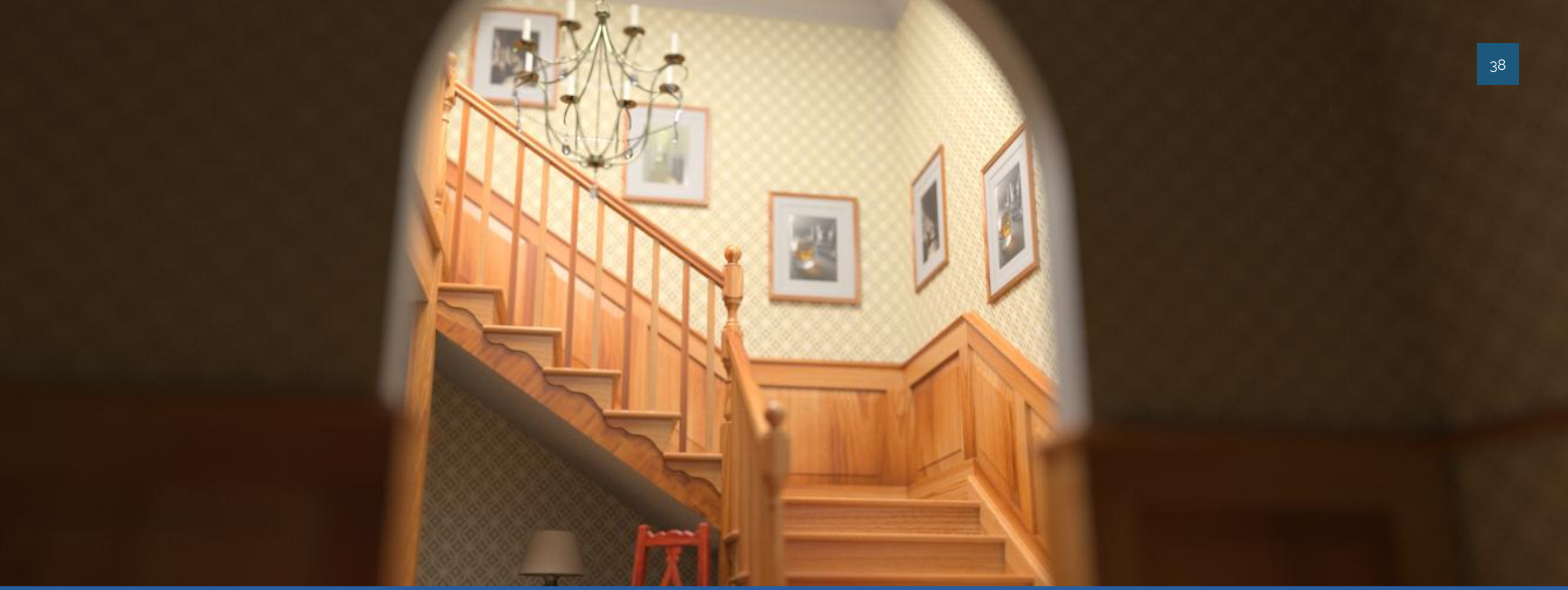
- **Modelos OBJ** (.obj)
  - Vértices.
  - Normales.
  - Coordenadas de textura.
  - Índices de triángulos.
- **Ficheros binarios** (.bin, aunque podría ser cualquier otra extensión)
  - std::vector<vec3>
  - std::vector<vec2>
  - std::vector<GLuint>



Si guardamos los datos de un vector, desconocemos el tamaño cuando volvamos a leer (fix: guarda tamaño y a continuación, los datos).

En formato binario, no ASCII, y por tanto, no legible.

Acoplado a una estructura de información (si hacemos cambios, habrá que borrar el binario o hacer un intérprete que transfiera los datos a la nueva estructura).



Práctica 6. Cámara virtual

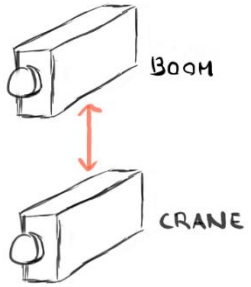
Curso académico 2022-2023

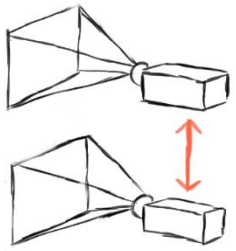
Programación de aplicaciones gráficas

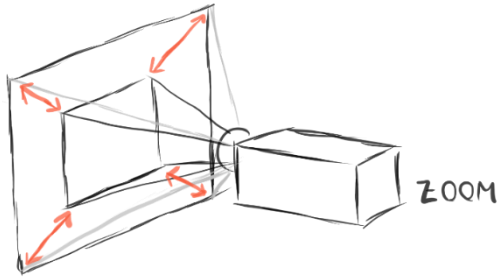
Grado en Ingeniería Informática

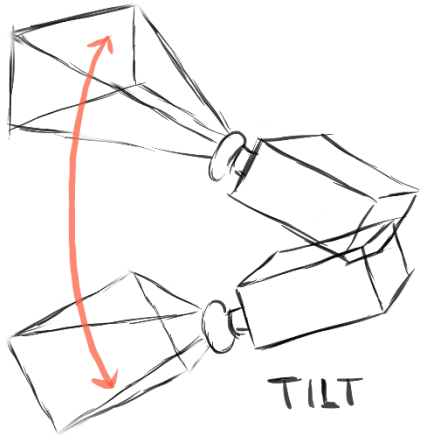
Objetivos de la práctica

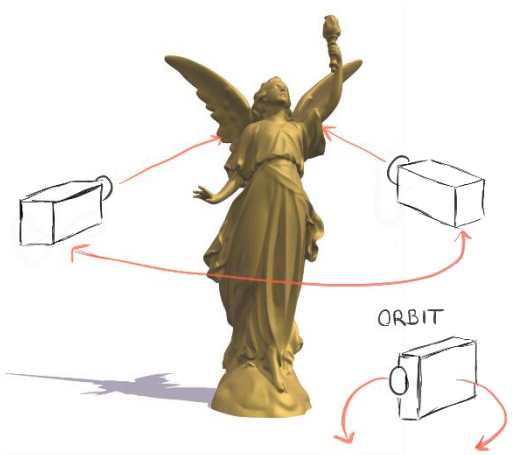
- **Implementación de una cámara con los siguientes movimientos:**
  - *Pan.*
  - *Tilt.*
  - *Dolly.*
  - *Boom/Crane.*
  - *Orbit.*
  - *Zoom.*

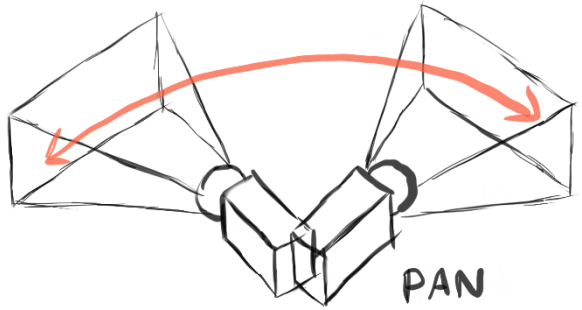


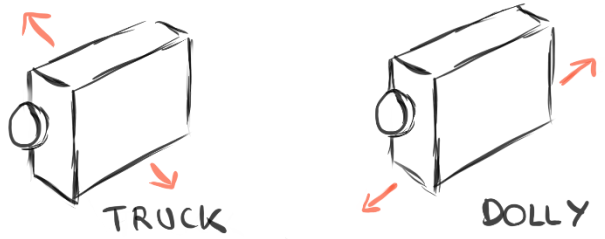


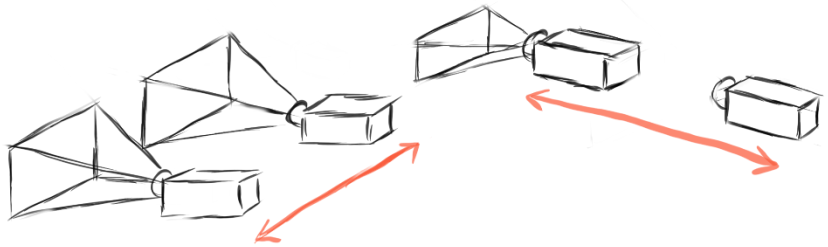












Objeto cámara

- **Sistema de coordenadas:**
  - *U, V, N*
- **View matrix:**
  - *Eye, look at, up.*
  - *View matrix.*
- **Projection matrix:**
  - *Field of view (Y), Z near, Z far, aspect ratio.*
  - *Projection matrix.*

Cámara

- **Facilitar la especificación de la cámara para un usuario.**
  - **Para el usuario es mucho más fácil...**
    - Definir relación de aspecto mediante anchura y altura.
    - Ángulos en grados.
      - `glm::radians`.
    - Ángulos horizontales en lugar de verticales (*field of view*).
      - Trigonometría conociendo tamaño de ventana.

Cámara

Comparaciones entre vectores y valores reales.

