



ugr | Universidad
de **Granada**

TRABAJO FIN DE GRADO
INGENIERÍA INFORMÁTICA

Implementación de la técnica Mesh Colors para pintar modelos 3D

Autor

Jose Manuel Torres Morales

Directores

Domingo Martín Perandrés



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

—
Granada, Junio de 2022

Implementación de la técnica Mesh Colors para pintar modelos 3D

Jose Manuel Torres Morales

Palabras clave: Informática gráfica, OpenGL, GPU, pintado de texturas, Mesh Colors, modelado 3D.

Resumen

Este proyecto se basa en la implementación de la técnica Mesh Colors creada por Cem Yuksel, John Keyser, Donald H. House en 2008 como una alternativa al pintado de texturas. En lugar de necesitar archivos de imagen externos, Mesh Colors guarda la información del pintado directamente en los modelos, enlazada las caras, vértices y aristas.

El objetivo principal del proyecto es crear una puerta de entrada para aprender programación en la GPU con OpenGL, aprendiendo también como crear y controlar interfaces de usuario en Qt.

Esta documentación muestra el proceso de creación del software the pintado de modelos 3D, desde el proceso de investigación hasta la implementación de Mesh Colors, los shaders y la interfaz de usuario.

El resultado del proyecto como herramienta de aprendizaje fueron positivos, ya que me permitió entender como funciona el cauce gráfico y a programar en la GPU.

La técnica Mesh Colors tiene un gran potencial para ser utilizado en entornos 3D, en especial en entornos con modelos simples y con visualización en tiempo real. Aun así para poder ser utilizada en entornos como Unity y Unreal, necesita un trabajo extra para poder adaptar las aplicaciones a las nuevas estructuras de color.

Implementation of Mesh Colors for painting 3D objects.

Jose Manuel Torres Morales

Keywords: Computer Graphics, OpenGL, GPU, Texture Paint, Mesh Colors, 3D modeling.

Abstract

This project shows the implementation of the technique Mesh Colors created by Cem Yuksel, John Keyser, Donald H. House in 2008 as an alternative to Texture Paint. Instead of relying in external image files, Mesh Colors stores all the painting information inside the geometry of the models attached directly to the faces, vertices and edges.

The main goal of the project is to serve as an entrance to learn GPU programming and OpenGL while also learning how to manage Qt and the interface scheme.

This documentation shows how the 3D object painting software was created, from the research process to the implementation of Mesh Colors in the shaders and the connection with the user interface.

The results of the realization of the project as a tool for learning OpenGL and Qt was positive and it helped me get a greater understanding of the graphics pipeline and shader programming.

The technique Mesh Colors show great potential to be used in the 3D modeling pipeline, it can be really useful for low poly real-time rendering environments but it will need code adaptation for the software to be able to be used in applications like Unity and Unreal.

Yo, **Jose Manuel Torres Morales**, alumno de la titulación Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 75573477P, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Jose Manuel Torres Morales

Granada a 7 de Julio de 2022.

D. Domingo Martín Perandrés, Profesor del Área de Informática Gráfica del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Implementación de la técnica Mesh Colors para pintar modelos 3D***, ha sido realizado bajo su supervisión por **Jose Manuel Torres Morales**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 8 de Julio de 2022.

Los directores:

Domingo Martín Perandrés

Índice general

1. Introducción	11
1.1. Motivación	11
1.2. Objetivos	12
2. Trabajos previos	13
2.1. Visualización	13
2.2. Vertex colors	14
2.3. Pintado de texturas	15
2.3.1. Mapeo UV	19
3. Desarrollo	24
3.1. OpenGL	24
3.1.1. Shaders	25
3.2. La técnica Mesh Colors	30
4. Implementación	34
4.1. Herramientas	34
4.2. Visualización 3D	34
4.2.1. Objetos 3D	34
4.2.2. Gl Widget	43
4.2.3. Shaders	53
4.3. Interfaz de usuario	59
4.3.1. Barra de herramientas	61
4.3.2. Menú de ayuda al usuario	62
4.3.3. Menú de acciones de archivo	63
4.3.4. Área de opciones	65
4.3.5. Área de dibujado	68
5. Metodología para el desarrollo del software	69
5.1. Requisitos funcionales	69
5.2. Requisitos no funcionales	69
6. Resultados	71
7. Conclusiones	74
7.1. Sobre la técnica MeshColors	74
7.2. Sobre este proyecto	74

