TRABAJOS FIN DE GRADO

Universidad de Jaén

Escuela Politécnica Superior de Jaén

TOULOUSE COMPILER 2.0: MEJORA DE UNA APLICACIÓN DE ESCRITORIO PARA PRUEBA DE SHADERS GLSL

Autor: José Antonio Montalbán Salvador

Grado: Ingeniería Informática

Tutores: Ángel Luis García Fernández y Francisco de Asís Conde Rodríguez
Departamento de Informática

Fecha: junio, 2024

Licencia CC

CREA

(Página intencionalmente en blanco)

Universidad de Jaén

Departamento de Informática

Don Ángel Luis García Fernández y Don Francisco de Asís Conde Rodríguez, tutores del Trabajo Fin de Grado titulado: '**Toulouse Compiler 2.0: Mejora de una aplicación de escritorio para prueba de shaders GLSL**', que presenta Don José Antonio Montalbán Salvador, otorgan el visto bueno para su entrega y defensa en la Escuela Politécnica Superior de Jaén.

Jaén, junio de 2024

El alumno:	Los tutores:
José Antonio Montalbán Salvador	Ángel Luis García Fernández Francisco de Asís Conde Rodríguez

(Página intencionalmente en blanco)

Agradecimientos

Quiero expresar mi más sincero agradecimiento a todas las personas que, de una manera u otra, han contribuido a la realización de este Trabajo de Fin de Grado.

En primer lugar, deseo agradecer a mis tutores Ángel Luis García Fernández y Francisco de Asís Conde Rodríguez, por su disposición para ayudarme con las dudas y por su orientación durante todo el proceso de investigación.

Mi agradecimiento especial a mis compañeros y amigos por su apoyo incondicional, sus valiosas sugerencias y por haber compartido conmigo momentos de estudio y reflexión que han sido fundamentales para la conclusión de este trabajo.

No puedo olvidar a mi familia, por su amor, comprensión y aliento constante. Su apoyo ha sido una fuente de fuerza y motivación durante todo este tiempo.

Gracias a todos, por hacer posible la culminación de este proyecto.

Ficha del Trabajo Fin de Título	
Titulación	Grado en Ingeniería Informática
Modalidad	Proyecto de Ingeniería
Especialidad (solo TFG)	Tecnologías de la Información
Mención (solo TFG)	Sistemas Gráficos
Idioma	Español
Tipo	Específico
TFT en equipo	No
Autor/a	José Antonio Montalbán Salvador
Fecha de asignación	16/04/2024
Descripción corta	Partiendo de un trabajo ya defendido en el curso 2019-2020, en este TFG se plantea la mejora de la aplicación desarrollada, incluyendo nuevas características, depuración de errores y refactorización del código, haciéndola portable al menos a los sistemas Windows y GNU/Linux.

NORMAS APLICADAS EN ESTE DOCUMENTO	
Locales	
TFT-UJA:2017	Normativa de Trabajos Fin de Grado, Fin de Máster y otros Trabajos Fin de Título de la Universidad de Jaén (Normativa marco UJA aprobada en Consejo de Gobierno)
TFT-EPSJ:2017	Normativa sobre Trabajos Fin de Grado y Fin de Máster en la Escuela Politécnica Superior de Jaén (Normativa EPSJ aprobada en Junta de Escuela)
TFT-EPSJ	Criterios de evaluación y normas de estilo para TFG y TFM de la Escuela Politécnica Superior de Jaén
Nacionales e Internacionales	
ISO 2145:1978	Documentación - Numeración de divisiones y subdivisiones en documentos escritos
UNE 50132:1994	Traducción de la ISO 2145
APA 6ª edición	Estilo de referencias y citas de APA (American Psychological Association)

NORMAS UTILIZADAS COMO BASE O REFERENCIA	
Nacionales	
UNE 157001:2014	Criterios generales para la elaboración formal de los documentos que constituyen un proyecto técnico
UNE 157801:2007	Criterios generales para la elaboración de proyectos de sistemas de información
*Estas normas se han utilizado **como base o referencia** para la inclusión de algunos contenidos y definiciones sobre elaboración de proyectos, entendiendo como **proyecto** la documentación consensuada entre una empresa y un cliente, que da lugar al perfeccionamiento de un contrato para la elaboración de una obra o la prestación de un servicio. Por consiguiente, no debe esperarse la aplicación de estas normas en cuanto a la completitud de los contenidos ni a la organización de los mismos.*	

(Página intencionalmente en blanco)

Contenido

1 Especificación del trabajo 19

1.1 Introducción 19
1.2 Objetivos del trabajo 20
1.3 Antecedentes y estado del arte 20
1.4 Descripción de la situación de partida 32
1.4.1 Descripción del entorno actual 35
1.4.2 Resumen de las deficiencias y carencias identificadas 41
1.5 Requisitos iniciales 42
1.6 Alcance 43
1.7 Hipótesis y restricciones 43
1.8 Estudio de alternativas y viabilidad 44
1.9 Descripción de la solución propuesta 45
1.10 Tecnologías utilizadas 45
1.11 Metodología de desarrollo de software 46
1.12 Estimación del tamaño y esfuerzo 48
1.13 Planificación temporal 50
1.14 Presupuesto 52

2 Diseño inicial 55

2.1 Especificaciones del sistema 55
2.2 Diseño del sistema 57
2.2.1 Diseño arquitectónico del sistema 59
2.2.2 Diseño de interfaz y storyboards 71

3 Desarrollo 74

3.1 Implementación 74
3.2 Pruebas finales 81

4 Conclusiones y trabajos futuros 99

5 Apéndices 101

5.1 Guía original del Trabajo Fin de Título 101
5.2 Instalación y configuración de la aplicación 104
5.2.1 Instalación en el sistema operativo Windows. 104
5.2.2 Instalación en sistemas GNU/Linux. 105
5.3 Manuales de usuario 105
5.3.1 Cargar un modelo 106
5.3.2 Cargar y guardar un shader 109
5.3.3 Alternar entre modos de visualización 112
5.3.4 Gestión de uniforms 115
5.3.5 Gestión de subrutinas 119
5.3.6 Cargar una textura 123
5.3.7 Edición de shaders y visualización 125

6 Bibliografía 126

Índice de ilustraciones

Ilustración 1 Sombreado de Lambert (Sombreado plano) (fuente: https://www4.ujaen.es/~demiras/cgex/#shading) 21
Ilustración 2 Sombreado de Gouraud (fuente: https://www4.ujaen.es/~demiras/cgex/#shading) 21
Ilustración 3 Sombreado de Phong (fuente: https://www4.ujaen.es/~demiras/cgex/#shading) 21
Ilustración 4 Cel shading con textura básica (fuente: https://es.wikipedia.org/wiki/Sombreado_plano_(animación) 22
Ilustración 5 Fragmento de código GLSL (fuente: elaboración propia) 23
Ilustración 6 Iluminación local y global (fuente: https://www.esi.uclm.es/www/cglez/fundamentos3D/03.01.Introduccion.html) 23
Ilustración 7 Vertex Array Object (fuente: Asignatura Programación de Aplicaciones Gráficas) 24
Ilustración 8 Vertex Buffer Object Entrelazado y no entrelazado (fuente: Asignatura Programación de Aplicaciones Gráficas) 25
Ilustración 9 Ejemplo de dos IBOS diferentes (fuente: Asignatura Programación de Aplicaciones Gráficas) 25
Ilustración 10 Proceso de compilación y enlazado de los shaders (fuente: Asignatura Programación de Aplicaciones Gráficas) 26
Ilustración 11 Pipeline de rendering (fuente: Asignatura Programación de Aplicaciones Gráficas) 27
Ilustración 12 Definición y uso de uniforms (fuente: elaboración propia) 28
Ilustración 13 Definición y uso de una subrutina (fuente: elaboración propia) 28
Ilustración 14 Mipmaps (fuente: https://docs.unity3d.com/Manual/texture-mipmaps-introduction.html) 29
Ilustración 15 Parámetros de la cámara (fuente: Asignatura Programación de Aplicaciones Gráficas) 30
Ilustración 16 Movimiento Orbit (fuente: Asignatura Programación de Aplicaciones Gráficas) 31
Ilustración 17 Movimiento Zoom (fuente: Asignatura Programación de Aplicaciones Gráficas) 32
Ilustración 18 Ventana principal de la aplicación original (fuente: elaboración propia) 34
Ilustración 19 Tabla de uniforms de la aplicación original (fuente: elaboración propia) 34
Ilustración 20 Clase OpenGLWidget (fuente: elaboración propia) 36
Ilustración 21 Clase PagShaderProgram (fuente: elaboración propia) 37
Ilustración 22 Clase Cámara (fuente: elaboración propia) 38
Ilustración 23 Diagrama modelos (fuente: elaboración propia) 39
Ilustración 24 Clase VAO (fuente: elaboración propia) 40
Ilustración 25 Diagrama Luces (fuente: elaboración propia) 40
Ilustración 26 Diagrama Texturas y Materiales (fuente: elaboración propia) 41
Ilustración 27 Diagrama de Gantt (fuente: elaboración propia) 51
Ilustración 28 Diagrama de casos de uso (fuente: elaboración propia) 57

Ilustración 29 Estructura de nodos de Assimp (fuente: https://learnopengl.com/Model-Loading/Assimp)..60
Ilustración 30 Construcción de sólido de revolución (fuente: https://www.aulafacil.com/cursos/matematicas-secundaria-eso/matematicas-segundo-eso-13-anos/cuerpos-de-revolucion-l7712) 61
Ilustración 31 Algoritmo de subdivisión de polilíneas (fuente: Asignatura Programación de Aplicaciones Gráficas)...62
Ilustración 32 Formato de sólido de revolución .SOR (fuente: elaboración propia) 63
Ilustración 33 Diagrama UML para la funcionalidad de cargar un modelo en formato .OBJ o .SOR (fuente: elaboración propia) 64
Ilustración 34 Diagrama UML para la gestión del vertex, fragment y geometry shader (fuente: elaboración propia)..65
Ilustración 35 Clase uniform (fuente: elaboración propia) 67
Ilustración 36 Clase subrutina (fuente: elaboración propia) 69
Ilustración 37 Clase cámara (fuente: elaboración propia) 70
Ilustración 38 Diagrama UML para la visualización de la escena (fuente: elaboración propia) 71
Ilustración 39 Interfaz principal de la aplicación (fuente: elaboración propia).........................72
Ilustración 40 Interfaz para la gestión de uniforms (fuente: elaboración propia) 73
Ilustración 41 Interfaz para la gestión de subrutinas (fuente: elaboración propia).................73
Ilustración 42 Diagrama UML para la gestión de funcionalidades e interfaces (fuente: elaboración propia)..74
Ilustración 43 Método setupMesh (fuente: elaboración propia)..76
Ilustración 44 Carga de modelos con Assimp (fuente: elaboración propia)...........................77
Ilustración 45 Método setUniform de la clase ShaderProgram (fuente: elaboración propia) .78
Ilustración 46 Método apply de la clase Uniform (fuente: elaboración propia) 79
Ilustración 47 Parámetros de la cámara virtual (fuente elaboración propia)..........................79
Ilustración 48 Método initializeGL de la clase Renderer (fuente: elaboración propia) 80
Ilustración 49 Comunicación de las distintas acciones de la intergaz principal (fuente: elaboración propia)..81
Ilustración 50 Modelos para realizar la prueba (fuente: elaboración propia) 82
Ilustración 51 Vertex shader para la prueba de visualización (fuente: elaboración propia) ...82
Ilustración 52 Fragment shader para la prueba de visualización (fuente: elaboración propia) 82
Ilustración 53 Resultado de la carga de cada uno de los modelos (fuente: elaboración propia) 83
Ilustración 54 Vertex shader para la prueba de uniforms (fuente: elaboración propia)..........84
Ilustración 55 Fragment shader para la prueba de uniforms (fuente: elaboración propia).....84
Ilustración 56 Asignación de uniforms (fuente: elaboración propia) 85
Ilustración 57 Resultado de la prueba de uniforms (fuente: elaboración propia)...................85
Ilustración 58 Resultado al aplicar modo de visualización de puntos (fuente: elaboración propia)..86
Ilustración 59 Resultado al aplicar modo de visualización de líneas (fuente: elaboración propia) 87
Ilustración 60 Resultado al aplicar modo de visualización de triángulos (fuente: elaboración propia)..88

Ilustración 61 Vertex shader para la prueba de subrutinas (fuente: elaboración propia) 88
Ilustración 62 Fragment shader para la prueba de subrutinas (fuente: elaboración propia) ..89
Ilustración 63 Asignación de la primera implementación de subrutina (fuente: elaboración propia) 89
Ilustración 64 Resultado al aplicar la primera implementación de subrutina (fuente: elaboración propia) 90
Ilustración 65 Asignación de la segunda implementación de subrutina (fuente: elaboración propia) 91
Ilustración 66 Resultado al aplicar la segunda implementación de subrutina (fuente: elaboración propia) 91
Ilustración 67 Vertex shader para la prueba de texturas (fuente: elaboración propia) 92
Ilustración 68 Fragment shader para la prueba de texturas (fuente: elaboración propia) 92
Ilustración 69 Textura usada en la prueba (fuente: Asignatura Programación de Aplicaciones Gráficas) 93
Ilustración 70 Resultado de la prueba de texturas (fuente: elaboración propia) 93
Ilustración 71 Geometry shader (fuente: elaboración propia) 94
Ilustración 72 Vertex shader para la prueba del geometry shader (fuente: elaboración propia) 95
Ilustración 73 Fragment shader para la prueba del geometry shader (fuente: elaboración propia) 95
Ilustración 74 Resultado del efecto explosión haciendo uso del geometry shader frame 1 (fuente: elaboración propia) 96
Ilustración 75 Resultado del efecto explosión haciendo uso del geometry shader frame 2 (fuente: elaboración propia) 97
Ilustración 76 Resultado del efecto explosión haciendo uso del geometry shader frame 3 (fuente: elaboración propia) 98
Ilustración 77 Formas de cargar un modelo (fuente: elaboración propia) 106
Ilustración 78 Cargando un modelo (fuente: elaboración propia) 107
Ilustración 79 Carga de modelo exitosa (fuente: elaboración propia) 107
Ilustración 80 Carga de modelo fallida (fuente: elaboración propia) 107
Ilustración 81 Ejemplo del formato .SOR 108
Ilustración 82 Formas de cargar y guardar shaders (fuente: elaboración propia) 110
Ilustración 83 Formato para la carga de shaders y carga múltiple (fuente: elaboración propia) 110
Ilustración 84 Editores de shaders una vez cargados (fuente: elaboración propia) 111
Ilustración 85 Error en shader (fuente: elaboración propia) 111
Ilustración 86 Opción para guardar un shader (fuente: elaboración propia) 112
Ilustración 87 Formas de alternar entre modos de visualización (fuente: elaboración propia) 113
Ilustración 88 Modo de visualización nube de puntos con textura (fuente: elaboración propia) 113
Ilustración 89 Modo de visualización líneas con textura (fuente: elaboración propia) 114
Ilustración 90 Modo de visualización triángulos con textura (fuente: elaboración propia) ...114
Ilustración 91 Formas de abrir tabla de uniforms (fuente: elaboración propia) 115
Ilustración 92 Tabla para la gestión de uniforms (fuente: elaboración propia) 116

Ilustración 93 Advertencia de error a la hora de añadir un uniform (fuente: elaboración propia) 117
Ilustración 94 Ejemplo de uso de uniforms (fuente: elaboración propia) 118
Ilustración 95 Formas de abrir la tabla de subrutinas (fuente: elaboración propia) 119
Ilustración 96 Tabla para la gestión de subrutinas (fuente: elaboración propia) 120
Ilustración 97 Advertencia de error al añadir subrutina (fuente: elaboración propia) 121
Ilustración 98 Ejemplo de uso de subrutinas (fuente: elaboración propia) 122
Ilustración 99 Formas de cargar una textura (fuente: elaboración propia) 123
Ilustración 100 Formatos para cargar una textura (fuente: elaboración propia) 124
Ilustración 101 Ejemplo de uso de textura (fuente: elaboración propia) 124
Ilustración 102 Editor de shaders (fuente: elaboración propia) 125

Índice de tablas

Tabla 1 Estimación de líneas de código 48
Tabla 2 Ecuaciones del modelo básico de COCOMO 49
Tabla 3 Perfiles para la realización del proyecto 52
Tabla 4 Tabla de costos por perfil 53
Tabla 5 Tabla de costos de recursos hardware 53
Tabla 6 Tabla de costos de recursos software 54
Tabla 7 Tabla de costos adicionales..54
Tabla 8 Presupuesto final...55

1 ESPECIFICACIÓN DEL TRABAJO

En este capítulo se presenta la especificación del trabajo, con una estructura y contenidos **inspirados** en los criterios y recomendaciones que establece la norma UNE 157801:2007 - "*Criterios Generales para la elaboración de proyectos de Sistemas de Información*".

1.1 Introducción

El objetivo principal del trabajo será la labor de refactorizar y mejorar una aplicación de escritorio basada en gráficos OpenGL que se realizó en el curso 2019-2020, corrigiendo los posibles errores que esta presente y añadiendo nuevas características. Debe permitir crear shaders escritos en lenguaje GLSL, los cuales se usarán para poder ser aplicados en modelos 3D que hayamos cargado previamente, y podrán ser visualizados en tiempo real para que de esta forma sirva como una herramienta de aprendizaje sobre la etapa de sombreado de un pipeline de rendering de una API de gráficos.

La motivación de realizar este proyecto es la de ocultar todos los aspectos que intervienen a la hora de realizar una aplicación gráfica, como pueden ser la elaboración de una interfaz, la gestión de memoria, la carga de modelos…, para centrarnos en las etapas de sombreado (shading), y por consiguiente ahorrar horas de trabajo y esfuerzo para obtener un shader que se podrá emplear en cualquier otra aplicación.

A lo largo del documento se profundizará sobre las clases empleadas, las tecnologías utilizadas y la metodología de trabajo que se ha seguido para finalmente ver el resultado obtenido del proyecto.

1.2 Objetivos del trabajo

- Poner en práctica los conocimientos y destrezas adquiridos durante los estudios de grado.
- Demostrar el nivel de madurez para el diseño e implementación de software.
- Producir un software funcional y eficiente, que pueda instalarse y usarse en computadoras de usuarios finales.
- Mejorar la aplicación existente, refactorizando el código, eliminando errores y añadiendo nuevas funcionalidades, tales como carga de modelos personalizados (como mínimo en formato OBJ), soporte para texturas personalizadas, soporte para subrutinas, múltiples luces configurables, etcétera.

1.3 Antecedentes y estado del arte

En este apartado vamos a realizar un recorrido a lo largo de los últimos años sobre la evolución de los shaders. Lo organizaremos por décadas, indicando las mejoras que se han adoptado. Finalmente, haremos un recorrido sobre cómo está formada una aplicación gráfica en la actualidad.

Década de 1960 y 1970: Durante estos años, la computación gráfica estaba en sus primeras etapas y se utilizaba principalmente en entornos académicos e industriales para aplicaciones científicas y de diseño asistido por computadora (CAD). Los gráficos por computadora eran simples y se basaban en modelos geométricos muy básicos, como líneas y polígonos. Los primeros algoritmos de sombreado gráfico eran simples y se centraban en el relleno de polígonos con colores sólidos, sin tener en cuenta efectos de iluminación o texturas, como se puede ver en la *Ilustración 1*.

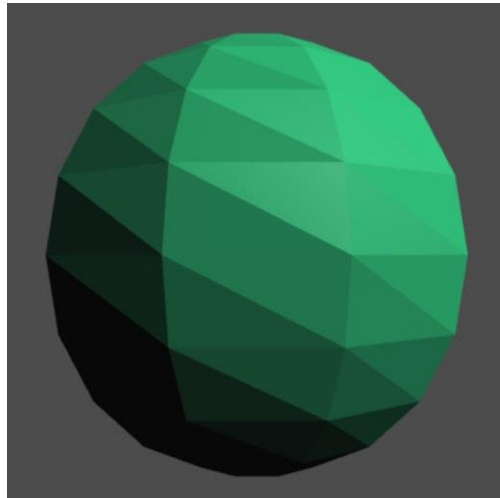


Ilustración 1 Sombreado de Lambert (Sombreado plano) (fuente: https://www4.ujaen.es/~demiras/cgex/#shading)

Década de 1980: Esta década marcó un punto de inflexión en los gráficos por computadora, con el desarrollo de los algoritmos de sombreado de Gouraud (*Ilustración 2*) y Phong (Ilustración 3). Estos algoritmos permitían simular efectos de iluminación más realistas, al calcular la iluminación en los vértices de los polígonos y luego interpolarla suavemente entre ellos. Esto permitió a los desarrolladores de software crear imágenes más suaves y realistas en comparación con los métodos de sombreado anteriores.

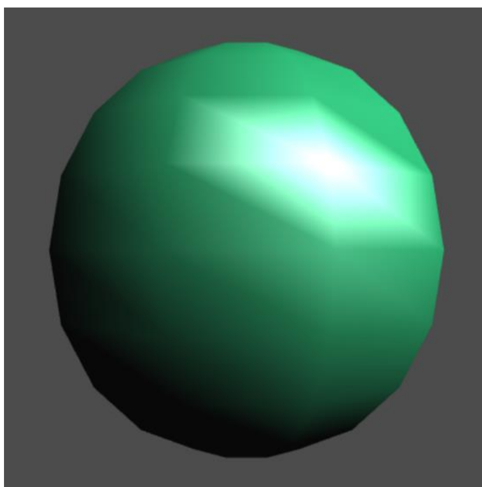


Ilustración 2 Sombreado de Gouraud (fuente: https://www4.ujaen.es/~demiras/cgex/#shading)

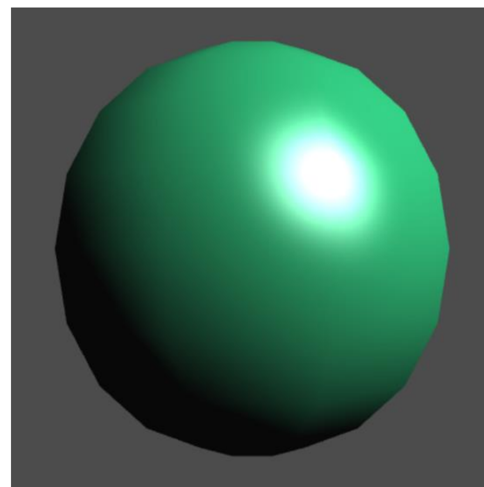


Ilustración 3 Sombreado de Phong (fuente: https://www4.ujaen.es/~demiras/cgex/#shading)

Década de 1990: Con el aumento de la potencia de procesamiento de las computadoras y el desarrollo de hardware gráfico dedicado, como las tarjetas gráficas, los gráficos por computadora se volvieron más comunes en aplicaciones comerciales y de entretenimiento, como los videojuegos y las películas CGI. En esta década, se introdujeron técnicas más avanzadas de sombreado, como el sombreado de texturas, que permitía mapear imágenes 2D sobre superficies 3D para obtener efectos visuales más detallados y realistas. También se desarrollaron algoritmos de sombreado específicos para efectos especiales, como el de cel shading, que simula el aspecto de la animación tradicional en dibujos animados, como se muestra en la *Ilustración 4*.

Ilustración 4 Cel shading con textura básica (fuente: https://es.wikipedia.org/wiki/Sombreado_plano_(animación)

Década de 2000: Esta década fue testigo de avances significativos en el hardware gráfico, con la introducción de la programabilidad en las unidades de procesamiento gráfico (GPU), lo que permitió la implementación de técnicas de sombreado más complejas en tiempo real.

Se introdujo el sombreado de píxeles programable (*Ilustración 5*), que permitía a los desarrolladores escribir sus propios programas (conocidos como shaders) para controlar el proceso de sombreado a un nivel más granular. Esto llevó a una mayor flexibilidad y realismo en los gráficos generados por computadora, y es aquí donde enmarcamos la labor del proyecto.

```
#version 410
out vec4 colorFragmento;
void main ()
{
colorFragmento = vec4 ( 0.0, 1.0, 0.0, 1.0 );
}
```

Ilustración 5 Fragmento de código GLSL (fuente: elaboración propia)

Década de 2010 en adelante: En los últimos años, los avances en los shaders gráficos se han centrado en la simulación de materiales realistas, la iluminación global y los efectos visuales avanzados. Los shaders de rayos se han vuelto más comunes, especialmente en aplicaciones de visualización no en tiempo real, como películas y efectos visuales.

Estos shaders simulan el comportamiento real de la luz al trazar rayos a través de la escena, lo que resulta en imágenes de alta calidad y muy realistas, como se puede observar en la *Ilustración 6*. Además, ha habido un enfoque en la optimización de los shaders para su uso en aplicaciones de realidad virtual y aumentada, lo que permite experiencias inmersivas y envolventes para los usuarios.

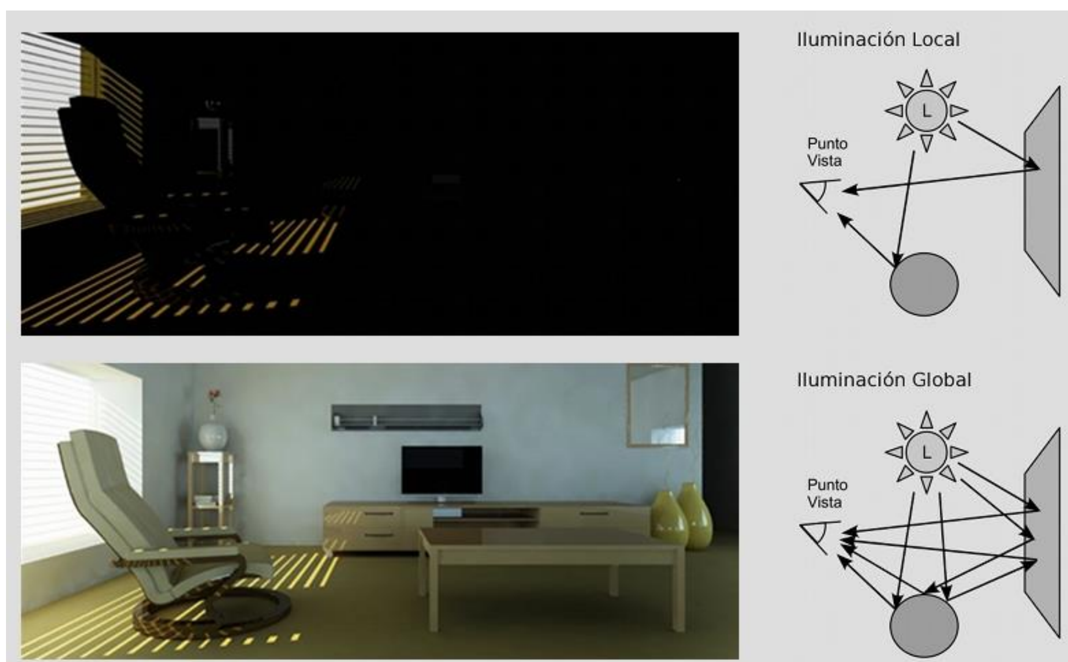


Ilustración 6 Iluminación local y global (fuente: https://www.esi.uclm.es/www/cglez/fundamentos3D/03.01.Introduccion.html)

En conjunto, la historia de los shaders gráficos refleja la evolución constante de la tecnología en el campo de la computación gráfica y la búsqueda continua de imágenes cada vez más realistas y convincentes. Desde los primeros días de la computación gráfica hasta la era actual de la visualización en tiempo real y la realidad virtual, los shaders gráficos han desempeñado un papel crucial en la evolución de los gráficos por computadora.

En la actualidad las aplicaciones gráficas que hacen uso de shaders requieren primero cargar un modelo 3D, que es una representación tridimensional discreta de un objeto, el cual está formado por una serie de vértices que contienen información sobre su posición, normal para el cálculo de iluminación, coordenadas de textura que indican qué parte de la textura le corresponde y tangente, entre otros.

Para pasar la geometría a la GPU es necesario cargarla previamente en la VRAM para que la tarjeta gráfica pueda trabajar con ella (Hughes, y otros, 2014). En la API de gráficos OpenGL, hay que definir un VAO (Vertex Array Object), que es una estructura de datos que guarda como su nombre indica arrays (*Ilustración 7*).

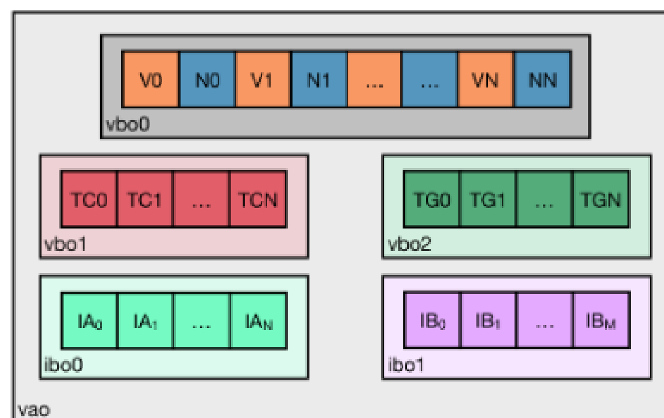


Ilustración 7 Vertex Array Object (fuente: Asignatura Programación de Aplicaciones Gráficas)

Dentro del VAO se pueden defininir varios VBOs (Vertex Buffer Object), los cuales pueden ser entrelazados o no entrelazados, atendiendo a si la información se guarda en un array de manera intercalada o en arrays independientes respectivamente (*Ilustración 8*), y contienen información relativa a los vértices (posición, normales, coordenadas de textura y tangentes).

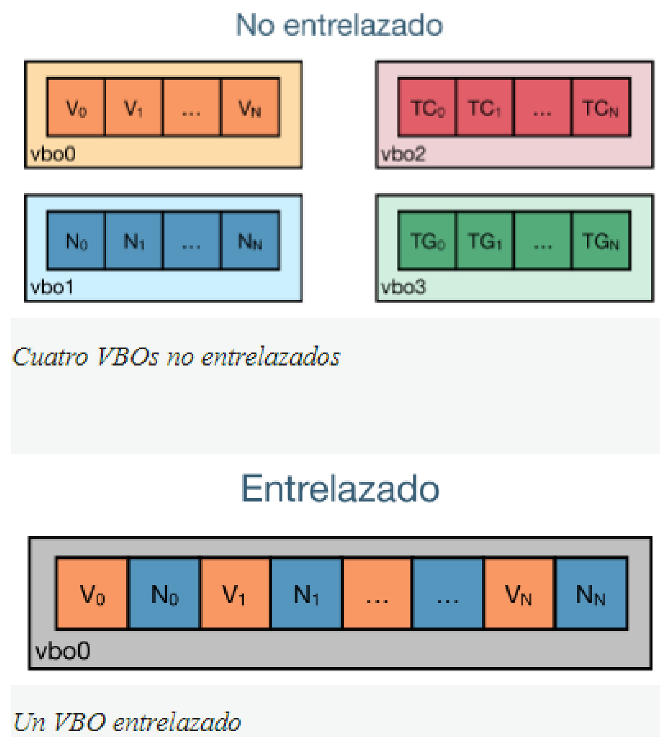


Ilustración 8 Vertex Buffer Object Entrelazado y no entrelazado (fuente: Asignatura Programación de Aplicaciones Gráficas)

De igual forma, un VAO puede tener uno o varios IBOs (Index Buffer Object), que guardan índices a un VBO almacenado en el mismo VAO. Estos índices, junto con el tipo de primitiva a dibujar, definen la topología de la geometría (*Ilustración 9*).

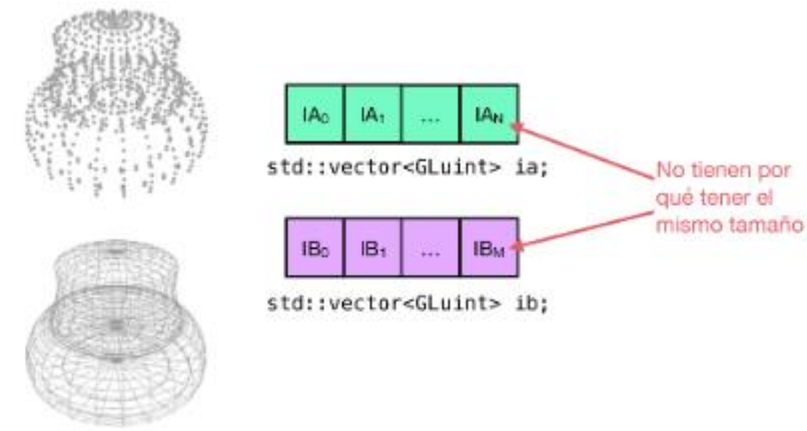


Ilustración 9 Ejemplo de dos IBOS diferentes (fuente: Asignatura Programación de Aplicaciones Gráficas)

Para la visualización de un modelo, después de haber cargado la geometría en VRAM, hay que cargar, compilar y enviar a la GPU un shader program (Wolff, 2018), que contiene las instrucciones que la GPU ha de seguir para procesar la geometría.

Un shader program está formado por shader objects (*Ilustración 10*). Estos son el vertex shader, geometry shader, compute shader y fragment shader.

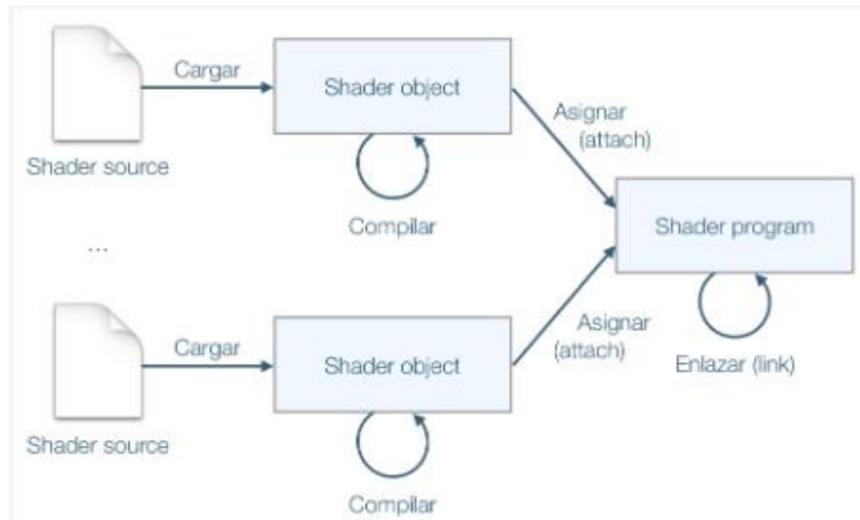


Ilustración 10 Proceso de compilación y enlazado de los shaders (fuente: Asignatura Programación de Aplicaciones Gráficas)

Cada uno de ellos se ejecutará en una etapa concreta del pipeline de rendering (Rodríguez, 2013). Como mínimo, el shader program debe estar formado por el vertex shader y fragment shader, donde el primero tiene como objetivo aplicar las transformaciones de modelado, visión y perspectiva, mientras que el segundo se ejecuta una vez por cada pixel que se obtiene en la fase de rasterización (Ilustración 11).

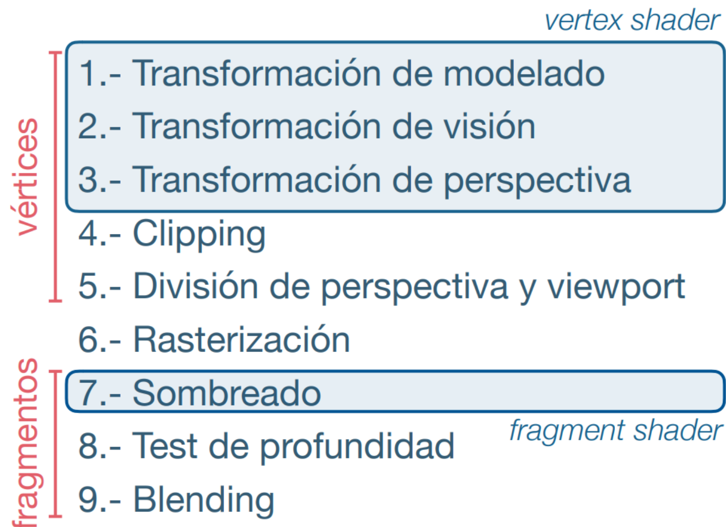


Ilustración 11 Pipeline de rendering (fuente: Asignatura Programación de Aplicaciones Gráficas)

El geometry shader es una etapa intermedia que se encuentra entre el vertex shader y el fragment shader. Su función principal es manipular las primitivas geométricas (como puntos, líneas y triángulos) que se generan en el vertex shader, antes de enviarlas al fragment shader para su rasterización (Movania, Lo, Wolff, & Lo, 2017).