PAG 2022/2023

```
[1]       v := (0, 1, 0)
[2]       if v == (0, 1, 0)
[3]          do things
[4]       print(v)
[Out]     [.00000000001, 1.000000001, .0]
```

```
#include <glm/glm.hpp>
#include <glm/ext/vector_relational.hpp>   // Para equal
#include "glm/gtc/constants.hpp"           // Constantes, como epsilon
#include <glm/gtc/epsilon.hpp>             // Para epsilonEqual
...
glm::bvec3 v3 = epsilonEqual(v1, v2, glm::epsilon<float>());
glm::bvec3 v3 = equal(v1, v2, glm::epsilon<float>()));
// En ambos casos, devuelve un vector con tres valores booleanos
```

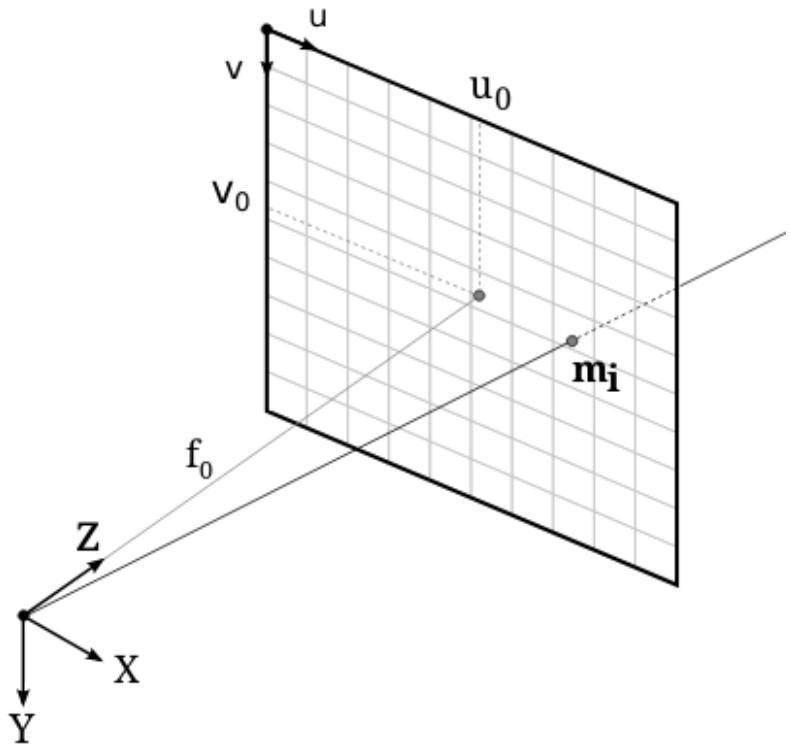
Cámara

Gestión de un número excesivo de eventos

```
void miFuncion ()
{
   static clock_t ultimaEjecucion = clock(); // Solo se ejecuta la primera vez
   if ((clock() - ultimaEjecucion) > 100)
   {
      ultimaEjecucion = clock ();
   }
}
```

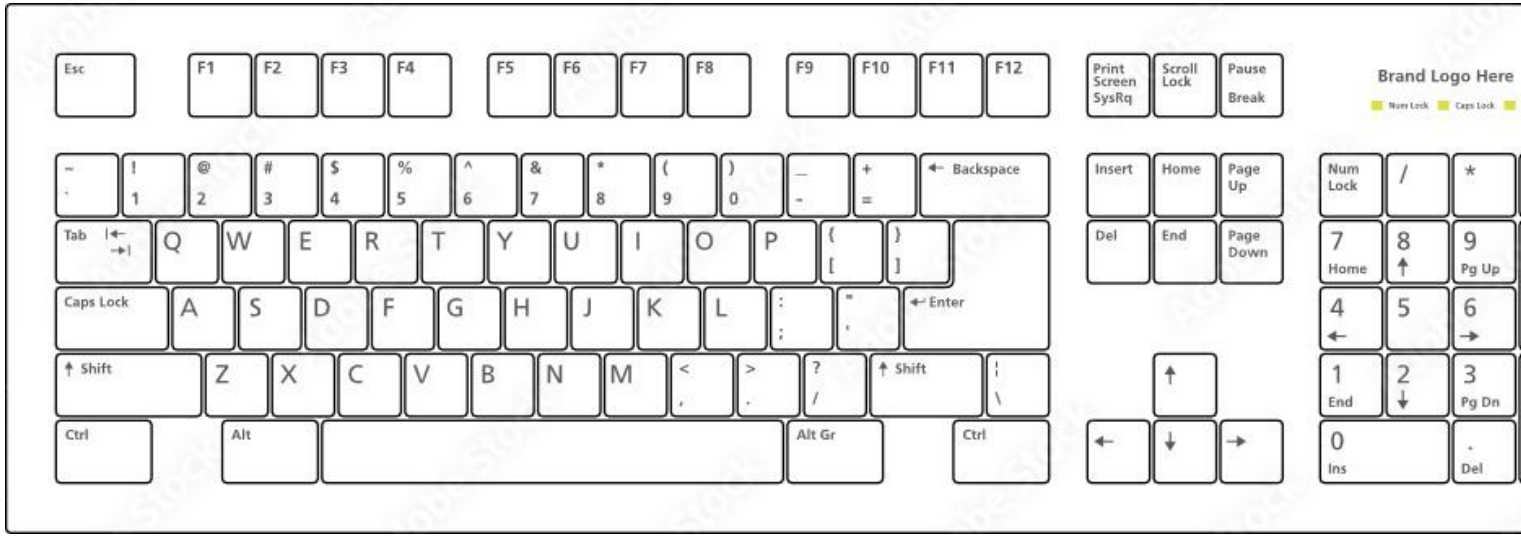
Cámara

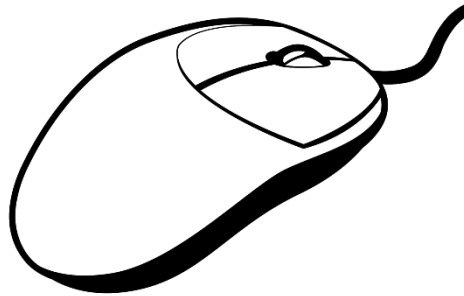
- **Recálculo eficiente de parámetros:**
  - Pensad qué ejes se modifican.
  - Pensad qué atributos se modifican.
  - Recalculad matrices en función de esto.



Cámara

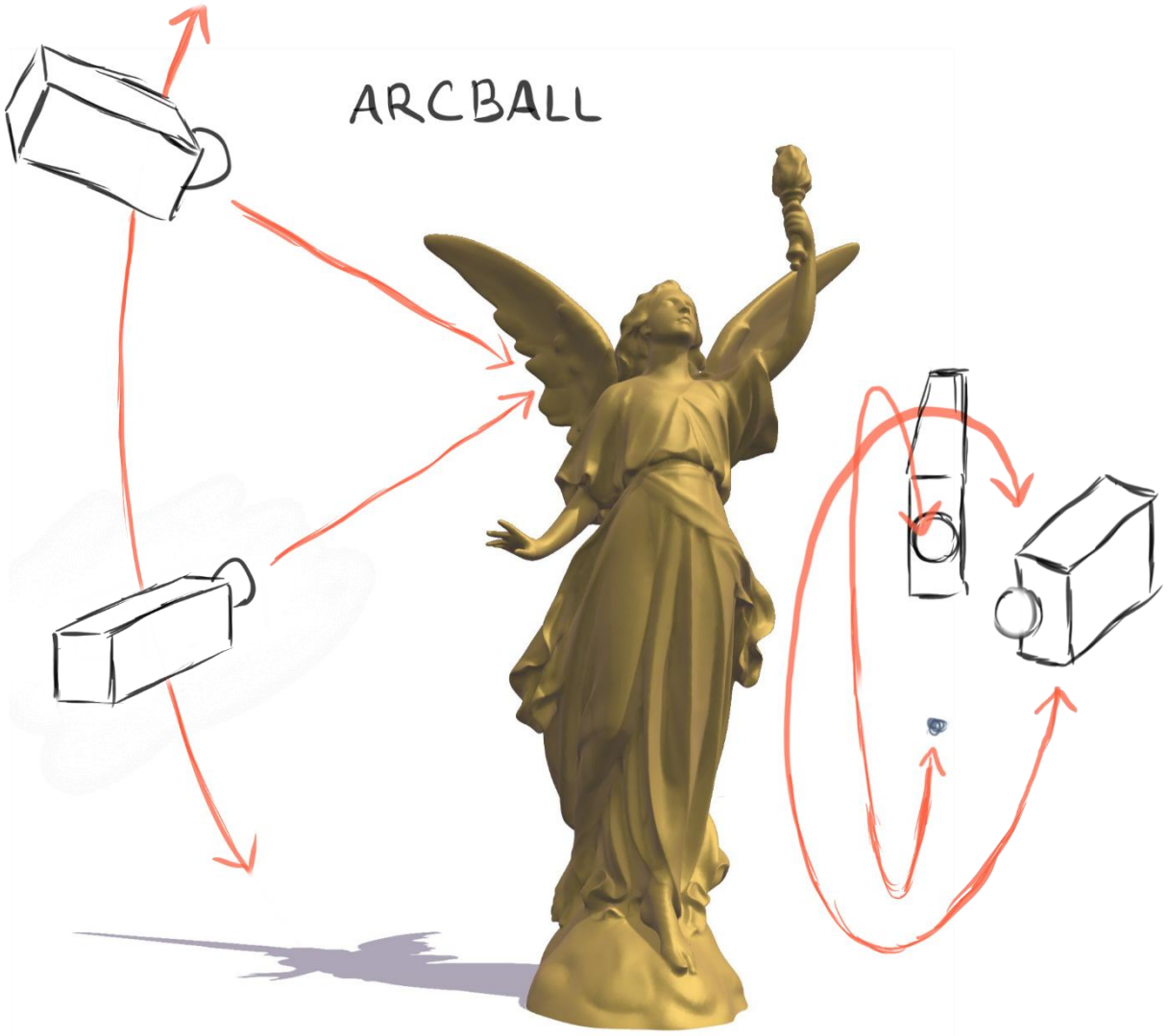
- **Asignación de teclas para movimientos:**
  - Libre elección.
    - Documentación.
  - Un ejemplo:
    - W, S: *Dolly* positivo y negativo.
    - A, D: *Truck* negativo y positivo.
    - *Scroll*: *zoom*.
    - X: *orbit*.
    - Y: *arcball*.
    - Ratón (manteniendo pulsado): *dolly*, *tilt*.