ÍNDICE GENERAL

7

Blibliografía	75
Referencias	77
Apéndice	78

Índice de figuras

2.1.	Visualización de una escena 3D realizado por el artista 3D francés Gilles Tran. (Referencia 1)	13
2.2.	Cubo pintado utilizando <i>Vertex Colors</i>	14
2.3.	Modelo de una cabeza pintado con <i>Vertex Colors</i> utilizando <i>Blender</i> . (Referencia 2)	15
2.4.	Modelo de una cabeza texturizado, al comparar con 2.3 podemos observar que el nivel de detalle aumenta pero la complejidad del modelo no es mayor. (Referencia 3)	16
2.5.	Textura a la izquierda, modelo sin textura en el centro y modelo con la textura aplicada a la derecha.(Referencia 4) . . .	16
2.6.	A la izquierda, modelo sin texturizar. A la derecha el modelo texturizado a mano por un artista.(Referencia 5)	17
2.7.	En la parte superior, texturas de roca y musgo generadas automáticamente. En la parte inferior operaciones contenidas en los recuadros y conectadas mediante líneas que representan el flujo de ejecución para generar las texturas de la parte superior.(Referencia 6)	18
2.8.	Ejemplo de una aplicación de pintado de texturas. Esta aplicación forma parte del software de modelado 3D de código abierto <i>Blender</i> .(Referencia 7)	19
2.9.	Ejemplo de un modelo 3D de una esfera cuyos vértices se esparcen sobre el mapa UV. Sobre ese mapa podemos colocar la textura de la tierra para crear el modelo 3D texturizado. (Referencia 8)	20
2.10.	A la derecha el modelo texturizado, a la izquierda el mapa de texturas. Las líneas y puntos naranjas representan los vértices y las aristas del modelo y se pueden mover manualmente para ajustar la influencia de la textura sobre el modelo. (Referencia 9)	20
2.11.	A la derecha la imagen del atlas de texturas y el mapa UV de los cubos. A la izquierda los cubos texturizados.(Referencia 10)	21
2.12.	Las aristas marcadas en verde tendrán sus vértices representados varias veces en el mapa de UV, ya que se encuentran en la frontera, pudiendo crear discontinuidades.(Referencia 11)	22
2.13.	Varios niveles de detalle de un modelo 3D.(Referencia 12) . .	22
2.14.	Visualización de una imagen con Mip-mapping.(Referencia 13)	23

3.1. Cauce gráfico de <i>OpenGL</i> .(Referencia 14)	26
3.2. La dirección a la que encaran los triángulos depende en el orden de los vértices.(Referencia 15)	28
3.3. Rasterización de un triángulo, transformando la geometría en el conjunto de fragmentos más cercanos a la forma.(Referencia 16)	29
3.4. Ejemplo de un modelo texturizado utilizando <i>Mesh Colors</i> .(Referencia 17)	30
3.5. Triángulo pintado utilizando <i>Vertex Colors</i> y asignando los colores rojo verde y azul en cada vértice y dejando a <i>OpenGL</i> encargarse de la interpolación para el resto de la superficie. .	31
3.6. Los puntos representan las muestras de color: azul para los vértices, verde para las aristas y rojo para las caras. Partiendo de la resolución 1 (a) en la que el dibujado se comporta como <i>Vertex Colors</i> , cada vértice tiene un color, podemos incrementar la resolución para aumentar el número de muestras.(Referencia 17)	31
3.7. Selección de las 3 muestras adyacentes, los puntos oscuros, que influyen el color de en la posición del punto rojo. (Referencia 17)	33
4.1. Visualización de los ejes de coordenadas.	35
4.2. Comparación de la dirección de las normales en función del orden de los vértices.(Referencia 18)	38
4.3. Estado inicial.	39
4.4. Muestras clave con color generadas tras incrementar la resolución.	39
4.5. Cálculo de una muestra generada en función de las muestras clave adyacentes.	40
4.6. Resultado de aplicar el paso 3 a todas las muestras generadas.	40
4.7. Cálculo del color de la muestra clave.	41
4.8. Cálculo de una muestra para la resolución inferior.	41
4.9. Resultado de la reducción de resolución.	41
4.10. Modelo 3D.	45
4.11. Aristas del modelo 3D.	45
4.12. Representación de color de los índices de las caras	46
4.13. Triángulo seleccionado.	46
4.14. Ejes de coordenadas	47
4.15. Visualización de índices de las caras y las muestras.	52
4.16. Diferencia de tamaño de pincel al pintar sobre el modelo. .	53
4.17. Colores aplicados aplicados en el geometry shader e interpolados para generar un mapa de coordenadas baricéntricas. .	54
4.18. Muestras de un triángulo dibujadas con y sin interpolación. .	57
4.19. Modelo 3D visualizado con y sin iluminación.	58
4.20. Ejemplo de distintos tipos de <i>widget</i> , cada número representa una clase distinta. (Referencia 19)	60