El shader program recibe de la CPU la información sobre geometría (posición, normal…) a través de layouts (ubicaciones) que están numeradas a partir del cero. En él también se pueden definir uniforms y subrutinas.

Un uniform es un tipo especial de variable que se emplea en el shader program para pasar datos constantes que son accesibles desde los diferentes tipos de shader para realizar cálculos. Para usarlos se emplea un nombre, el cual deberá coincidir con el nombre especificado en el shader, y estas variables pueden ser de diferentes tipos (bool, int, float…). En la *Ilustración 12* se puede ver la información sobre la geometría, donde el layout cero corresponde con la posición del vértice y el layout uno corresponde a su normal. También la definición y uso de tres uniform de tipo matriz 4x4 en un shader.

```
#version 330 core

layout(location = 0) in vec3 vPosition;
layout(location = 1) in vec3 vNormal;

out vec3 position;
out vec3 normal;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    normal = vec3(view * model * vec4(vNormal, 1.0));
    position = vec3(view * model * vec4(vPosition,1.0));

    gl_Position = projection * view * model * vec4(vPosition, 1.0);
}
```

Ilustración 12 Definición y uso de uniforms (fuente: elaboración propia)

Las subrutinas son una característica avanzada que permiten cambiar dinámicamente las funciones ejecutadas dentro de los shaders sin necesidad de recompilar o reenlazar el shader program. Son útiles para crear shaders más flexibles y reutilizables, y para reducir el número de shader programs empleados en una aplicación. En la *Ilustración 13* se muestra la definición de una subrutina que tiene dos implementaciones.

```
#version 410
out vec4 colorFragmento;

subroutine vec4 a();
subroutine uniform a colorA;

subroutine(a)
vec4 red(){
    return vec4(1.0,0.0,0.0,1.0);
}

subroutine(a)
vec4 green(){
    return vec4(0.0,1.0,0.0,1.0);
}

void main ()
{
colorFragmento = colorA();
}
```

Ilustración 13 Definición y uso de una subrutina (fuente: elaboración propia)

Las aplicaciones gráficas también hacen uso de texturas. Una textura es una imagen bidimensional que se puede emplear en los shaders para proporcionar

información para realizar cálculos de iluminación, normales, valores de intensidad especular del material, coeficientes de brillo para los cálculos del modelo especular, rugosidad... Los mipmaps son una serie de texturas preprocesadas que son versiones reducidas de una textura original, cada una con la mitad de la resolución de la anterior. Estas versiones se generan automáticamente cuando se crea la textura y se almacenan en memoria junto con la textura principal (*Ilustración 14*).

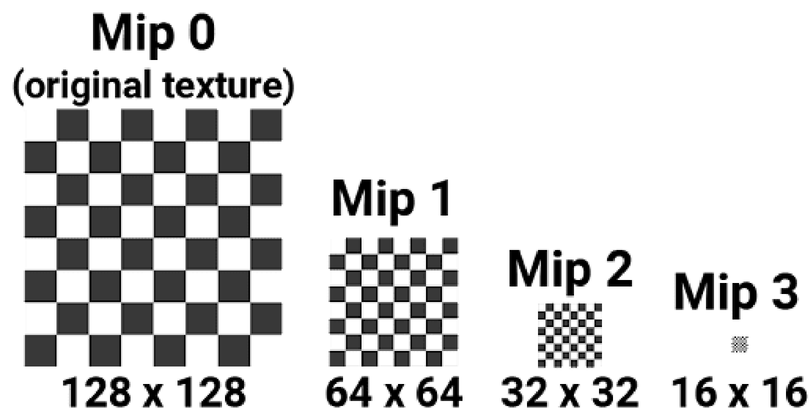


Ilustración 14 Mipmaps (fuente: https://docs.unity3d.com/Manual/texture-mipmaps-introduction.html)

Para aplicar una textura a un modelo, es necesario definir una correspondencia entre los vértices del modelo y los píxeles de la imagen de textura. Es por ello que los vértices guardan información sobre coordenadas de textura que están normalizados al rango [0,1] y se interpolan en el proceso de rasterización, donde coordenadas de textura resultantes se convierten al espacio de la textura.

Las tarjetas gráficas tienen una serie de circuitos, llamados unidades de textura, encargados específicamente de convertir de coordenadas de textura a coordenadas en el espacio de la textura, calcular los mipmaps, leer e interpolar valores de las imágenes de textura...

Por otro lado, las aplicaciones gráficas cuentan con cámaras virtuales que tienen una serie de parámetros para poder visualizar la escena de manera correcta (*Ilustración 15*). Estos son:

- Posición.

- Punto al que mira.
- Vector hacia arriba.
- Ángulo de visión.
- Relación de aspecto.
- Plano cercano y plano lejano.

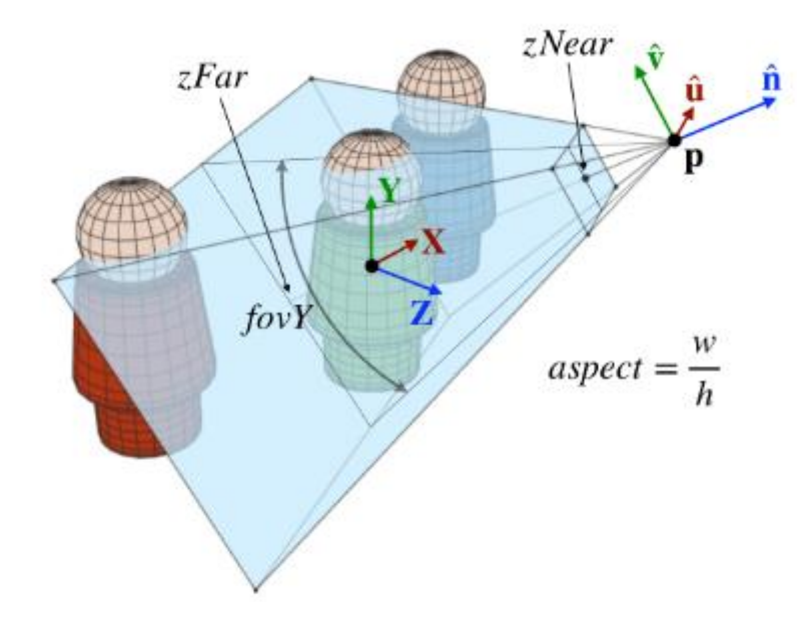


Ilustración 15 Parámetros de la cámara (fuente: Asignatura Programación de Aplicaciones Gráficas)

Con estos parámetros se calcularán los vectores v, u y n que representan el sistema de coordenadas local de la cámara.

Los parámetros de posición, punto al que mira y vector v se usarán para construir la matriz de visión, que se usa para mover y orientar la cámara y escena, para que la cámara pase a estar situada en el origen de coordenadas y mirando a la dirección negativa del eje Z (transformación de visión (Akenine-Möller, Haines, & Hoffman, 2008)).

Por otro lado, los parámetros de ángulo de visión, relación de aspecto, plano cercano y plano lejano se emplearán para construir la matriz de proyección, que se aplica para deformar los modelos de la escena, que reducen su tamaño cuanto más

alejados están de la cámara (transformación de perspectiva (Akenine-Möller, Haines, & Hoffman, 2008)).

La combinación de las matrices de visión y perspectiva junto a la del modelo (transformaciones aplicadas a este), se usan para transformar las coordenadas de los vértices de los objetos desde el espacio local del objeto hasta las coordenadas de pantalla.

La cámara puede definir algunos tipos de movimiento para moverse por la escena. Algunos de ellos son:

- El movimiento **orbit,** que es una composición de dos rotaciones, una en latitud y otra rotación en longitud alrededor del punto al que mira la cámara. Este movimiento se realiza aplicando una matriz que es la composición de una translación del punto al que mira la cámara respecto del origen, dos rotaciones (una en latitud y otra en longitud) y finalmente otra translación en sentido inverso a la primera translación. En la *Ilustración 16* se puede ver de manera más clara el tipo de movimiento.

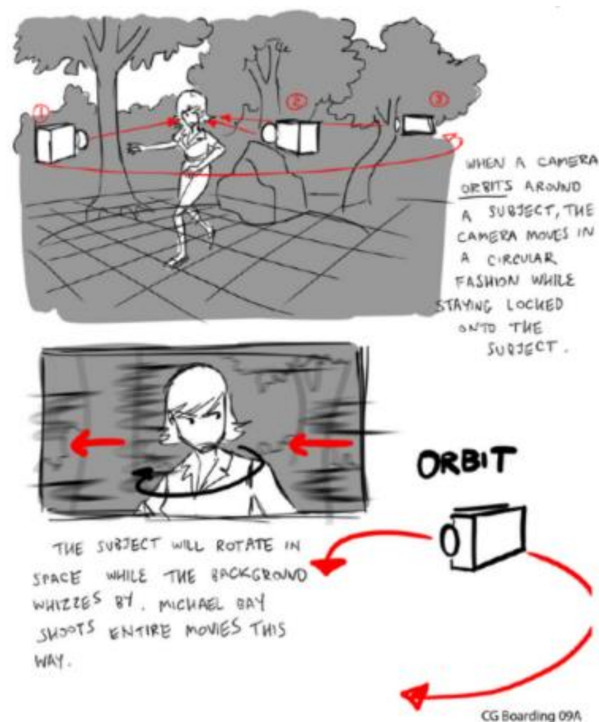


Ilustración 16 Movimiento Orbit (fuente: Asignatura Programación de Aplicaciones Gráficas)

- El movimiento **zoom** que, aunque no es realmente un movimiento, sí es considerado como tal y se realiza variando el ángulo de visión de la cámara.

Cuanto más aumenta el ángulo de visión, menos zoom y viceversa (*Ilustración 17).*

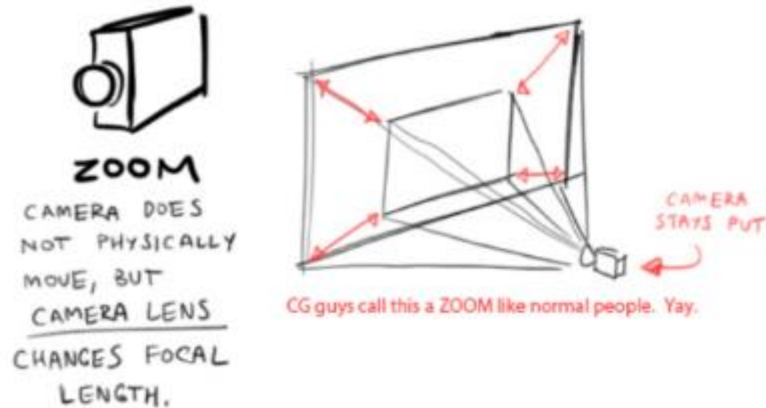


Ilustración 17 Movimiento Zoom (fuente: Asignatura Programación de Aplicaciones Gráficas)

1.4 Descripción de la situación de partida

Parto de un proyecto realizado en el año 2019-2020 por Antonio Jesús Viso Sánchez, el cual incluye un editor de shaders de vértices y fragmentos que permite elegir entre distintos modos de visualización con unos modelos fijos los cuales son sólidos de revolución (peón, copa, joystick). Cuenta con distintos modos de visualización y opciones para activar texturas y materiales predefinidos. La interfaz principal de la aplicación se muestra en la *Ilustración* 18. También cuenta con una interfaz para asignar uniforms como se puede ver en la *Ilustración 19*.

En la parte inferior tiene una pequeña consola que nos da información del estado de la aplicación, así como los posibles errores que presenta. En la parte central izquierda nos encontramos con los editores de shaders, que se pueden alternar haciendo clic en la pestaña correspondiente. En la parte central derecha está situada la escena de la aplicación, donde se va a mostrar uno de los modelos comentados anteriormente con el shader que se haya escrito. Finalmente, en la parte superior nos encontramos con una barra de herramientas con las diferentes funcionalidades representadas con iconos, además de distintos menús para acceder a las funcionalidades si la barra de herramientas estuviera oculta.

Todas las interfaces de la aplicación están realizadas con el framework Qt (Qt Documentation, s.f.). Qt cuenta con un editor de interfaces muy completo donde se pueden usar numerosos widgets, que son clases para realizar diferentes funcionalidades (barra de herramientas, separadores, texto, renderer, etiquetas…). Para la comunicación entre objetos en Qt se hace uso de funciones signal, que emiten una señal cuando ocurre un evento (pulsar un botón, por ejemplo) y funciones slot, que son llamadas en respuesta a los signals. Se conectan mediante el método connect().

La aplicación funciona cargando los shaders o escribiéndolos directamente sobre sus editores, los cuales remarcan de colores las diferentes palabras clave. Una vez cargados se procedería a añadir uniforms si los shaders los requieren. Posteriormente, se selecciona el modo de visualización o se activan el material o textura. Se elige el modelo de los tres disponibles. Finalmente, se pulsa en la opción ejecutar, donde si todo ha salido correctamente, se visualizará el modelo con el shader aplicado y de lo contrario se mostrarán los errores en la consola.

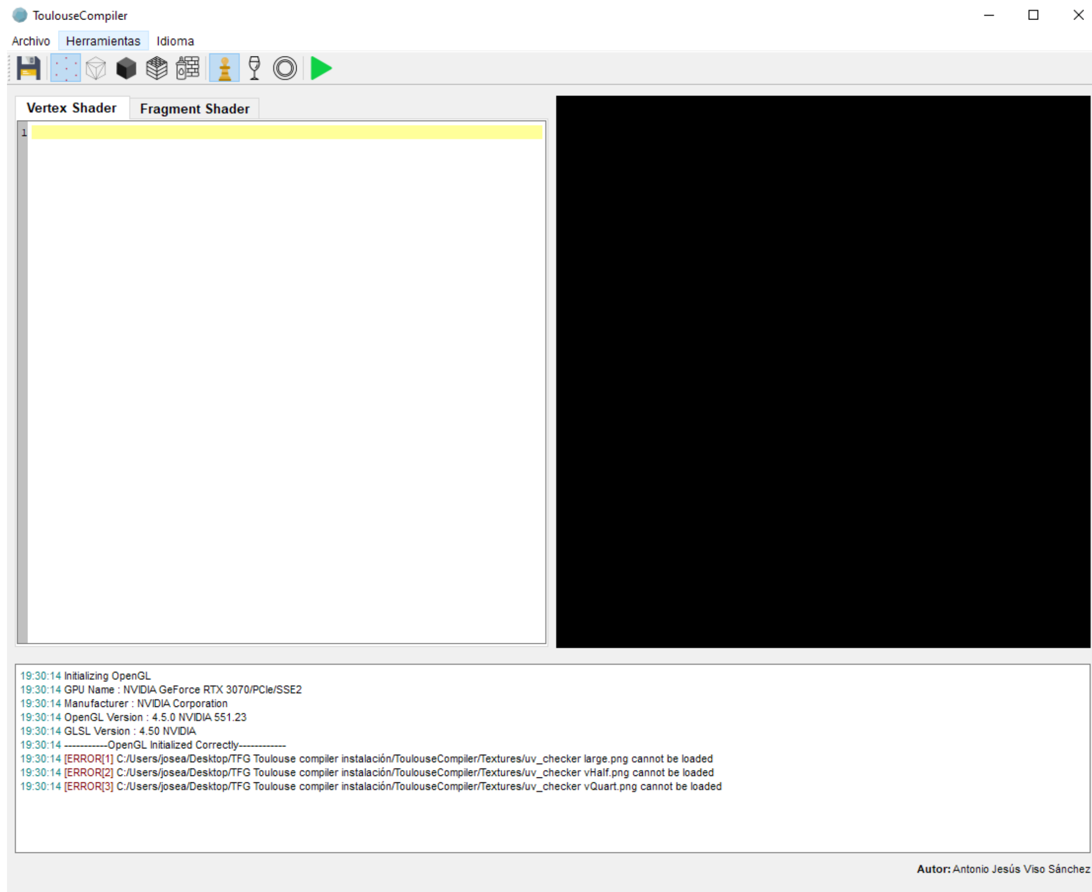


Ilustración 18 Ventana principal de la aplicación original (fuente: elaboración propia)

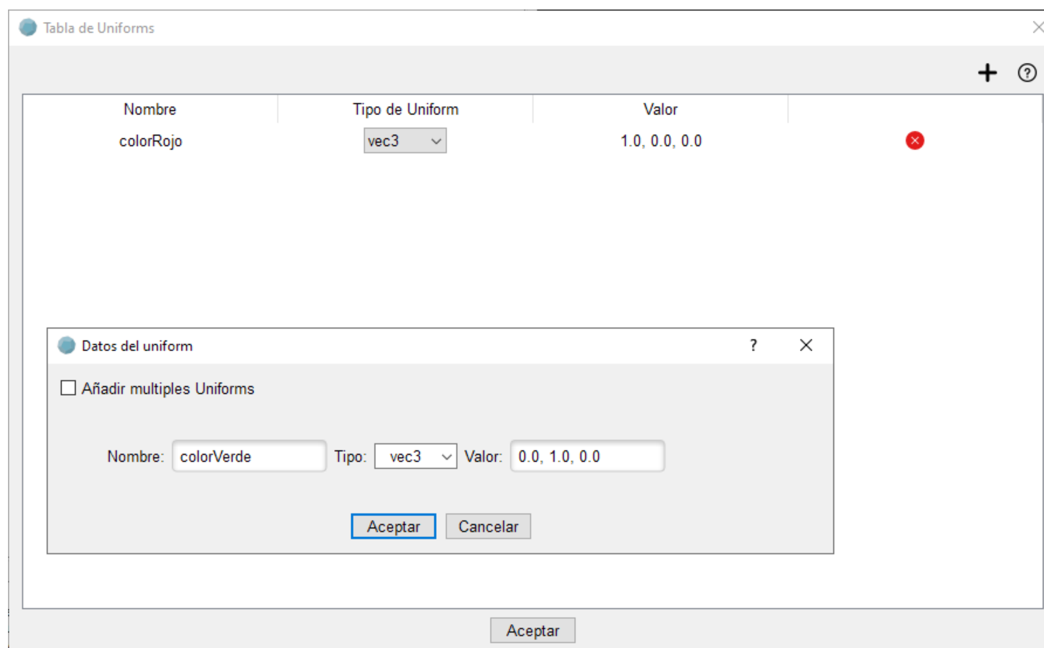


Ilustración 19 Tabla de uniforms de la aplicación original (fuente: elaboración propia)

En este trabajo voy a analizar el código de esta aplicación, refactorizarlo e implementar nuevas características, para finalmente obtener una versión ampliada de la aplicación la cual servirá de herramienta de software libre para el aprendizaje en el campo de los shaders.

1.4.1 Descripción del entorno actual

En este apartado voy a realizar un análisis de la arquitectura de la aplicación actual para entender las decisiones adoptadas, ya que no dispongo de la documentación del proyecto original.

Para gestionar el canvas OpenGL, el cual está situado a la derecha al abrir la aplicación, hay definida una clase que se llama OpenGLWidget (*Ilustración 20*) que trabaja directamente con Qt, la cual contiene todos los métodos de visualización, así como parámetros importantes como pueden ser los shaders y los uniforms.

```
OpenGLWidget
----------------------------------------
- camera: PagCamera*
- w: int
- h: int
- posX: int
- posY: int
- instance: OpenGLWidget*
- m_animating: bool
- m_context: QOpenGLContext*
- m_device: QOpenGLPaintDevice*
- viewMatrix: glm::mat4
- projMatrix: glm::mat4
- aspect: float
- ambiental: AmbientLight*
- puntual: PointLight*
- puntual2: PointLight*
- objetos: std::vector<Pag3DGroup>
- shaderProgram: PagShaderProgram*
- uniforms: QVector<dataForUniform>
- modeTrial: bool
- typeTrial: int
- pushCompile: bool
- TypeObject: int
- shaderPath: string
----------------------------------------
+ render(painter: QPainter*): void
+ render(): void
+ resize(w: int, h: int): void
+ initialize(): void
+ setAnimating(animating: bool): void
+ static getInstance(): OpenGLWidget*
+ refreshCallback(): void
+ prepareOpenGL(): void
+ changeMode(nuevo: int): void
+ changeObject(nuevo: int): void
+ setPathShader(path: string): void
+ setUniforms(uniforms: QVector<dataForUniform>): void
+ compile(): void
+ chargeUniforms(): void
+ deleteUniforms(): void
+ mouseMoveEvent(ev: QMouseEvent*): void
+ mousePressEvent(ev: QMouseEvent*): void
+ leaveEvent(ev: QMouseEvent*): void
+ mouseReleaseEvent(e: QMouseEvent*): void
+ wheelEvent(e: QWheelEvent*): void
+ renderLater(): void
+ renderNow(): void
```

Ilustración 20 Clase OpenGLWidget (fuente: elaboración propia)

Esta clase recibe los shaders del editor para aplicarlos a la hora de ejecutar la aplicación, por ello tenemos la clase PagShaderProgram (*Ilustración 21*), la cual es la encargada de compilar los shaders escritos en lenguaje GLSL. De la misma forma, aplica los uniforms que recibe de la tabla para finalmente usarlos en la visualización del modelo.

PagShaderProgram
- gl: QOpenGLFunctions* - handler: GLuint - linked: bool - logString: std::string - showUniformDebugMessages: bool
+ setGLFunctions(gl: QOpenGLContext*): void + createShaderProgram(filename: const string&): GLuint + getHandler(): GLuint + use(): bool + setUniform(name: string, value: GLfloat): bool + setUniform(name: string, value: GLint): bool + setUniform(name: string, value: glm::mat4): bool + setUniform(name: string, value: glm::vec3): bool + setUniform(name: string, value: bool): bool + getShaderError(): string - compileShader(filename: const string&, shaderType: GLenum): GLuint

Ilustración 21 Clase PagShaderProgram (fuente: elaboración propia)

Para poder movernos por la escena son necesarias las matrices de visión y perspectiva, las cuales serán recibidas en el shader program para poder visualizar de manera correcta el modelo. Estas matrices las obtendremos de la cámara de la escena, la cual define una serie de movimientos, así como la posibilidad de hacer zoom, por todo ello también está definida la clase Camera (*Ilustración 22*).

```
Camera
- FoV: float
- cameraPos: glm::vec3
- cameraLookAt: glm::vec3
- cameraUp: glm::vec3
- n: glm::vec3
- u: glm::vec3
- v: glm::vec3
- angleOrbitX: float
- updateCameraAxis(): void
- projectionMatrix: glm::mat4
+ moveFoward(): void
+ moveBackward(): void
+ moveLeft(): void
+ moveRight(): void
+ moveUp(): void
+ moveDown: void
+ tiltAndPan(cantidadX: float, cantidadY: float): void
+ orbitX(cantidad: float): void
+ orbitY(cantidad: float): void
+ zoom(factor: float): void
+ getWorldToProjecMatrix(_width: float, _height: float): glm::mat4
+ getWorldToViewMatrix(): glm::mat4
```

Ilustración 22 Clase Cámara (fuente: elaboración propia)

En la escena se debe visualizar algún modelo. Por ello, también existe la clase Pag3dGroup, la cual define un vector de Pag3DElements que representan a los modelos, que en este caso se tratan de sólidos de revolución, como se puede ver en la *Ilustración 23*.

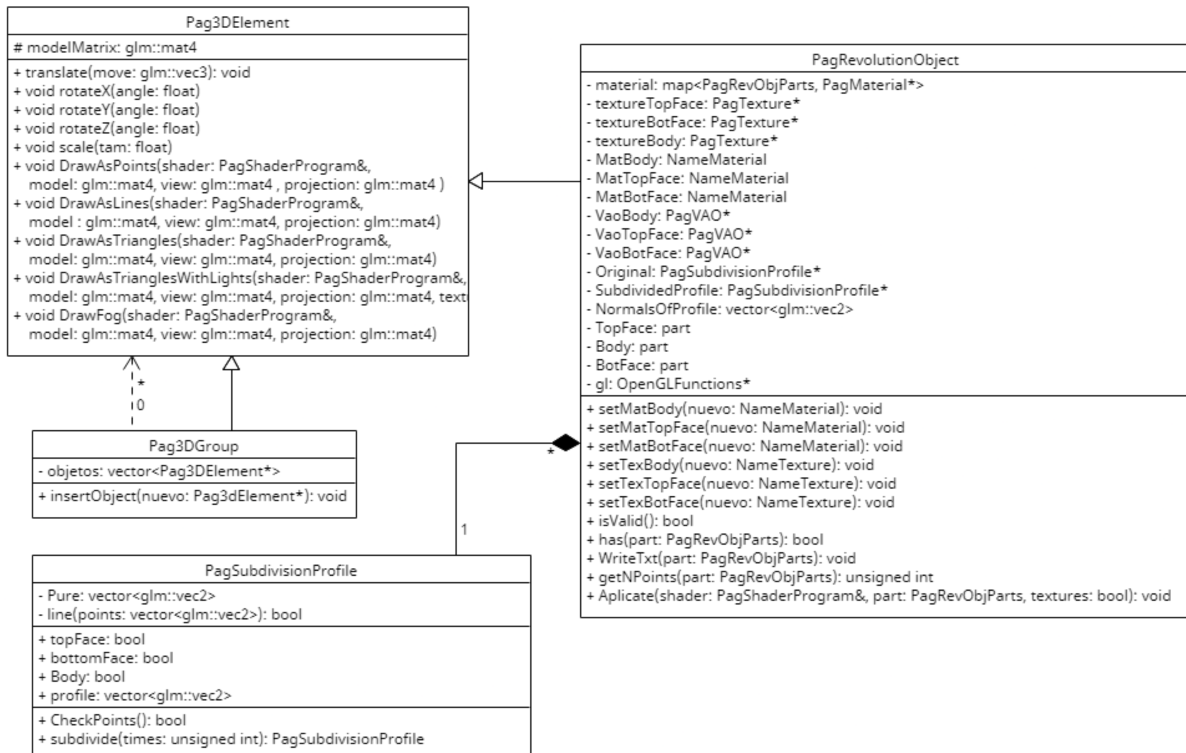


Ilustración 23 Diagrama modelos (fuente: elaboración propia)

Como ya se comentó en la sección 1.3, todos los modelos hay que pasarlos a una estructura de datos que llamamos VBO (vertex buffer object) para que la tarjeta gráfica pueda procesar posiciones, normales, tangentes, coordenadas de textura… e IBO (index buffer object) el cual indica su topología por medio de índices a vértices para generar los triángulos del modelo. Por ello está definida la clase PagVAO (vertex array object), la cual es la encargada de almacenar VBOs e IBOs para procesar los sólidos de revolución y que la tarjeta gráfica pueda trabajar con ellos mediante funciones de OpenGL. Todo ello se puede ver en el diagrama de clases de la *Ilustración 24*.

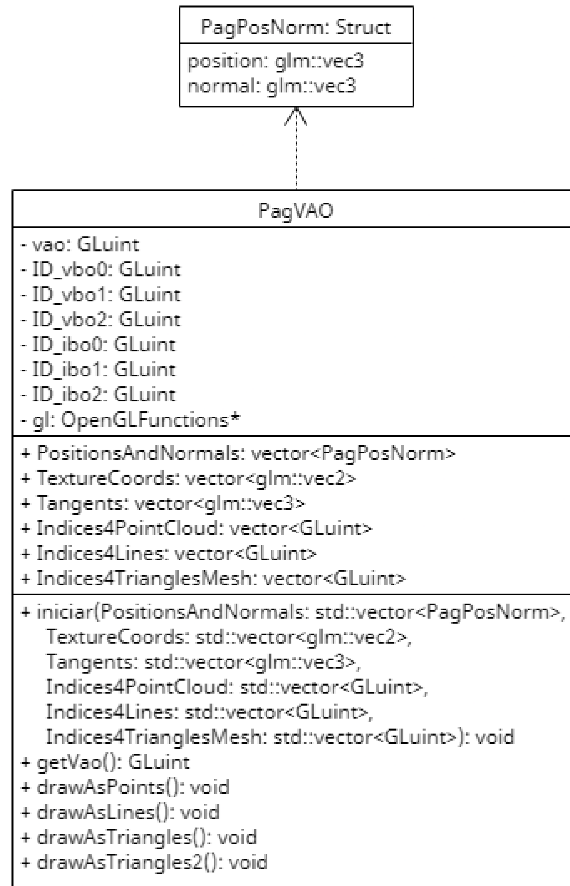


Ilustración 24 Clase VAO (fuente: elaboración propia)

También hay definidas una serie de luces, las cuales no dejan de ser uniforms que se aplican al shader program para poder usarlos en los shaders, como se puede observar en el diagrama de clases de la *Ilustración 25*.

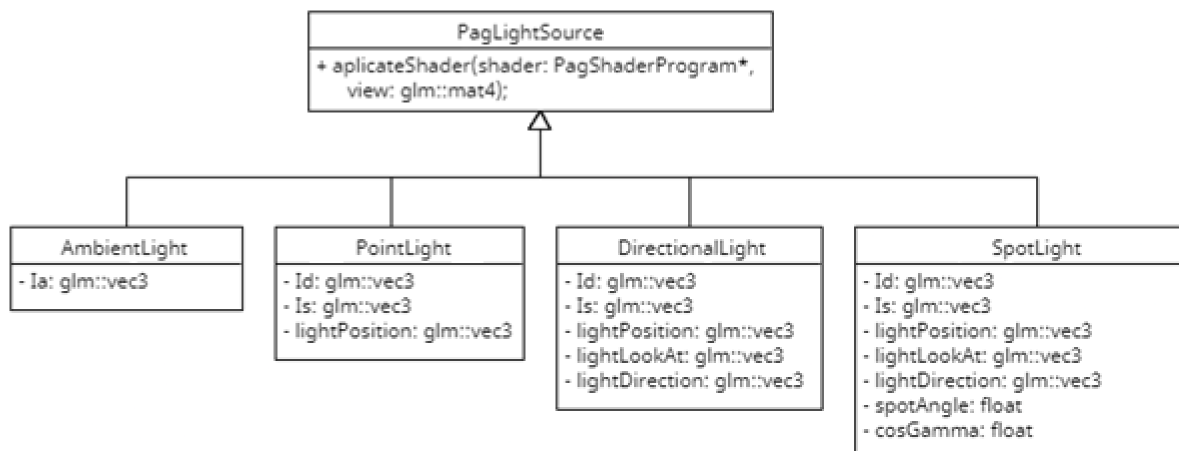


Ilustración 25 Diagrama Luces (fuente: elaboración propia)

La aplicación también cuenta con una serie de clases para gestionar las texturas y materiales de los modelos, donde las clases TextureLibrary y MaterialLibrary implementan el patrón singleton (*Ilustración 26*).

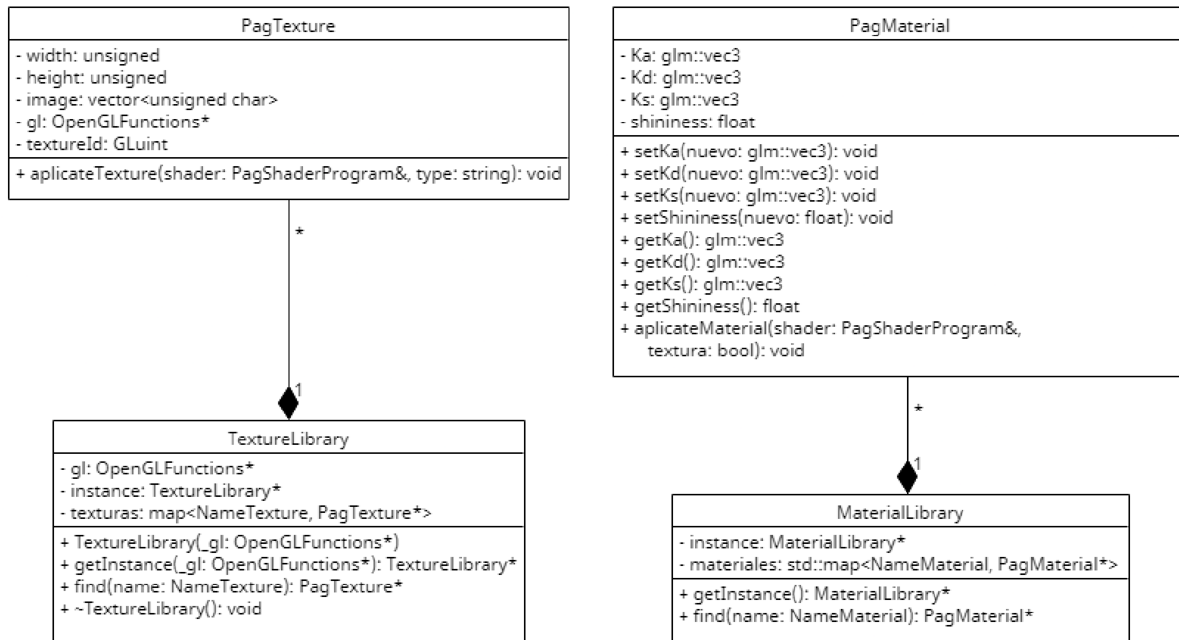


Ilustración 26 Diagrama Texturas y Materiales (fuente: elaboración propia)

Finalmente, se ha usado Qt para implementar la interfaz gráfica con diferentes elementos como son los botones, la consola y el Widget OpenGL, así como la creación de la tabla de uniforms.

1.4.2 Resumen de las deficiencias y carencias identificadas

Una vez expuesta la arquitectura de la aplicación, y habiendo probado la misma, he identificado las siguientes deficiencias:

- Modelos predefinidos sin la opción de poder elegir nuevos modelos.
- Falta de abstracción en clases como PagVAO, la cual está orientada a cargar un sólido de revolución en lugar de cualquier modelo.
- Falta de un espacio de nombres para estructurar el código y evitar posibles colisiones de nombres entre bibliotecas.
- Está implementada la opción de visualizar un conjunto de modelos, pero solo se carga uno a la vez.

- Modos de visualización poco intuitivos, además de creación innecesaria de topologías de nube de puntos, ejes y triángulos, ya que OpenGL nos permite definir el modo de visualización. De igual forma, define uniforms para el modo de texturas y material, haciéndolo poco flexible.
- Al ejecutar los shaders, el modelo no sale centrado en el canvas OpenGL. Este se posiciona si redimensionas la pantalla, y además no tiene límite de redimensión, lo cual hace que se pueda redimensionar hasta que el canvas desaparece.
- Uso innecesario del patrón singleton para materiales, ya que se pueden definir como uniforms y texturas, que no aparecen en ningún lugar de la aplicación.
- Al hacer clic en la barra de herramientas de la interfaz aparece un menú vacío, y si hacemos clic en él se oculta la barra de herramientas, haciendo difícil volver a habilitarla.
- Copia de la clase OpenGLWidget de Qt, pero se podría haber usado una simple herencia y utilizarla en una clase Renderer, por ejemplo.
- Falta de comentarios en varias clases, lo cual hace difícil su comprensión.
- Clases muy dispersas de enumerados y constantes.
- Abstracción de luces, las cuales también se pueden definir como uniforms.
- La clase para resaltar colores en palabras reservadas tiene un pequeño error que resalta clases de Qt cuando la sintaxis es GLSL.
- Definición de varios movimientos de cámara cuando los únicos interesantes y que usa son el movimiento de orbitar alrededor del modelo y la posibilidad de hacer zoom para la aplicación.

1.5 Requisitos iniciales

Estos son los requisitos principales de la aplicación para prueba de shaders escritos en GLSL:

1. Debe poder importar modelos como mínimo en formato .obj.
2. Debe permitir la posibilidad de escribir geometry shaders.