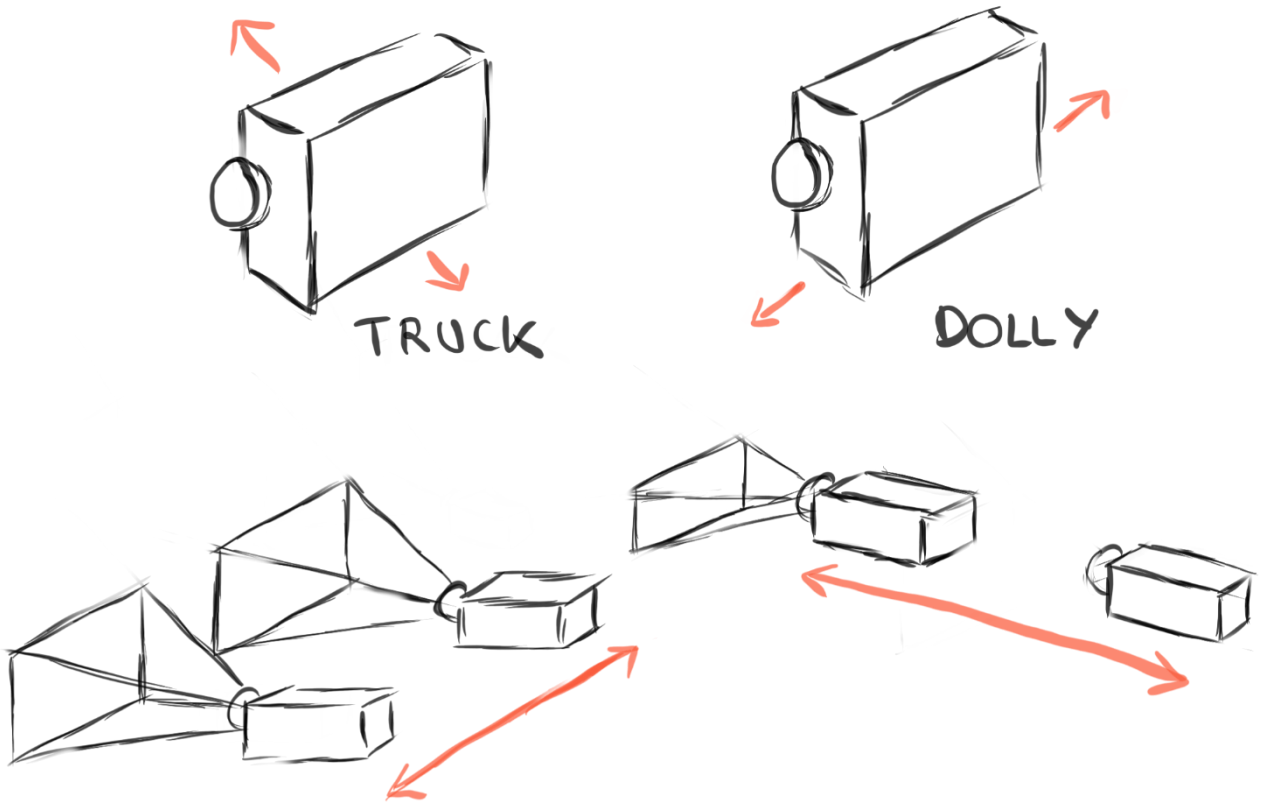
Ejercicios opcionales

- **Movimiento arcball.**



Ejercicios opcionales

- **Aceleración de movimientos.**
  - Movimientos en ráfaga (seguidos): contador.
    - Velocidad de movimiento en función de este valor.



Práctica 7. Subrutinas GLSL y materiales

Curso académico 2022-2023

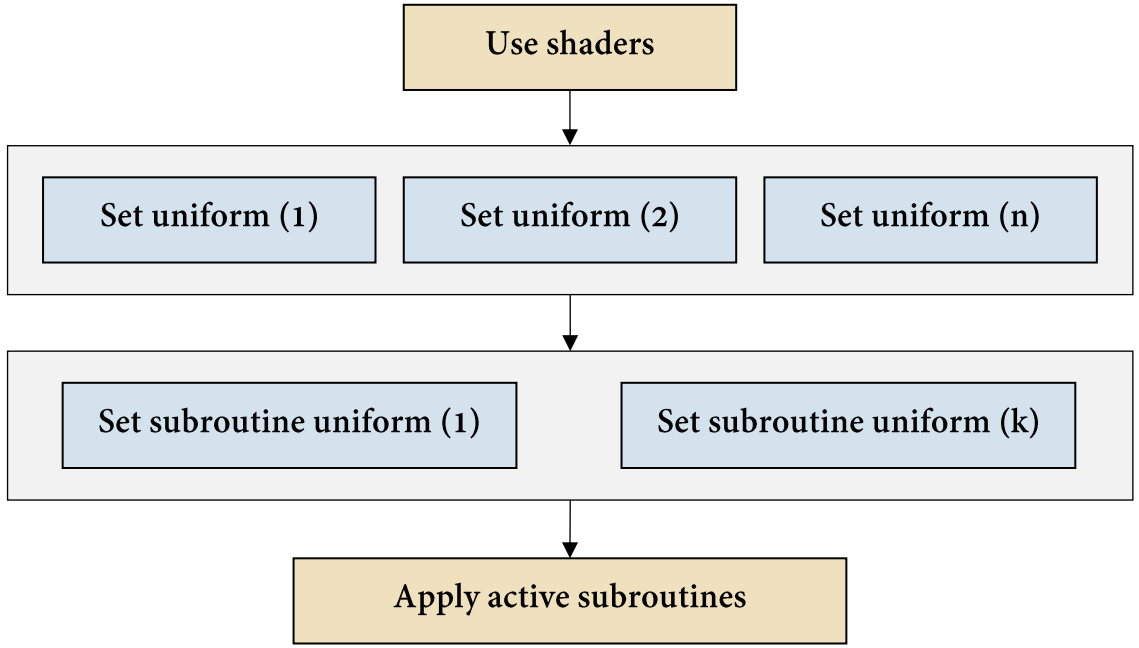
Programación de aplicaciones gráficas

Grado en Ingeniería Informática

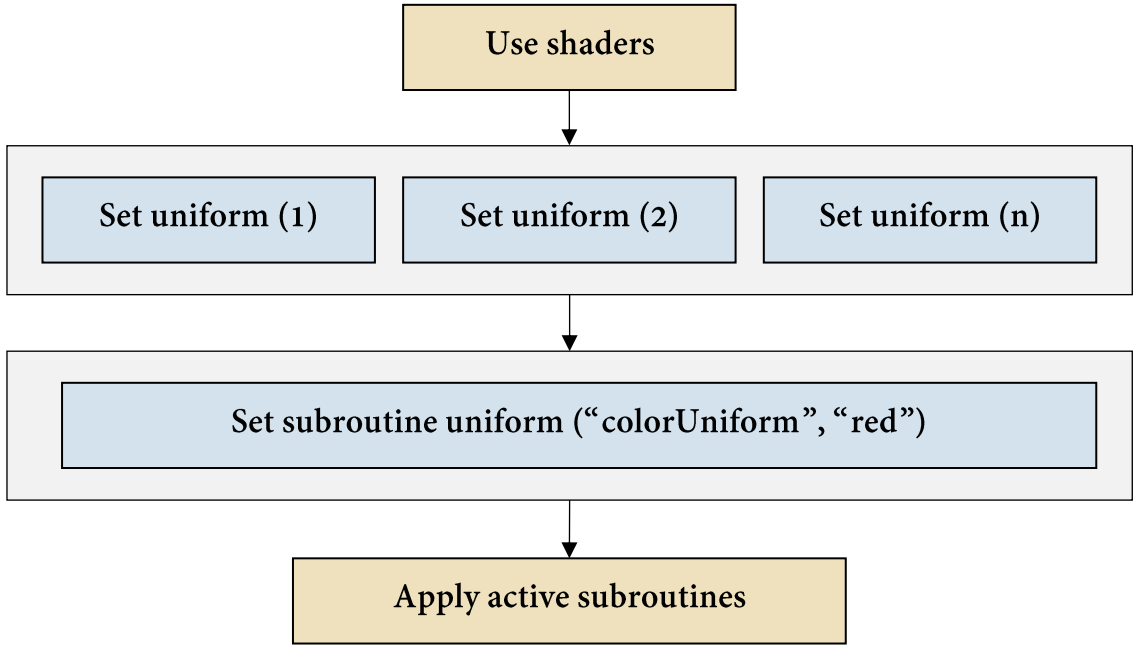
Objetivos de la práctica

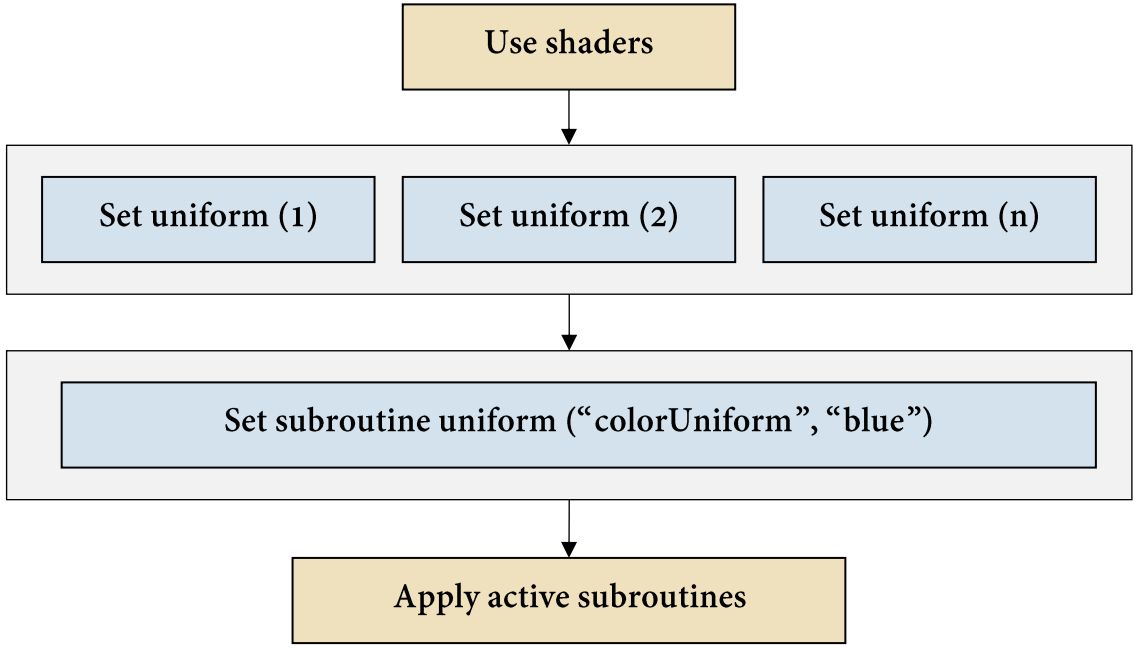
- **Incluir subrutinas en nuestros shaders.**
  - Permite elegir diferentes rutas de ejecución.
  - Evita duplicar código; en su lugar, se implementan varias posibilidades.
  - Cambios de comportamiento de dibujado en tiempo de ejecución.