4.21. Áreas principales de la interfaz de la aplicación	61
4.22. Menús accesibles desde la barra de herramientas	62
4.23. Ventana de ayuda para explicar los controles	63
4.24. Ventana de selección de archivos para cargar el modelo en formato ply.	64
4.25. Opciones de pincel de dibujado: selección de color y configuración de tamaño y opacidad del pincel	66
4.26. Ventana de selección de color.	66
4.27. Botón de selección de color, cambia de color para mostrar al usuario el que ha seleccionado.	67
4.28. Botones para incrementar y decrementar resolución	67
4.29. Casillas para activar y desactivar las opciones de visualización. .	68
6.1. Pantalla de selección de modelos.	71
6.2. Visualización de modelo iluminado.	72
6.3. Pintado sobre el modelo.	72
6.4. Pantalla de selección de archivos para guardar los colores pintados.	73
7.1. Ventana de la aplicación con los controles numerados.	79
7.2. Aristas del modelo 3D.	80
7.3. Modelo 3D visualizado con y sin iluminación.	81
7.4. Muestras de un triángulo dibujadas con y sin interpolación. .	81

1. Introducción

El objetivo principal de este trabajo es dar los primeros pasos en el mundo de la investigación y el desarrollo utilizando un proyecto que me permita pasar por todas las etapas, desde la adquisición de conocimiento hasta el desarrollo del programa, resolviendo los problemas que van surgiendo a partir de una idea inicial que se toma como objetivo.

En mi caso esa idea es desarrollar el algoritmo *Mesh Colors* planteado en un artículo [?]. Además este proyecto requiere aprender a programar el uso de la GPU utilizando programas específicos para controlar y planificar su funcionamiento conocidos como *shaders* y aprender los varios pasos del proceso de visualización 3D que es la transformación de una escena 3D en una imagen 2D.

El proyecto consiste en un programa de dibujo de texturas usando la librería *OpenGL* basado en la técnica *MeshColors*, en el que se aportan herramientas esenciales para cargar, visualizar y pintar colores sobre un modelo 3D.

1.1. Motivación

La GPU (*Graphics Processing Unit*), unidad de procesamiento gráfico en español, es un componente básico del computador que se encuentra en constante y rápido desarrollo. Además de su propósito inicial para dibujado de modelos 3D (*rendering*) las GPU son máquinas de procesado paralelo que se utilizan con problemas que requieren una gran cantidad de cálculos, como edición de video, blockchain y machine learning debido a su potencia y capacidad de computación.

La Informática Gráfica siempre ha sido un campo muy interesante para mí ya que requiere conocimientos técnicos sobre el funcionamiento de la GPU y que permite un gran rango de creatividad compaginando la creación artística con los conocimientos técnicos de ingeniería informática. Además ofrece una gran cantidad de oferta profesional en industrias de importancia y peso económico como la medicina, inteligencia artificial o el entretenimiento.

El proyecto que vamos a desarrollar se basa en gran medida en el uso de programas para la GPU o *shaders*. Los *shaders* son un pilar fundamental de la informática gráfica ya que permiten el control a bajo nivel sobre el

comportamiento de la unidad de procesamiento gráfico para sacar todo su potencial. Y, aunque requieren un conocimiento técnico profundo, ofrecen un entorno creativo e interesante para crear una gran variedad de programas como el de este proyecto.

1.2. Objetivos

El proyecto tiene como objetivo principal la iniciación a la investigación y el desarrollo a través de la implementación de un método para la creación de texturas expuesto en un artículo. Esto implica:

- Tener que aprender y documentar conceptos
- Aprender a programar la GPU mediante el uso de shaders
- Desarrollar una interfaz de usuario que facilite el uso de la aplicación interactivamente

Otros objetivos son:

- Entender las capacidades y limitaciones de la GPU como la utilización de la memoria y estructuras de datos y la computación paralela.
- Aprender sobre el flujo de las interfaces de usuario en Qt.
- Ampliar el conocimiento sobre los modelos de iluminación y las texturas.

2. Trabajos previos

2.1. Visualización

En informática gráfica la visualización es el proceso de convertir una escena 3D en imágenes 2D.[2]



Figura 2.1: Visualización de una escena 3D realizado por el artista 3D francés Gilles Tran. (Referencia 1)

Obtener una imagen a partir de un modelo 3D es un proceso complejo y costoso. El mismo se puede describir como una serie de etapas que se van cubriendo hasta que se alcanza el objetivo final. Este conjunto de etapas es el cauce gráfico (*graphics pipeline*). Las interfaces de programación centradas en gráficos de computador como *OpenGL* y *DirectX* ofrecen cauces que pueden ser editados mediante el uso de *shaders*.

El proceso de visualización será explicado en más profundidad mostrando en concreto cómo funciona en *OpenGL*.