3. Debe visualizar de manera correcta el modelo, siempre y cuando sea topológicamente correcto.
4. Debe poder indicar qué subrutina se va a usar en un momento dado.
5. Debe poder definir los uniforms correspondientes.
6. Debería tener una experiencia de usuario mejorada respecto a la aplicación original.
7. Debería poder definir sólidos de revolución de manera sencilla.
8. Debe poder cargar texturas para usarlas en los modelos.
9. Debería poder ser ejecutada en sistemas operativos como Windows y Linux como mínimo.
10. Debería remarcar errores sobre la sintaxis GLSL.

1.6 Alcance

El alcance del trabajo incluirá:

- **ToulouseCompiler 2.0:** Versión ampliada y mejorada de la aplicación, la cual se podrá ejecutar y usar para la creación de shaders en tiempo real. Se entregará tanto el ejecutable como el código fuente.
- **Documentación** elaborada durante todo el proyecto, que contiene cómo éste se ha desarrollado en sus diferentes etapas.
- **Manual de usuario,** en el cual se expondrá el funcionamiento de cada una de las partes de la aplicación, así como la navegación en el renderer.
- **Vídeo demostrativo** para ver su funcionamiento.

1.7 Hipótesis y restricciones

El TFG se define como una asignatura de 12 créditos, lo que supone que la duración total del proyecto será de **300 horas**, incluyendo todas las etapas del ciclo de vida, con la excepción del mantenimiento. Por consiguiente, la principal restricción aplicable es la limitación de la **duración del trabajo**.

1.8 Estudio de alternativas y viabilidad

Lo primero que he estudiado es la refactorización del código, ya que el código existente está muy acoplado, con falta de comentarios y poco definido para mis objetivos, lo que me supondrá una mayor dificultad y gasto de tiempo del cual estoy restringido. Otra opción es rehacer el proyecto basándome en la versión anterior conservando aspectos como puede ser la interfaz, y de esta forma tener un mayor control sobre el proyecto e intentar realizar una mayor abstracción para versiones futuras y por ende agilizar el proceso de implementación.

En cuanto al IDE, el proyecto original se realizó en Visual Studio Community 2019, pero puesto que en la asignatura "Programación de Aplicaciones Gráficas" ya habíamos usado CLion junto al gestor de paquetes Conan, el cual agiliza el proceso para importar bibliotecas, he decidido realizarlo en él y puesto que es multiplataforma al igual que Qt puedo generar los ejecutables de manera sencilla.

Para importar modelos tengo varias opciones, como incluir algún importador de archivos .OBJ o incluir Assimp, que es muy utilizada en la industria por la variedad de archivos soportados como son Autodesk (.FBX), Wavefront Object (.OBJ) … por lo que de esta forma habrá una gran variedad de formatos con los que añadir modelos.

Otro punto es la creación del formato para el sólido de revolución. Una opción es conservar el código que se utiliza para crear los modelos que hay predefinidos en la aplicación, el cual solo toma una serie de puntos que representa el perfil 2D que se va a revolucionar. Por otro lado, puedo volver a implementarlo para añadirle características como subdivisiones horizontales y verticales, así como vértices no afectados en las subdivisiones verticales, puesto que esta característica no la tiene en cuenta el código original y es fija.

Finalmente, en cuanto a la interfaz va a haber varios cambios, puesto que es necesaria la importación de modelos, texturas, añadir subrutinas… y para que sea algo más intuitiva se eliminará la consola de mensajes, siendo sustituida por algún otro elemento.

1.9 Descripción de la solución propuesta

Expuestas las alternativas, al final voy a optar por rehacer el proyecto para tener más control de este y una mayor abstracción, permitiéndome reducir el tiempo de implementación y posibles problemas que me podrían haber surgido reutilizando lo existente, incluyendo comentarios para una mejor comprensión.

Voy a realizarlo en IDE CLion ya que es un IDE que me ha sorprendido y en el que más cómodo me siento.

Finalmente voy a usar la biblioteca Assimp para poder usar una gran variedad de formatos a la hora de cargar los modelos en la aplicación.

En cuanto al formato de sólidos de revolución, he optado por reimplementarlo, puesto que de esta forma me permitirá abstraer tanto la creación de modelos vía Assimp como la importación de modelos vía sólido de revolución, además de añadir características ya mencionadas como divisiones horizontales y verticales, así como vértices invariables.

1.10 Tecnologías utilizadas

En esta sección vamos a realizar un recorrido por todas las tecnologías que se utilizarán para la realización del proyecto:

- **C++:** Es un lenguaje de programación de alto nivel y multiparadigma el cual combina la programación orientada a objetos con características de programación procedural. Es ampliamente utilizado debido a su eficiencia, flexibilidad y potencia.
- **CLion:** Entorno de desarrollo integrado (IDE) creado específicamente para el lenguaje de programación C++ que proporciona herramientas avanzadas de edición de código, depuración e integración con sistemas de control de versiones, todo dentro de una interfaz de usuario intuitiva.
- **OpenGL (Open Graphics Library):** Especificación estándar de una API (Interfaz de Programación de Aplicaciones) de gráficos en 2D y 3D. Proporciona una interfaz de programación unificada para interactuar con

hardware gráfico, por lo que permite crear aplicaciones gráficas de alto rendimiento en una variedad de plataformas.

- **GLM (OpenGL Mathematics):** Biblioteca matemática de código abierto diseñada específicamente para trabajar con OpenGL y gráficos en 3D en general. Proporciona funciones y clases para realizar operaciones matemáticas comunes utilizadas en el desarrollo de aplicaciones gráficas, como transformaciones geométricas, álgebra lineal, manejo de vectores y matrices, y operaciones relacionadas con gráficos en tres dimensiones.
- **Assimp (Open Asset Import Library):** Biblioteca de código abierto desarrollada en C++ que permite cargar, procesar y gestionar varios formatos de archivo de modelos 3D y escenas. Proporciona una interfaz unificada para acceder a una variedad de formatos de archivo comunes utilizados en gráficos 3D, como OBJ, FBX, COLLADA, STL…
- **Qt6:** Conjunto de herramientas de desarrollo de software que permite crear aplicaciones con interfaces gráficas de usuario (GUI) y no-GUI de manera eficiente y multiplataforma. Ofrece soporte para una amplia gama de plataformas, incluyendo escritorio, móvil y embebido, y proporciona herramientas como Qt Widgets, Qt Quick y QML para crear interfaces de usuario modernas y dinámicas. Es utilizado en una gran variedad de campos, desde aplicaciones de escritorio hasta aplicaciones móviles, y ofrece un conjunto de bibliotecas y herramientas de desarrollo para facilitar el proceso de desarrollo de aplicaciones.
- **Conan:** Plataforma que proporciona una solución para la gestión de dependencias de software en C++ y otros lenguajes de programación. Conan permite a los desarrolladores de software administrar y distribuir bibliotecas y paquetes de código de manera eficiente, lo que facilita la construcción y el mantenimiento de proyectos en entornos de desarrollo en C++ y otros lenguajes compatibles.

1.11 Metodología de desarrollo de software

La metodología de desarrollo en cascada es un enfoque tradicional para el desarrollo de software que sigue un proceso lineal y estructurado, dividido en fases

bien definidas (Pressman, 2010). Cada fase debe completarse antes de pasar a la siguiente, lo que significa que el proceso se desarrolla de manera secuencial y progresiva, como una cascada que fluye de una etapa a la siguiente. Sus fases son:

- **Análisis:** En esta fase inicial, se identifican cuáles son las necesidades y objetivos del sistema, es decir, las funcionalidades y características que debe tener el software para satisfacer las necesidades del cliente o usuario final. También se realiza un estudio de viabilidad donde se evalúan costes, la rentabilidad y la factibilidad del proyecto software.
- **Diseño:** Una vez que se han establecido los requisitos, se procede a diseñar la arquitectura del sistema y la estructura general del software. Esto incluye definir las estructuras de datos, la interfaz de usuario, y cualquier otro aspecto relacionado con la implementación del software.
- **Implementación (Codificación):** En esta etapa, los programadores comienzan a escribir el código del software utilizando el diseño establecido anteriormente. Se trata de traducir los requisitos y el diseño a código ejecutable.
- **Pruebas:** Una vez que se ha completado la implementación del software, se procede a realizar pruebas para asegurarse de que el software funciona según lo previsto y cumple con los requisitos establecidos. Estas pruebas pueden incluir pruebas de unidad, pruebas de integración y pruebas de sistema.
- **Despliegue (o implantación):** Una vez que el software ha pasado con éxito las pruebas, se implanta en el entorno de producción para su uso por parte de los usuarios finales.
- **Mantenimiento:** Finalmente, después del despliegue, se lleva a cabo el mantenimiento del software para corregir errores, realizar actualizaciones y añadir mejoras según sea necesario a lo largo de su ciclo de vida. Esta etapa no se tendrá en cuenta en el TFG.

He optado por usar esta metodología de desarrollo software puesto que habiendo analizado la situación de partida e identificado las carencias y deficiencias, tengo ya un enfoque de lo que voy a realizar e implementar, y por lo tanto puedo exponer de manera clara y concisa qué se va a realizar en cada etapa y de qué manera.

1.12 Estimación del tamaño y esfuerzo

Ya que el presente proyecto es un TFG, no existen restricciones de tipo económico, sino de tipo temporal (un número aproximado de horas). Por consiguiente, los cálculos de tamaño del proyecto están supeditados al tiempo disponible. En cuanto al esfuerzo, se dispone de tan un solo efectivo (la persona autora del trabajo).

Vamos a obtener una tabla con el número estimado de líneas de código para cada requisito inicial (ver *Tabla 1*) y a partir de ello realizaremos el modelo constructivo de costos o COCOMO para estimar el tamaño y esfuerzo del proyecto.

Requisito inicial	Líneas de código (LOC)
Importación de modelos (.OBJ)	400 LOC
Integración de geometry shaders	200 LOC
Visualización y refactorización	600 LOC
Implementación subrutinas	300 LOC
Gestión de uniforms	400 LOC
Mejoras de interfaz	200 LOC
Formato sólido de revolución	500 LOC
Carga de texturas	200 LOC
Gestión de errores	200 LOC
TOTAL	**3000 LOC**

Tabla 1 Estimación de líneas de código

Una vez contamos con las líneas de código aproximadas que se van a escribir en la realización el proyecto, podemos proceder a realizar el modelo constructivo de costo (COCOMO 81 (Adriana Gómez & Silvina Migani, 2000)).

Voy a emplear el modelo **básico** y las fórmulas son:

$$\boldsymbol{PM} = A * (KSLOC)^B \qquad \textbf{TDEV} = \mathrm{C} \times (PM)^D$$

Donde:

- **PM** es el esfuerzo estimado. Representa los meses-persona necesarios para ejecutar el proyecto.

- **KSLOC** es el tamaño del software a desarrollar en miles de líneas de código.
- **TDEV** representa los meses de trabajo que se necesitan para ejecutar el proyecto.
- **A**, **B, C** y **D** son coeficientes que varían según el modo de desarrollo según sea Orgánico, Semiacoplado o Empotrado (ver *Tabla 2*).

Modo de desarrollo	Esfuerzo	Cronograma
Orgánico	PM=2,4 x $(KSLOC)^{1.05}$	TDEV=2,5 x $(PM)^{0.38}$
Semiacoplado	PM=3,0 x $(KSLOC)^{1.12}$	TDEV=2,5 x $(PM)^{0.35}$
Empotrado	PM=3,6 x $(KSLOC)^{1.20}$	TDEV=2,5 x $(PM)^{0.32}$

Tabla 2 Ecuaciones del modelo básico de COCOMO

Al ser un proyecto pequeño donde el entorno es conocido, flexible y en el cual trabaja una sola persona, vamos a utilizar las fórmulas del modo de desarrollo orgánico.

Lo primero que vamos a realizar es una conversión de líneas de código a miles de líneas de código realizando una simple división entre mil:

$$\boldsymbol{KSLOC} = \frac{3000}{1000} = 3\ KSLOC$$

Para continuar, calculamos el esfuerzo estimado:

$$\boldsymbol{PM} = 2{,}4 * 3^{1.05} \approx 7{,}61 \text{ meses-persona}$$

Finalmente calculamos los meses de trabajo aproximados del proyecto.

$$\boldsymbol{TDEV} = 2{,}5 * 7{,}61^{0.38} \approx 5{,}4 \text{ meses de trabajo}$$

1.13 Planificación temporal

En este apartado se va a mostrar la planificación temporal del proyecto mediante un diagrama de Gantt realizado en la aplicación **GanttProject**[1]**,** en el que se muestra la duración de cada etapa del modelo en cascada, añadiendo también una etapa de estudio preliminar donde se analiza la aplicación de partida y se realiza un estudio de alternativas y viabilidad. En la etapa de desarrollo, además, se hace un desglose de la duración de cada requisito.

El proyecto ha tenido una duración de 160 días, donde se incluyen festivos y algunos días donde no se ha podido avanzar en él debido a situaciones adversas y horarios, lo cual hace un promedio aproximado de dos horas diarias, puesto que el TFG tiene un máximo de 300 horas.

Todo lo comentado anteriormente se puede ver en el diagrama de Gantt que se encuentra en la *Ilustración 27.*

[1] GanttProject: https://www.ganttproject.biz/

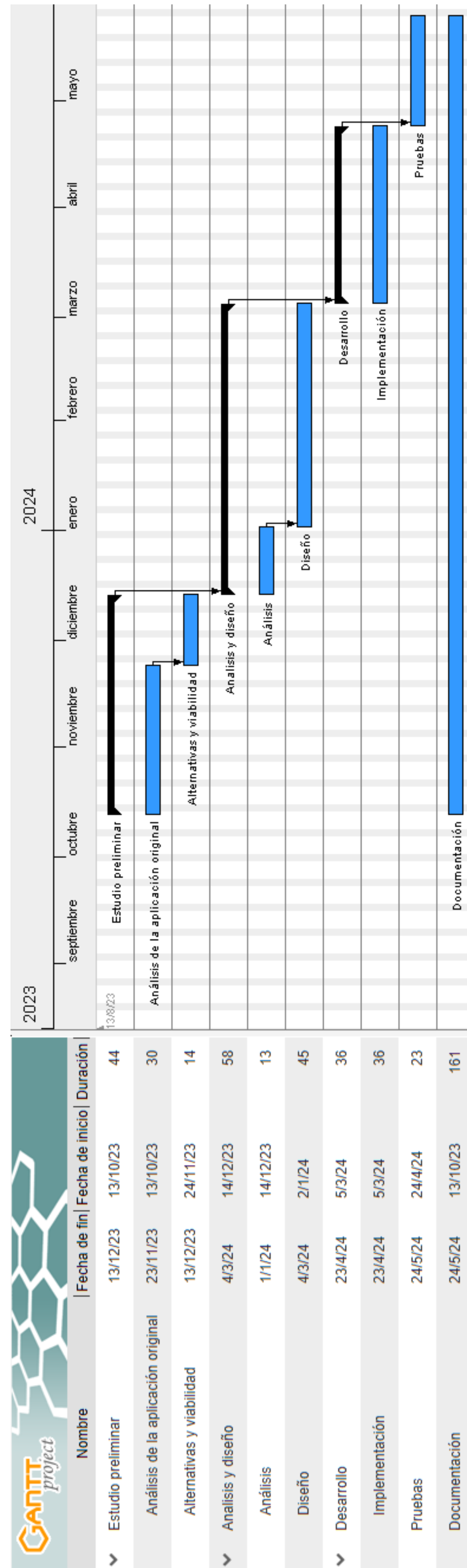


Ilustración 27 Diagrama de Gantt (fuente: elaboración propia)

1.14 Presupuesto

Para la realización del presupuesto, vamos a realizar un estudio de los recursos necesarios para la realización del proyecto, así como sus costes asociados. Finalmente, se realizará un presupuesto final.

Los recursos los desglosaremos en:

- Recursos humanos.
- Recursos hardware.
- Recursos software.
- Otros recursos.

Cabe destacar que todos estos recursos están ligados directamente con la planificación temporal, ya que el coste de cada recurso es directamente proporcional al uso de este.

Recursos humanos

Vamos a abordar los diferentes perfiles que van a intervenir a la hora de la realización del proyecto. Estos perfiles están recogidos en la *Tabla 3*.

<table>
<tr><th>Perfil</th><th>Tarea</th><th>Horas</th></tr>
<tr><td>Jefe de proyectos</td><td>Redacción de la memoria</td><td>80</td></tr>
<tr><td rowspan="4">Analista</td><td>Búsqueda y revisión bibliográfica</td><td>10</td></tr>
<tr><td>Análisis de requisitos</td><td>20</td></tr>
<tr><td>Análisis de usuario</td><td>20</td></tr>
<tr><td>Análisis de Arquitectura</td><td>50</td></tr>
<tr><td rowspan="2">Analista-Programador</td><td>Diseño de casos de uso</td><td>12</td></tr>
<tr><td>Diseño de interfaz</td><td>24</td></tr>
<tr><td rowspan="3">Programador</td><td>Implementación</td><td>56</td></tr>
<tr><td>Manual de usuario</td><td>8</td></tr>
<tr><td>Pruebas de aceptación</td><td>20</td></tr>
</table>

Tabla 3 Perfiles para la realización del proyecto

Una vez recogidos todos los perfiles necesarios para la realización del proyecto, vamos a elaborar la tabla de costes asociados a las horas de trabajo para cada perfil (*Tabla 4*). Los sueldos han sido obtenidos del **BOE**[2]**,** donde todos los perfiles pertenecen al "Área 3: Desarrollo de software, Programación y Explotación de Sistemas". El perfil jefe de proyectos pertenece al grupo A, el perfil de analista al grupo B nivel 1, el perfil de analista-programador al grupo B nivel 2 y el perfil de programador al grupo C nivel 2. Consideramos que se trabajan 1736 horas anuales.

Perfil	Duración	Precio por hora	Total
Jefe de proyectos	80 horas	16,62 euros/hora	**1329,6 €**
Analista	100 horas	16,12 euros/hora	**1612 €**
Analista-Programador	36 horas	15,64 euros/hora	**563,04 €**
Programador	84 horas	14,29 euros/hora	**1200,36 €**
		TOTAL	**4705 €**

Tabla 4 Tabla de costos por perfil

Recursos Hardware

Puesto que es solo una persona la que va a realizar todos los perfiles descritos anteriormente, solo será necesario un ordenador para el desarrollo de todo el proyecto. En la *Tabla 5* se muestran desglosados los recursos hardware necesarios. Para este proyecto solo se empleará un ordenador personal, que hoy en día tiene una amortización de 3 a 5 años. Vamos a considerar 5 años de amortización, y 700 horas de uso al año.

Recurso hardware	Cantidad	Precio unitario	Amortización	Tiempo de uso	Total amortizado
Ordenador personal	1	1755,67€	5 años (3500 horas)	300 horas	**150,49 €**

Tabla 5 Tabla de costos de recursos hardware

[2] https://www.boe.es/eli/es/res/2023/07/13/(5). XVIII Convenio colectivo estatal de empresas de consultoría, tecnologías de la información y estudios de mercado y de la opinión pública.

Recursos software

En esta sección se van a recoger todos los recursos software con costo que van a intervenir a la hora de la realización del proyecto. En la *Tabla 6* se muestra todo el software desglosado.

Recurso software	Cantidad	Precio unitario	Amortización	Tiempo de uso	Total amortizado
SO Windows 10 Pro	1	200 €	3 años (2100 horas)	300 horas	**28,57 €**
CLion	1	229 €	1 año (560 horas)	76 horas	**31,08 €**
Microsoft Office 2019	1	233,71 €	3 años (420 horas)	80 horas	**44,52 €**
Pencil	1	-	-	-	**0 €**
GanttProject	1	-	-	-	**0 €**
Qt6	1	-	-	-	**0 €**
				Total	**104,17 €**

Tabla 6 Tabla de costos de recursos software

Otros costos

En este apartado veremos otros costes asociados, como puede ser el coste de la luz, el coste del uso del ordenador, servicio de Internet… Todo ello se ve reflejado en la *Tabla 7*.

Descripción	Cantidad	Precio unitario	Total
Consumo luz	300 h	0,1199 kW/hora	**35,97€**
Consumo de Internet	5,4 meses	30,90 euros/mes	**166,86 €**
		Total	**202,83 €**

Tabla 7 Tabla de costos adicionales

- **Presupuesto**

Teniendo en cuenta los costes descritos anteriormente, aplicando un beneficio del 30% y el sobrecargo del 21% de I.V.A, en la *Tabla 8* se muestra el presupuesto final.

Descripción	Coste
Recursos humanos	4705 €
Recursos hardware	150,49 €
Recursos software	104,17 €
Otros costes	202,83 €
Total costes	**5162,49 €**
Beneficios (30%)	1548,75 €
Total sin I.V.A	**6711,24 €**
I.V.A (21%)	1409,36 €
Total	**8120,6 €**

Tabla 8 Presupuesto final

2 DISEÑO INICIAL

El este apartado vamos a realizar el análisis de las especificaciones del sistema, es decir, requisitos funcionales, no funcionales y casos de uso del mismo. Finalmente realizaremos el diseño arquitectónico del sistema, así como el diseño de las interfaces de la aplicación.

2.1 Especificaciones del sistema

En este apartado se recogerán los requisitos funcionales y no funcionales de la aplicación, así como sus casos de uso.

Requisitos funcionales:

- El usuario debe poder escribir vertex, fragment y geometry shaders en lenguaje GLSL.

- El usuario debería poder importar modelos al menos en formato .OBJ
- El modelo debería poder visualizarse como nube de puntos, líneas o triángulos.
- Se debe poder elegir una subrutina en un momento dado.
- El usuario debe poder cargar texturas.
- Se debe poder especificar uniforms para su uso.
- Se debe visualizar el modelo siempre y cuando éste y los shaders sean correctos.
- Debe poder cargar modelos ya creados y guardarlos cuando quiera el usuario.
- La aplicación debe dar información sobre los errores que ésta presente.
- El usuario debe poder navegar de manera cómoda por la escena para visualizar el modelo completo.
- La interfaz de usuario debe aportar información sobre cada uno de los elementos y atajos de teclado para una mayor fluidez.
- La aplicación debe permitir importar sólidos de revolución en un formato sencillo de entender.

Requisitos no funcionales:

- La aplicación debe ser estable y no colapsar durante el uso normal.
- La aplicación debe ser capaz de renderizar modelos y aplicar shaders en tiempo real, con una tasa de cuadros mínima de 60 FPS.
- La interfaz debe ser intuitiva y fácil de usar, permitiendo a los usuarios cargar modelos y shaders, ajustar parámetros y visualizar resultados sin dificultad.
- Se debe proporcionar retroalimentación clara sobre las acciones realizadas y el estado de la aplicación.
- El código debe estar bien documentado y estructurado para facilitar su mantenimiento y futuras actualizaciones.

- Debe soportar una amplia gama de tarjetas gráficas, tanto integradas como dedicadas.
- Debe poder ser distribuida y ejecutada sin configuraciones complejas, idealmente como un ejecutable independiente o a través de un instalador simple.

Casos de uso:

En el diagrama de la *Ilustración 28* se muestran los casos de uso identificados en la aplicación.

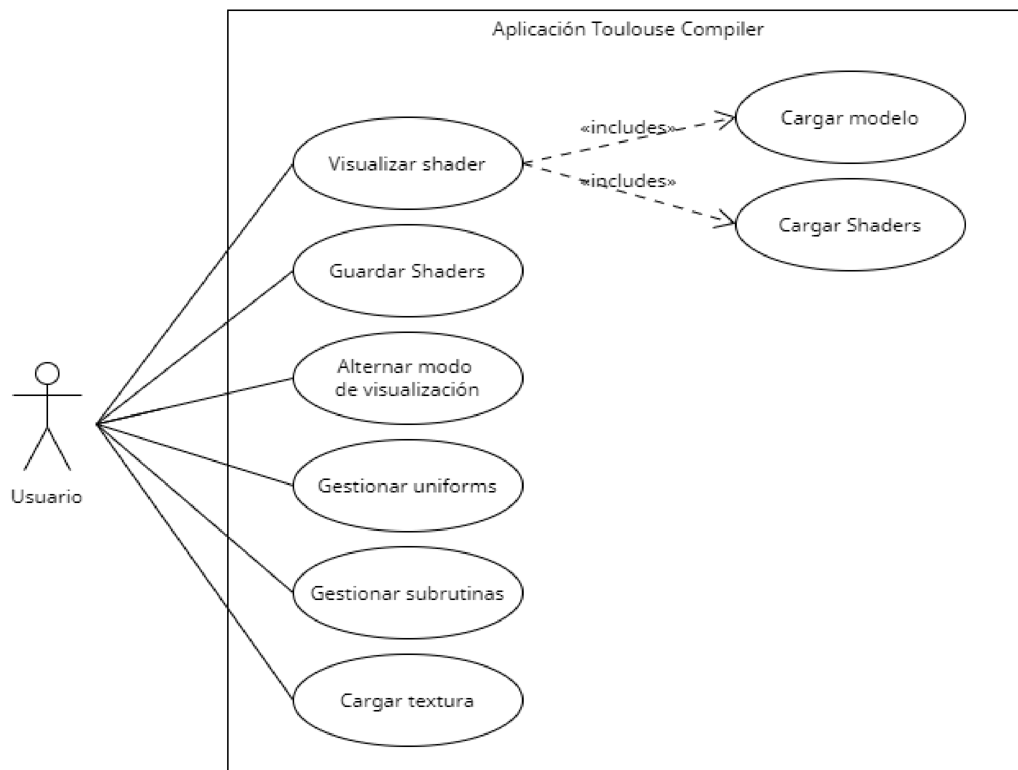


Ilustración 28 Diagrama de casos de uso (fuente: elaboración propia)

2.2 Diseño del sistema

Vamos a analizar las distintas opciones que ha de tener la aplicación desde el punto de vista del usuario para que cumpla los requisitos del sistema:

- El usuario quiere cargar un modelo, por lo que lo hace seleccionando la opción “cargar modelo”, navega por el sistema de archivos y selecciona el modelo que desea cargar.

- Para visualizar el modelo es necesario que se le aplique un shader. Para ello, el usuario escribe shaders directamente desde el editor o bien los carga con la opción correspondiente, buscándolos en el sistema de archivos.
- El usuario ha especificado algunos uniforms en los shaders, por lo que requiere de algún mecanismo para asignarles un valor a los mismos y poder usarlos para realizar algún tipo de cálculo.
- El usuario ha hecho uso de subrutinas en el shader, y por ello necesita algún mecanismo para especificar el nombre de la subrutina, el shader en el que se encuentra y la implementación que quiere usar.
- El usuario quiere aplicarle una textura al modelo, por lo que requerirá de algún mecanismo para cargar una imagen en el shader y usarla para aplicarla al modelo.
- El usuario quiere comprobar la topología del modelo y quiere alternar entre los distintos modos de visualización (nube de puntos, líneas y triángulos), por lo que será necesario especificar algún mecanismo para ello.
- El usuario quiere ver de forma completa el modelo, y es por ello que será necesario definir algún mecanismo para navegar en la escena.
- Una vez el usuario ha conseguido el resultado esperado, guarda el shader con la opción de "guardar", especificando la ruta y el nombre con el que desea guardarlo.

Una vez enumeradas las distintas opciones que ha de tener la aplicación podemos identificar claramente distintas funcionalidades deben ser implementadas. Estas son:

- Funcionalidad para cargar un modelo en formato .OBJ o sólidos de revolución.
- Funcionalidad para cargar un shader desde un archivo y mostrarlo en el editor.
- Funcionalidad para guardar el shader escrito en el editor.
- Funcionalidad para alternar entre los distintos modos de visualización.
- Funcionalidad para gestionar uniforms presentes en el shader.

- Funcionalidad para gestionar subrutinas presentes en el shader.
- Funcionalidad para cargar una textura en algún formato de imagen.
- Mecanismos de control para moverse en la escena.
- Funcionalidad para visualizar el resultado.

2.2.1 Diseño arquitectónico del sistema

Vamos a enfocar el diseño atendiendo a las funcionalidades analizadas en el apartado anterior. Lo dividiremos en diferentes subsecciones, atendiendo a la funcionalidad desempeñada:

Funcionalidad para cargar un modelo

Para la funcionalidad de cargar un modelo se definirá una clase malla, la cual comunicará la geometría a la GPU y será la encargada de gestionar el VAO, cuyo contenido será un VBO entrelazado empleando un array de una estructura Vertex, que almacena información sobre el vértice como posición, normal, coordenada de textura y tangente. Por otro lado, el VAO también almacenará un IBO empleando un array de enteros, que serán índices a vértices del VBO entrelazado. También tendrá un parámetro encargado de indicar el tipo de primitiva, que se utilizará en un método de dibujado para visualizar la topología de manera correcta.

Cabe destacar que OpenGL tiene una serie de funciones que se emplean para reservar recursos, y les asigna un id que podremos almacenar para acceder a ellos como si de una referencia se tratase.

Crearé una clase abstracta que almacenará la matriz de modelado, que define las distintas transformaciones aplicadas a los modelos, como pueden ser translación, rotación, escalado… Además, un modelo puede estar formado por diferentes mallas, y es por eso por lo que almacena las diferentes mallas si las tuviera, que se visualizarán con el método de dibujado correspondiente. Por otro lado, para posicionar la cámara correctamente respecto al modelo, emplearé una clase caja envolvente, que calculará los vértices con mayor y menor valor en los ejes de coordenadas (x, y,

z), para de esta forma posicionar la cámara respecto al centro de la misma cada vez que se cargue un modelo y que no se pierda la referencia.

Para cargar un modelo en formato .OBJ, crearé una clase que herede de la clase modelo, que empleará la biblioteca Assimp (Vries, 2020) para cargar el modelo procesando nodos de una estructura jerárquica (*Ilustración 29*). Al usar esta biblioteca, nos permite cargar un modelo en una variedad de formatos distintos al formato .OBJ, por lo que flexibiliza la carga de modelos.

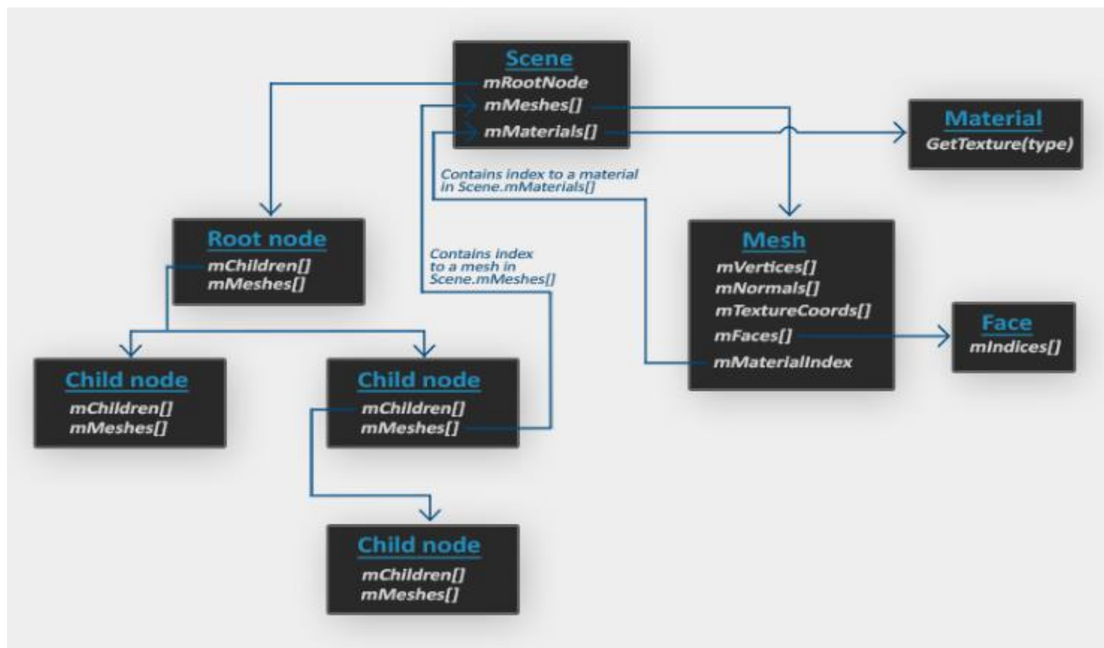


Ilustración 29 Estructura de nodos de Assimp (fuente: https://learnopengl.com/Model-Loading/Assimp)

Por otro lado, un requisito es que la aplicación permita cargar sólidos de revolución de manera sencilla. Los sólidos de revolución se definen como la rotación de una polilínea o un polígono (generatriz) alrededor de un eje (llamado eje de revolución). En la *Ilustración 30* se muestra un ejemplo de cómo se construye.

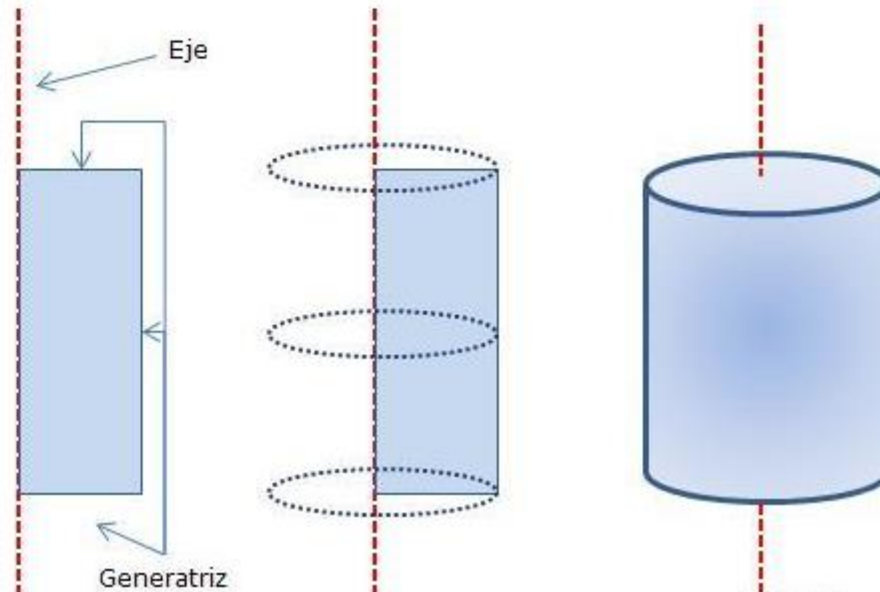


Ilustración 30 Construcción de sólido de revolución (fuente: https://www.aulafacil.com/cursos/matematicas-secundaria-eso/matematicas-segundo-eso-13-anos/cuerpos-de-revolucion-l7712)

Puesto que se tiene que rotar 360º alrededor de un eje, será necesario definir el número de cortes longitudinales, donde el mínimo será tres, debido que es la cantidad mínima necesaria para definir un sólido de revolución con volumen.

El perfil de revolución, como se ha comentado, puede ser un polígono o una polilínea, y los vértices deberán tener una abscisa mayor o igual a 0 para que no se produzcan fallos en la visualización del sólido. Por otra parte, si el perfil es una polilínea, habrá que tener en cuenta el primer y último vértice, puesto que, si están en el eje de revolución, tendremos tantos vértices como cortes longitudinales en la misma posición y esto puede crear posibles problemas a la hora de visualizar el sólido. Para solucionar esto, se deberá aplicar otro tipo de topología en el primer y último vértice.

Adicionalmente, para crear sólidos de revolución con bordes suavizados, se puede aplicar un algoritmo de subdivisión aplicado a los vértices del perfil 2D. El algoritmo que voy a emplear es el cálculo de promedios basado en iteraciones, donde por cada conjunto de tres vértices adyacentes de la polilínea:

- El primer y último vértice se conserva en la polilínea subdividida.
- Se crean vértices en los puntos medios de los segmentos.

- Finalmente se desplaza el vértice central a una posición promedio de los nuevos vértices.

En la *Ilustración 31* se puede ver un boceto del resultado del algoritmo de subdivisión de polilíneas basado en promedios.