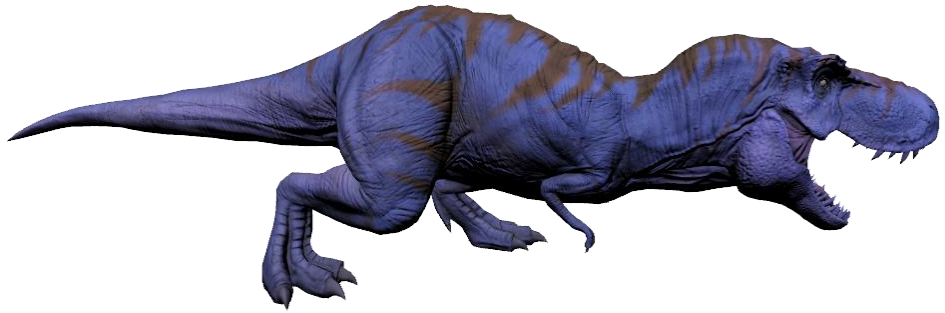
-1 si no consigue establecer la subrutina



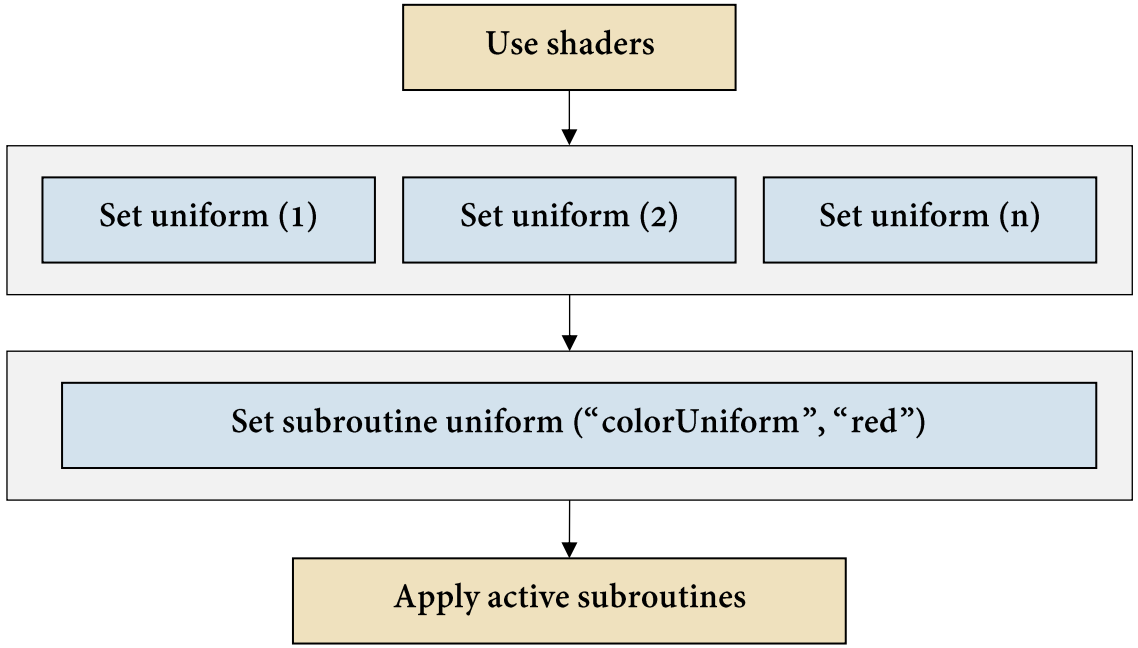
Subrutinas GLSL

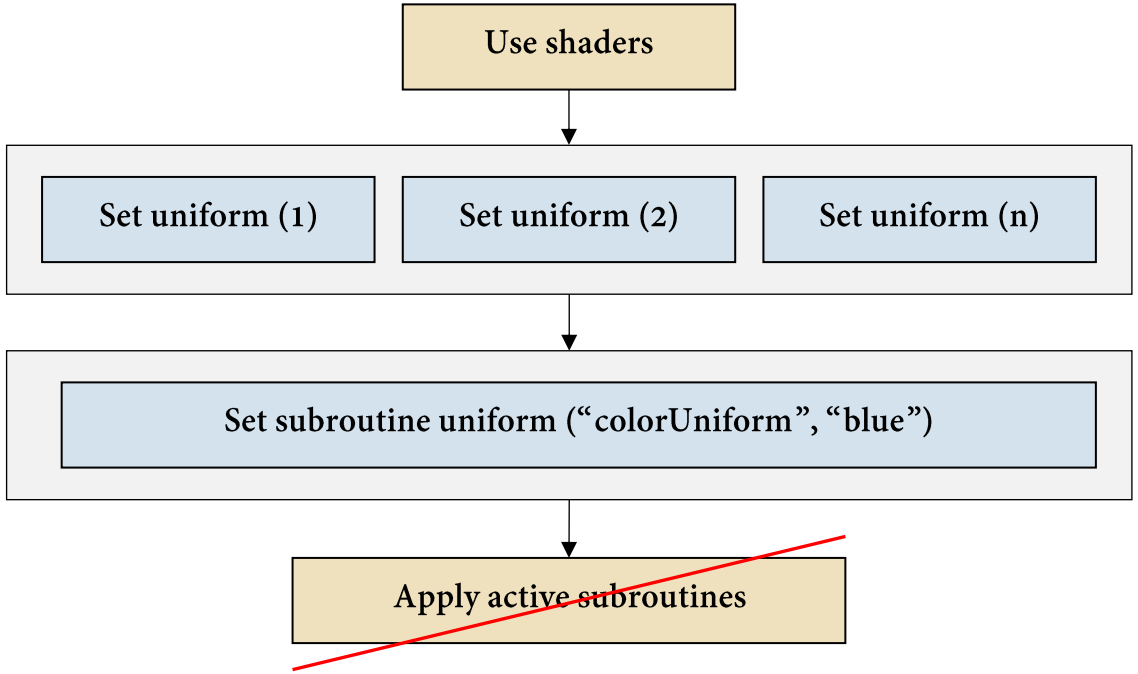




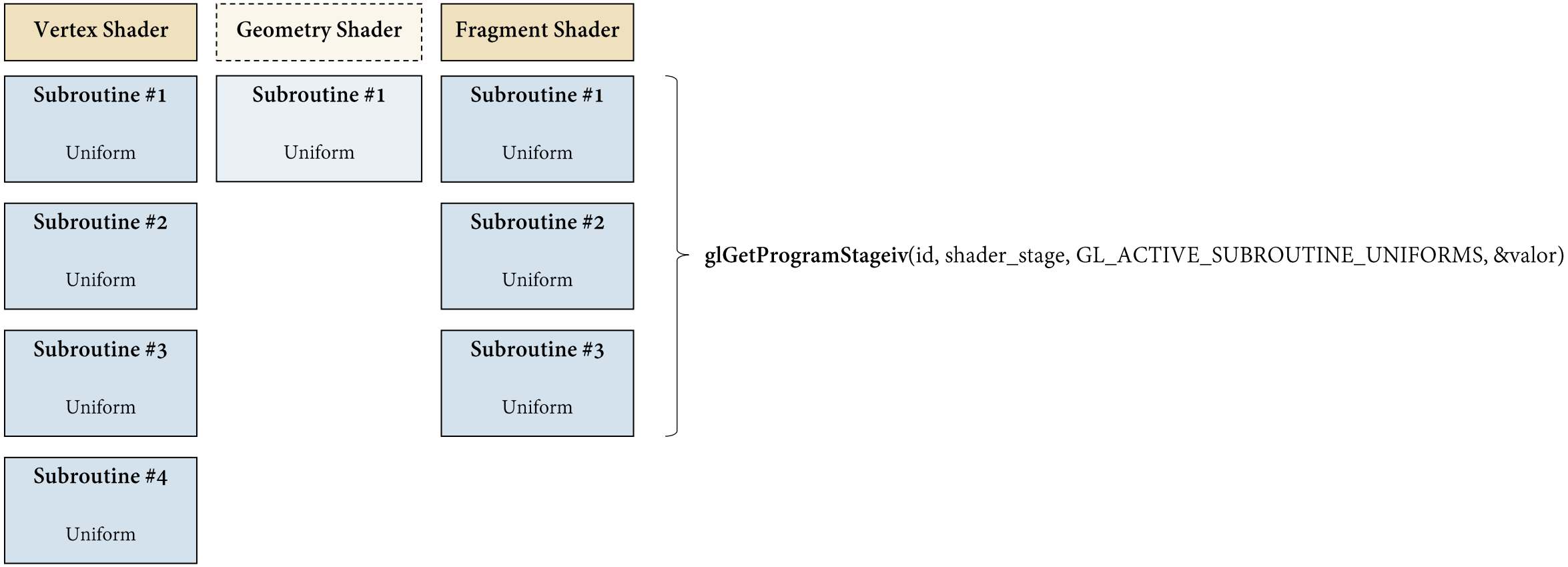


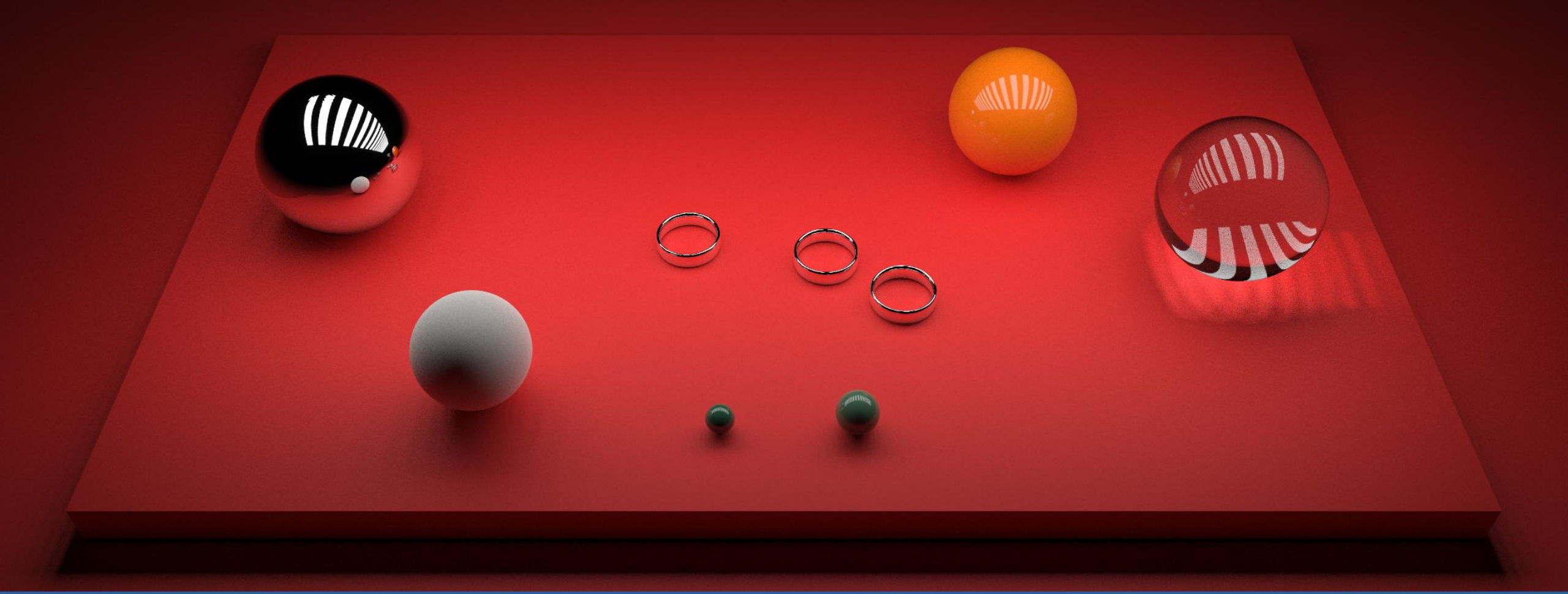
Subrutinas GLSL





Subrutinas GLSL





Práctica 8. Iluminación y sombreado

Curso académico 2022-2023

Programación de aplicaciones gráficas

Grado en Ingeniería Informática

Luces

- **Ambient Light**

Luces

Ambient Light

Luces