2.2. Vertex colors

OpenGL por defecto ofrece una forma muy simple de aplicar color a un modelo 3D. Cada vértice tiene asociado un color con formato RGBA¹, que se interpola entre los colores de los otros dos vértices para definir el color en la superficie del triángulo.

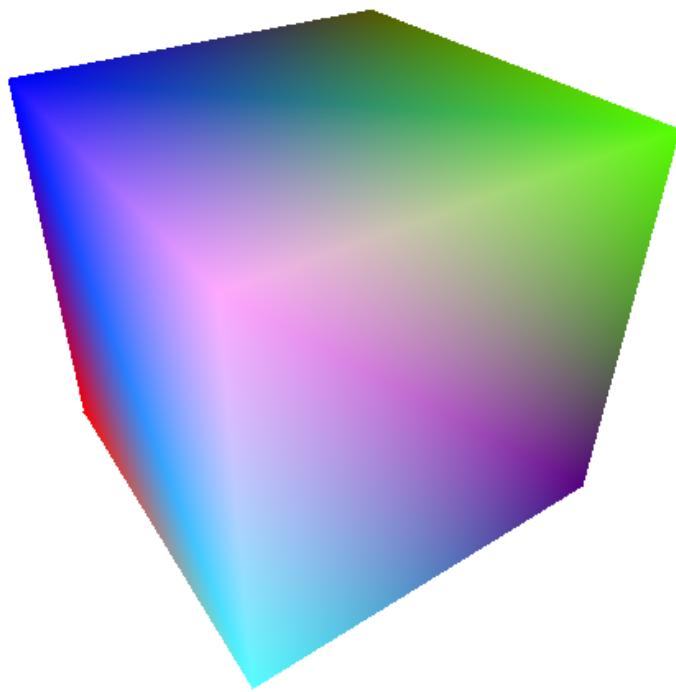


Figura 2.2: Cubo pintado utilizando *Vertex Colors*

Esta técnica no solo se utiliza para mostrar color, si no también para enviar información de cada vértice a la GPU y suele ser utilizadas para crear animaciones procedurales o efectos especiales con programas de la GPU.

¹**Formato de color RGBA:** contiene cuatro componentes numéricos que representan las intensidades de: rojo, verde, azul y alpha, el valor de los tres primeros representa la cantidad entre 1 y 0 de ese color. La componente alpha representa la transparencia, 1 es completamente opaco y 0 completamente transparente.



Figura 2.3: Modelo de una cabeza pintado con *Vertex Colors* utilizando *Blender*. (Referencia 2)

2.3. Pintado de texturas

Vertex Colors permite dar color a los modelos, pero no con alto nivel de detalle. Debido a que los colores van asociados a los vértices, si quisieramos colorear un modelo con detalle deberíamos crear muchos vértices adicionales que no aplicarían ningún valor a la geometría del modelo.

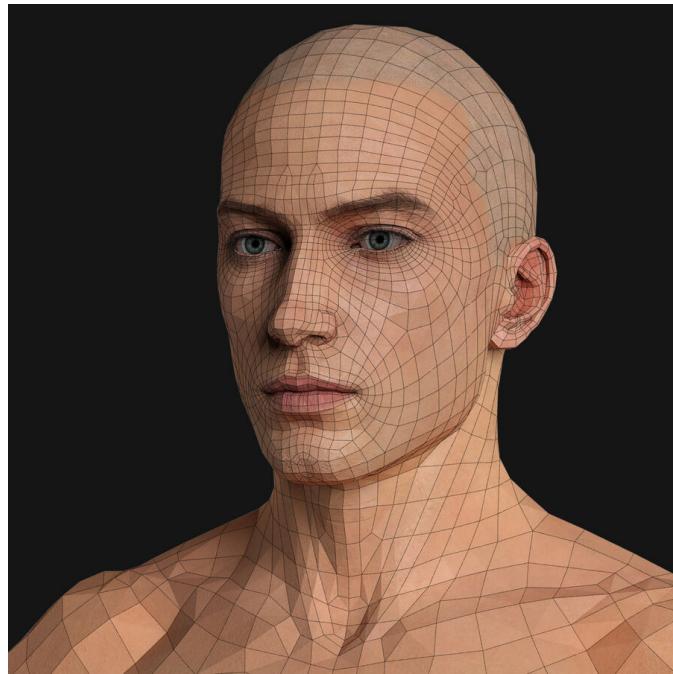
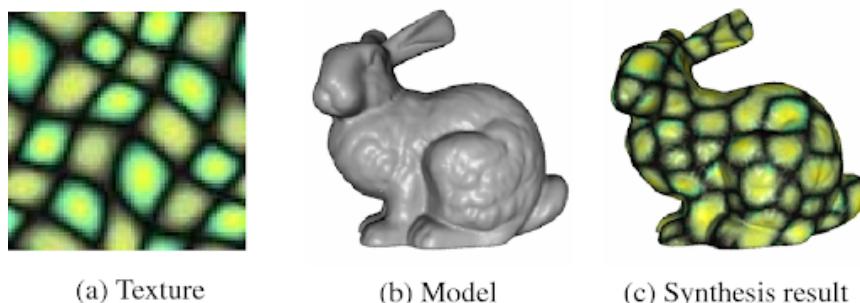