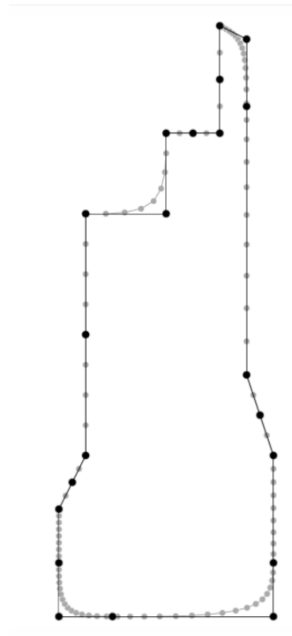


Ilustración 31 Algoritmo de subdivisión de polilíneas (fuente: Asignatura Programación de Aplicaciones Gráficas)

Otro aspecto a tener en cuenta es permitir de alguna forma que un vértice no sea suavizado, para de esta forma tener mayor control sobre el algoritmo de subdivisión.

Para la carga de sólidos de revolución he pensado en definir un formato de texto con extensión .SOR (Solid Of Revolution), el cual se escribirá línea a línea y estará formado por un identificador asignando un valor de la siguiente forma:

- Un identificador para las divisiones longitudinales, el cual se llamará “slices”, al que se asignará un valor entero que no podrá ser inferior a tres para generar volumen.
- Un identificador para el número de iteraciones en el algoritmo de subdivisión de polilíneas, el cual se llamará “subdivisions”, al que se puede asignar un valor de tipo entero mayor o igual a cero.

- Un identificador para indicar la posición de un vértice que se llamará "v", al que se le asignará un valor en formato "coordenada x, coordenada y".
- Un identificador para indicar que un vértice no se modifique a la hora de aplicar el algoritmo de subdivisión de polilíneas, llamado "sv", y que tendrá de la misma forma un valor en formato "coordenada x, coordenada y".

En la *Ilustración 32* se puede ver un ejemplo del formato creado para definir un sólido de revolución, el cual aplicaría dos iteraciones en el algoritmo de subdivisión de polilíneas, realizaría treinta y seis cortes longitudinales, y tiene diecinueve vértices, de los cuales tres no son afectados por el algoritmo.

```
subdivisions=2
slices=36
sv=2.0,4.0
sv=2.0,0.0
v=2.0,-4.0
v=1.5,-5.0
sv=1.0,-5.5
v=1.0,-7.0
v=1.0,-9.0
v=3.0,-9.0
v=10.0,-9.0
v=10.0,-7.0
v=10.0,-4.0
v=9.0,-2.5
v=8.0,-1.0
v=8.0,6.5
v=8.0,8.5
v=7.0,9.0
v=7.0,6.0
v=5.0,6.0
v=5.0,4.0
```

Ilustración 32 Formato de sólido de revolución .SOR (fuente: elaboración propia)

Durante la carga del modelo me encargaré de controlar los posibles casos de error para que el usuario pueda saber en todo momento lo que ocurre.

La carga del modelo se realizará mediante un botón en la interfaz, el cual abrirá el explorador de archivos, permitiendo al usuario navegar por el sistema de almacenamiento hasta encontrar el modelo que quiera cargar. Una vez cargado, se mostrará el estado del modelo, indicando si se ha cargado correctamente o si se ha producido un error en una barra de estado de texto.

En el diagrama de la (*Ilustración 33*) se puede ver la arquitectura para la carga de un modelo 3D en la aplicación.

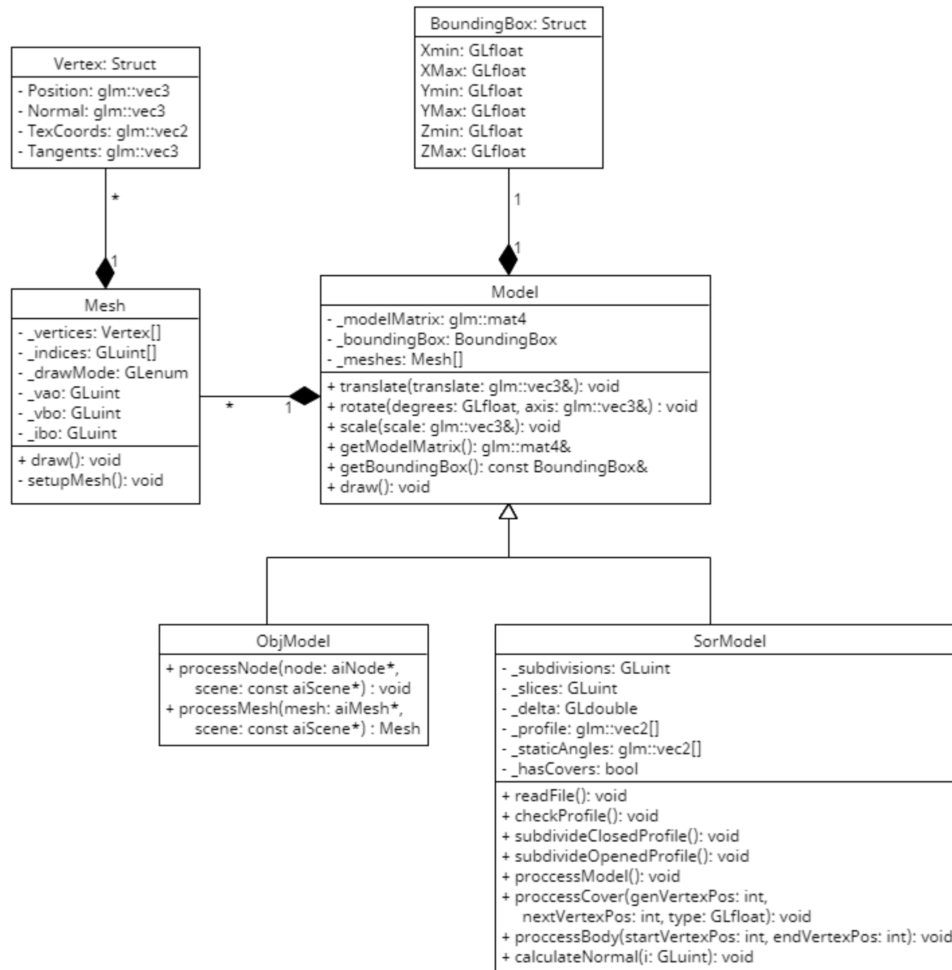


Ilustración 33 Diagrama UML para la funcionalidad de cargar un modelo en formato .OBJ o .SOR (fuente: elaboración propia)

Funcionalidad para cargar y guardar un shader

Para la funcionalidad de cargar y guardar un shader, definiré una clase Shader para compilar un shader. Esta clase tendrá un enumerado que definirá el tipo, además del código fuente a compilar. De igual forma, definiré la clase shader program, que compilará un shader de cada tipo, es decir, un vertex shader, un geometry shader y un fragment shader. Durante la carga controlaré todos los posibles errores que se puedan dar a la hora de compilar y enlazar cada uno de los shaders, para que el usuario sea capaz de encontrar el error.

Debido a que todos los shaders se cargarán desde archivos de texto que tendrán la extensión .GLSL, los cuales contendrán el código fuente de los mismos, tendré que implementar algún mecanismo de diferenciación de estos. Este mecanismo estará basado en el nombre del shader, donde el vertex shader deberá tener el formato “nombreVertexShader-vs.glsl”, el geometry shader “nombreGeometryShader-gs.glsl” y el fragment shader “nombreFragmentShader-fs.glsl”. De esta forma, seré capaz de diferenciar cada uno de los shaders y ubicarlos en el editor correspondiente para que se compilen y ejecuten en el shader program. Debido a que los únicos indispensables para la visualización, como se ha comentado anteriormente, son el vertex shader y fragment shader, el geometry shader será opcional, y no se compilará si su editor de código está completamente vacío.

Puesto que son varios tipos de shaders, se permitirá la selección múltiple de archivos para cargarlos de manera directa. El guardado se realizará de manera similar, donde el usuario podrá pulsar un botón, y a través del sistema de archivos especificará el directorio donde desea guardar los shaders compilados, junto con el nombre para su guardado. En el guardado de los shaders se seguirá el mismo formato especificado anteriormente para la carga de shaders.

El diagrama de la *Ilustración 34* muestra la abstracción del shader program y los shader objects. Se emplerá en el renderer para la visualización del modelo.

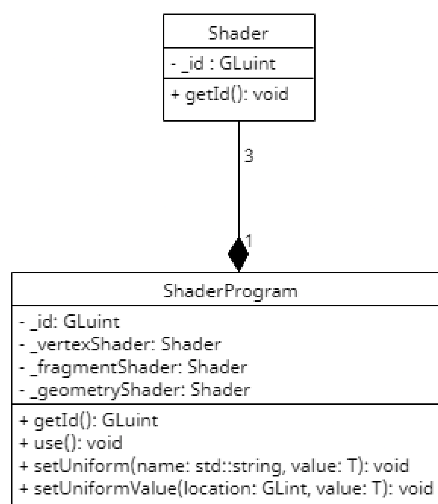


Ilustración 34 Diagrama UML para la gestión del vertex, fragment y geometry shader (fuente: elaboración propia)

Funcionalidad para alternar entre modos de visualización

Esta funcionalidad se puede implementar utilizando un enumerado para indicar el modo de visualización deseado. Los modos de visualización serán nube de puntos donde solo se visualizarán los vértices del modelo como puntos en la escena, y de la misma forma el modo líneas y el modo triángulos. Esto es posible puesto que si cargamos el modelo con un Index Buffer Object con la primitiva triángulos, podemos usar la función de OpenGL "glPolygonMode(GLenum face, GLenum mode)" (OpenGL® 4.5 Reference Pages, s.f.), donde el primer parámetro indica la orientación de la cara, que en este caso será tanto la frontal como la trasera (GL_FRONT_AND_BACK), y el modo se indicará según el parámetro de visualización indicado, y puede ser GL_POINT (nube de puntos), GL_LINE (líneas) o GL_FILL (triángulos).

En la interfaz habrá tres botones, cada uno indicando un tipo de visualización, donde por defecto estará activo el modo nube de puntos. Estos botones se podrán pulsar para alternar entre los modos de visualización, esto es, si se pulsa un modo de visualización diferente al actual, se desactivará el modo que estaba activo.

Funcionalidad para gestionar uniforms

Para la gestión de uniforms definiré la clase Uniform, que contendrá un nombre y un valor. Puesto que hay diferentes tipos de uniforms emplearé el tipo "variant" de C++, que me permitirá almacenar valores de distintos tipos para los uniforms. Estos podrán ser uniforms de tipo booleano (bool), entero (int), flotante (float), vector de dos flotantes (vec2), vector de tres flotantes (vec3), matriz de cuatro filas y cuatro columnas (mat4). Todo ello haciendo uso de la biblioteca GLM.

Los uniforms se gestionarán en la aplicación a través de una interfaz que representará una tabla con los uniforms que se han definido. Tanto el nombre del uniform como su valor se tomarán de un campo de texto de la interfaz, por lo que tendré que depurar para encontrar posibles errores que se puedan dar a la hora de añadir un uniform, los cuales son:

- No puede haber dos uniforms con el mismo nombre.

- Un uniform no puede tener un nombre que empiece por un número.
- El campo valor debe tener un formato de acuerdo al manual de usuario.
- El nombre del uniform no puede contener espacios.
- El campo nombre no puede estar vacío.
- El campo valor no puede estar vacío.

En el caso de que no se cumpla alguna de estas restricciones, se notificará al usuario mediante un cuadro de advertencia, indicando el error cometido para que pueda ser corregido.

Los uniforms se almacenarán en un vector que, una vez concluida la gestión de uniforms, se aplicarán en el shader program a través de un método al cual se le pasará el nombre y su valor.

Para gestionar los uniforms, la interfaz principal tendrá un botón que abrirá la tabla para poder asignar uniforms.

En la *Ilustración 35* se puede ver la clase que representará un uniform.

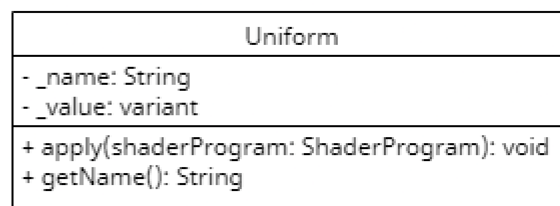


Ilustración 35 Clase uniform (fuente: elaboración propia)

Funcionalidad para gestionar subrutinas

Para especificar subrutinas en OpenGL, se emplea un método que devuelve el número de subrutinas activas en un shader determinado, por lo que se tendrá que hacer una diferenciación de las mismas según el shader. Por cada subrutina en los shaders, se deberá especificar el nombre y la implementación de subrutina que se quiera usar en un momento determinado. OpenGL gestiona tanto el nombre como la

implementación con identificadores de tipo entero, y para indicarlo en el shader program es necesario pasarlos como un array de enteros donde la posición que indica el identificador del nombre en el array contendrá el identificador de la implementación que se quiera utilizar.

Es por ello, que definiré una clase Subrutina la cual contendrá el nombre y la implementación de la misma. El shader al que se aplica, se gestionará desde el renderer.

Para su gestión en la interfaz de la aplicación, crearé una tabla en la que se puedan añadir y eliminar subrutinas, y para definirlas se tendrá que especificar el nombre, el shader al que se aplica, y finalmente la implementación que se quiera utilizar. De la misma forma que los uniforms, al recogerse los datos en campos de texto se deberá comprobar una serie de restricciones:

- El nombre de la subrutina no podrá comenzar por un número.
- La implementación de la subrutina no podrá comenzar por un número.
- El campo del nombre no puede estar vacío ni contener espacios.
- El campo de la implementación no puede estar vacío ni contener espacios.
- No puede haber dos subrutinas con el mismo nombre en el mismo shader.

De igual forma que en la gestión de uniforms, si no se cumple alguna de estas restricciones, se notificará al usuario mediante un cuadro de advertencia, indicando el error.

Las subrutinas se almacenarán en un mapa, donde la clave será el tipo de shader y el valor el array de subrutinas asociado a ese shader, para de esta forma poder gestionarlas de manera independiente. Se gestionará en el renderer mediante un método que gestionará las subrutinas según el shader al que se aplican.

En la *Ilustración 36* se puede ver la clase que representará una subrutina.

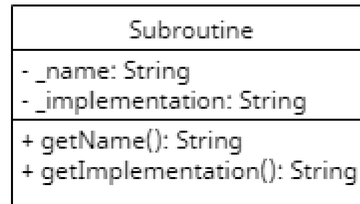


Ilustración 36 Clase subrutina (fuente: elaboración propia)

Funcionalidad para cargar una textura

A la hora de cargar la textura, se tendrá que asignar una unidad de textura para poder ser empleada en el shader, además de especificar el modo de visualización de la textura si excede el intervalo [0,1], como es repetir la textura, y la generación de mipmaps para mejorar el resultado a la hora de aplicar texturas.

Para la carga de textura, la interfaz tendrá un botón que permitirá al usuario navegar en el sistema de archivos hasta encontrar la textura que desea aplicar. En el shader se emplea como un uniform de tipo “sampler2d” al que le daré un nombre específico para su uso en el mismo.

Mecanismos de control para moverse en la escena.

Debido a que la escena constará de un modelo en cada momento, y para reducir la complejidad y evitar posibles “pérdidas” del usuario por la misma, los movimientos de cámara que considero más oportunos son el movimiento orbit y zoom.

Para aplicar estos mecanismos, se tendrá que posicionar el ratón sobre el canvas OpenGL y se usarán los clics del ratón para realizar el movimiento orbit calculando en cada frame el vector de dirección del ratón una vez ha sido arrastrado. En cuanto al zoom, se usará la rueda del ratón, aumentando el zoom al subir la rueda y disminuyendo zoom al bajarla.

En la *Ilustración 37* se puede ver la clase Cámara que representará la cámara virtual de la escena.

Camera
- _position: glm::vec3 - _lookAt: glm::vec3 - _up: glm::vec3 - _fovY: GLfloat - _aspect: GLfloat - _zNear: GLfloat - _zFar: GLfloat - _nAxis: glm::vec3 - _uAxis: glm::vec3 - _vAxis: glm::vec3
+ setAspect(aspect: GLfloat): void + Orbit(xOffset: GLfloat, yOffset: GLfloat): void + zoom(increase: GLfloat): void + getViewMatrix(): glm::mat4 + getProjectionMatrix(): glm::mat4 + reposition(position: glm::vec3, lookAt: glm::vec3): void

Ilustración 37 Clase cámara (fuente: elaboración propia)

Funcionalidad para visualizar una escena

Para esta funcionalidad, se creará una clase encargada de gestionar el renderizado de la aplicación, y se mostrará en un canvas OpenGL. Es el corazón de la aplicación, ya que se encargará de recibir toda la información como son el modelo, shader program, uniforms, subrutinas y textura para procesarla y mostrar el resultado final.

Con el fin de crear shaders dinámicos, es decir, que cambien a lo largo del tiempo, añadiré un temporizador que se empleará para establecer una tasa de refresco en el canvas OpenGL de sesenta fotogramas por segundo. Es decir, se tendrá que actualizar aproximadamente cada dieciséis milisegundos. Se añadirá un uniform con nombre “time”, con el valor del tiempo transcurrido en la aplicación.

Puesto que la visualización del resultado se realizará en un canvas OpenGL, definiré una cámara virtual para que el usuario pueda moverse por la misma.

La matriz de modelado, obtenida del modelo, y las matrices de visión y proyección, obtenidas de la cámara, se aplicarán como uniforms independientes con los nombres “model”, “view” y “projection” respectivamente, para que el usuario pueda combinarlas como quiera y de esta forma añadir flexibilidad a la aplicación.

En la *Ilustración 38* se puede ver el diagrama de la funcionalidad para visualizar el resultado.

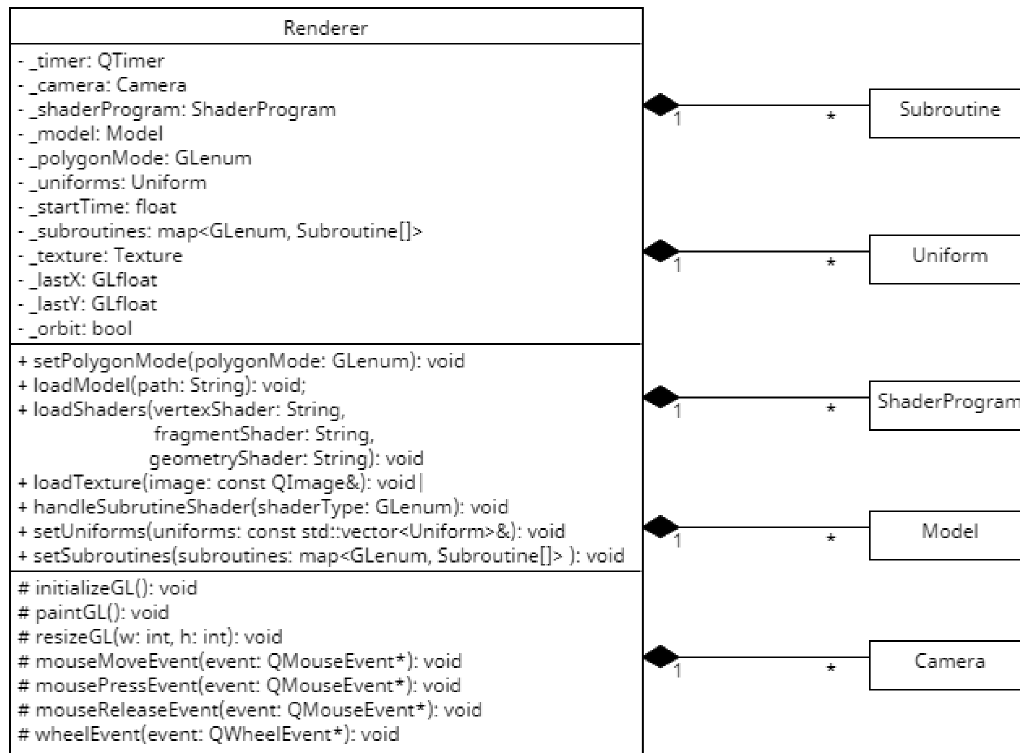


Ilustración 38 Diagrama UML para la visualización de la escena (fuente: elaboración propia)

2.2.2 Diseño de interfaz y storyboards

En este apartado se va a aportar el diseño para las diferentes interfaces contenidas en la aplicación.

En la *Ilustración 39* se puede ver el diseño de la interfaz de la ventana principal, en la cual tendremos una barra de herramientas con iconos para las diferentes acciones en la parte superior (cargar modelo, cargar shader, guardar shader, alternar entre modos de visualización, tabla de uniforms, tabla de subrutinas y cargar textura). En la parte central izquierda, se encuentra el editor de shaders en formato pestañas. En la parte central derecha se puede observar la ventana de la escena donde se va a visualizar el modelo aplicándole el shader. Finalmente, en la parte inferior tenemos un par de cuadros de texto que se actualizarán con la información del modelo y shaders cargados, para de esta forma aportar retroalimentación al usuario.

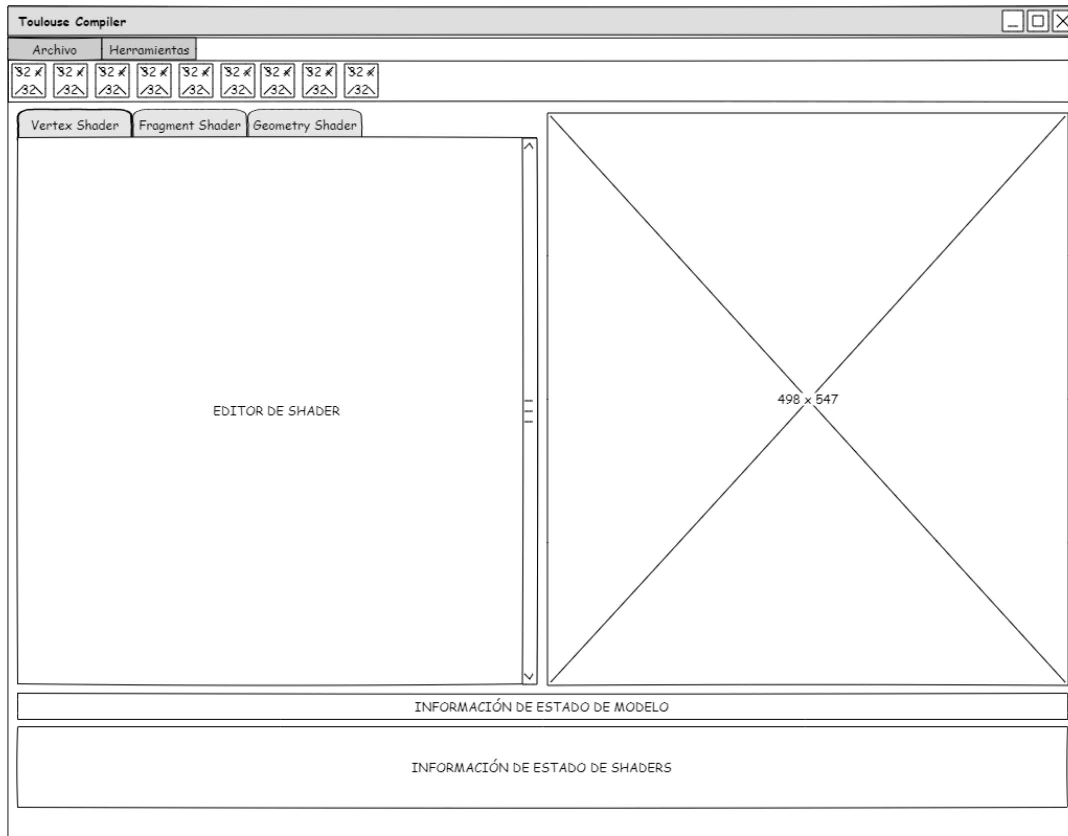


Ilustración 39 Interfaz principal de la aplicación (fuente: elaboración propia)

El diseño de interfaz de la *Ilustración 40* muestra la ventana encargada de gestionar los uniforms aplicados al shader. La idea es que se muestren una serie de filas con los uniforms activos, los cuales podremos seleccionar y eliminar con su correspondiente botón. En la parte inferior se muestra el área para añadir un uniform indicando su tipo (float, int, vec3, mat3…), su nombre en el shader y como último parámetro, su valor asociado. Cabe destacar que para indicar si el uniform tiene un valor correcto, se resaltará con una ventana emergente de aviso indicando el error en el caso de que, por ejemplo, se pusieran dos uniforms con el mismo nombre. El botón de la parte inferior derecha se usará para salir de la ventana de uniforms.

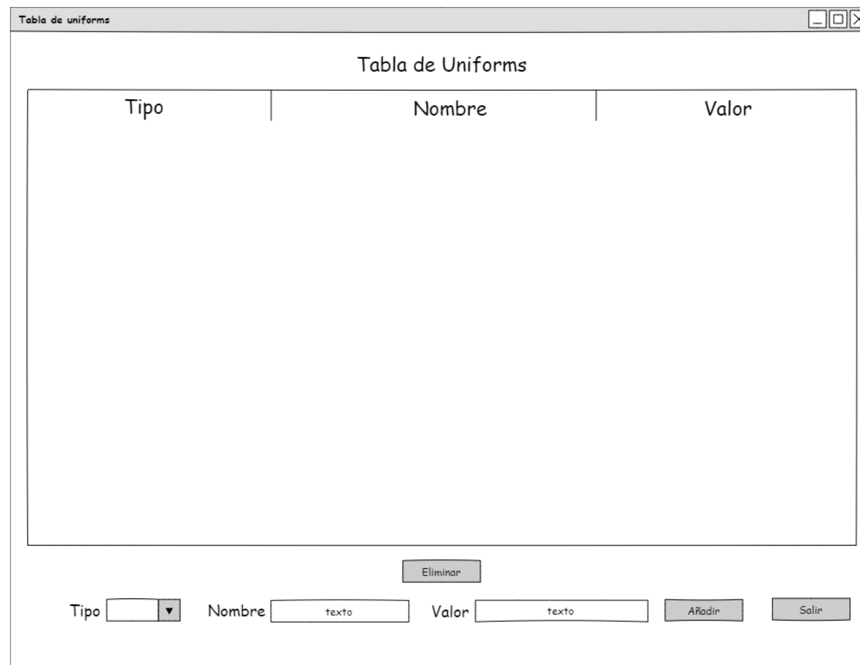


Ilustración 40 Interfaz para la gestión de uniforms (fuente: elaboración propia)

La última interfaz de la aplicación, que se muestra en la *Ilustración 41,* es la tabla de subrutinas, la cual tiene un comportamiento idéntico a la tabla de uniforms, donde podremos seleccionar una subrutina y eliminarla. La diferencia es que para una subrutina se tiene que especificar el nombre de la misma, el shader al que se aplica y su implementación a usar, ya que OpenGL las gestiona de esa forma.

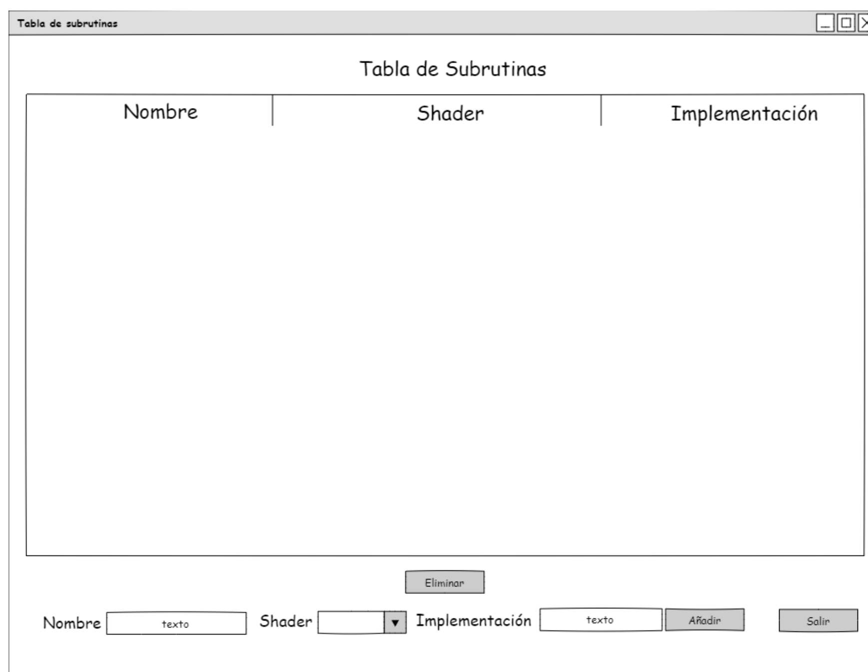


Ilustración 41 Interfaz para la gestión de subrutinas (fuente: elaboración propia)

En la *Ilustración 42* se puede ver el diagrama UML para la gestión de funcionalidades e interfaces.

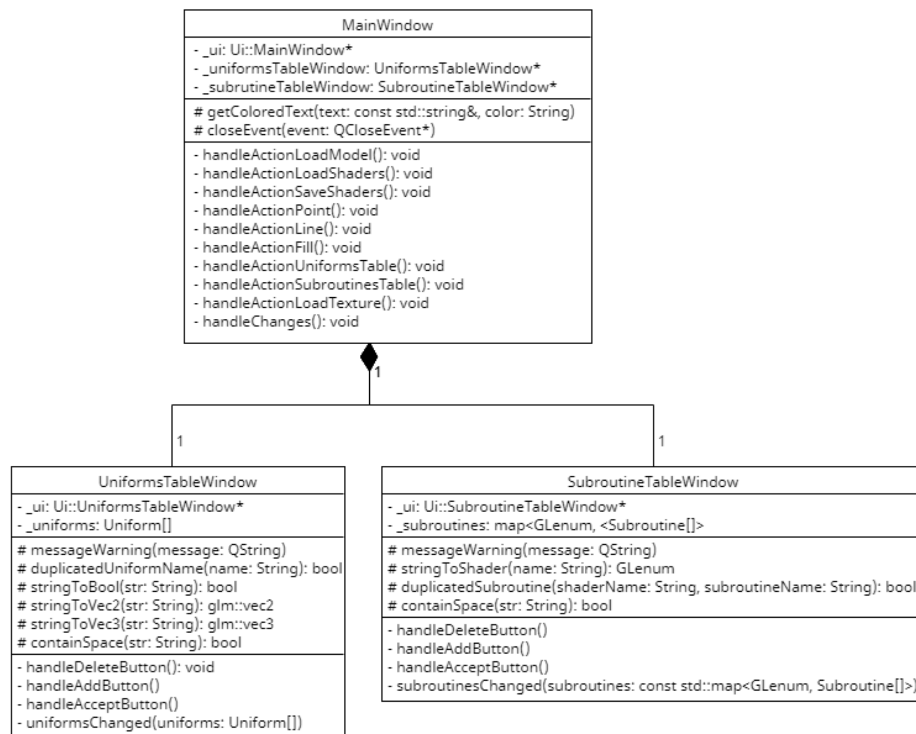


Ilustración 42 Diagrama UML para la gestión de funcionalidades e interfaces (fuente: elaboración propia)

3 DESARROLLO

En este apartado vamos a comentar el desarrollo de la aplicación. Se expondrán las diferentes funcionalidades de la aplicación por separado, indicando los detalles de implementación, que seguirá los diagramas de clases que se han mostrado en apartados anteriores.

3.1 Implementación

En este apartado vamos a mostrar los detalles de implementación del proceso de desarrollo de la aplicación.

Hay que destacar que el contexto de OpenGL lo crea la clase Renderer, y es necesario pasar ese contexto a las diferentes clases que vayan a hacer de funciones OpenGL, Es por ello que muchas de las clases tienen un atributo **"_functions",** que pertenece a la clase **QOpenGLFunctions** o **QOpenGLExtraFunctions** según el caso de uso.

El método setupMesh() (*Ilustración 43*) de la clase Mesh es el encargado de cargar la geometría en la GPU. Primero crea el VAO y lo enlaza, acto seguido se crea y enlaza al VAO el VBO entrelazado, al cual se le asigna el vector de la estructura Vertex que contiene la información de cada vértice. Después, se crea y enlaza al VAO el IBO donde se asigna el vector de enteros que contiene la topología del modelo. Tanto el VBO como el IBO se especifican con el uso "GL_STATIC_DRAW", que indica que los datos de los buffers se especificarán una sola vez y se usarán muchas veces para operaciones de dibujo.

Puesto que debe ser usado en el shader program, se activan los layouts donde las posiciones corresponderán con el layout (location = 0), las normales corresponderán con el layout (location = 1), las coordenadas de textura corresponderán con el layout (location = 2) y las tangentes corresponderán con el layout (location = 3). Finalmente se desenlaza el VAO.

```
void Mesh::setupMesh() {

    // - Creamos el Vertex Array Object
    _functions->glGenVertexArrays( n: 1,  arrays: &_vao);
    _functions->glBindVertexArray( array: _vao);

    // - Creamos el Vertex Buffer Object y asignamos el vector de vértices
    _functions->glGenBuffers( n: 1,  buffers: &_vbo);
    _functions->glBindBuffer( target: GL_ARRAY_BUFFER,  buffer: _vbo);
    _functions->glBufferData( target: GL_ARRAY_BUFFER,  size: _vertices.size() * sizeof(Vertex),  data: &_vertices[0],  usage: GL_STATIC_DRAW);

    // - Creamos el Index Buffer Object y asignamos el vector de indices
    _functions->glGenBuffers( n: 1,  buffers: &_ibo);
    _functions->glBindBuffer( target: GL_ELEMENT_ARRAY_BUFFER,  buffer: _ibo);
    _functions->glBufferData( target: GL_ELEMENT_ARRAY_BUFFER,  size: _indices.size() * sizeof(GLuint),  data: &_indices[0],
                             usage: GL_STATIC_DRAW);

    // - Posiciones (layout = 0)
    _functions->glEnableVertexAttribArray( index: 0);
    _functions->glVertexAttribPointer( indx: 0,  size: 3,  type: GL_FLOAT,  normalized: GL_FALSE,  stride: sizeof(Vertex),
                                      ptr: (void *) offsetof(Vertex, Position));

    // - Normales (layout = 1)
    _functions->glEnableVertexAttribArray( index: 1);
    _functions->glVertexAttribPointer( indx: 1,  size: 3,  type: GL_FLOAT,  normalized: GL_FALSE,  stride: sizeof(Vertex),
                                      ptr: (void *) offsetof(Vertex, Normal));

    // - Coordenadas de textura (layout = 2)
    _functions->glEnableVertexAttribArray( index: 2);
    _functions->glVertexAttribPointer( indx: 2,  size: 2,  type: GL_FLOAT,  normalized: GL_FALSE,  stride: sizeof(Vertex),
                                      ptr: (void *) offsetof(Vertex, TexCoords));

    // - Tangentes (layout = 3)
    _functions->glEnableVertexAttribArray( index: 3);
    _functions->glVertexAttribPointer( indx: 3,  size: 3,  type: GL_FLOAT,  normalized: GL_FALSE,  stride: sizeof(Vertex),
                                      ptr: (void *) offsetof(Vertex, Tangents));

    // - Desenlazamos el vao
    _functions->glBindVertexArray( array: 0);

}
```

Ilustración 43 Método setupMesh (fuente: elaboración propia)

Para cargar un modelo .OBJ con la biblioteca Assimp es necesario usar la clase **"Importer"**, que crea una escena para procesar los nodos. Esta clase lee un archivo de una ruta que se le pase y además requiere de una serie de flags (configuraciones). Los flags que he usado han sido:

- **aiProcess_Triangulate:** Sirve para especificar que se desea que todas las primitivas geométricas sean trianguladas. Se triangulan todas las primitivas para posteriormente poder alternar entre modos de visualización.
- **aiProcess_GenSmoothNormals:** Sirve para generar normales suavizadas automáticamente en aquellos vértices donde no están explícitamente definidas. Esta es una opción para que, a la hora de aplicar la iluminación, no se generen errores de visualización si hubiera algún vértice sin normales.

- **aiProcess_PreTransformVertices:** Sirve para aplicar transformaciones de vértices antes de procesarlos. Es útil para cargar modelos con varias mallas que no tienen las transformaciones aplicadas.