Point Light

Luces

Directional Light

Luces

Spot Light

Luces

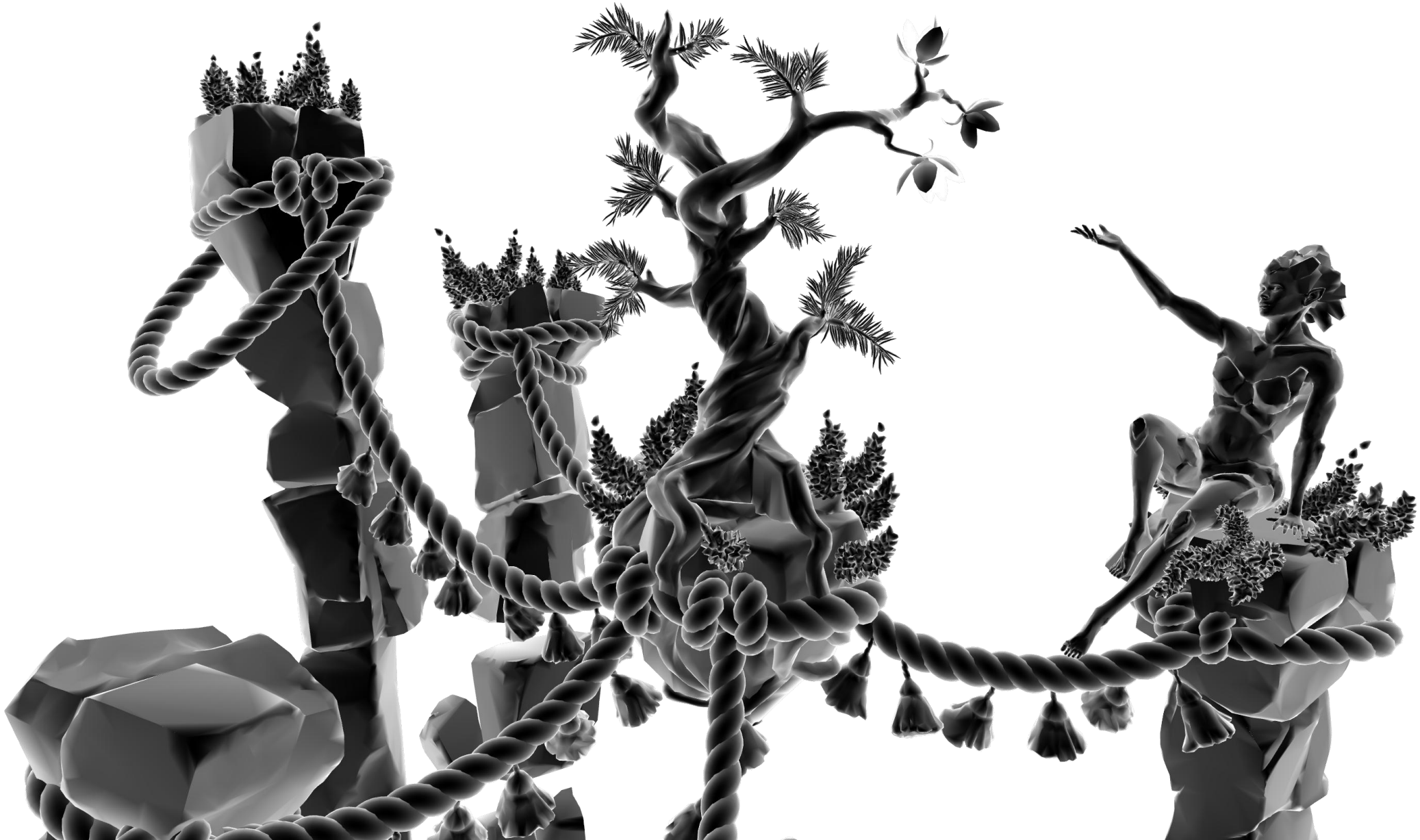
Spot Light

Luces

Rim Light

Luces

Rim Light



Práctica 9. Texturas, sombreado y práctica opcional

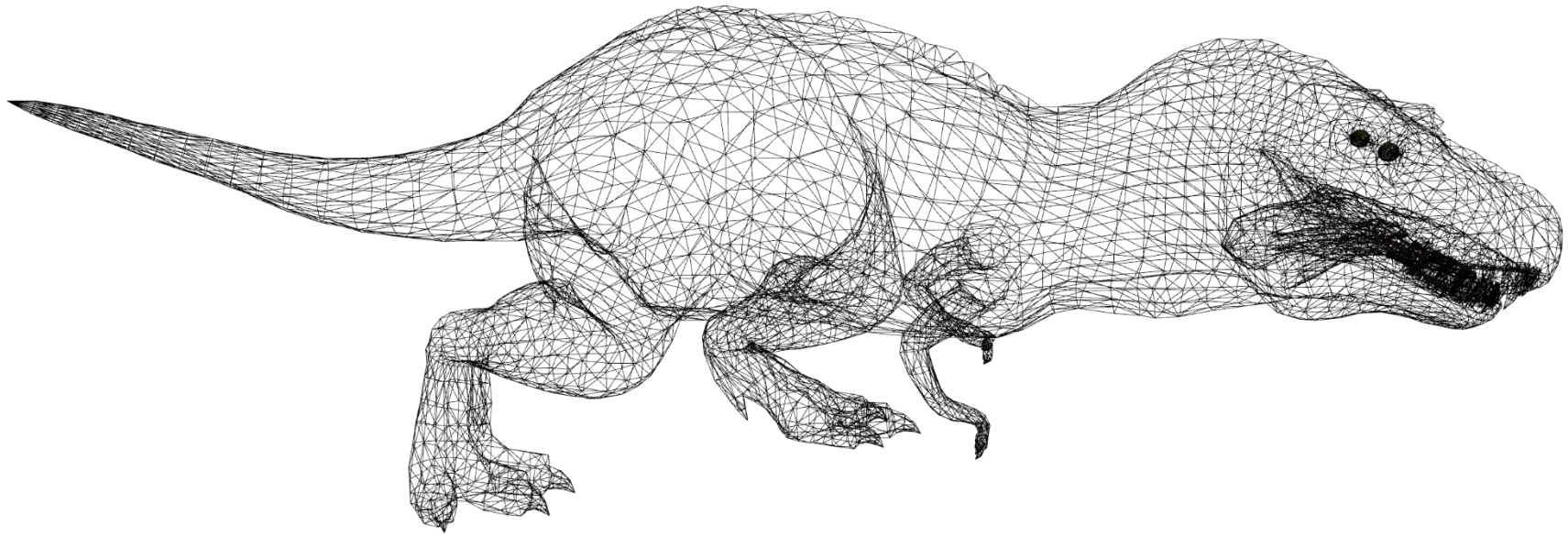
Curso académico 2022-2023

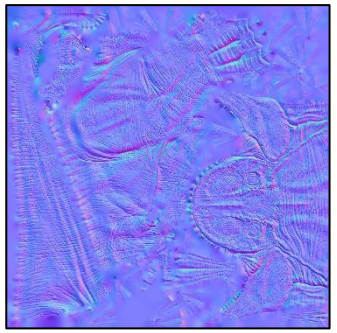
Programación de aplicaciones gráficas

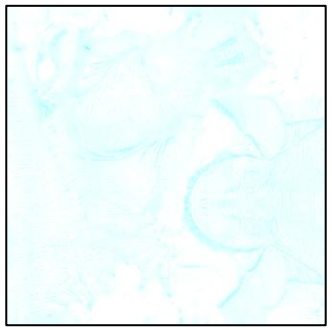
Grado en Ingeniería Informática

Normal Mapping

- Selección de vector normal utilizando la información de la textura.