Figura 2.4: Modelo de una cabeza texturizado, al comparar con 2.3 podemos observar que el nivel de detalle aumenta pero la complejidad del modelo no es mayor. (Referencia 3)

El texturizado consiste en añadir datos a la superficie geométrica de un modelo 3D para aumentar el detalle o la calidad gráfica y dar color, normalmente mediante imágenes que cubren el modelo. Es una práctica estándar en las industrias de los videojuegos, la animación y el cine, ya que permite crear gráficos por computador detallados.



(a) Texture

(b) Model

(c) Synthesis result

Figura 2.5: Textura a la izquierda, modelo sin textura en el centro y modelo con la textura aplicada a la derecha.(Referencia 4)

El modelo sin texturizado también puede incluir información del color.

La información va asignada en los vértices del modelo y cuando se visualiza cada triángulo se interpola entre los colores de los tres vértices, esta técnica se conoce como *Vertex Colors* y solo permite utilizar puntos de color en los vértices sin posibilidad de añadir detalles en otras partes del modelo.

En el ejemplo 2.5 podemos ver cómo la imagen se aplica sobre el modelo generando el texturizado, en este caso es un ejemplo simple. Pero este proceso incluye artistas con un conocimiento técnico extenso que trabajan para generar de forma manual o automática la imagen que envolverá al modelo.

En concreto, el pintado manual es un proceso en el que el artista dibuja sobre la región de la textura, o sobre el modelo directamente, los colores que generarán la textura. Es un proceso artístico estandarizado para el diseño de personajes en la industria de la animación y los videojuegos.



Figura 2.6: A la izquierda, modelo sin texturizar. A la derecha el modelo texturizado a mano por un artista.(Referencia 5)

Otro estándar de estas industrias son los programas para la GPU generados por artistas y que sirven para, a través de patrones, colores y cálculos matemáticos, crear texturas de entorno como rocas, charcos y vegetación o incluso para generar efectos sobre los modelos como fuego, luz o distorsiones.

En esta imagen vemos un ejemplo de una textura generada automáticamente por un *shader* en el editor de nodos de *Blender*, un programa de modelado 3D con la funcionalidad de crear programas para la GPU de texturizado a través de un lenguaje de programación visual basado en nodos. El programa recibe unos parámetros de entrada y genera texturas para rocas añadiendo detalles de musgo.

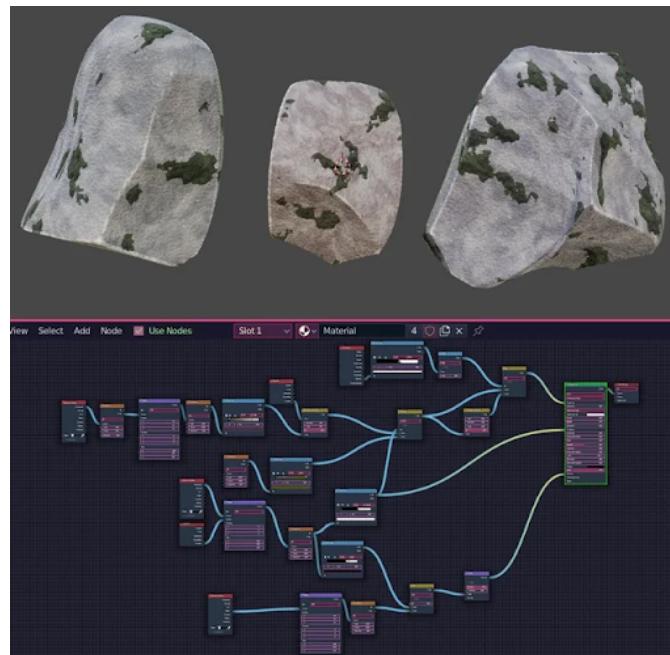


Figura 2.7: En la parte superior, texturas de roca y musgo generadas automáticamente. En la parte inferior operaciones contenidas en los recuadros y conectadas mediante lineas que representan el flujo de ejecución para generar las texturas de la parte superior.(Referencia 6)

Muchos programas de modelado 3D como por ejemplo *Blender* y *Maya*, implementan sus propias herramientas de texturizado como parte del paquete de modelado. Y existen otros programas que se centran específicamente en texturizado y que son utilizados muy frecuentemente en dichas industrias, por ejemplo *Substance Painter*. Estos programas suelen incluir tanto la habilidad para dibujar las texturas manualmente como un editor de nodos para generarlas, incluso permitiendo combinar los dos para crear texturas más dinámicas y detalladas.