En la *Ilustración 44* se puede ver el constructor de la carga de modelos con Assimp, donde se aplican los flags comentados, se comprueba si se ha cargado la escena correctamente, es decir, si el modelo es correcto, y finalmente se procesan los nodos donde se almacenará la geometría en los atributos del modelo para ser visualizada.

```
ObjModel::ObjModel(QOpenGLContext *glContext, const std::string &path) : Model(glContext, path) {

    Assimp::Importer import;
    const aiScene *scene = import.ReadFile( pFile: _path,  pFlags: aiProcess_Triangulate |
                                                          aiProcess_GenSmoothNormals |
                                                          aiProcess_PreTransformVertices);

    if (!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode) {

        throw std::runtime_error(import.GetErrorString());

    }

    processNode( node: scene->mRootNode, scene);

}
```

Ilustración 44 Carga de modelos con Assimp (fuente: elaboración propia)

Para la lectura de un sólido de revolución, se ha realizado una depuración del mismo para evitar posibles errores en su carga. Estas comprobaciones son:

- El número de vértices debe ser como mínimo tres.
- La abscisa del primer y último vértice debe ser mayor o igual a cero y del resto de vértices mayor estricto a cero.

Otro aspecto a tener en cuenta es el sentido en el que se han escrito los vértices, ya que han de estar en sentido antihorario para generar las normales de manera correcta, y que de esta forma se visualice bien el modelo. Es por ello que se comprueba su sentido, y si este es horario, se invierten los vértices.

Para los sólidos de revolución que empiezan y acaban en el eje de revolución, se crearán tres mallas, donde dos de ellas son las tapas, a las cuales se les aplicará un modo de dibujado “GL_TRIANGLE_FAN”. Estas tapas están formadas por el primer o último vértice y el conjunto de vértices de la siguiente subdivisión horizontal. La otra malla representa el cuerpo, al que se aplica el modo de dibujado “GL_TRIANGLE_STRIP”.

En cuanto a la clase ShaderProgram, el geometry shader será un parámetro opcional que solo se procesará si existe texto en su editor correspondiente. Dado que existen numerosos tipos de uniforms, y para evitar la creación de numerosos métodos para aplicar cada tipo, he hecho uso de los templates de C++ para que a través del mismo método se puedan aplicar los diferentes uniforms siempre y cuando estén definidos (*Ilustración 45*).

```
    /**
     * @brief Método para asignar un uniform.
     * @param name - Nombre del uniform.
     * @param value - Valor a asignar.
     */
    template<typename T>
    void setUniform(const std::string &name, const T &value);

    /**
     * @brief Método para asignar un uniform a una localización específica.
     * @param location - Id del uniform.
     * @param value - Valor a asignar.
     */
    template <typename T>
    void setUniformValue(GLint location, const T& value);

};

template<typename T>
void ShaderProgram::setUniform(const std::string &name, const T &value){

    GLint location = _functions->glGetUniformLocation( program: _id,  name: name.c_str());

    if (location >= 0) {

        setUniformValue(location, value);

    }

};
```

Ilustración 45 Método setUniform de la clase ShaderProgram (fuente: elaboración propia)

Puesto que el valor de los uniforms es de tipo “variant”, es necesario el uso de la función “visit” de C++ para aplicar correctamente el valor almacenado, puesto que el compilador no sabe de antemano qué tipo se va a aplicar (*Ilustración 46*).

```
void Uniform::apply(const std::shared_ptr<ShaderProgram> &shaderProgram) {
    std::visit( visitor: [this, shaderProgram](const auto &val) -> void  {
        shaderProgram->setUniform( name: _name,  value: val);
    },  &: _value);
}
```

Ilustración 46 Método apply de la clase Uniform (fuente: elaboración propia)

Los parámetros iniciales que se usan para definir la cámara se pueden ver en la *Ilustración 47*. La relación de aspecto viene dada por el renderer OpenGL que define la clase Renderer, y será actualizado cada vez que se redimensione la ventana.

```
Camera::Camera(GLfloat aspect) {
    _position = glm::vec3( x: 0.0f,  y: 5.0f,  z: 20.0f);
    _lookAt = glm::vec3( x: 0.0f,  y: 0.0f,  z: 0.0f);
    _up = { x: 0.0f,  y: 1.0f,  z: 0.0f};

    _fovY = 40.0f;
    _aspect = aspect;
    _zNear = 0.1;
    _zFar = 10000;

    updateAxis();
}
```

Ilustración 47 Parámetros de la cámara virtual (fuente elaboración propia)

En el método initializeGL (*Ilustración 48*) de la clase Renderer se inicializa el contexto OpenGL y sus parámetros globales, que son:

- Habilitar test de profundidad con la función GL_LEQUAL (menor o igual) para comparar. Se usa para asegurar que los objetos más cercanos a la cámara virtual se dibujen sobre los objetos más lejanos
- Especificar el índice para reiniciar la secuencia de primitivas durante el dibujado de una malla de triángulos para la carga de sólidos de revolución. El índice que se ha empleado ha sido 0xFFFF, que está almacenado en PRIMITIVE_RESTART_INDEX.
- Habilitar el antialiasing (GL_MULTISAMPLE) para suavizar los bordes de los objetos renderizados.
- Habilitar la eliminación de caras (GL_CULL_FACE) traseras (GL_BACK), para que no haya confusión a la hora de visualizar el modelo en modo nube de puntos y modo alambre.

```
void Renderer::initializeGL() {

    // - Llamada al método para inicializar las funciones OpenGL.
    initializeOpenGLFunctions();

    // - Habilitamos el z-buffer con funcion menor igual.
    glEnable( cap: GL_DEPTH_TEST);
    glDepthFunc( func: GL_LEQUAL);

    // - Establecemos el índice de reinicio de la malla de triangulos.
    glPrimitiveRestartIndex( index: PRIMITIVE_RESTART_INDEX);

    // - Habilitamos el uso de antialiasing de múltiples muestras para suavizado de bordes.
    glEnable( cap: GL_MULTISAMPLE);

    // - Habilitamos la eliminación de caras ocultas.
    glEnable( cap: GL_CULL_FACE);
    glCullFace( mode: GL_BACK);

    _camera = std::make_unique<tc::Camera>( aspect: (GLfloat) width() / (GLfloat) height());

    // Iniciar el tiempo
    startTime = QDateTime::currentMSecsSinceEpoch() / 1000.0;
}
```

Ilustración 48 Método initializeGL de la clase Renderer (fuente: elaboración propia)

Para comunicar la interfaz con la lógica de la aplicación, he conectado las funciones signal de las distintas acciones de la aplicación con las funciones slot que han sido implementadas para manejar la lógica. En la *Ilustración 49* se puede ver la comunicación de las distintas acciones de la interfaz principal.

```
MainWindow::MainWindow(QWidget *parent) :
        QMainWindow(parent), _ui(new Ui::MainWindow) {

    _ui->setupUi( tc_MainWindow: this);

    // - Creamos la tabla de uniforms
    _uniformsTableWindow = new UniformsTableWindow();

    // - Creamos la tabla de subrutinas
    _subrutineTableWindow = new SubroutineTableWindow();

    // - Acción cargar modelo
    connect( sender: _ui->_actionLoadModel, signal: &QAction::triggered, context: this, slot: &MainWindow::handleActionLoadModel);

    // - Acción cargar shader
    connect( sender: _ui->_actionLoadShader, signal: &QAction::triggered, context: this, slot: &MainWindow::handleActionLoadShaders);

    // - Acción guardar shader
    connect( sender: _ui->_actionSaveShader, signal: &QAction::triggered, context: this, slot: &MainWindow::handleActionSaveShaders);

    // - Acciones para el modo de visualización de los polígonos
    connect( sender: _ui->_actionPoint, signal: &QAction::triggered, context: this, slot: &MainWindow::handleActionPoint);
    connect( sender: _ui->_actionLine, signal: &QAction::triggered, context: this, slot: &MainWindow::handleActionLine);
    connect( sender: _ui->_actionFill, signal: &QAction::triggered, context: this, slot: &MainWindow::handleActionFill);

    // - Acciónes para la tabla de uniforms
    connect( sender: _ui->_actionUniformsTable, signal: &QAction::triggered, context: this, slot: &MainWindow::handleActionUniformsTable);
    connect( sender: _uniformsTableWindow, signal: &UniformsTableWindow::uniformsChanged, context: _ui->_renderer, slot: &Renderer::setUniforms);

    // - Acciónes para la tabla de subrutinas
    connect( sender: _ui->_actionSubroutinesTable, signal: &QAction::triggered, context: this, slot: &MainWindow::handleActionSubroutinesTable);
    connect( sender: _subrutineTableWindow, signal: &SubroutineTableWindow::subroutinesChanged, context: _ui->_renderer, slot: &Renderer::setSubroutines);

    // - Acción para visualizar el resultado
    connect( sender: _ui->_vertexShaderText, signal: &QPlainTextEdit::textChanged, context: this, slot: &MainWindow::handleChanges);
    connect( sender: _ui->_fragmentShaderText, signal: &QPlainTextEdit::textChanged, context: this, slot: &MainWindow::handleChanges);
    connect( sender: _ui->_geometryShaderText, signal: &QPlainTextEdit::textChanged, context: this, slot: &MainWindow::handleChanges);

    // - Acción para cargar texturas
    connect( sender: _ui->_actionLoadTexture, signal: &QAction::triggered, context: this, slot: &MainWindow::handleActionLoadTexture);
}
```

Ilustración 49 Comunicación de las distintas acciones de la intergaz principal (fuente: elaboración propia)

3.2 Pruebas finales

En este apartado se comentarán las pruebas finales del sistema, es decir, el conjunto de pruebas de todas las funcionalidades implementadas en la etapa de desarrollo para validar los requisitos definidos.

Prueba para la visualización de un modelo

Esta prueba tiene como objetivo evaluar la visualización de un modelo. Para ello es necesaria la carga tanto de un modelo como de al menos un vertex shader y un fragment shader. Se hará la prueba de diferente modelos en formato .OBJ, .FBX y .SOR (*Ilustración 50*).

Ajax.obj	25/10/2023 22:28	3D Object	16.029 KB
copa.sor	01/11/2023 20:25	Archivo SOR	1 KB
corona.sor	01/11/2023 20:26	Archivo SOR	1 KB
dado.obj	25/10/2023 22:28	3D Object	1 KB
example.sor	05/11/2023 0:59	Archivo SOR	1 KB
JakeElPerro.fbx	24/02/2024 11:51	3D Object	632 KB
lata.sor	11/02/2024 12:06	Archivo SOR	1 KB
Mandalorian.obj	25/10/2023 21:50	3D Object	5.408 KB
objetoPag.sor	04/11/2023 22:56	Archivo SOR	1 KB
Panther.obj	25/10/2023 22:28	3D Object	13.353 KB
peon.sor	04/11/2023 22:50	Archivo SOR	1 KB
SonicModel.fbx	05/04/2024 11:48	3D Object	253 KB
toroide.sor	11/02/2024 12:19	Archivo SOR	1 KB
t-rex.obj	06/12/2022 3:10	3D Object	1.047 KB
trompo.sor	12/11/2023 14:02	Archivo SOR	1 KB
vacaSinNormales.obj	25/10/2023 22:28	3D Object	323 KB

Ilustración 50 Modelos para realizar la prueba (fuente: elaboración propia)

Se cargará un vertex shader (*Ilustración 51*) y un fragment shader (*Ilustración 52)* sencillos, donde este último asignará los píxeles de color verde. Se hará uso del modo de visualización de líneas para ver mejor el resultado.

```
1 #version 410
2 layout (location = 0) in vec3 posicion;
3
4 uniform mat4 model;
5 uniform mat4 view;
6 uniform mat4 projection;
7
8 void main ()
9 {
10  gl_Position = projection * view * model * vec4 ( posicion, 1 );
11 }
```

Ilustración 51 Vertex shader para la prueba de visualización (fuente: elaboración propia)

```
1 #version 410
2 out vec4 colorFragmento;
3 void main ()
4 {
5 colorFragmento = vec4 ( 0.0, 1.0, 0.0, 1.0 );
6 }
```

Ilustración 52 Fragment shader para la prueba de visualización (fuente: elaboración propia)

Todos los modelos se visualizan de manera correcta. Se puede ver en la *Ilustración 53*.

Ilustración 53 Resultado de la carga de cada uno de los modelos (fuente: elaboración propia)

Prueba de uso de uniforms y modos de visualización

Esta prueba tiene como objetivo evaluar la funcionalidad para gestionar uniforms presentes en el shader. Para ello emplearemos el modelo "Ajax.obj" y un vertex shader (Ilustración 54) y fragment shader (*Ilustración 55*) en el cual definiremos a través de uniforms un material (Ka, Kd, Ks, shininess) para el modelo y una luz puntual (lightPosition, Ia, Id, Is).

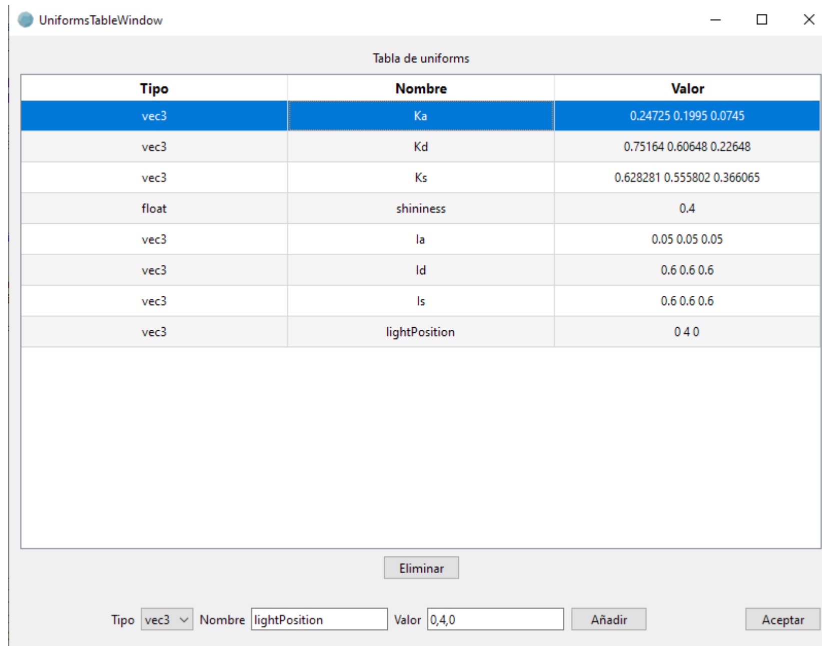
```
1 #version 330 core
2
3 layout(location = 0) in vec3 vPosition;
4 layout(location = 1) in vec3 vNormal;
5
6 out vec3 position;
7 out vec3 normal;
8
9 uniform mat4 model;
10 uniform mat4 view;
11 uniform mat4 projection;
12
13 void main()
14 {
15
16     normal = vec3(view * model * vec4(vNormal,0.0));
17     position = vec3(view * model * vec4(vPosition, 1.0));
18
19     gl_Position = projection * view * model * vec4(vPosition,
   1.0);
20 }
```

Ilustración 54 Vertex shader para la prueba de uniforms (fuente: elaboración propia)

```
1 #version 330 core
2
3 in vec3 position;
4 in vec3 normal;
5
6 uniform vec3 Ka;
7 uniform vec3 Kd;
8 uniform vec3 Ks;
9 uniform float shininess;
10
11 uniform vec3 Ia;
12 uniform vec3 Id;
13 uniform vec3 Is;
14 uniform vec3 lightPosition;
15
16 layout (location = 0) out vec4 fragColor;
17
18 vec3 ads(){
19     vec3 n = normalize(normal);
20     vec3 l = normalize(lightPosition - position);
21     vec3 v = normalize(-position);
22     vec3 r = reflect(-l, n);
23     vec3 ambient =  (Ia * Ka);
24     vec3 diffuse = (Id * Kd * max(dot(l,n),0.0));
25     vec3 specular = (Is * Ks * pow(max(dot(r,v),0.0),
   shininess));
26
27     return ambient + diffuse + specular;
28 }
29
30 void main()
31 {
32     fragColor = vec4(ads(), 1.0);
33 }
```

Ilustración 55 Fragment shader para la prueba de uniforms (fuente: elaboración propia)

Asignamos todos los uniforms tanto para el material (que simulará oro) como para la luz puntual (que será una luz blanca) en la tabla (*Ilustración 56*).



Tipo	Nombre	Valor
vec3	Ka	0.24725 0.1995 0.0745
vec3	Kd	0.75164 0.60648 0.22648
vec3	Ks	0.628281 0.555802 0.366065
float	shininess	0.4
vec3	Ia	0.05 0.05 0.05
vec3	Id	0.6 0.6 0.6
vec3	Is	0.6 0.6 0.6
vec3	lightPosition	0 4 0

Ilustración 56 Asignación de uniforms (fuente: elaboración propia)

Una vez asignados los uniforms y pulsado el botón aceptar se visualiza el resultado esperado, donde se puede ver el modelo con los efectos de iluminación (*Ilustración 57*)

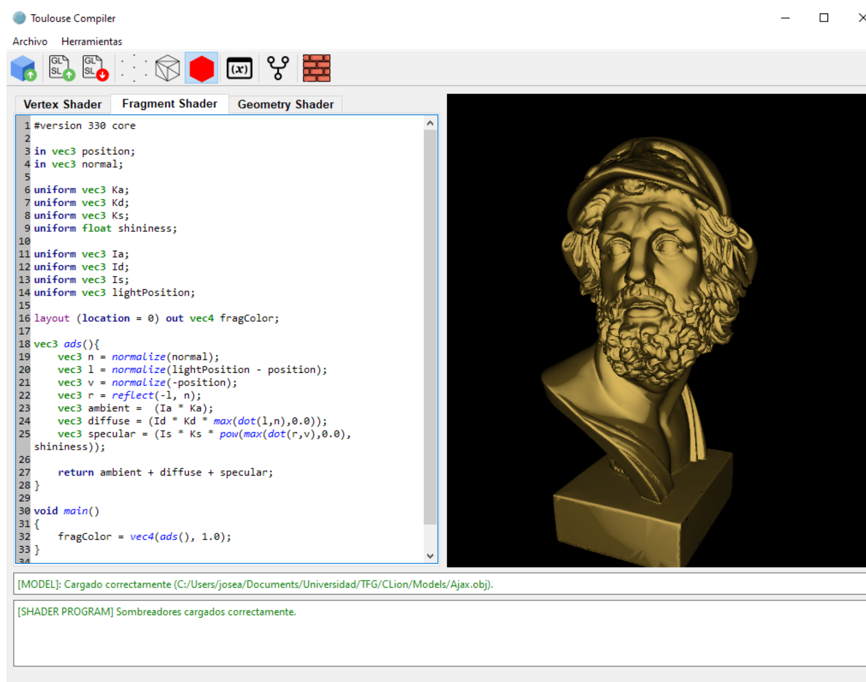


Ilustración 57 Resultado de la prueba de uniforms (fuente: elaboración propia)

Para la prueba sobre los modos de visualización se va a usar otro material (esmeralda) y el modelo “Mandalorian.obj”. Basta con alternar entre los distintos modos de visualización. El resultado de la aplicación del modo de visualización de puntos se puede ver en la *Ilustración 58*.

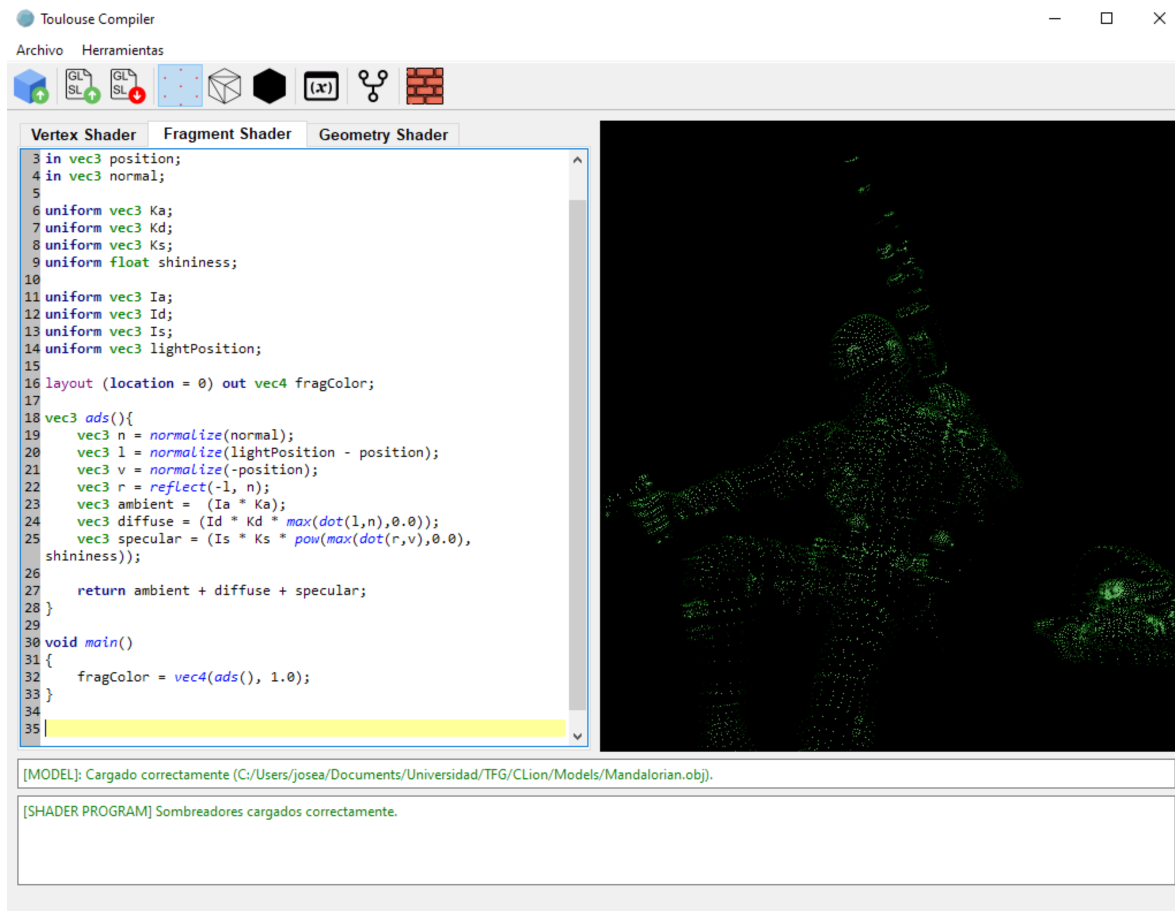


Ilustración 58 Resultado al aplicar modo de visualización de puntos (fuente: elaboración propia)

En la *Ilustración 59* se puede ver el resultado al aplicar el modo de visualización de líneas.

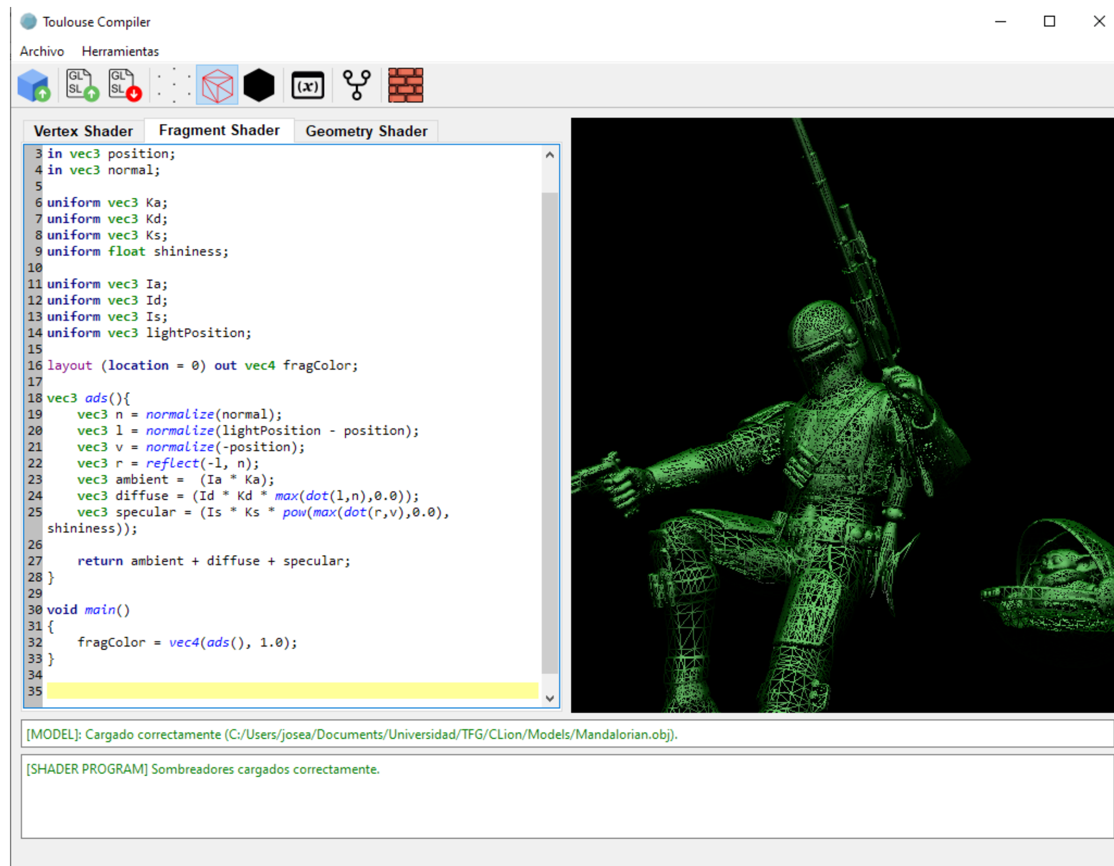


Ilustración 59 Resultado al aplicar modo de visualización de líneas (fuente: elaboración propia)

Finalmente, en la *Ilustración 60* se puede ver el resultado tras aplicar el modo de visualización de triángulos.

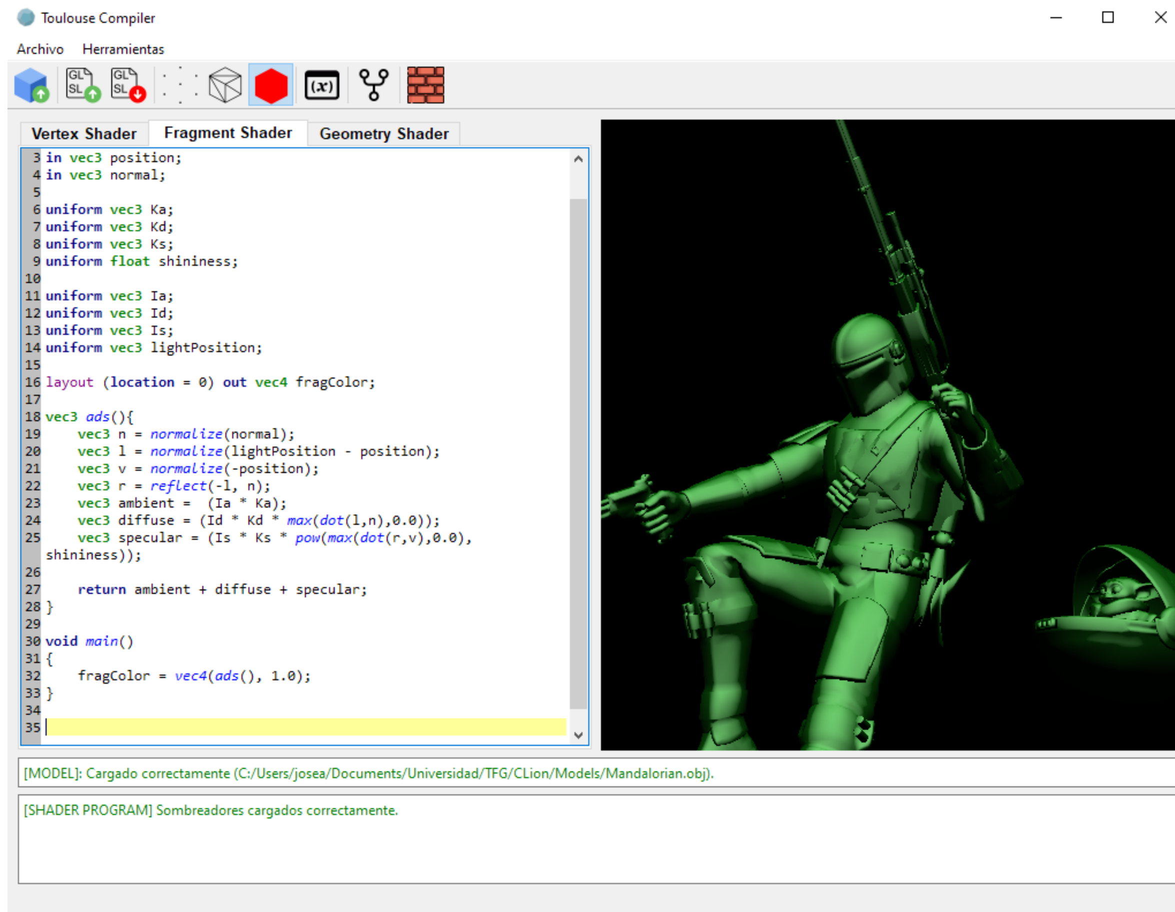


Ilustración 60 Resultado al aplicar modo de visualización de triángulos (fuente: elaboración propia)

Prueba de subrutinas

El objetivo de esta prueba es evaluar la funcionalidad para gestionar subrutinas presentes en el shader. Para ello vamos a realizar una prueba sencilla de cambio de color del modelo a través de una subrutina. El vertex shader y fragment shader empleados se pueden ver en la *Ilustración 61* e *Ilustración 62* respectivamente.

```
1  #version 410
2  layout (location = 0) in vec3 posicion;
3
4  uniform mat4 model;
5  uniform mat4 view;
6  uniform mat4 projection;
7
8  void main ()
9  {
10  gl_Position = projection * view * model * vec4 ( posicion, 1 );
11 }
```

Ilustración 61 Vertex shader para la prueba de subrutinas (fuente: elaboración propia)

```
1  #version 410
2  out vec4 colorFragmento;
3
4  subroutine vec4 a();
5  subroutine uniform a colorA;
6
7  subroutine(a)
8  vec4 red(){
9      return vec4(1.0,0.0,0.0,1.0);
10 }
11
12 subroutine(a)
13 vec4 green(){
14     return vec4(0.0,1.0,0.0,1.0);
15 }
16
17 void main ()
18 {
19 colorFragmento = colorA();
20 }
```

Ilustración 62 Fragment shader para la prueba de subrutinas (fuente: elaboración propia)

Asignamos en la tabla de subrutinas el nombre de la subrutina, el shader en el que se aplica y la implementación a usar (*Ilustración 63*).

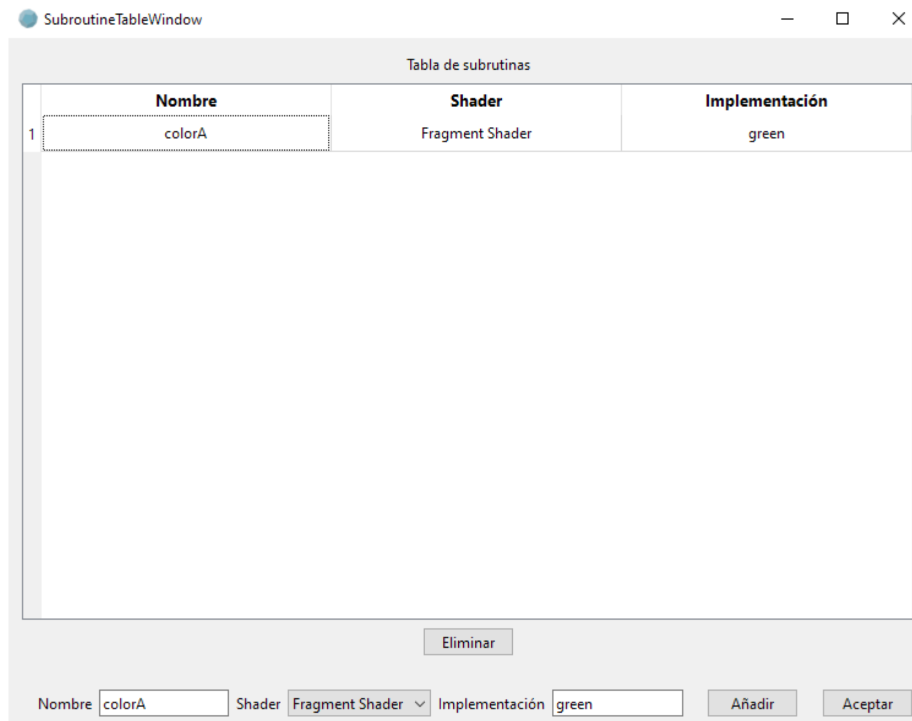


Ilustración 63 Asignación de la primera implementación de subrutina (fuente: elaboración propia)

Pulsamos en el botón aceptar y obtenemos un resultado correcto, el modelo aparece del color asignado (*Ilustración 64*).

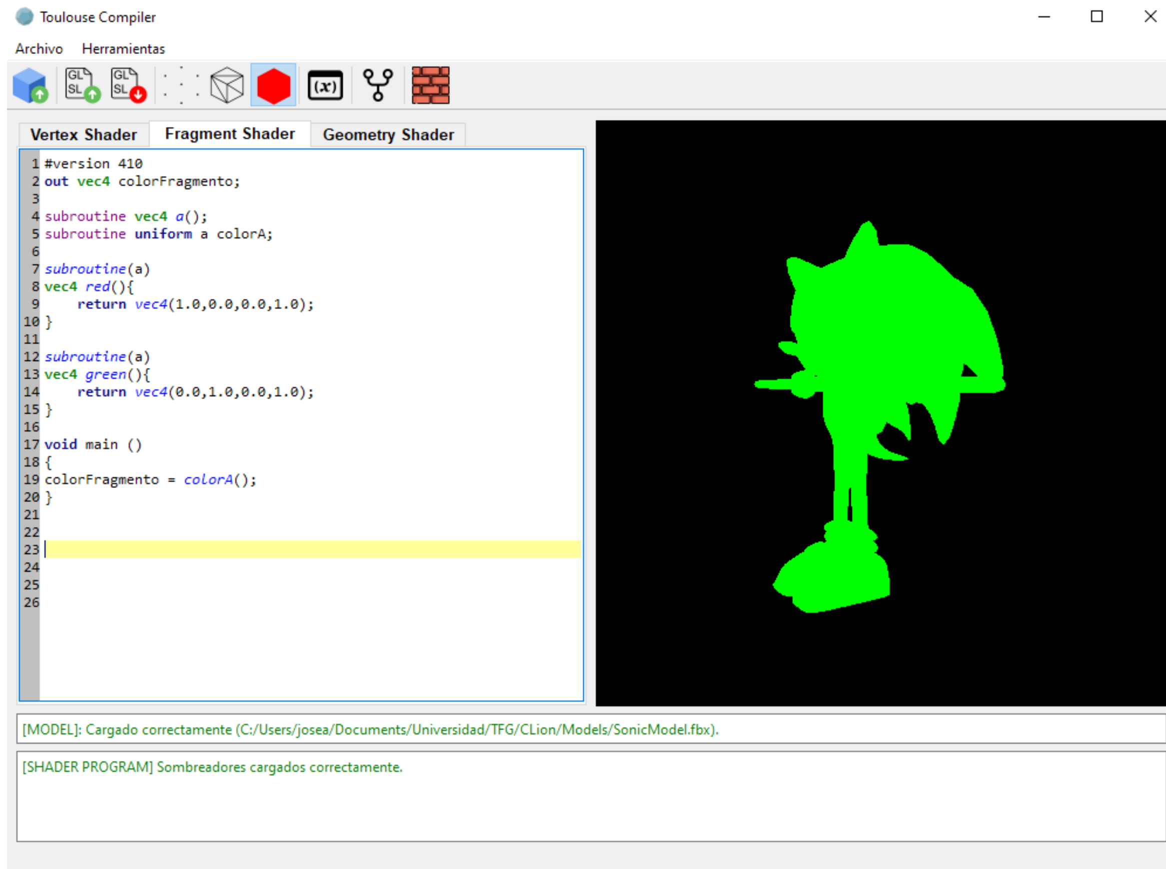


Ilustración 64 Resultado al aplicar la primera implementación de subrutina (fuente: elaboración propia)

Para completar la prueba, alternamos de implementación para comprobar que las subrutinas se aplican de manera correcta (*Ilustración 65*).