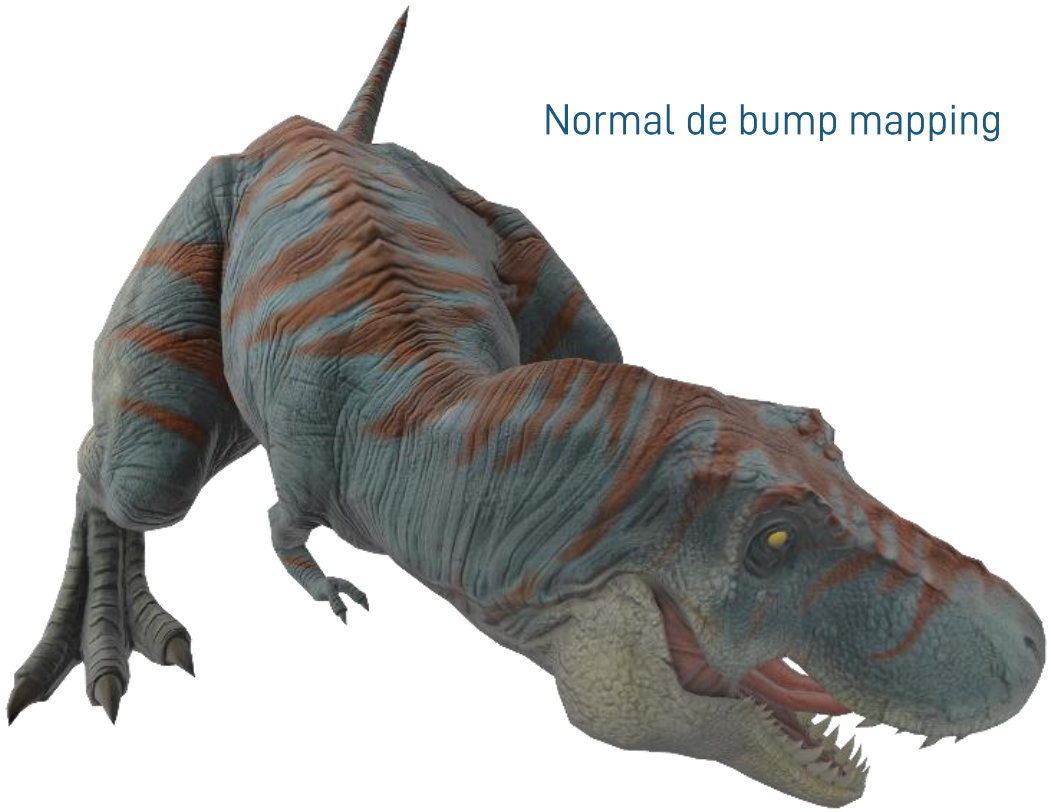




Normal Mapping

- **Selección de vector normal utilizando la información de la textura.**

Normal de VBO



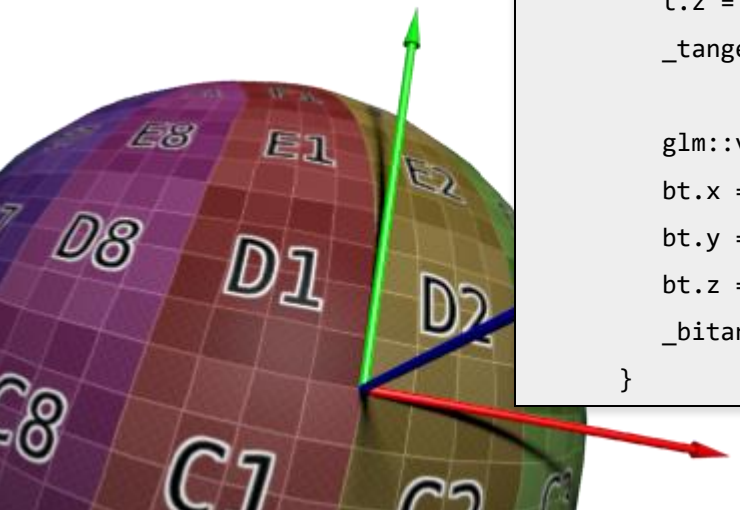
Normal de bump mapping

Normal Mapping

Cálculo de tangentes y bitangentes.

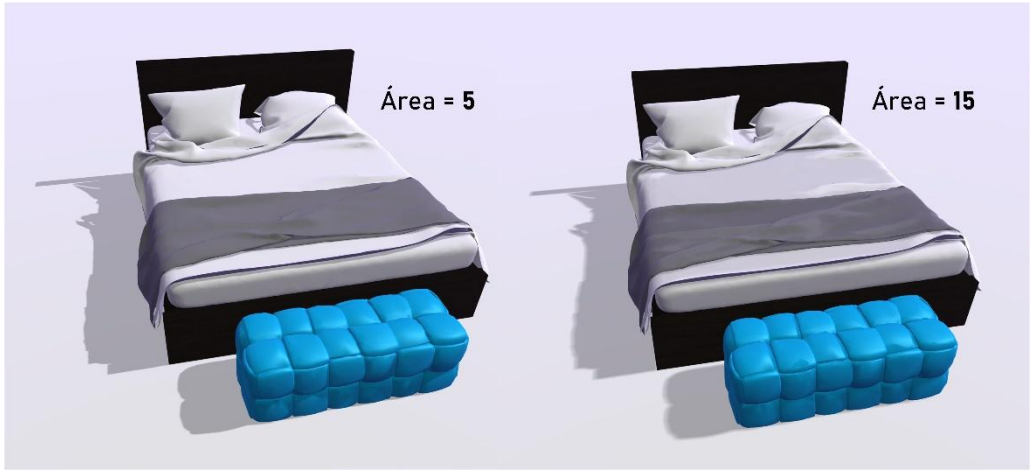
```
Assimp::Importer importador;
const aiScene* escena = importador.ReadFile ( rutaArchivo,
                                               aiProcess_JoinIdenticalVertices
                                               | aiProcess_Triangulate
                                               | aiProcess_GenSmoothNormals
                                               | aiProcess_CalcTangentSpace );
```

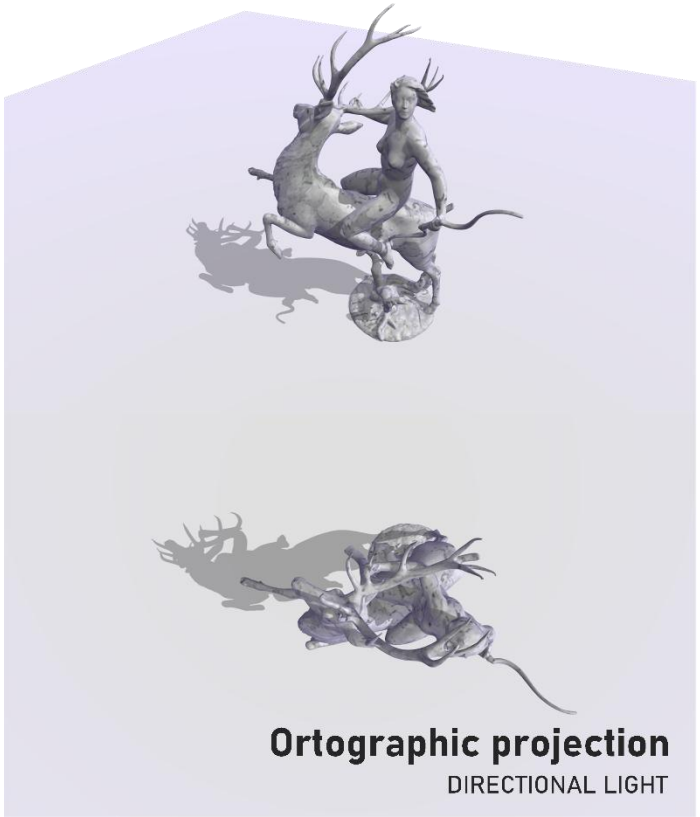
```
if ( malla->mTangents )   // Si hay tangentes, también habrá bitangentes
        glm::vec3 t;
        t.x = malla->mTangents[i].x;
        t.y = malla->mTangents[i].y;
        t.z = malla->mTangents[i].z;
        _tangentes.push_back ( t );

        glm::vec3 bt;
        bt.x = malla->mBitangents[i].x;
        bt.y = malla->mBitangents[i].y;
        bt.z = malla->mBitangents[i].z;
        _bitangentes.push_back ( bt );
     }
```

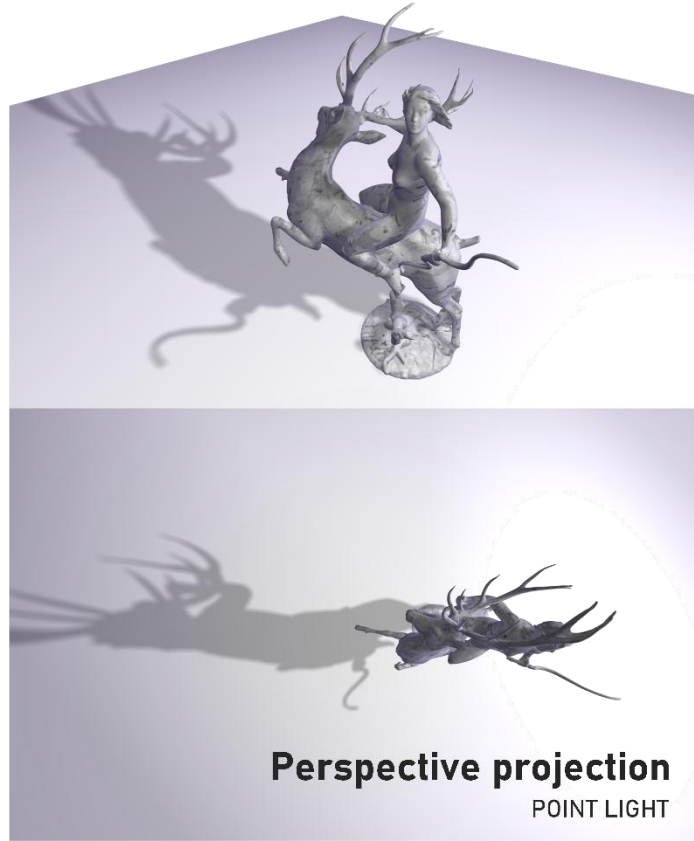


Shadow Mapping

- **Hard shadows vs soft shadows.**
  - Se puede suavizar la información de profundidad procedente de nuestra textura de shadow mapping.
  - Fragment shader.
- **La sombra varía en función de la proyección empleada.**

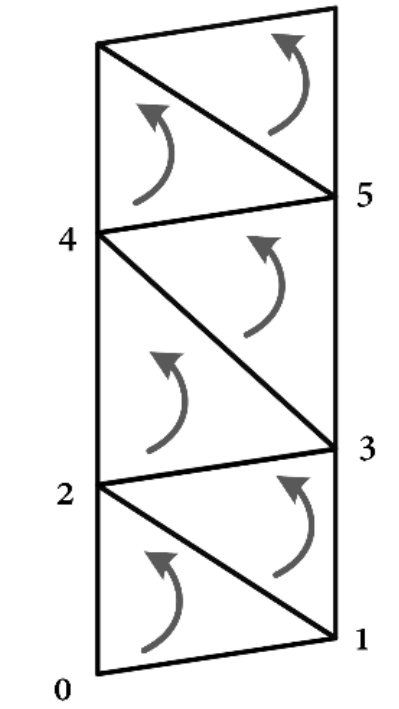




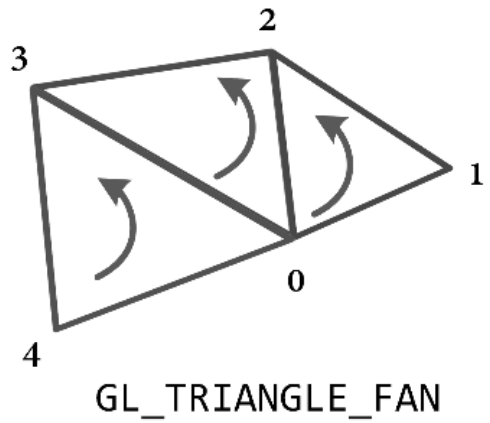


Superficies de revolución

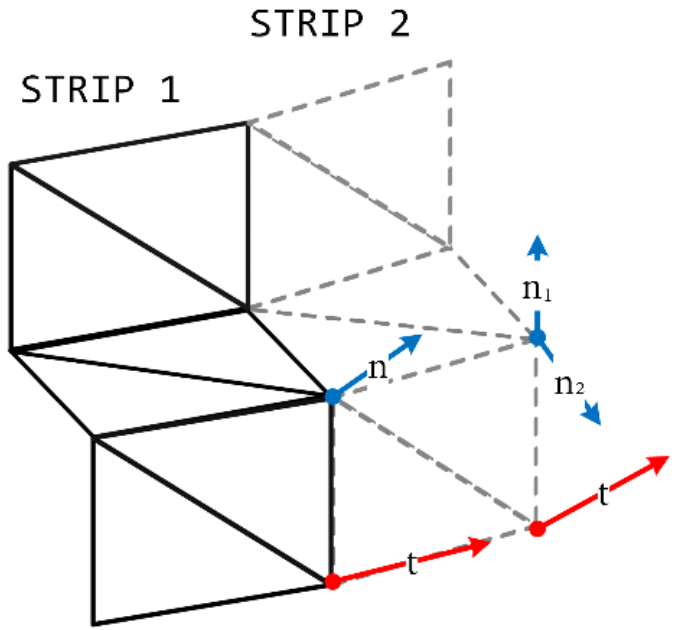
- **Construcción de superficies de revolución.**



a)