En este proyecto nos centramos en concreto en el pintado de texturas manual mediante una herramienta en 3D, pudiendo también dibujar directamente sobre el mapa de textura y ver el resultado en el modelo a la vez.

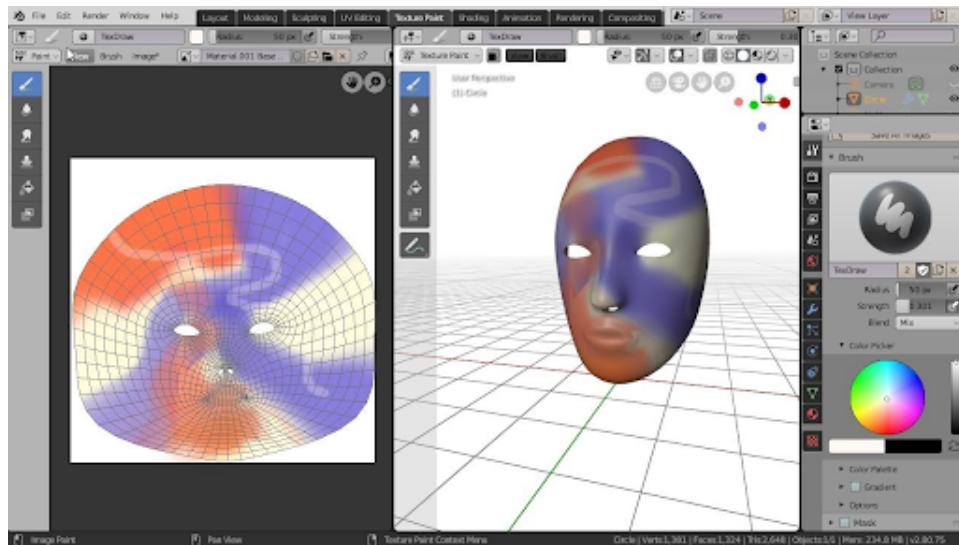


Figura 2.8: Ejemplo de una aplicación de pintado de texturas. Esta aplicación forma parte del software de modelado 3D de código abierto *Blender*. (Referencia 7)

Como vemos en el ejemplo 2.8 este el resultado de este proceso queda guardado en un archivo de imagen al que llamamos textura.

2.3.1. Mapeo UV

Para poder identificar qué parte de la textura se corresponde con cada parte del modelo utilizamos el Mapeo UV que consiste en:

1. Las imágenes se hacen corresponder con un espacio paramétrico normalizado para independizar la implementación del tamaño de cada imagen.
2. Cada vértice del modelo tiene asignadas unas coordenadas paramétricas de textura. Indican que posición de la textura se hace corresponder con cada vértice.
3. Un procedimiento relativamente complejo se encarga de hacer la correspondencia entre los píxeles de la imagen de la textura con los píxeles de la imagen resultado.

Las letras U y V denotan las coordenadas bidimensionales del cuadrado ya que X, Y y Z ya están siendo utilizadas por el modelo 3D para la posición de los vértices.

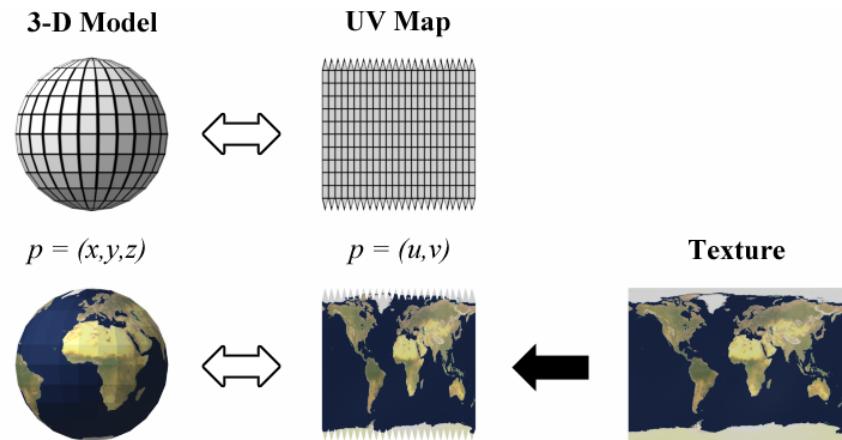


Figura 2.9: Ejemplo de un modelo 3D de una esfera cuyos vértices se esparcen sobre el mapa UV. Sobre ese mapa podemos colocar la textura de la tierra para crear el modelo 3D texturizado. (Referencia 8)

Las coordenadas UV se encuentran almacenadas en la información de los vértices junto con las coordenadas de la posición 3D y, aunque existen programas que generan estas coordenadas automáticamente, normalmente requieren que el artista ajuste manualmente y de forma optimizada las coordenadas del modelo debido a que no todas las zonas requieren la misma cantidad de detalles e incluso puede ser necesario para poder encajar todas las partes del modelo sin que se superpongan.