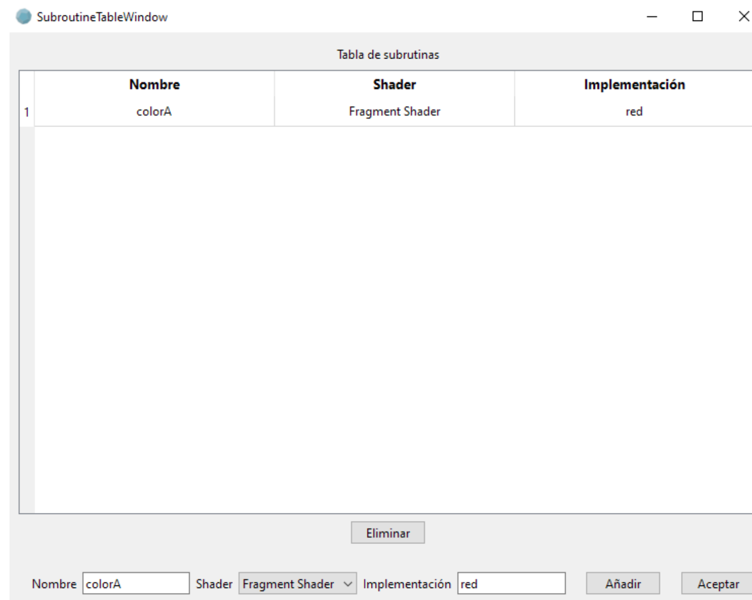


Ilustración 65 Asignación de la segunda implementación de subrutina (fuente: elaboración propia)

El resultado es correcto, se aplica la implementación de subrutina que hemos asignado (*Ilustración 66*).

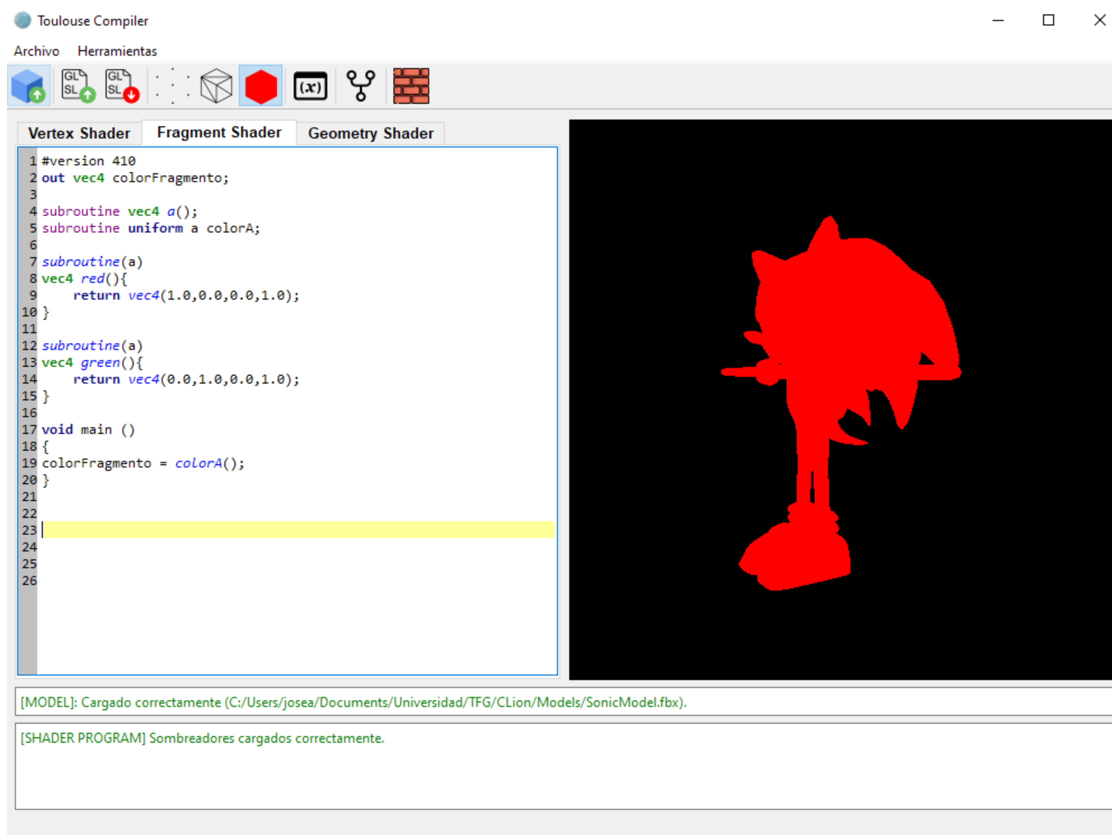


Ilustración 66 Resultado al aplicar la segunda implementación de subrutina (fuente: elaboración propia)

Prueba de textura

El objetivo de esta prueba es evaluar la carga de la textura en la aplicación. Para ello, definiremos un vertex shader y un fragment shader que hagan uso de ella. El vertex shader se puede ver en la *Ilustración 67* y el fragment shader en la *Ilustración 68.*

```
1 #version 330 core
2 layout (location = 0) in vec3 aPos;
3 layout (location = 1) in vec3 aColor;
4 layout (location = 2) in vec2 aTexCoord;
5
6 out vec3 ourColor;
7 out vec2 TexCoord;
8
9 uniform mat4 model;
10 uniform mat4 view;
11 uniform mat4 projection;
12
13 void main()
14 {
15     gl_Position = projection * view * model * vec4(aPos, 1.0);
16     ourColor = aColor;
17     TexCoord = aTexCoord;
18 }
```

Ilustración 67 Vertex shader para la prueba de texturas (fuente: elaboración propia)

```
1 #version 330 core
2 out vec4 FragColor;
3
4 in vec3 ourColor;
5 in vec2 TexCoord;
6
7 uniform sampler2D textureSampler;
8
9 void main()
10 {
11     FragColor = texture(textureSampler, TexCoord);
12 }
```

Ilustración 68 Fragment shader para la prueba de texturas (fuente: elaboración propia)

Para la prueba haremos uso del modelo "t-rex.obj" y cargaremos su textura (*Ilustración 69*).

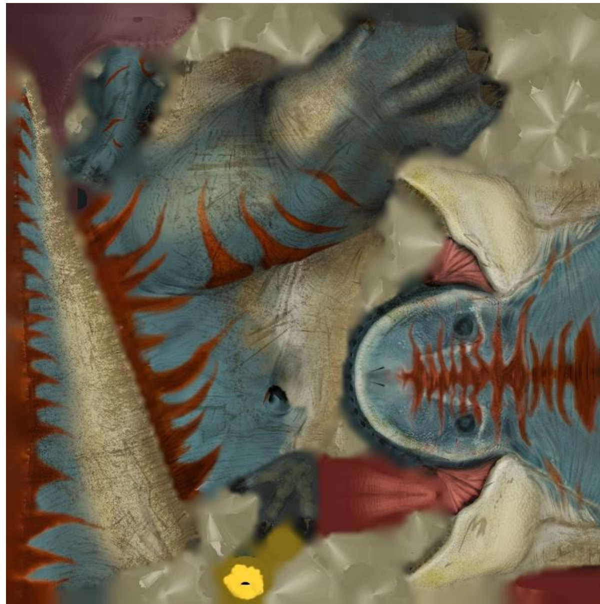


Ilustración 69 Textura usada en la prueba (fuente: Asignatura Programación de Aplicaciones Gráficas)

Una vez cargada la textura, ésta se visualiza de manera correcta sobre el modelo (*Ilustración 70*).

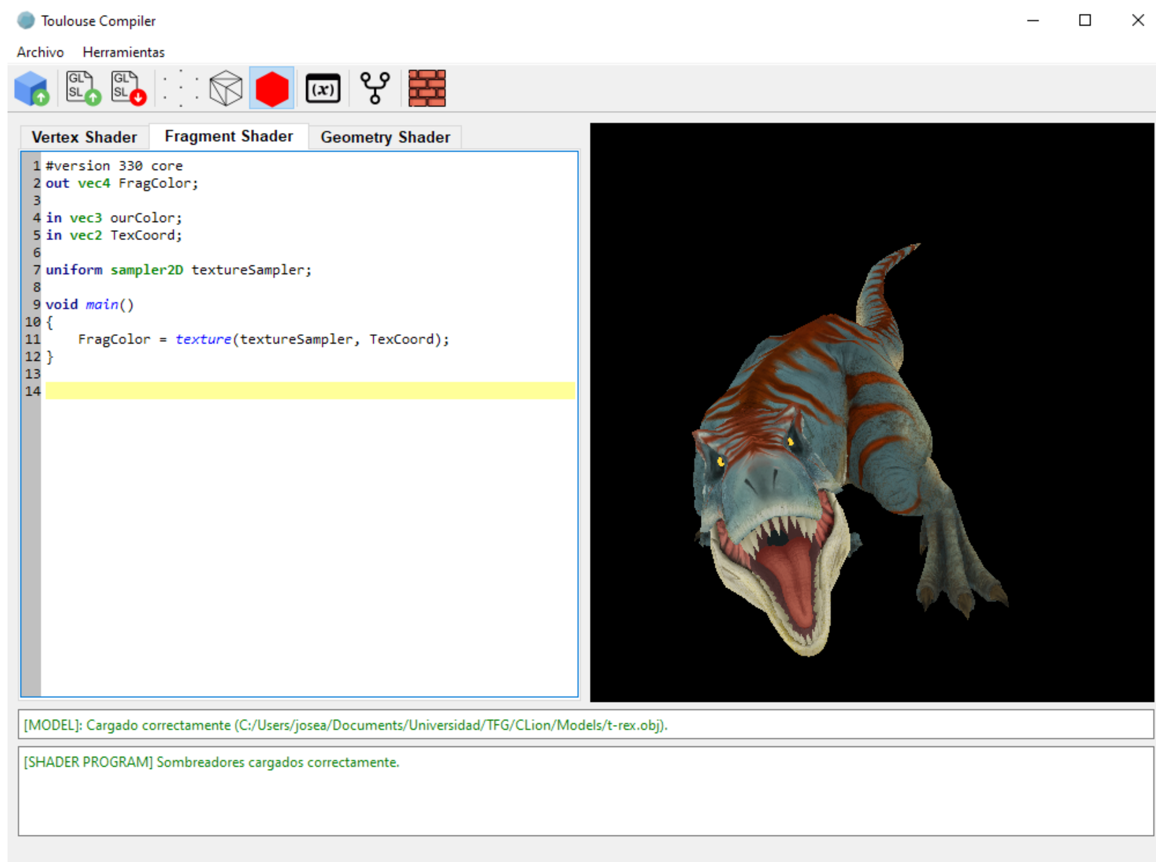


Ilustración 70 Resultado de la prueba de texturas (fuente: elaboración propia)

Prueba de geometry shader

El objetivo de esta prueba es evaluar la funcionalidad del geometry shader. Para ello voy a escribir un geometry shader el cual simula el efecto de explosión del modelo, haciendo uso del uniform definido por la aplicación time (*Ilustración 71*). Es necesaria también la defininición de un vertex shader (*Ilustración 72*) y geometry shader (*Ilustración 73*).

```
1 #version 330 core
2 layout (triangles) in;
3 layout (triangle_strip, max_vertices = 3) out;
4
5 in VS_OUT {
6     vec2 texCoords;
7 } gs_in[];
8
9 out vec2 TexCoords;
10 uniform float time;
11
12 vec4 explode(vec4 position, vec3 normal){
13     float magnitude = 2.0;
14     vec3 direction = normal * ((sin(time) + 1.0) / 2.0) * magnitude;
15     return position + vec4(direction, 0.0);
16 }
17
18 vec3 GetNormal(){
19    vec3 a = vec3(gl_in[0].gl_Position) - vec3(gl_in[1].gl_Position);
20    vec3 b = vec3(gl_in[2].gl_Position) - vec3(gl_in[1].gl_Position);
21    return normalize(cross(a, b));
22 }
23
24 void main() {
25     vec3 normal = GetNormal();
26     gl_Position = explode(gl_in[0].gl_Position, normal);
27     TexCoords = gs_in[0].texCoords;
28     EmitVertex();
29     gl_Position = explode(gl_in[1].gl_Position, normal);
30     TexCoords = gs_in[1].texCoords;
31     EmitVertex();
32     gl_Position = explode(gl_in[2].gl_Position, normal);
33     TexCoords = gs_in[2].texCoords;
34     EmitVertex();
35     EndPrimitive();
36 }
```

Ilustración 71 Geometry shader (fuente: elaboración propia)

```
1 #version 330 core
2 layout (location = 0) in vec3 aPos;
3 layout (location = 1) in vec3 aColor;
4 layout (location = 2) in vec2 aTexCoord;
5
6 out vec3 ourColor;
7 out vec2 TexCoord;
8
9 uniform mat4 model;
10 uniform mat4 view;
11 uniform mat4 projection;
12
13 out VS_OUT {
14     vec2 texCoords;
15 } vs_out;
16
17 void main()
18 {
19     gl_Position = projection * view * model * vec4(aPos, 1.0);
20     ourColor = aColor;
21     TexCoord = aTexCoord;
22     vs_out.texCoords = aTexCoord;
23 }
```

Ilustración 72 Vertex shader para la prueba del geometry shader (fuente: elaboración propia)

```
1 #version 330 core
2 out vec4 FragColor;
3
4 in vec3 ourColor;
5 in vec2 TexCoords;
6
7 uniform sampler2D ourTexture;
8
9 void main()
10 {
11     FragColor = texture(ourTexture, TexCoords);
12 }
```

Ilustración 73 Fragment shader para la prueba del geometry shader (fuente: elaboración propia)

Una vez definidos todos los shaders, obtenemos el efecto de explosión esperado. Se pueden ver tres frames de cómo varía la geometría del modelo a lo largo del tiempo simulando el efecto descrito. Se pueden ver en la *Ilustración 74, Ilustración 75* e *Ilustración 76*.

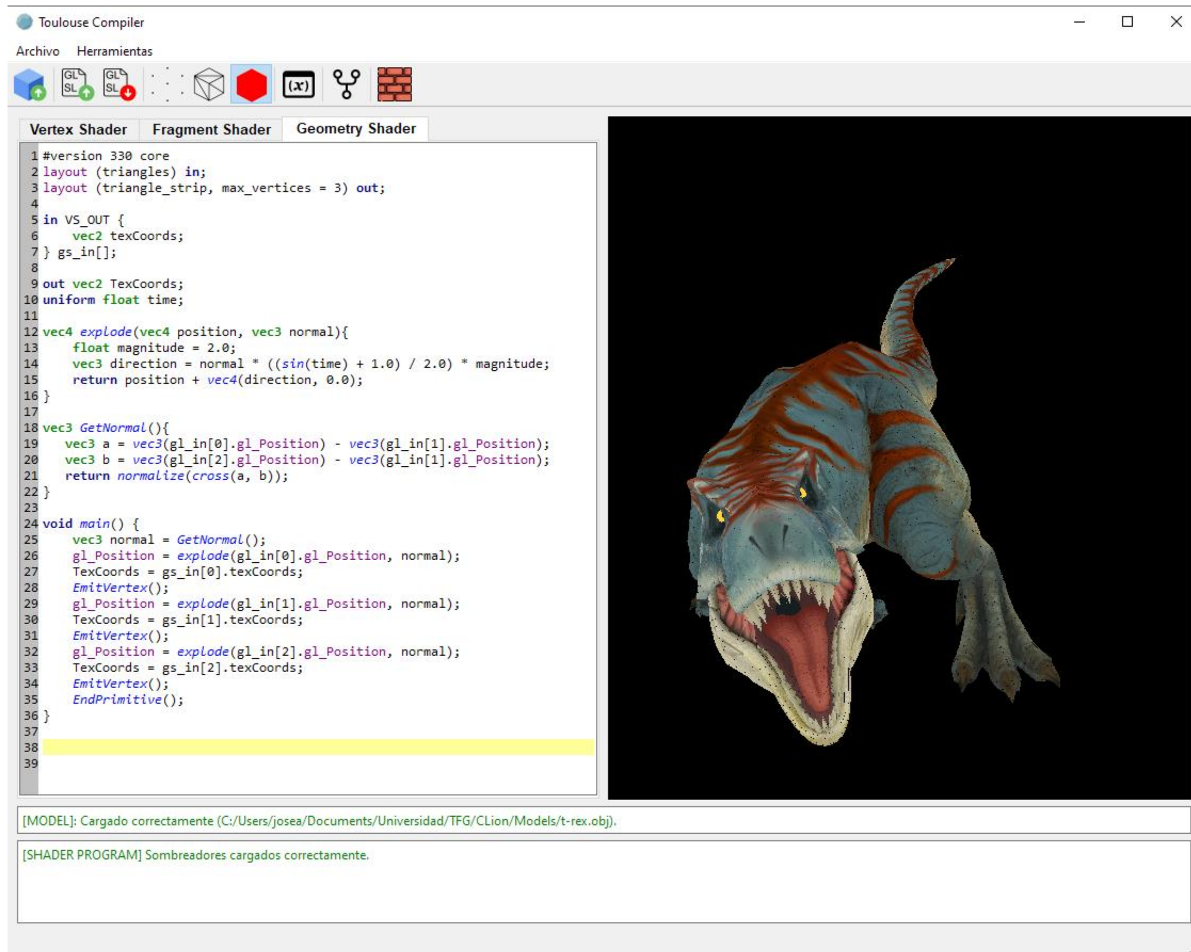


Ilustración 74 Resultado del efecto explosión haciendo uso del geometry shader frame 1 (fuente: elaboración propia)

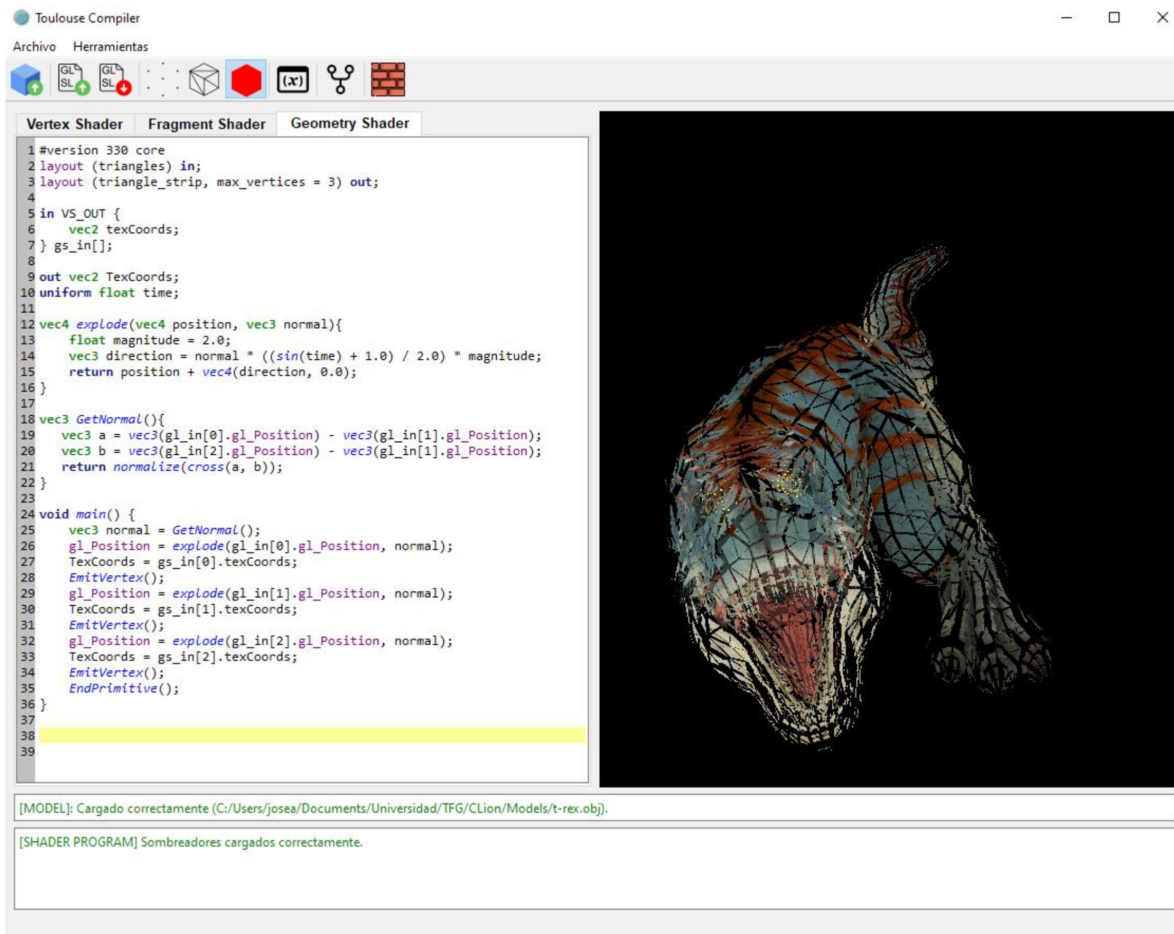


Ilustración 75 Resultado del efecto explosión haciendo uso del geometry shader frame 2 (fuente: elaboración propia)

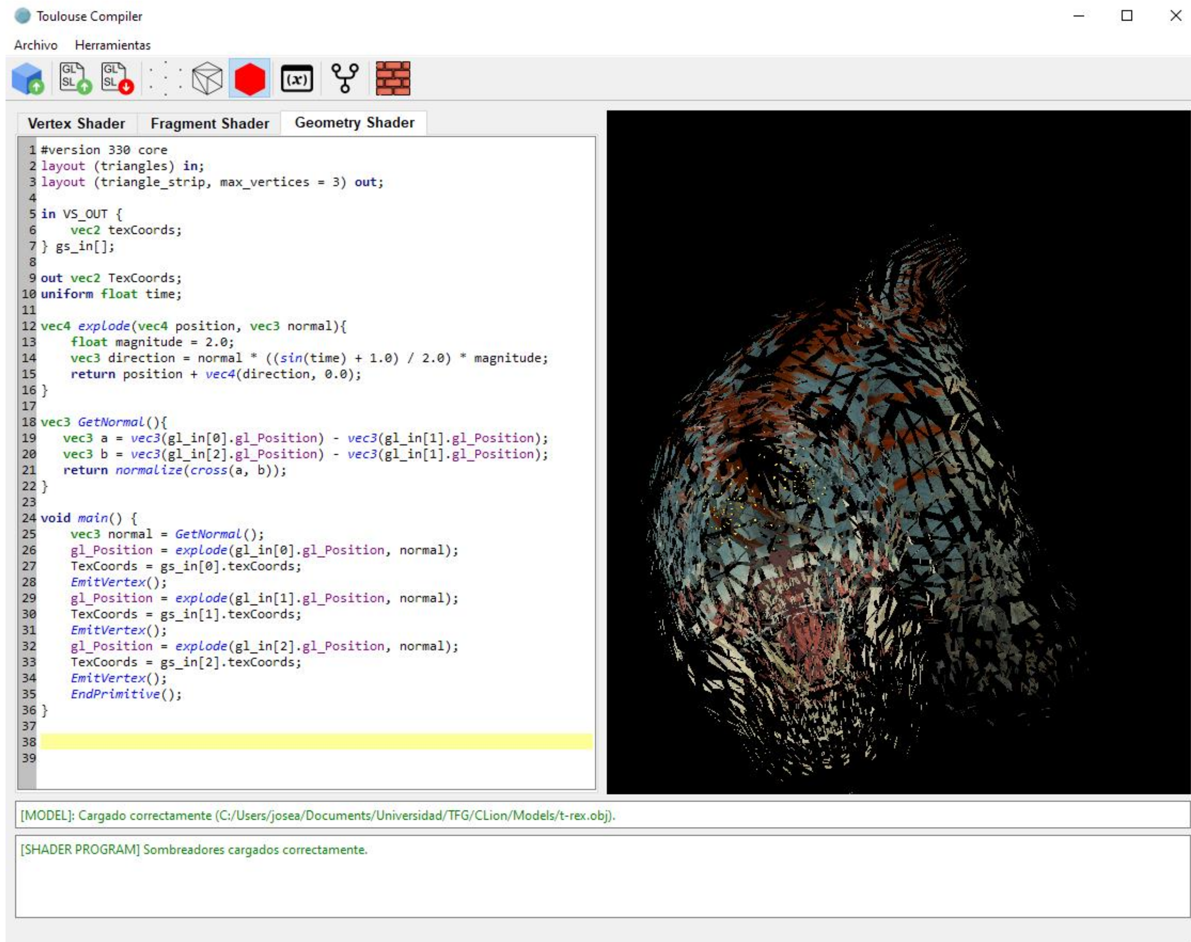


Ilustración 76 Resultado del efecto explosión haciendo uso del geometry shader frame 3 (fuente: elaboración propia)

4 CONCLUSIONES Y TRABAJOS FUTUROS

Este Trabajo de Fin de Grado (TFG) ha supuesto un aprendizaje significativo en el campo del desarrollo de aplicaciones gráficas utilizando OpenGL. A través del diseño y la implementación de diversas funcionalidades, se ha profundizado en los elementos esenciales que componen una aplicación gráfica avanzada. Los conocimientos adquiridos no solo abarcan la teoría, sino también la práctica de implementar soluciones eficientes y flexibles para la gestión de geometría, shaders, texturas, y la interacción en tiempo real con la escena renderizada.

La realización del Trabajo de Fin de Grado ha estado enfocada al desarrollo de los objetivos que estaban fijados. Durante su realización se me han ocurrido múltiples áreas de mejora y expansión que podrían ser abordadas en futuros trabajos. A continuación, se presentan algunas de las posibles líneas de desarrollo:

- **Carga múltiple de texturas:** Sería un trabajo interesante puesto que, debido a la limitación de tiempo, la aplicación solo es capaz de cargar una textura. Con la carga de múltiples texturas se podrían realizar técnicas avanzadas como el “bump mapping” para añadir más realismo a los diferentes modelos.
- **Carga de varios modelos simultáneamente:** La aplicación está restringida a la carga de un solo modelo. Sería interesante cargar varios modelos aplicando a cada uno shaders diferentes.
- **Rendering multipasada:** El rendering que se hace en la aplicación actualmente se realiza en una sola pasada, lo cual limita las técnicas y efectos visuales que se pueden implementar eficazmente.
- **Aplicación de transformaciones a los modelos:** La aplicación tiene implementados mecanismos como translación, rotación y escalado que se aplican a la matriz de modelado. Sin embargo, no se hace uso de ello puesto que no lo he visto necesario, pero podría ser interesante para añadir flexibilidad a la aplicación.

- **Añadir compatibilidad para el sistema operativo macOS:** Dado que el objetivo era añadir compatibilidad para sistemas operativos Windows y GNU/Linux, no se ha abordado la posibilidad de realizar la compatibilidad con el sistema operativo macOS. Esto no sería tarea muy difícil, puesto que tanto el IDE CLion como el framework Qt son compatibles con este sistema.
- **Añadir técnicas como el cube mapping para cambiar el fondo:** El fondo de la aplicación es un fondo negro. Al añadir técnicas como el cube mapping podría mejorar significativamente el aspecto de la aplicación.

5 APÉNDICES

5.1 Guía original del Trabajo Fin de Título

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior de Jaén

PROPUESTA DE TRABAJO DE FIN DE GRADO
GRADO EN: Grado en Ingeniería Informática **ESPECIALIDAD:** General **MENCIÓN:** General
TÍTULO DEL TRABAJO: Toulouse Compiler 2.0: Mejora de una aplicación de escritorio para prueba de shaders GLSL **Idioma:** Castellano
DESCRIPCION CORTA DEL TFG: Partiendo de un trabajo ya defendido en el curso 2019-2020, en este TFG se plantea la mejora de la aplicación desarrollada, incluyendo nuevas características, depuración de errores y refactorización del código, haciéndola portable al menos a los sistemas Windows y GNU/Linux.

DATOS DEL TRABAJO FIN DE GRADO:

Modalidad del TFG

Proyecto de Ingeniería

Tipo de TFG

- ○ **TFG General**
- ◉ **TFG Específico**

TFG en equipo

- ☐ **TFG en Equipo**

(Debe ajuntarse memoria justificativa)

Número máximo de estudiantes

1

TUTOR DEL TRABAJO FIN DE GRADO:

Tutor/a

Nombre:

ANGEL LUIS GARCIA FERNANDEZ

Departamento:

Informática

A. Conocimiento:

LENGUAJES Y SISTEMAS INFORMÁTICOS

- ☑ **Este TFG tiene un segundo tutor**
  - ◉ **El tutor es de la Escuela**
  - ○ **El tutor es ajeno al Centro**

Segundo Tutor

FRANCISCO DE ASIS CONDE RODRIGUEZ

DATOS DEL ALUMNO ASIGNADO

Alumno/a asignado/a

Nombre:
José Antonio Montalbán Salvador
DNI:
26506732Z
Dirección:
Avda. de España, 26. 23700 Linares
Teléfono:
601095168
Correo-E:
jams0022@red.ujaen.es

CONOCIMIENTOS/REQUISITOS PREVIOS RECOMENDADOS

- Conocimientos avanzados de C++, OpenGL y GLSL.
- Se recomienda experiencia previa con alguna biblioteca de diseño de interfaces de usuario (GUI) multiplataforma, como DearImgui, QT, wxWidgets o GTK.
- Haber cursado la asignatura de Programación de Aplicaciones Gráficas

OBJETIVOS DEL TFG:

- Poner en práctica los conocimientos y destrezas adquiridos durante los estudios de grado.
- Demostrar el nivel de madurez para el diseño e implementación de software.
- Producir un software funcional y eficiente, que pueda instalarse y usarse en computadoras de usuarios finales.
- Mejorar la aplicación existente, refactorizando el código, eliminando errores y añadiendo nuevas funcionalidades, tales como carga de modelos personalizados (como mínimo en formato OBJ), soporte para texturas personalizadas, soporte para subrutinas, múltiples luces configurables, etcétera.

METODOLOGÍA A DESARROLLAR:

Se podrá aplicar tanto una metodología ágil basada en iteraciones como una metodología tradicional en cascada.
Así mismo, durante todo el proceso de desarrollo se irá elaborando la correspondiente documentación.

DOCUMENTOS Y FORMATOS DE ENTREGA:

Toda la información se entregará en formato digital, constando de:

- Código fuente completo de la aplicación.
- Vídeo demostrativo del uso de la aplicación.
- Memoria del trabajo, incluyendo toda la documentación generada durante el desarrollo del mismo, manuales de instalación y uso de la aplicación, y anexos.

DOCUMENTOS ENTREGADOS:

Documentos entregados

☑ **Solicitud de propuesta de TFG cumplimenada y firmada.**

5.2 Instalación y configuración de la aplicación

En este apartado se va a mostrar el proceso de instalación y configuración de la aplicación para los sistemas operativos Windows y Linux.

5.2.1 Instalación en el sistema operativo Windows.

Instrucciones para la instalación en el sistema operativo Windows:

1. Descargar la versión de la aplicación Toulouse Compiler 2.0 para Windows.
2. Una vez descargado, nos encontraremos con un archivo .zip el cual tendremos que descomprimir. Al descomprimirlo nos encontraremos con una carpeta la cual tiene la aplicación, así como las bibliotecas para ejecutarla.

3. Mover la carpeta de la aplicación a un lugar deseado del sistema de archivos.
4. Finalmente, para usar la aplicación bastará con ejecutar el archivo .EXE que está contenido en dicha carpeta, que tiene un dodecaedro azul como icono.

El proceso de desinstalación es muy sencillo, ya que es una aplicación portable, es decir, la carpeta contiene todos los archivos necesarios para su ejecución, por lo que simplemente bastará con eliminar la carpeta de la aplicación.

5.2.2 Instalación en sistemas GNU/Linux.

Instrucciones para la instalación en el sistema operativo Linux:

1. Instalar las librerías GLM, GLFW, GLEW, Assimp y Qt la orden: "**sudo apt-get install libglm-dev libglfw3-dev libglew-dev libassimp-dev qt6-base-dev qt6-tools-dev**".
2. Descargar el archivo ejecutable de la aplicación.
3. Mover el archivo a un lugar deseado y ejecutarlo.

Para su desinstalación bastará con eliminar las librerías con la orden: "**sudo apt-get remove --purge libglm-dev libglfw3-dev libglew-dev libassimp-dev qt6-base-dev qt6-tools-dev**", además de eliminar el archivo ejecutable de la aplicación.

5.3 Manuales de usuario

En este apartado se va a presentar las instrucciones a seguir para poder utilizar la aplicación de manera adecuada probando todas las funcionalidades implementadas en el proceso de desarrollo de la misma. Se dividirá en diferentes subapartados atendiendo a la funcionalidad.

5.3.1 Cargar un modelo

Hay tres formas para cargar un modelo. La primera sería desde el menú de herramientas, a través del icono correspondiente, la segunda forma sería mediante el menú archivo→Cargar modelo y una última forma, para usuarios más avanzados, con la combinación de teclas Ctrl+M. En la *Ilustración 77* se muestran de manera gráfica las opciones comentadas.

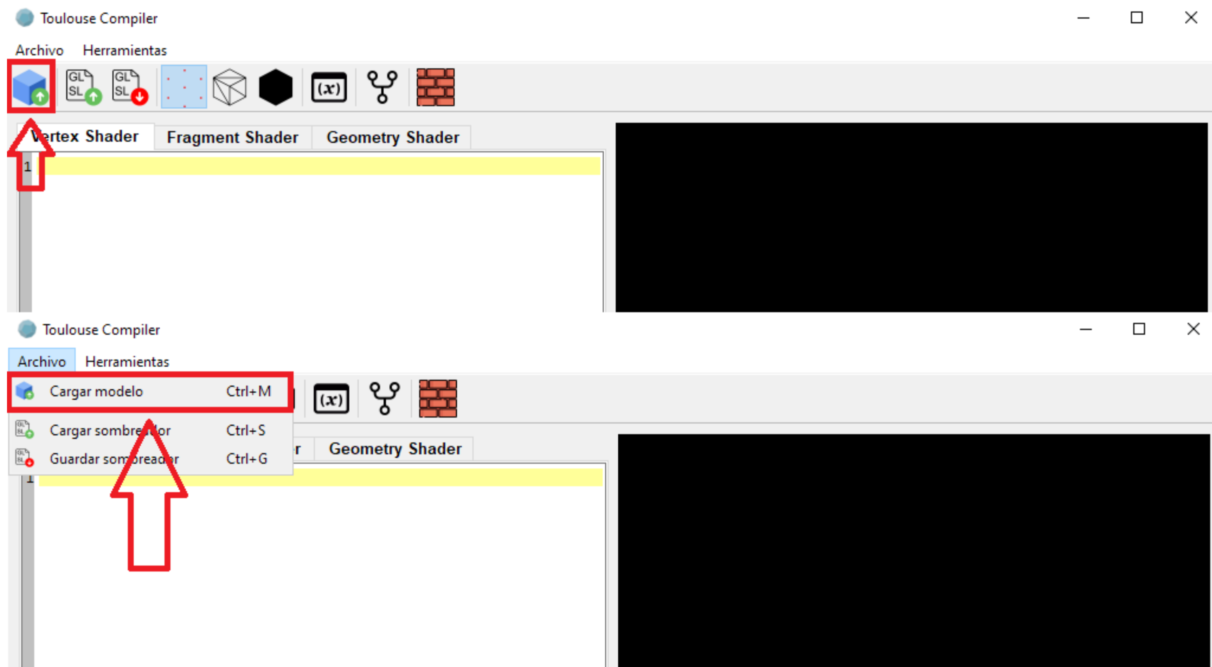


Ilustración 77 Formas de cargar un modelo (fuente: elaboración propia)

Una vez seleccionada la opción de cargar un modelo se nos abrirá el sistema de archivos y tendremos que navegar por él hasta encontrar el modelo que se quiera cargar (*Ilustración 78*).