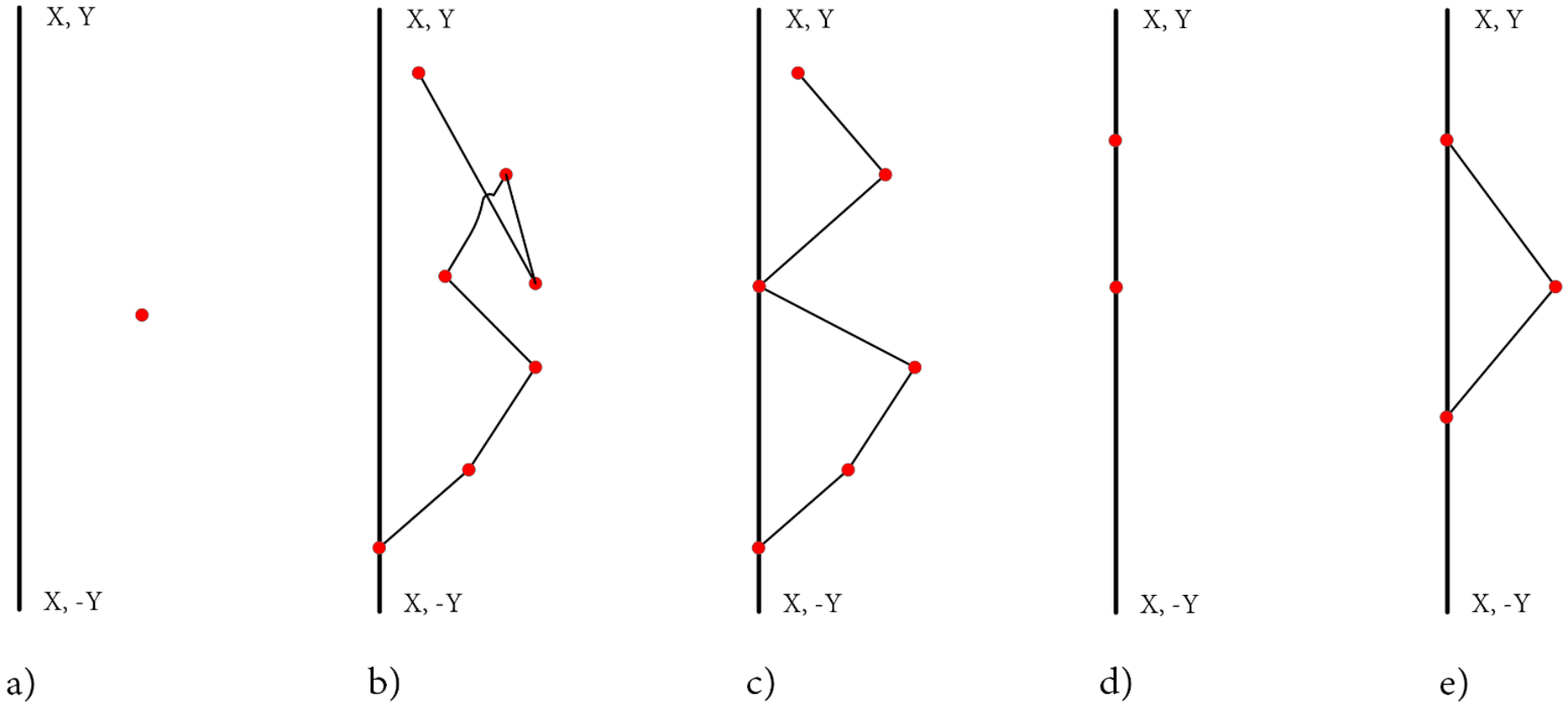
b)



c)

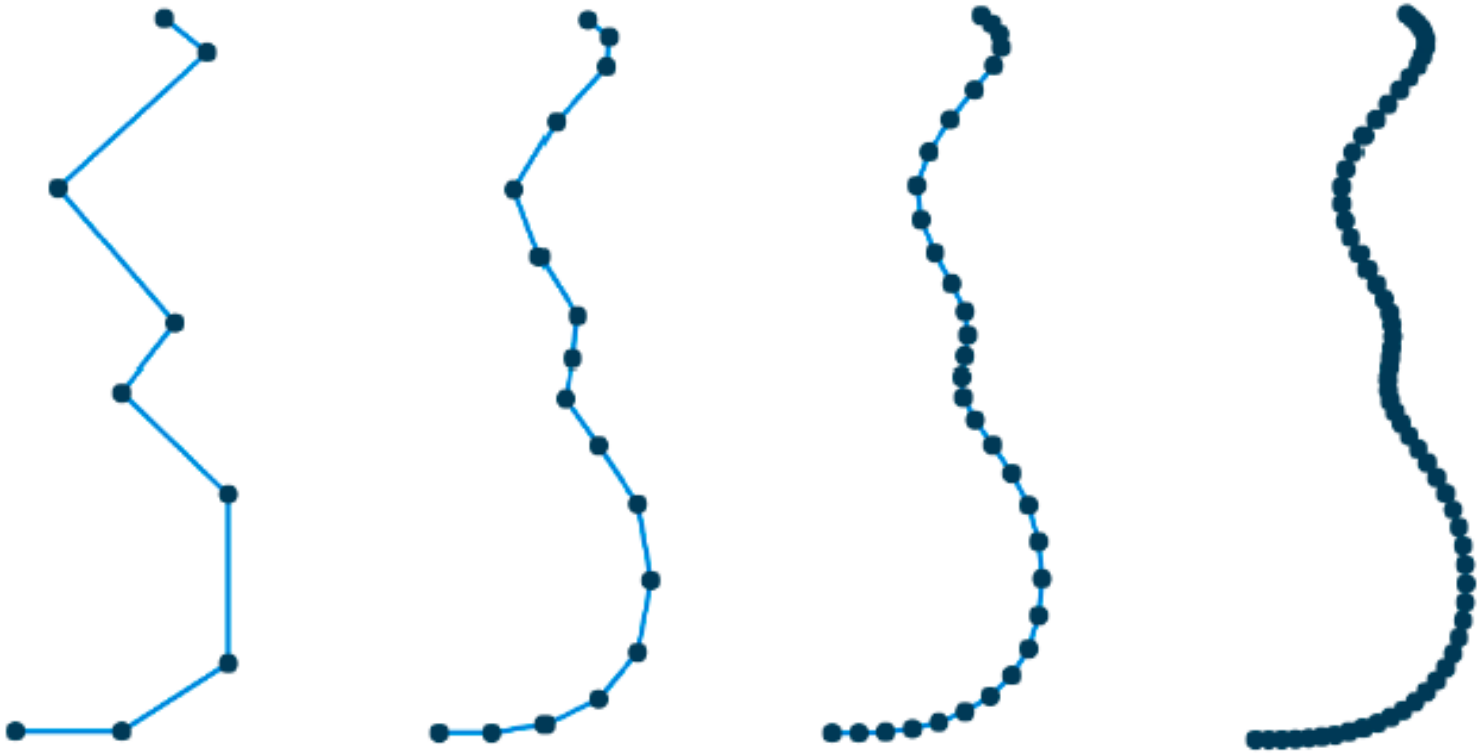
Superficies de revolución

- **Perfil de revolución procedente de datos de usuario.**
  - Habrá que tener en cuenta ciertos casos de error.