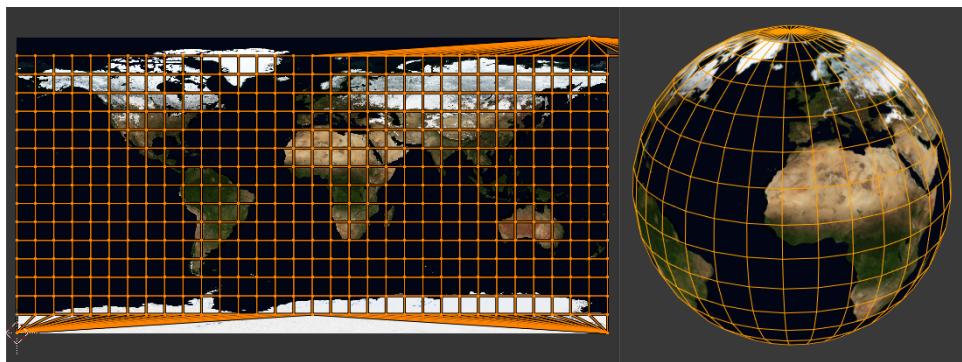


Figura 2.10: A la derecha el modelo texturizado, a la izquierda el mapa de texturas. Las líneas y puntos naranjas representan los vértices y las aristas del modelo y se pueden mover manualmente para ajustar la influencia de la textura sobre el modelo. (Referencia 9)

Para modelos más simples los mapas de UV se preparan para encajar

varios modelos en una sola imagen, reduciendo así el tamaño del proyecto ya que una textura sirve para cubrir varios modelos. Este proceso se llama atlas de texturas.

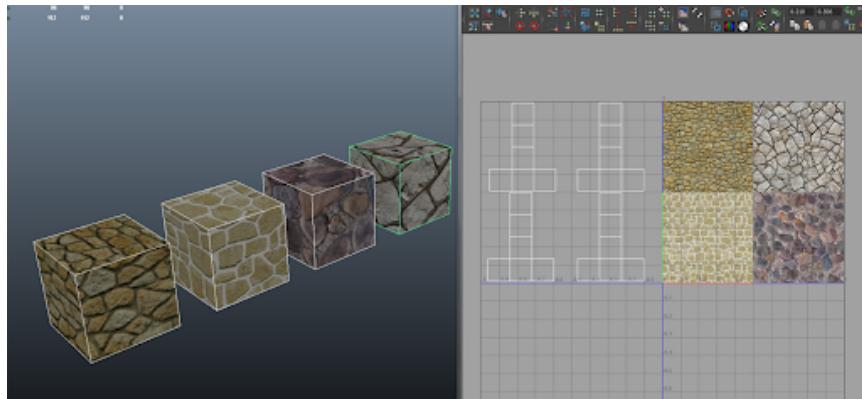


Figura 2.11: A la derecha la imagen del atlas de texturas y el mapa UV de los cubos. A la izquierda los cubos texturizados.(Referencia 10)

El mapa UV requiere que todos los polígonos del modelo estén mapeados correctamente ya que la desconexión de polígonos adyacentes podría generar artefactos no deseados en la superficie del modelo. Esto se conoce como discontinuidad y ocurre cuando uno o varios vértices son representados múltiples ocasiones en el mapa de UV debido a que representan la frontera entre distintas áreas de texturizado.