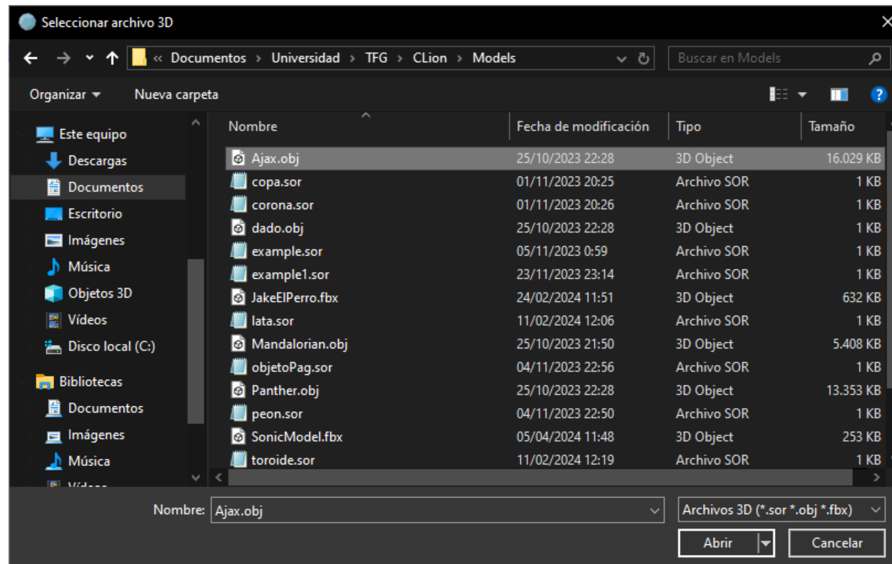


Ilustración 78 Cargando un modelo (fuente: elaboración propia)

Por defecto, solo permitirá cargar archivos con extensión .OBJ, .FBX y .SOR, pero lo podremos modificar desde el mismo navegador para cargar otras extensiones compatibles con Assimp.

Finalmente, al cargar el modelo se podrán dar dos casos. El primero es que el modelo se haya cargado correctamente y se mostrará en la barra de estado mediante un mensaje de color verde indicándolo (*Ilustración 79*). El segundo caso es que se haya producido algún error y se mostrará en la barra de estado mediante un mensaje de color rojo (*Ilustración 80*).

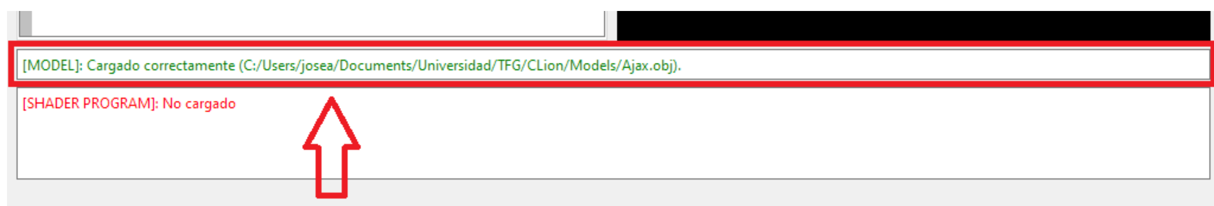


Ilustración 79 Carga de modelo exitosa (fuente: elaboración propia)

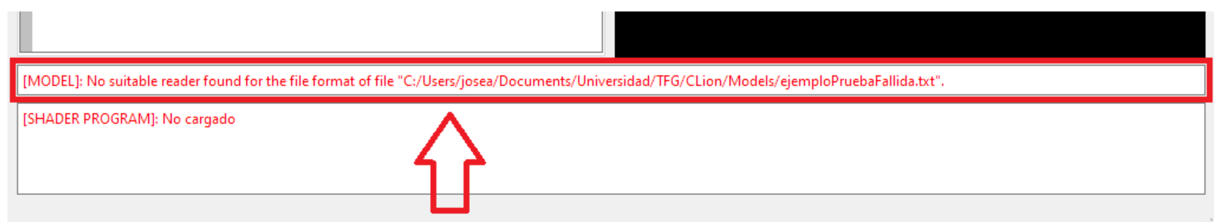


Ilustración 80 Carga de modelo fallida (fuente: elaboración propia)

Es posible cargar sólidos de revolución creados por nosotros mismos. Para ello, se deberá crear un archivo de texto con extensión .SOR que deberá contener el siguiente contenido:

- Una línea para indicar las subdivisiones verticales, de la forma "slices=(valor entero)".
- Una línea para indicar cuántas veces aplicar el algoritmo de suavizado del perfil definido, de la forma "subdivisions=(valor entero)".
- Una serie de líneas indicando los vértices en orden. Estos podrán tener la forma "sv=(coordenada x), (coordenada y)" para indicar que el vértice sea excluido del algoritmo de subdivisión. De lo contrario, deberá tener la forma "v =(coordenada x), (coordenada y)"".

En la *Ilustración 81* se puede ver un ejemplo del formato, donde se aplica cuatro veces el algoritmo de subdivisión del perfil, se realizan 36 subdivisiones longitudinales, tiene tres vértices a los cuales se aplica el algoritmo de subdivisión y doce vértices a los cuales no.

```
subdivisions=4
slices=36
v=0,0
v=1.4,0.05
sv=2.4,0.2
v=2.5,0.27
sv=2.6,0.4
sv=3.13,0.77
sv=3.13,5.5
sv=3.13,10.25
sv=2.54,11.5
sv=2.55,11.7
sv=2.6,11.7
sv=2.6,11.9
sv=2.5,11.9
sv=2.5,11.7
sv=0,11.7
```

Ilustración 81 Ejemplo del formato .SOR

5.3.2 Cargar y guardar un shader

La mecánica de esta funcionalidad es similar a la carga de un modelo. Tanto la carga como el guardado se pueden realizar desde la barra de herramientas, así como desde el menú archivo o con los atajos de teclado Ctrl+S y Ctrl+G respectivamente (*Ilustración 82*).

La carga de un shader se realizará de manera análoga a la carga de modelo: se navegará a través del sistema de archivos hasta encontrar el shader/shaders deseados. Para incorporarlo en el editor correspondiente, el nombre del shader deberá tener el siguiente formato:

- **"Nombre-vs.glsl"** para cargar el código fuente del vertex shader.
- **"Nombre-gs.glsl"** para cargar el código fuente del geometry shader.
- **"Nombre-fs.glsl"** para cargar el código fuente del fragment shader.

Donde **"Nombre"** podrá ser personalizado por el usuario. En la *Ilustración 83* se puede ver un ejemplo. Además, se pueden seleccionar varios archivos para cargar los distintos shaders de una sola vez.

Una vez cargados los shaders, se abrirá cada uno en su editor correspondiente para ser compilados y enlazados (mostrando un mensaje de éxito de color verde si son correctos y no tienen errores) en el shader program, además de permitir su edición en tiempo real para visualizar el resultado (*Ilustración 84*).

Si algún shader no se cargara correctamente o presentará algún error se notifica en la barra de estado del shader en color rojo (*Ilustración 85*).

La opción guardar shaders guardará todo el contenido de los editores siguiendo el mismo formato que se ha indicado según el tipo de shader. Una vez seleccionada la opción guardar se deberá especificar un nombre (*Ilustración 86*), y posteriormente seleccionar el directorio dentro del sistema de archivos donde se quieran guardar los shaders escritos.

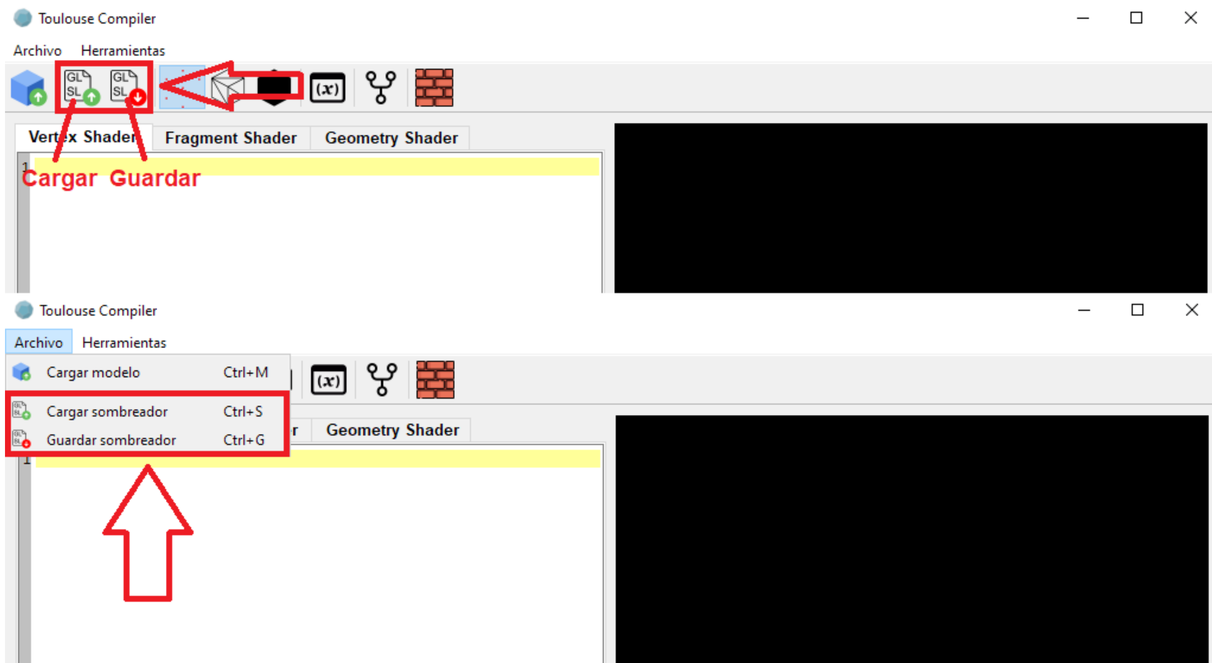


Ilustración 82 Formas de cargar y guardar shaders (fuente: elaboración propia)

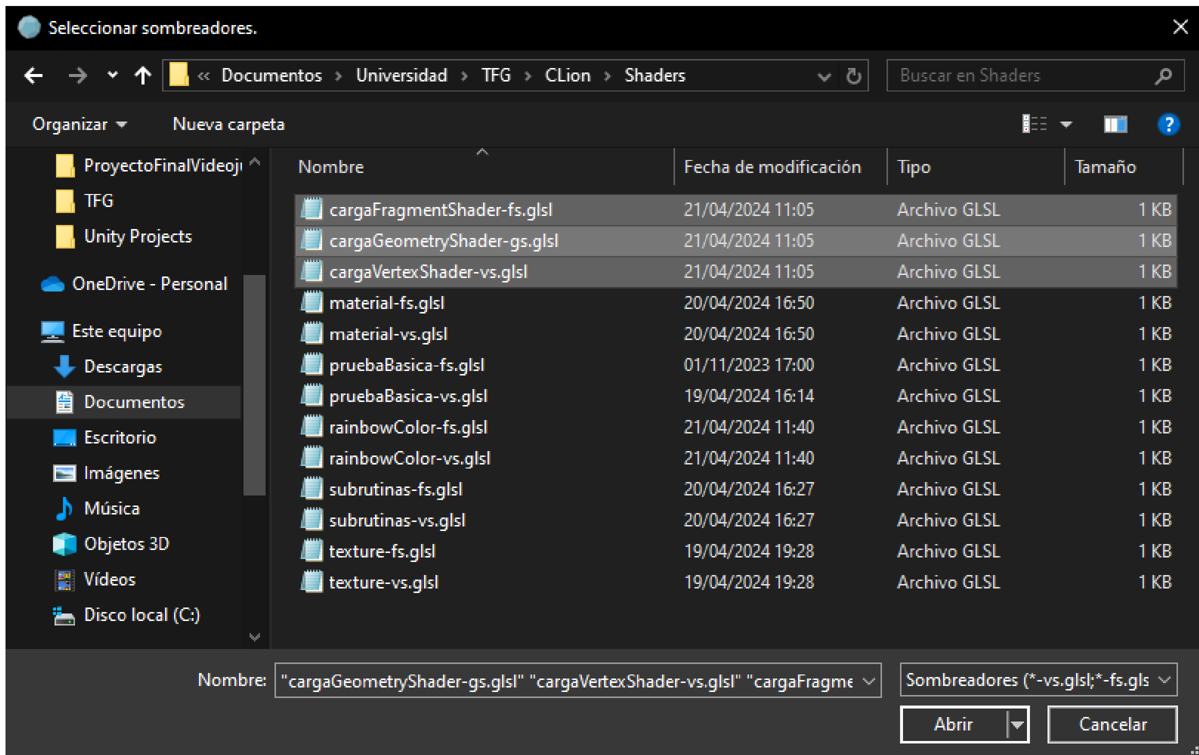


Ilustración 83 Formato para la carga de shaders y carga múltiple (fuente: elaboración propia)

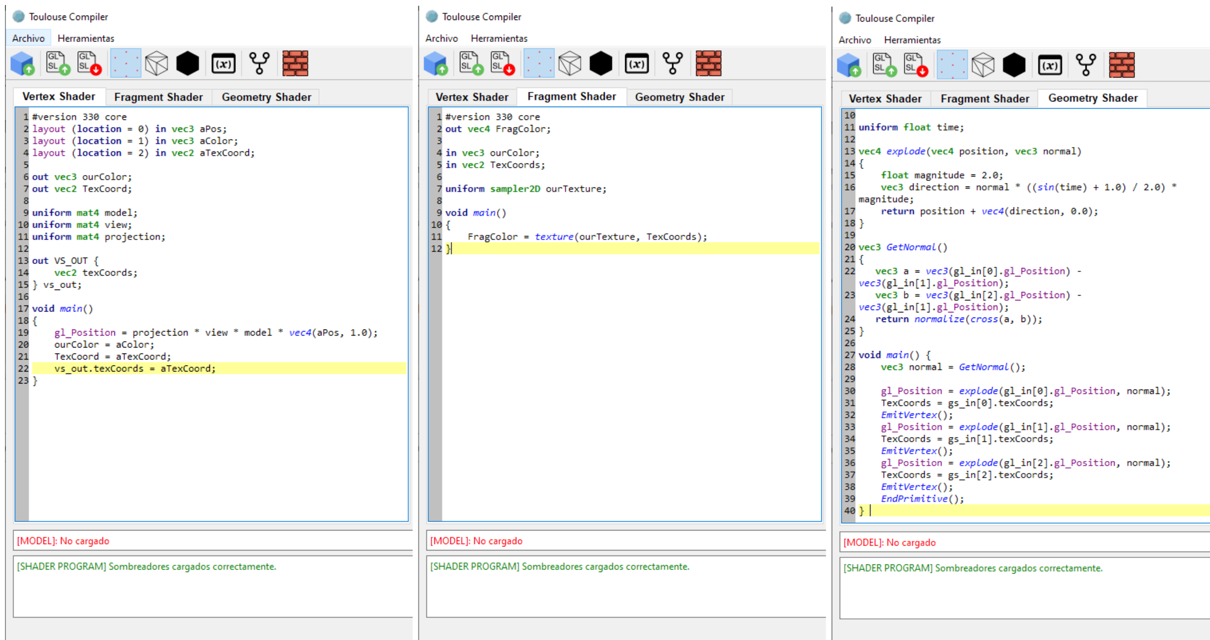


Ilustración 84 Editores de shaders una vez cargados (fuente: elaboración propia)

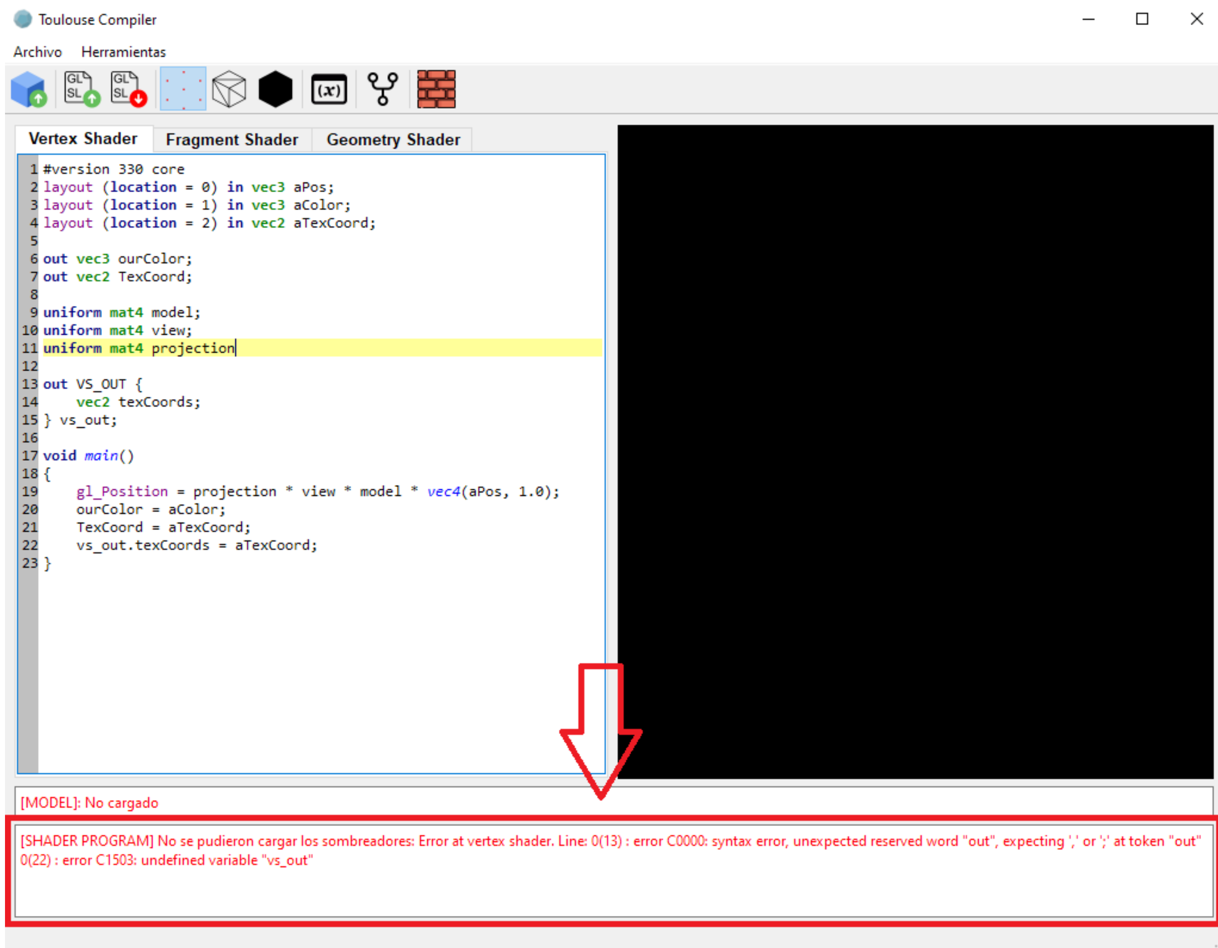


Ilustración 85 Error en shader (fuente: elaboración propia)

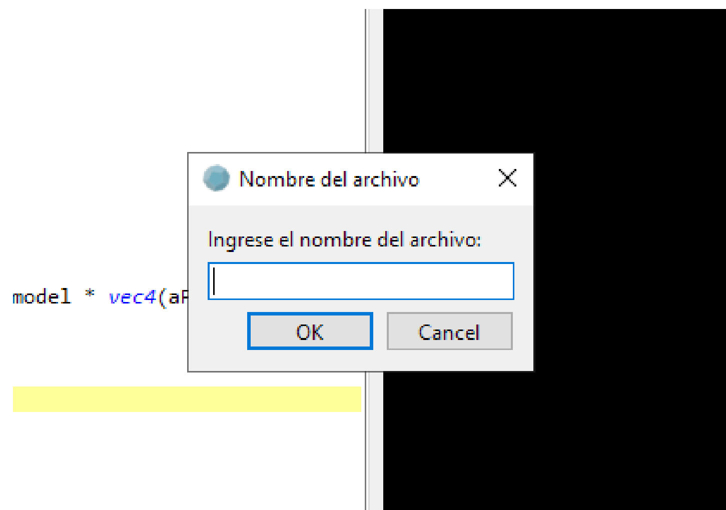


Ilustración 86 Opción para guardar un shader (fuente: elaboración propia)

5.3.3 Alternar entre modos de visualización

Hay tres formas para indicar el modo de visualización deseado. El primero sería a través de la barra de herramientas, donde aparecen tres iconos de un cubo: el primero representa los vértices del mismo (nube de puntos), el segundo las aristas (líneas) y el tercero los polígonos (triángulos). La segunda forma para definirlo sería a través del menú herramientas→Modo del polígono y seleccionamos el modo deseado. La última opción sería empleando los atajos de teclado Ctrl+P, Ctrl+L y Ctrl+F para alternar entre los modos nube de puntos, líneas y triángulos respectivamente (*Ilustración 87*).

En la *Ilustración 88* se pueden ver los resultados tras aplicar el modo de visualización de nube de puntos a un modelo, en el cual se pueden ver los vértices que lo componen. En la *Ilustración 89* se emplea el modo de visualización de líneas al mismo modelo, y se pueden ver las uniones entre vértices. Finalmente, en la *Ilustración 90* se usa el último modo de visualización, el cual sería modo triángulos o relleno, que da la sensación de que es un modelo con volumen, aunque solo esté representado por la frontera.

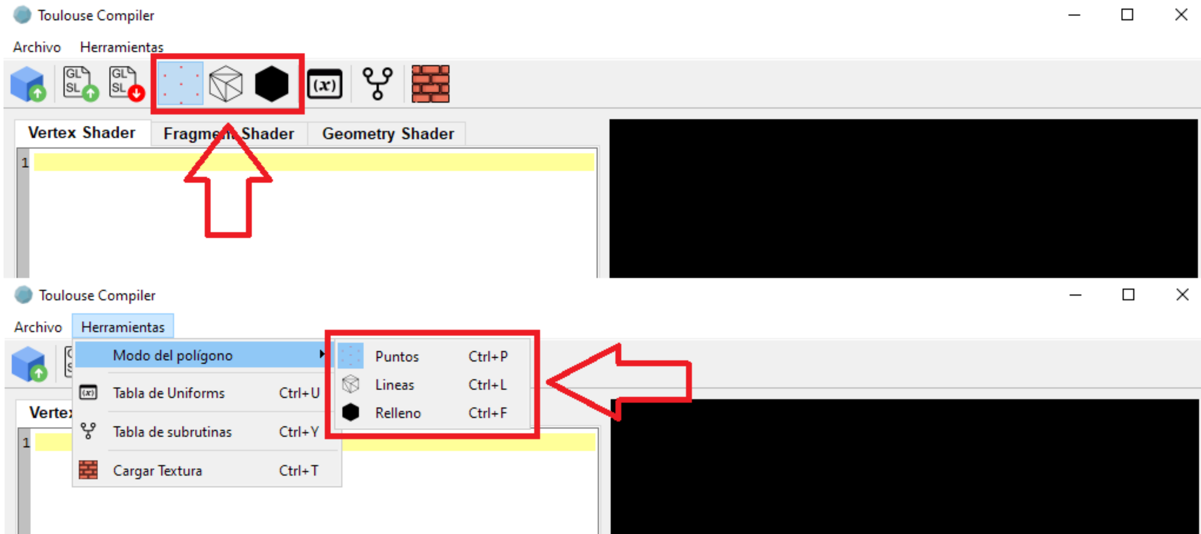


Ilustración 87 Formas de alternar entre modos de visualización (fuente: elaboración propia)

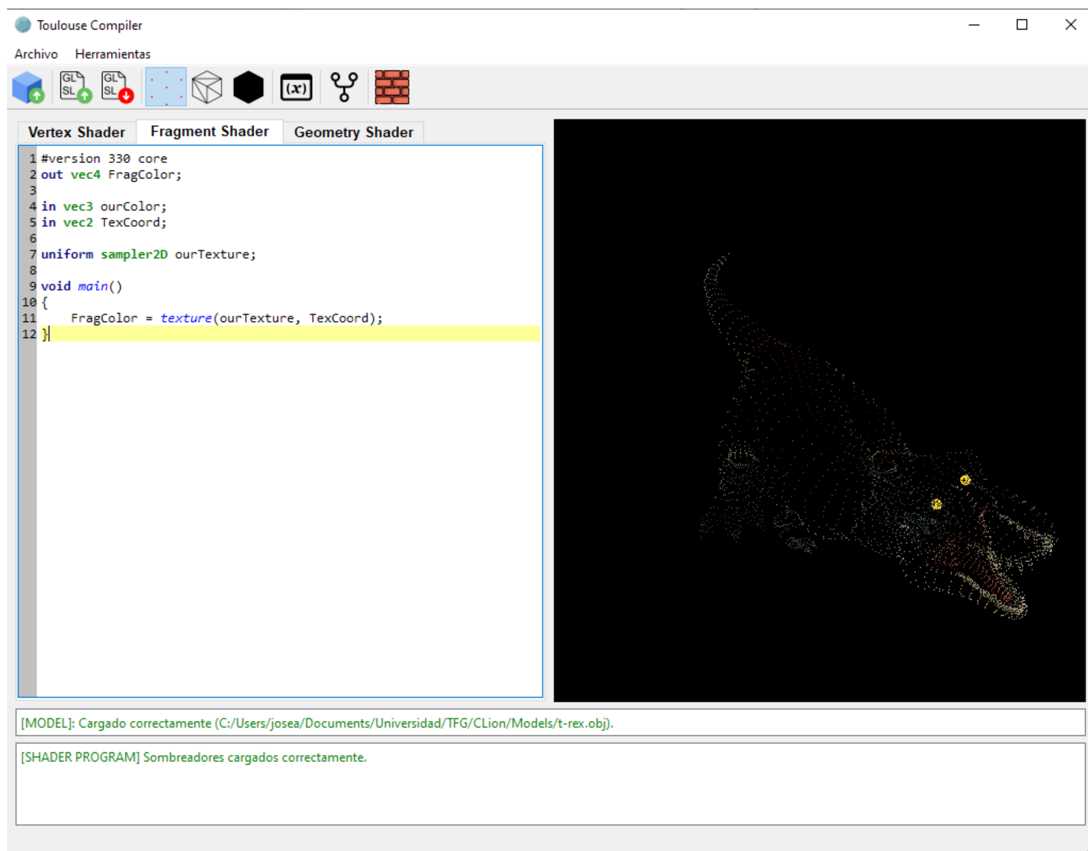


Ilustración 88 Modo de visualización nube de puntos con textura (fuente: elaboración propia)

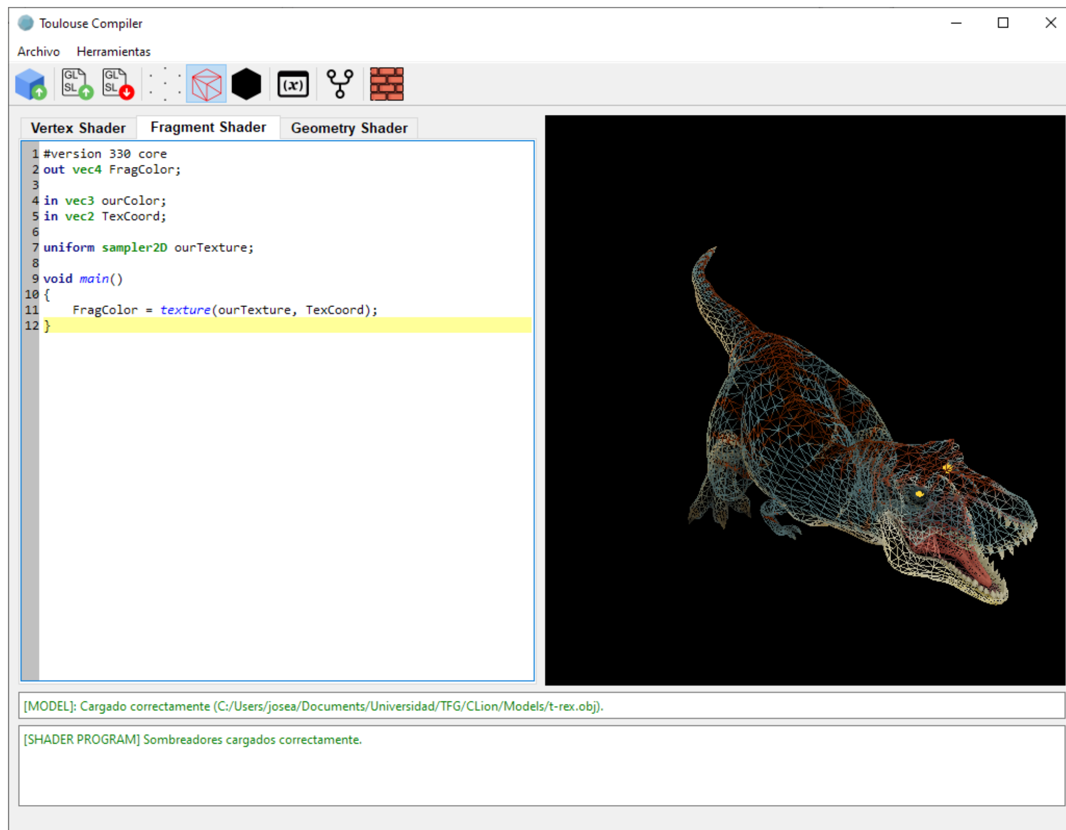


Ilustración 89 Modo de visualización líneas con textura (fuente: elaboración propia)

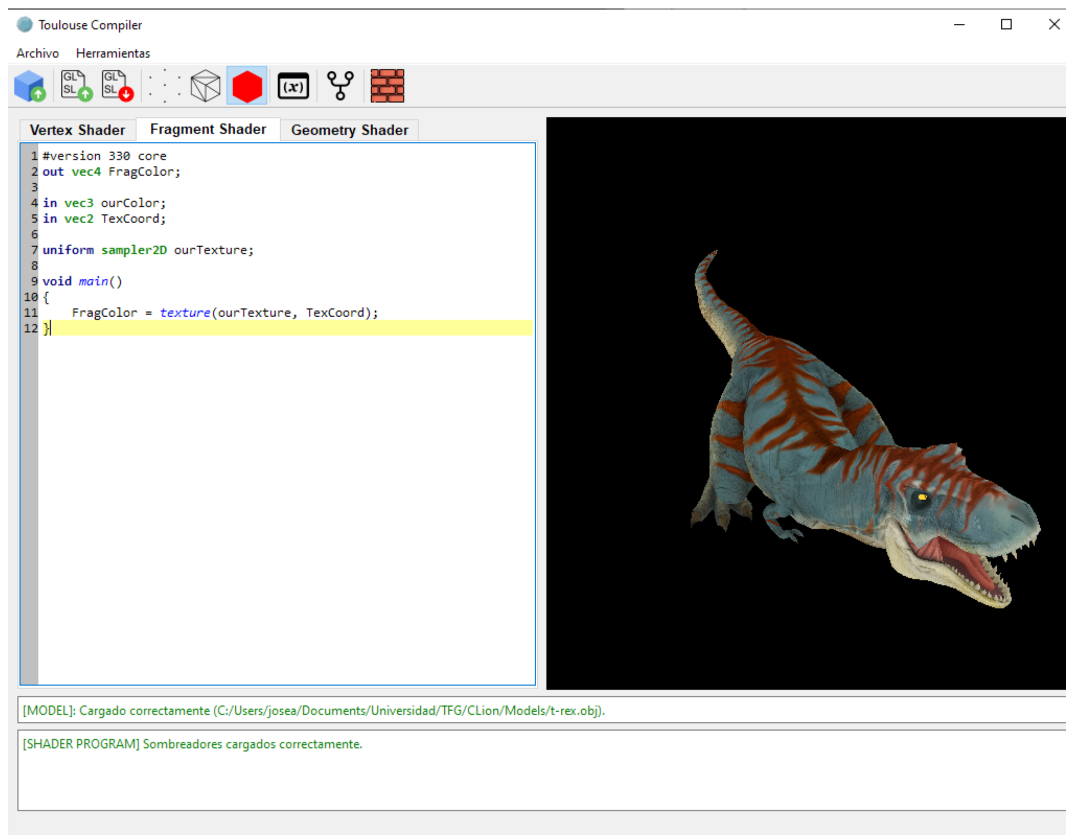


Ilustración 90 Modo de visualización triángulos con textura (fuente: elaboración propia)

5.3.4 Gestión de uniforms

Los uniforms se gestionan a través de una tabla. Hay tres formas para acceder a ella: la primera es a través de su correspondiente icono en la barra de herramientas, la segunda desde el menú herramientas→Tabla de uniforms, y la tercera mediante la combinación de teclas Ctrl+U (*Ilustración 91*).

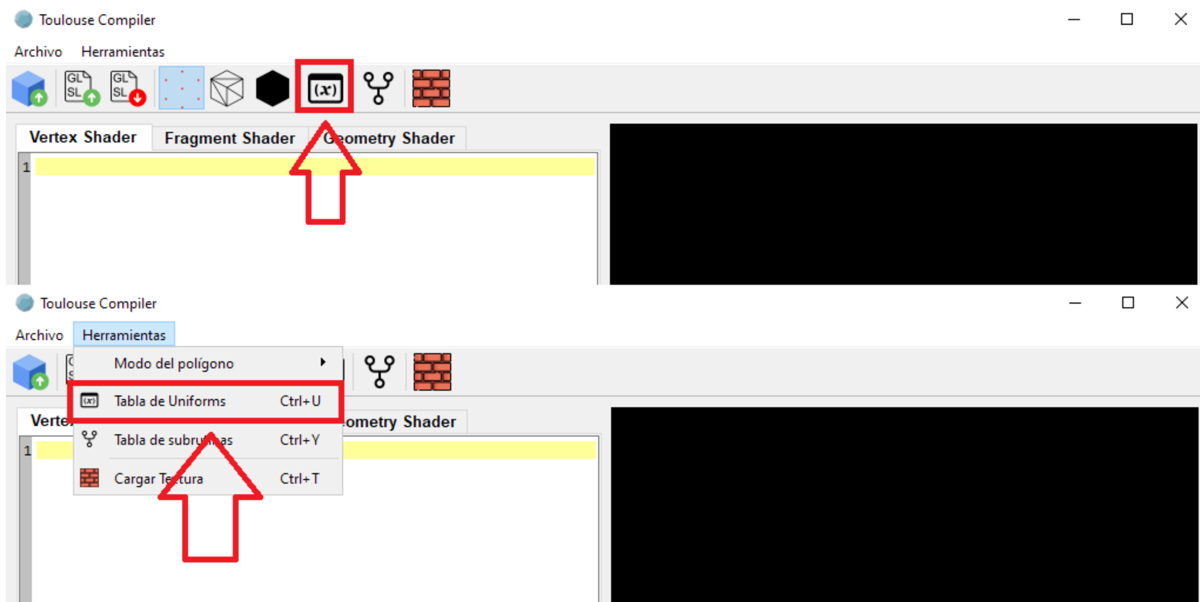


Ilustración 91 Formas de abrir tabla de uniforms (fuente: elaboración propia)

Una vez seleccionada la opción, se nos abrirá una tabla donde podremos gestionar los uniforms indicando el tipo, nombre y valor de cada uno (*Ilustración 92*).