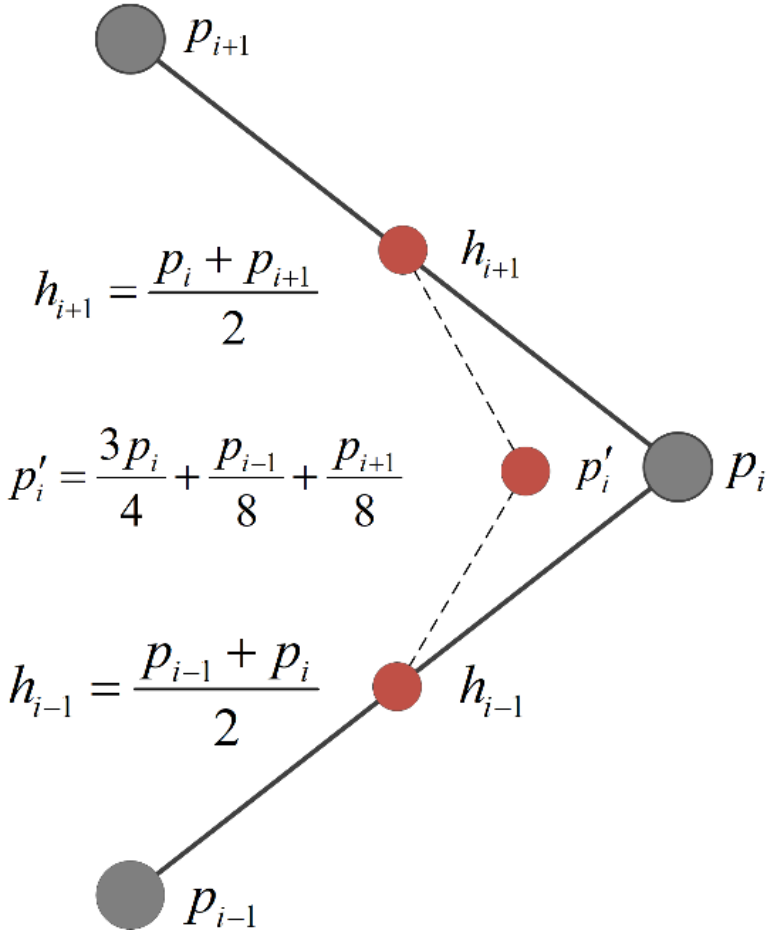
Superficies de revolución

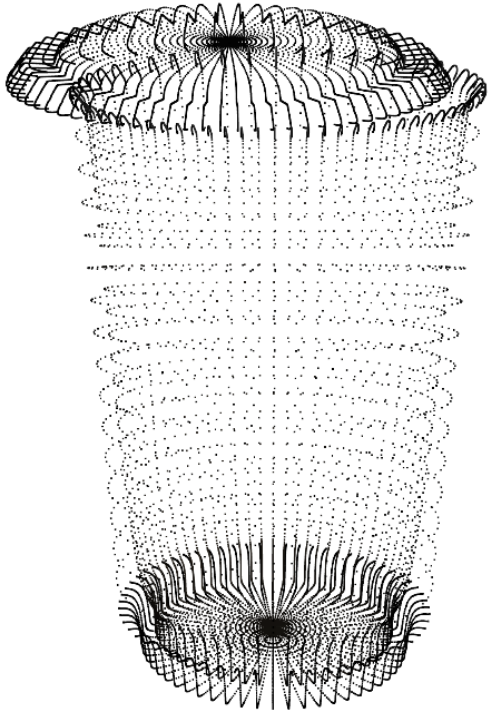
- **Subdivisión de perfiles de revolución.**

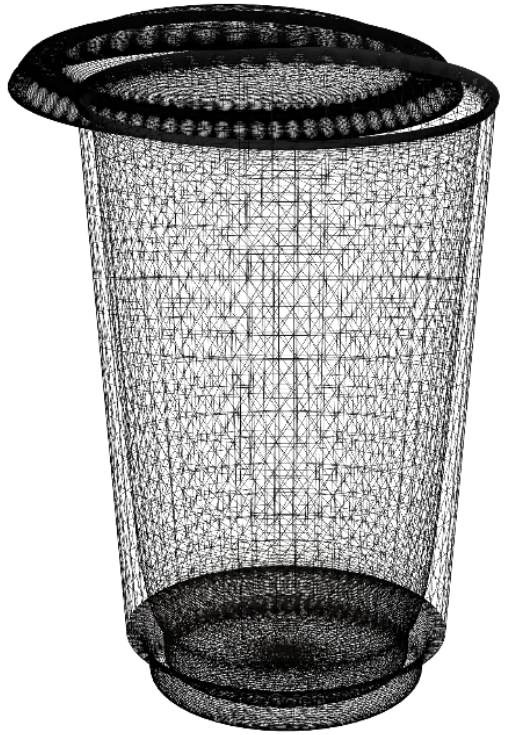


Superficies de revolución

- **Subdivisión de perfiles de revolución.**

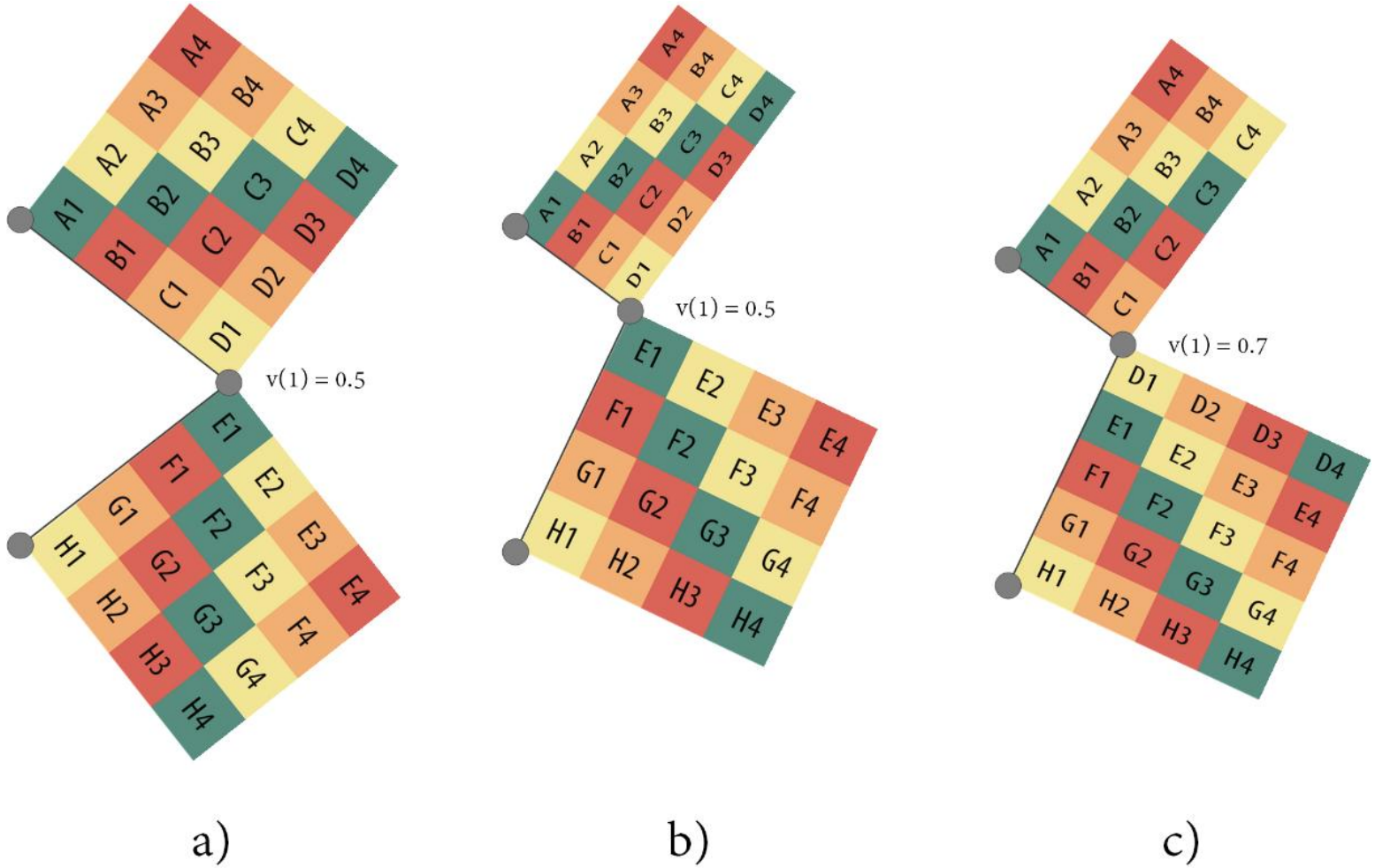






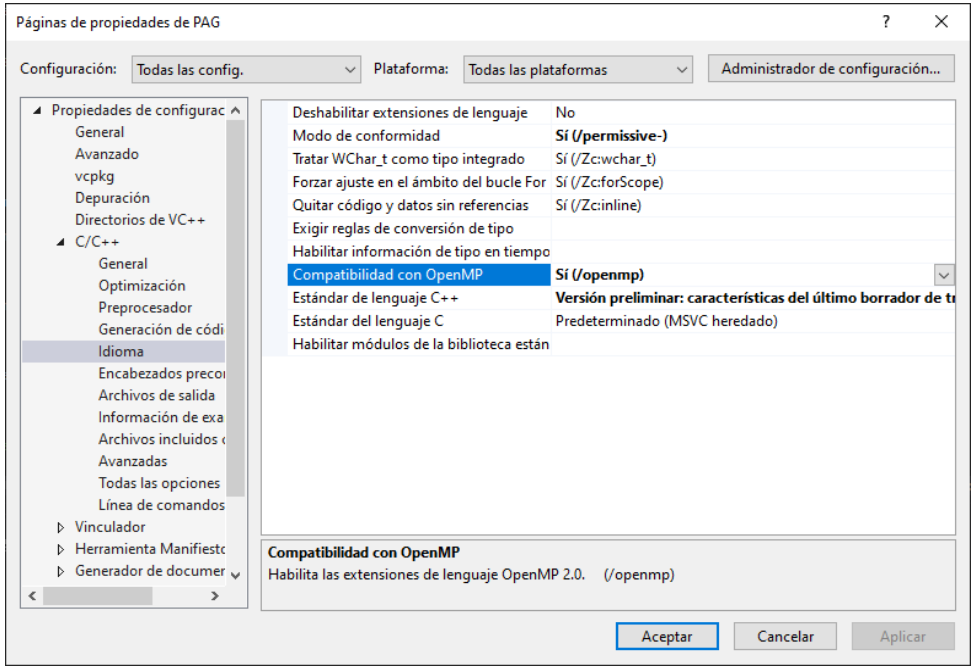
Superficies de revolución

- **Información de textura en perfiles de revolución.**



Superficies de revolución

- **Procesamiento en paralelo en CPU.**
  - OpenMP.



··· / Parallel programming / OpenMP / OpenMP library reference /

OpenMP Directives

Article • 18/05/2022 • 10 minutes to read • 11 contributors

Provides links to directives used in the OpenMP API.

Visual C++ supports the following OpenMP directives.

For parallel work-sharing:

Directive	Description
parallel	Defines a parallel region, which is code that will be executed by multiple threads in parallel.
for	Causes the work done in a `for` loop inside a parallel region to be divided among threads.
sections	Identifies code sections to be divided among all threads.
single	Lets you specify that a section of code should be executed on a single thread, not necessarily the main thread.

For main thread and synchronization:

Directive	Description
master	Specifies that only the main thread should execute a section of the program.
critical	Specifies that code is only executed on one thread at a time.
barrier	Synchronizes all threads in a team; all threads pause at the barrier, until all threads execute the barrier.
atomic	Specifies that a memory location that will be updated atomically.
flush	Specifies that all threads have the same view of memory for all shared objects.

Superficies de revolución

- **Procesamiento en paralelo en CPU.**
  - OpenMP.

```
#pragma omp parallel for
for (int idx = 0; idx < n; ++idx)
     DoSomething();

     #pragma omp atomic
     var += i;

     #pragma omp critical
     {
        vector.push_back(var);
     }
```