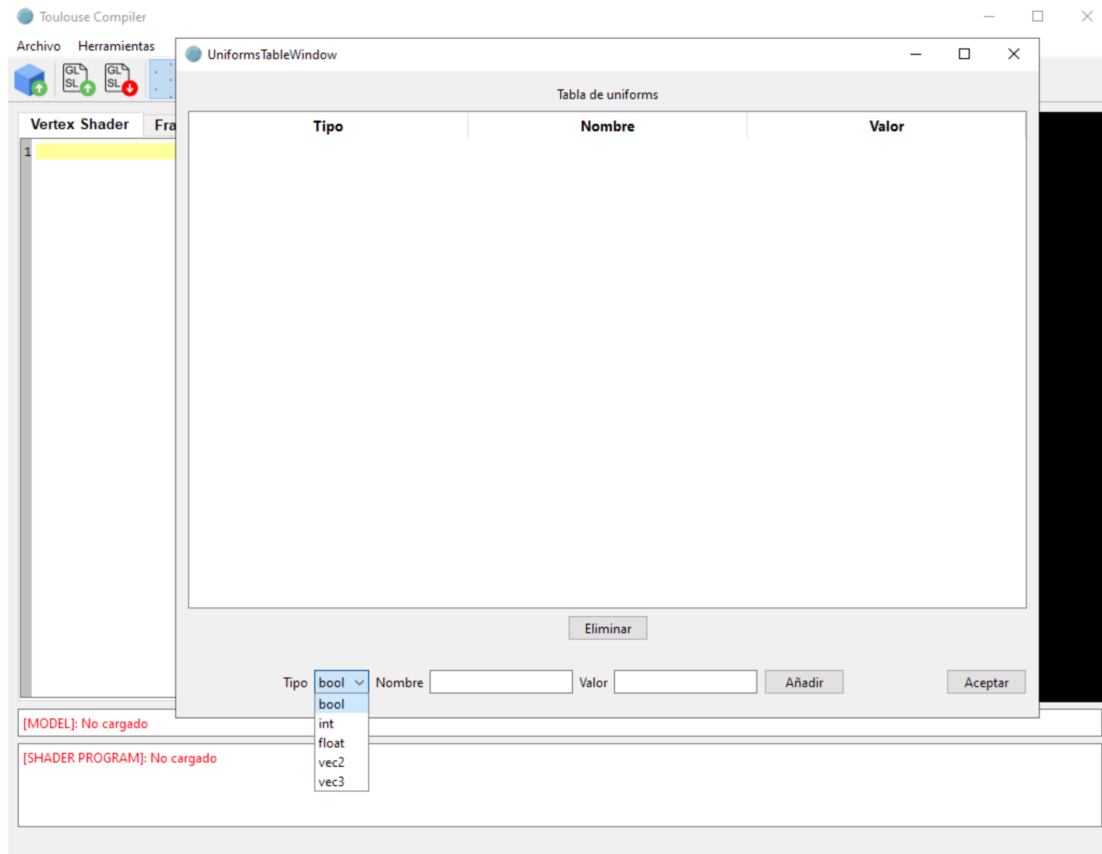


Ilustración 92 Tabla para la gestión de uniforms (fuente: elaboración propia)

Los tipos de uniforms permitidos en la aplicación son:

- **Bool:** deberá tener un valor true o false.
- **Int:** deberá tener un valor numérico entero.
- **Float:** deberá tener un valor numérico real donde el separador de la parte decimal será el punto.
- **Vec2:** vector de dos números reales que deberá seguir el formato "a,b" Ejemplo: "1.17, 10.5".
- **Vec3:** vector de tres números reales que deberá seguir el formato "a,b,c". Ejemplo: "2.23, 5.7, 4.3".

El campo **nombre** tiene una serie de restricciones y son:

- No puede comenzar por un número.
- No puede ser el mismo que alguno ya existente en la tabla.

- No puede contener espacios.
- No puede estar vacío.

Si se diera algunas de estas restricciones a la hora de añadir un uniform, se notificaría mediante un cuadro de diálogo informando sobre el error cometido (*Ilustración 93*).

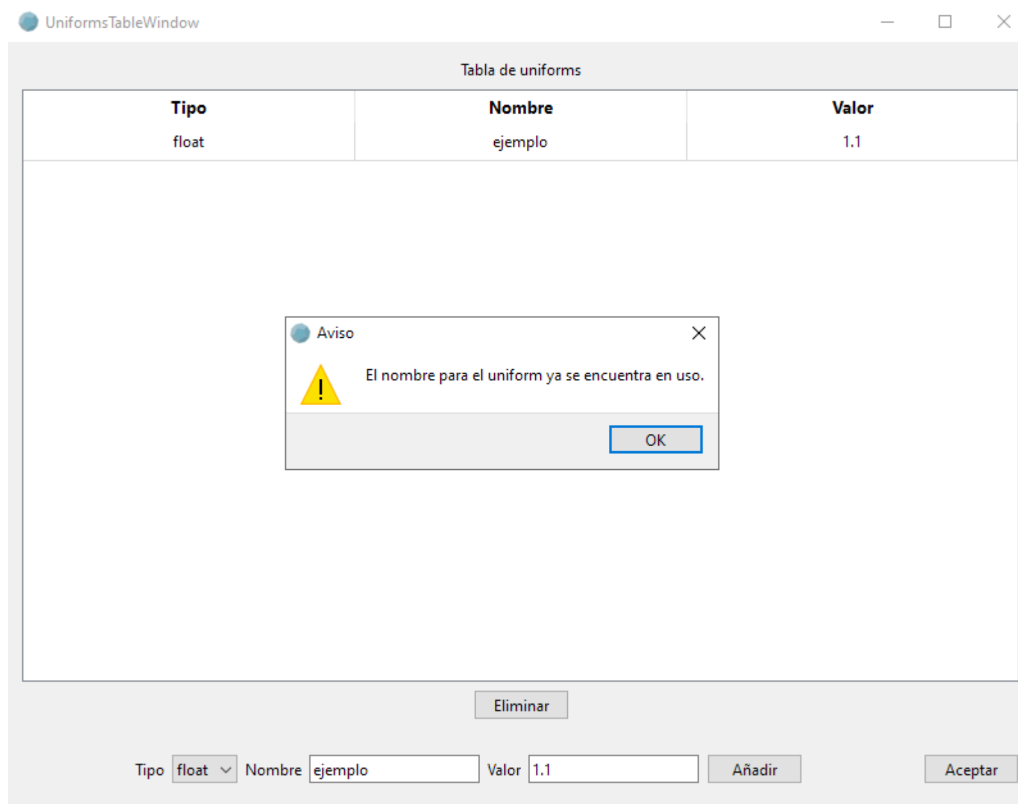


Ilustración 93 Advertencia de error a la hora de añadir un uniform (fuente: elaboración propia)

Para eliminar algún uniform, deberá ser seleccionado en la tabla y habrá que pulsar el botón "Eliminar". Finalmente, una vez finalizada la gestión de uniforms se deberá pulsar el botón "Aceptar" y, de esta forma, ya se encontrarían en uso los uniforms añadidos. En la *Ilustración 94* se puede ver un ejemplo básico de uso, donde se usa un uniform para mostrar el modelo de color rojo.

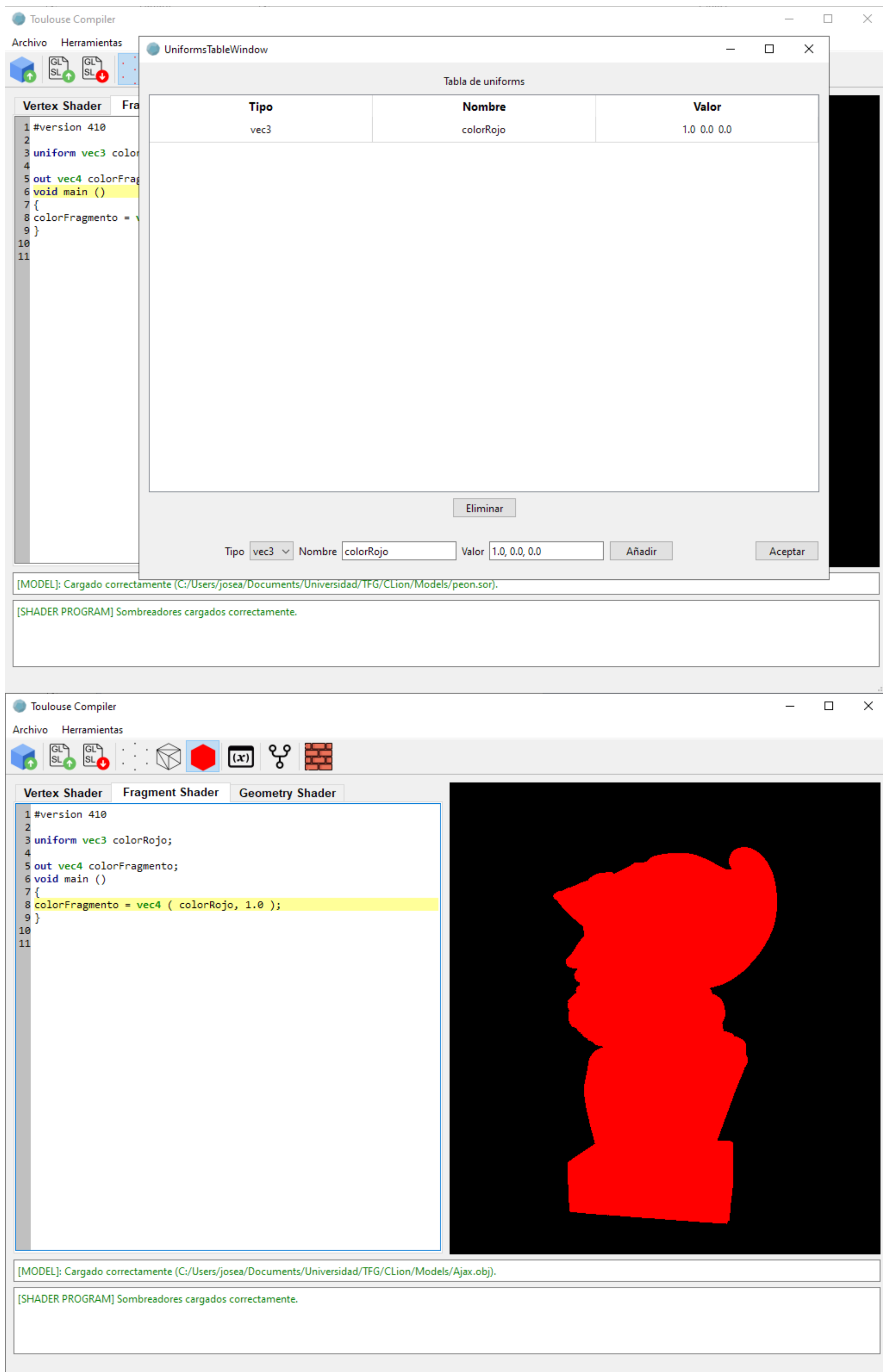


Ilustración 94 Ejemplo de uso de uniforms (fuente: elaboración propia)

5.3.5 Gestión de subrutinas

La funcionalidad para gestionar las subrutinas, al igual que los uniforms, se basa en una tabla. Hay tres formas de acceder a ella: la primera a través de la barra de herramientas, la segunda a través del menú herramientas→Tabla de subrutinas, y la última forma a través de la combinación de teclas Ctrl+Y (*Ilustración 95*).

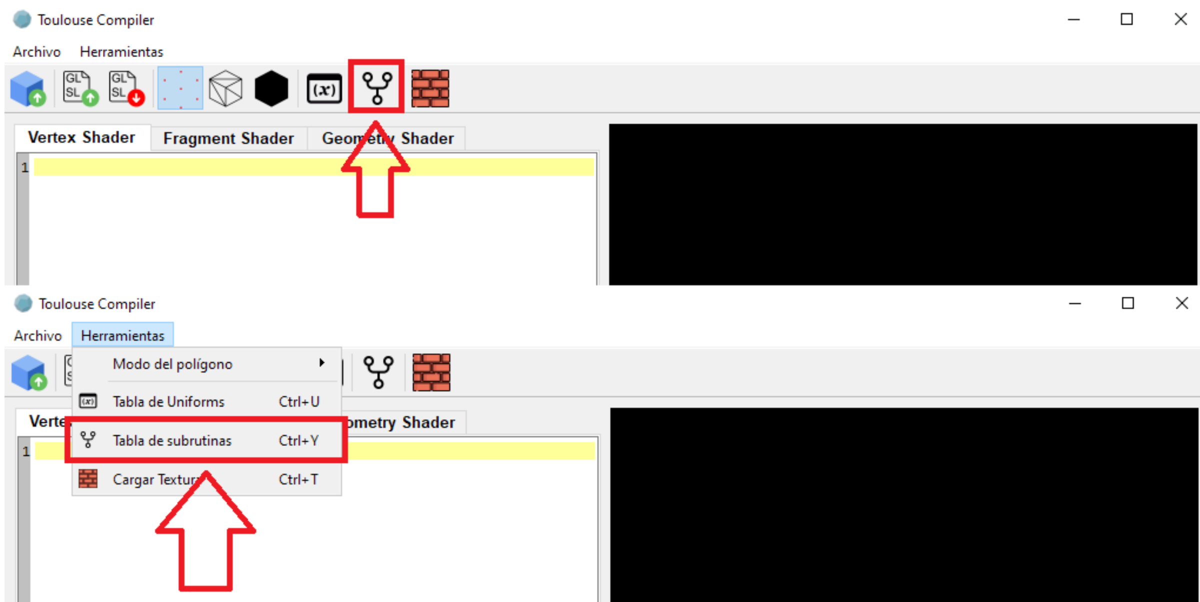


Ilustración 95 Formas de abrir la tabla de subrutinas (fuente: elaboración propia)

Una vez seleccionada la opción se nos abrirá una tabla donde podremos gestionar las subrutinas indicando un nombre, shader en el que se encuentra y la implementación a usar (*Ilustración 96*).

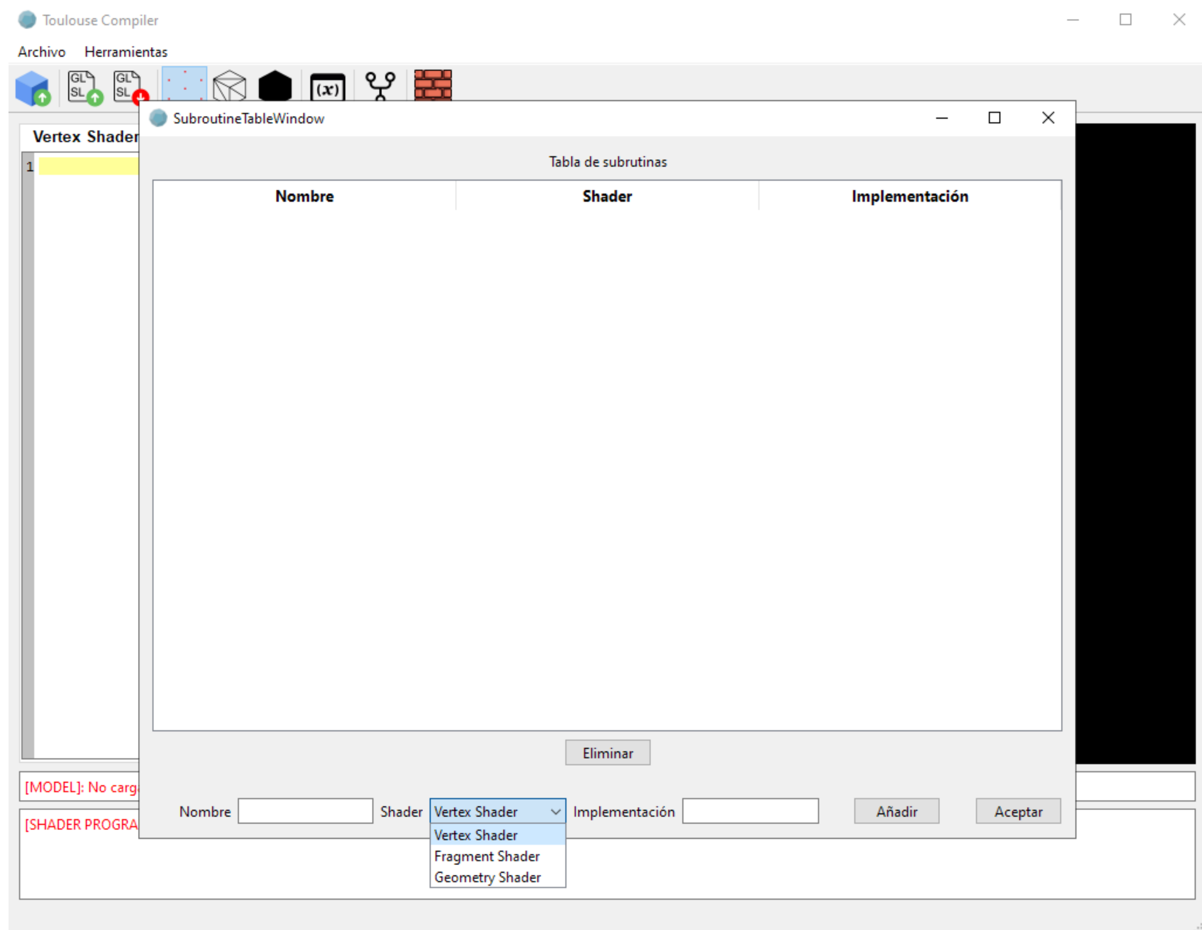


Ilustración 96 Tabla para la gestión de subrutinas (fuente: elaboración propia)

Para definir una subrutina hay una serie de restricciones:

- No puede haber dos subrutinas con el mismo nombre en un tipo de shader.
- Tanto el nombre como la implementación no pueden empezar por un número.
- Tanto el nombre como la implementación no pueden estar vacíos ni contener espacios en blanco.

Si se incumple alguna de estas restricciones, se muestra un cuadro de aviso indicando el error cometido (*Ilustración 97*).

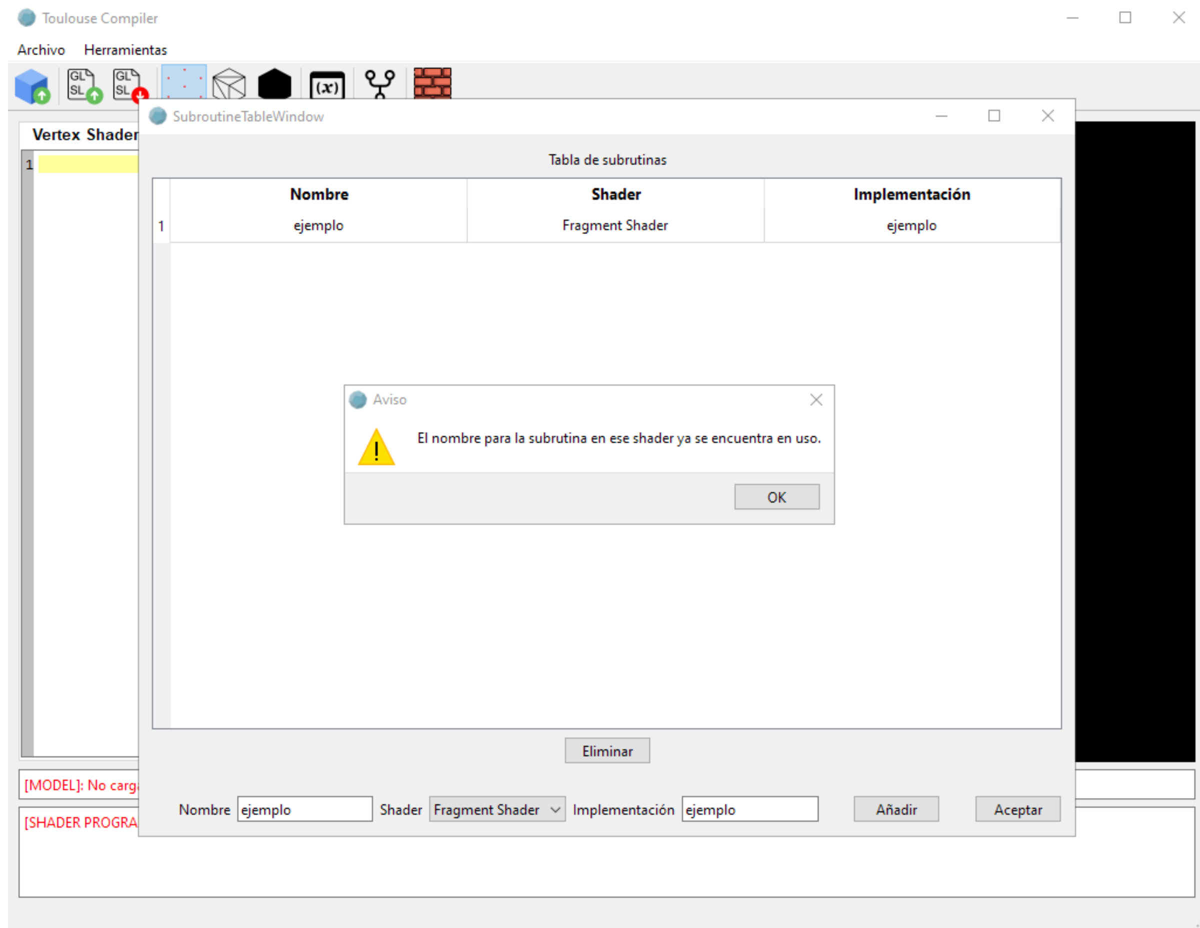


Ilustración 97 Advertencia de error al añadir subrutina (fuente: elaboración propia)

Para eliminar alguna subrutina, deberá ser seleccionada en la tabla y habrá que pulsar el botón "Eliminar". Finalmente, una vez finalizada la gestión de subrutinas se deberá pulsar el botón "Aceptar", y de esta forma ya se encontrarían en uso las subrutinas añadidas. En la *Ilustración 98* se puede ver un ejemplo básico de uso, donde se define una implementación de subrutina para cambiar a color verde un modelo.

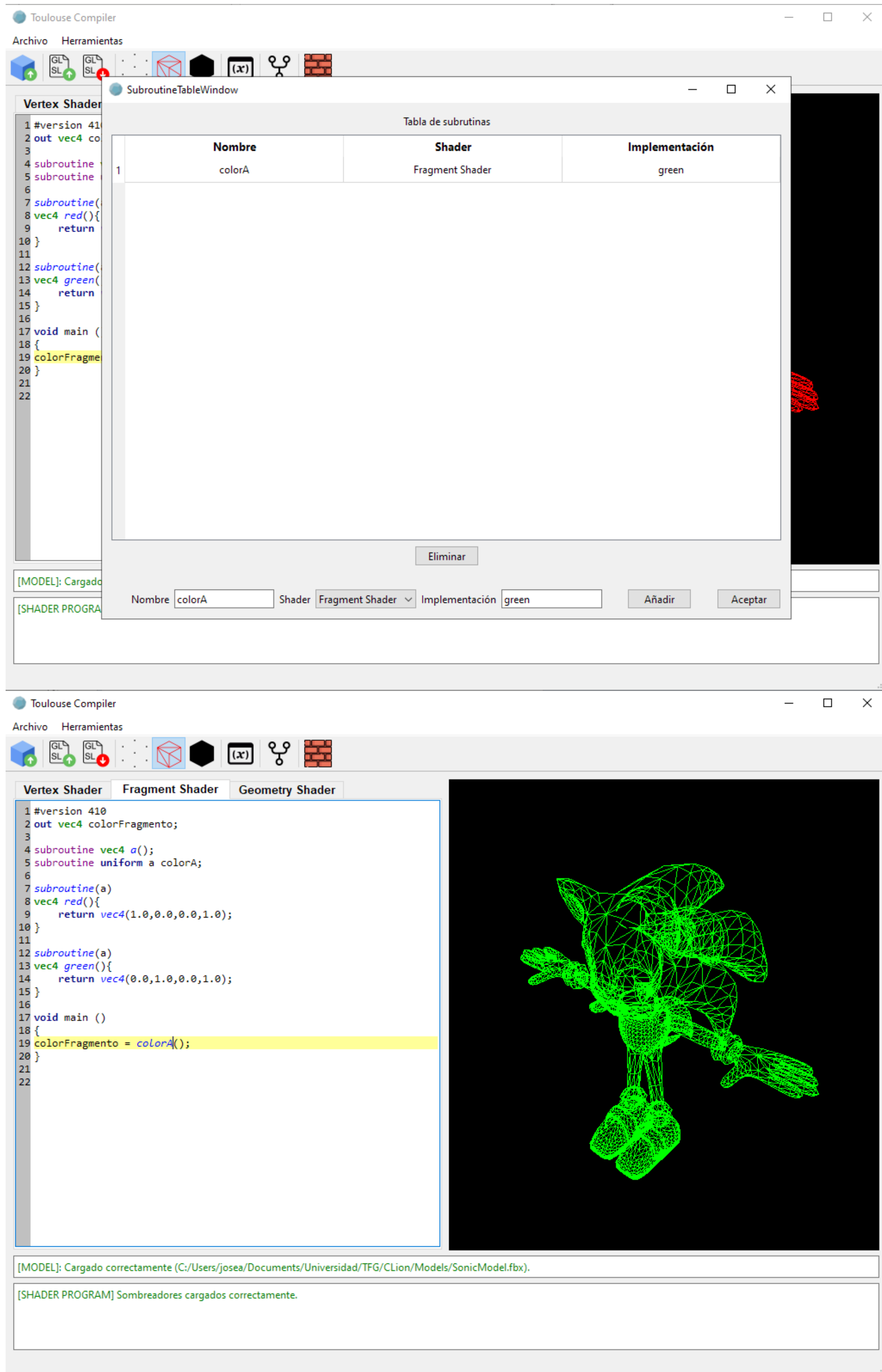


Ilustración 98 Ejemplo de uso de subrutinas (fuente: elaboración propia)

5.3.6 Cargar una textura

Al igual que todas las funcionalidades, hay tres formas de cargar una textura: la primera de ellas es a través del icono en la barra de herramientas, la segunda mediante el menú herramientas→Cargar Textura, y la tercera a través de la combinación de teclas Ctrl+T (*Ilustración 99*).

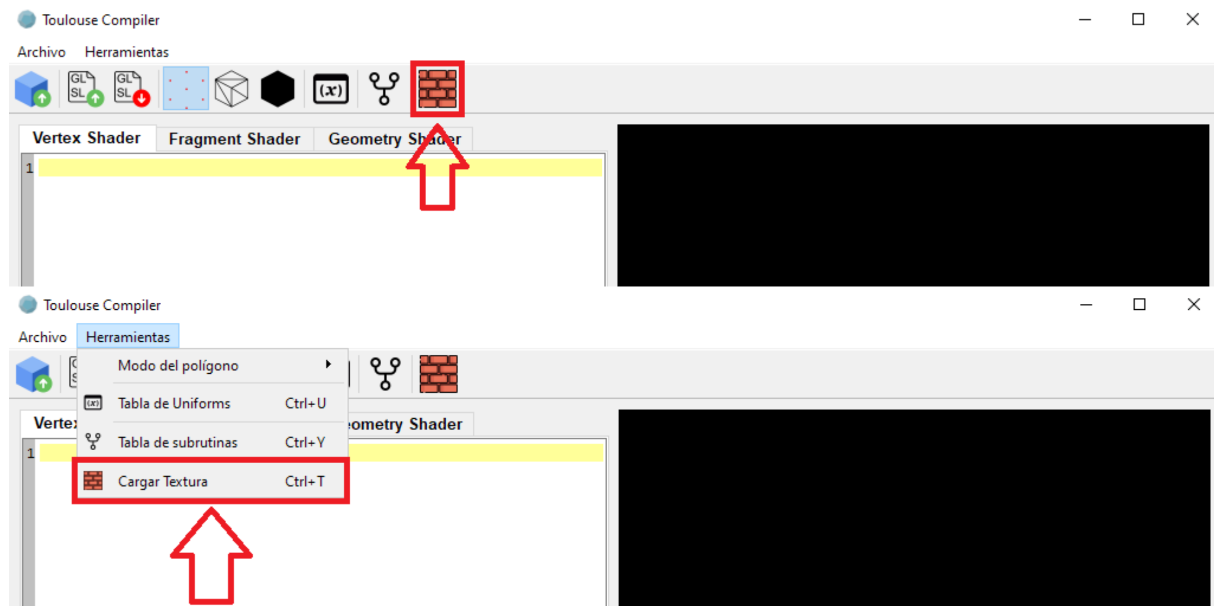


Ilustración 99 Formas de cargar una textura (fuente: elaboración propia)

Una vez seleccionada la opción, se nos abrirá el sistema de archivos, que nos permitirá navegar hasta encontrar la textura que se quiera aplicar. El formato para cargar una textura puede ser .PNG, .JPG o .BMP (*Ilustración 100*).

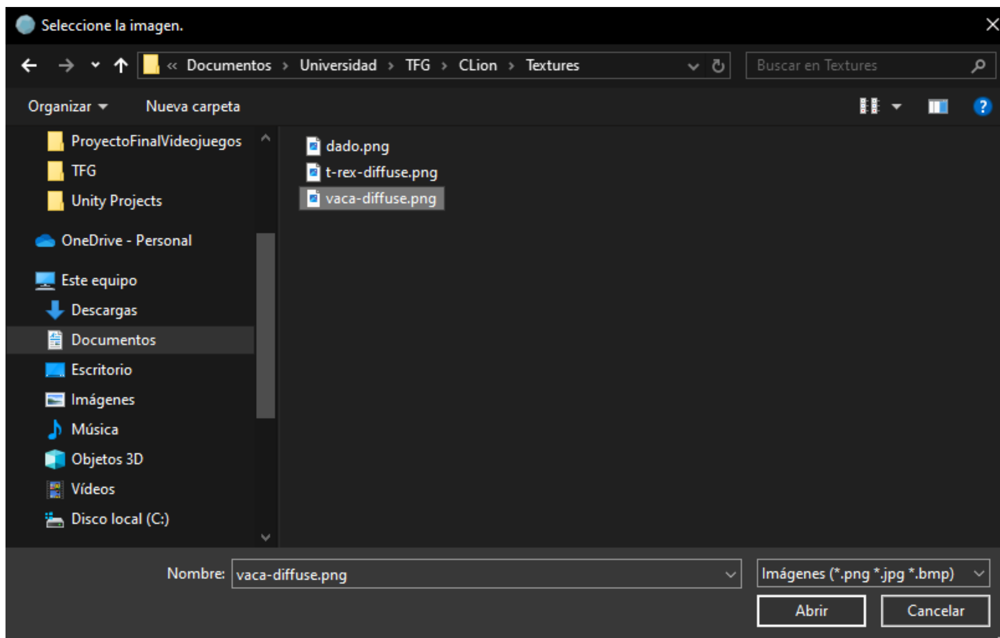


Ilustración 100 Formatos para cargar una textura (fuente: elaboración propia)

Una vez cargada, para usarla en el shader se deberá definir un uniform sampler2D con el nombre “textureSampler”, ya que es el nombre del uniform que emplea la aplicación en el shader program (*Ilustración 101*).

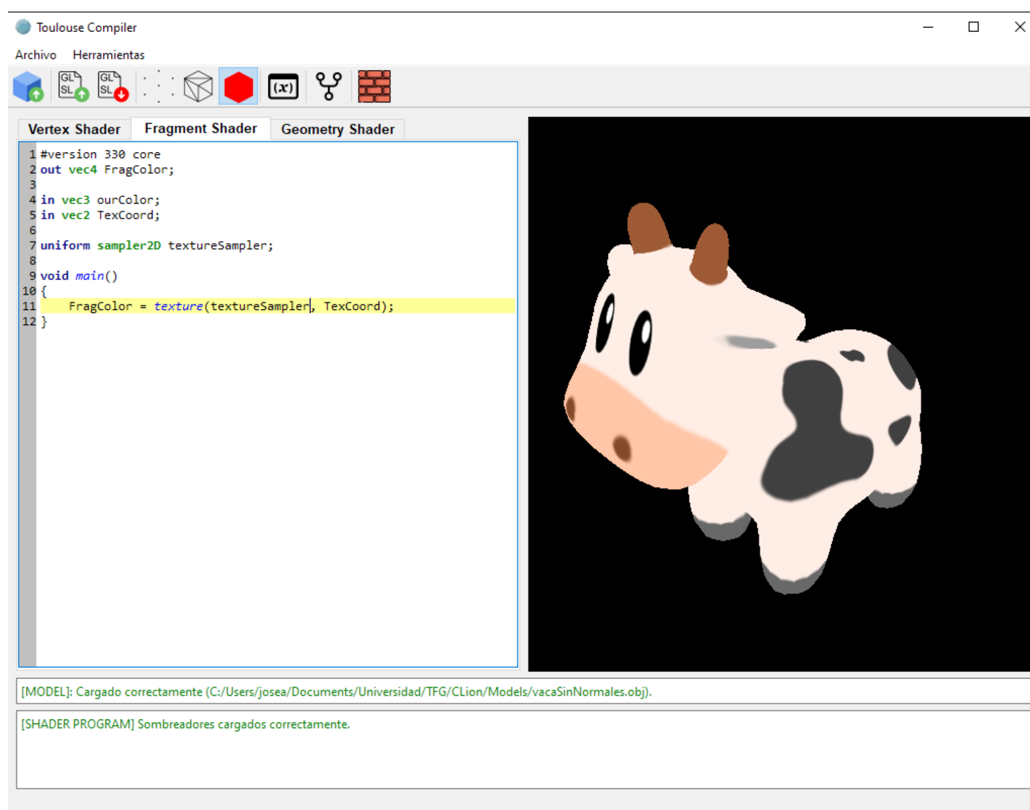


Ilustración 101 Ejemplo de uso de textura (fuente: elaboración propia)

5.3.7 Edición de shaders y visualización

Para editar los shaders existen una serie de pestañas (Vertex Shader, Fragment Shader, Geometry Shader) que sirven para alternar entre los distintos editores de código .GLSL(*Ilustración 102*).

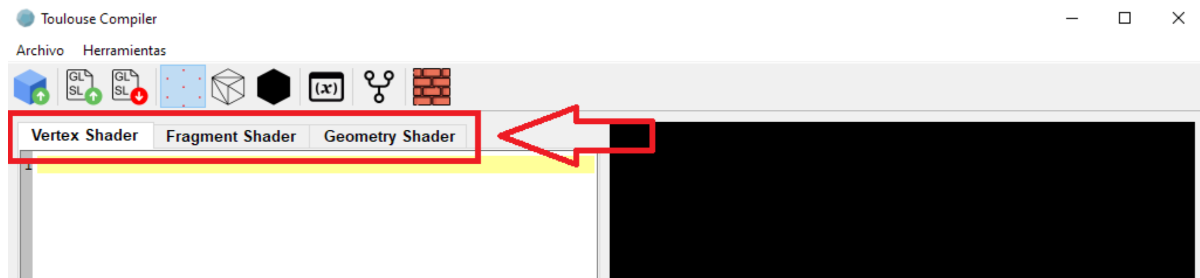


Ilustración 102 Editor de shaders (fuente: elaboración propia)

Para poder hacer uso de la geometría cargada, la aplicación gestiona por cada vértice:

- Posición, situada en el “layout (location = 0)”, que es un vector de tres números flotantes (vec3).
- Normal, situada en el “layout (location = 1)”, que es un vector de tres números flotantes (vec3).
- Coordenada de textura, situada en el “layout (location = 2)”, que es un vector de dos números flotantes (vec2).
- Tangente, situada en el “layout(location = 3)”, que es un vector de tres números flotantes (vec3).

Para visualizar un shader aplicado a un modelo será necesario cargar el modelo y cargar o escribir al menos un vertex shader y fragment shader correctos. Si se produce algún error en la carga del modelo o en la carga de shaders será mostrado los cuadros de texto situados en la parte inferior de la interfaz.

6 BIBLIOGRAFÍA

Adriana Gómez, M. d., & Silvina Migani, A. O. (3 de Junio de 2000). *COCOMO II Modelo para la Estimación de Costo.* Obtenido de https://blogadmi1.wordpress.com/wp-content/uploads/2010/11/cocom0llfull.pdf

Akenine-Möller, T., Haines, E., & Hoffman, N. (2008). *Real-Time Rendering.* Wellesley, MA: A K Peters, Ltd.

Hughes, J. F., Dam, A. v., McGuire, M., Sklar, D. F., Foley, J. D., Feiner, S. K., & Akeley, K. (2014). *Computer graphics principles and practice.* Upper Saddle River, N.J.: Addison-Wesley.

Movania, M. M., Lo, W. C., Wolff, D., & Lo, R. C. (2017). *OpenGL: Build High Performance Graphics.* Birmingham, UK: Packt Publishing.

OpenGL® 4.5 Reference Pages. (s.f.). Obtenido de https://registry.khronos.org/OpenGL-Refpages/gl4/

Pressman, R. (2010). *Ingeniería del Software.* McGraw-Hill.

Qt Documentation. (s.f.). Obtenido de https://doc.qt.io/qt.html

Rodríguez, J. (2013). *GLSL essentials.* Birmingham, UK: Packt Publishing.

Vries, J. d. (21 de Enero de 2020). *Learn OpenGL - Graphics Programming.* Kendall & Welling. Obtenido de https://learnopengl.com/About

Wolff, D. (2018). *OpenGL 4 Shading Language Cookbook: Build High-Quality, Real-Time 3D Graphics with OpenGL 4.6, GLSL 4.6, and C++17.* Packt Publishing Ltd.