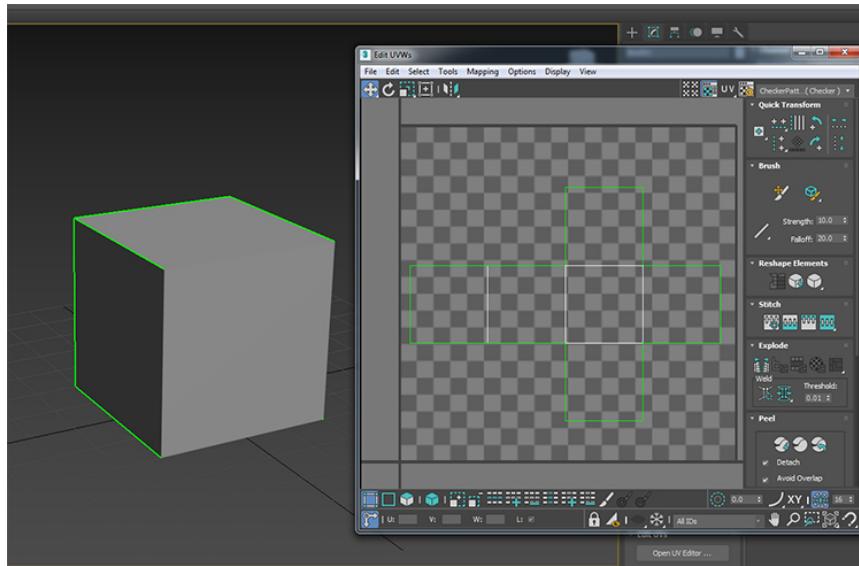


Figura 2.12: Las aristas marcadas en verde tendrán sus vértices representados varias veces en el mapa de UV, ya que se encuentran en la frontera, pudiendo crear discontinuidades.(Referencia 11)

La calidad de la imagen es fija y, aunque se puede ajustar el tamaño de los polígonos dentro del área de coordenadas UV, la resolución máxima no se puede cambiar. Esto implica que si se necesita un polígono con un texturizado muy detallado y el resto de polígonos son muy simples, la imagen necesitará una resolución de textura lo suficientemente alta, aunque solo sea para un solo polígono.

Este problema es especialmente importante en entornos visualizados en tiempo real. Para mitigar este problema se utiliza la técnica del nivel de detalle, en la que se crean varias versiones del modelo con distinta cantidad de polígonos y se interpola entre ellas en función de la distancia al objeto.

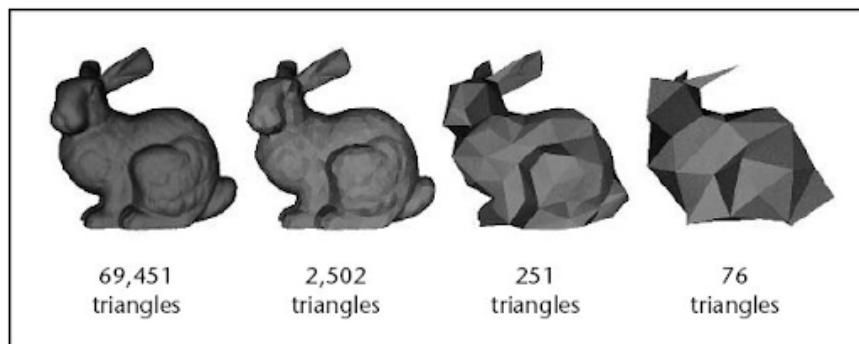


Figura 2.13: Varios niveles de detalle de un modelo 3D.(Referencia 12)

El nivel de detalle también se puede aplicar a las texturas con la técnica Mip-mapping, en la que una imagen almacena varias resoluciones progresivamente más bajas de la misma textura para poder asignar a los distintos niveles de detalle del modelo. [4]



Figura 2.14: Visualización de una imagen con Mip-mapping.(Referencia 13)

Además de pintar la textura principal, el pintado de texturas clásico requiere trabajo extra y conocimiento técnico para crear los mapas de UV y, en el caso de la visualización 3D en tiempo real, crear texturas para distintos niveles de detalle.

3. Desarrollo

Dentro de las áreas de investigación para la realización de este proyecto existen algunas en específico que requieren mucha atención ya que son clave para el proyecto y a la vez contienen una gran cantidad de información que hay que conocer. Estas áreas son *OpenGL*, los *shaders* y *Mesh Colors*. Una parte importante de este proyecto ha sido aprender a usar la GPU este proceso ha requerido adquirir un conocimiento técnico sobre la visualización 3D y el cauce gráfico y ha ocupado gran parte del tiempo de investigación. La intención de este apartado es compartir ese conocimiento y explicar la técnica *Mesh Colors*.

3.1. OpenGL

OpenGL es una interfaz de programación de aplicaciones (API) multilenguaje y multiplataforma que permite crear aplicaciones que producen gráficos 3D o 2D. La API se creó en 1992 como una alternativa multiplataforma a *IrisGL*, que se utilizaba para programar en computadores gráficos de la empresa estadounidense *Silicon Graphics*.

OpenGL permitía el uso de sus funciones en sistemas que no eran de *Silicon Graphics*. Desde entonces *OpenGL* ha seguido siendo actualizado mayoritariamente con adiciones incrementales que aportan nuevas funcionalidades y herramientas, con su versión más reciente, la 4.6, lanzada en 2017.

La principal competencia de *OpenGL* es *DirectX*, otra API creada por *Microsoft* que se desarrolló originalmente para la programación de videojuegos y sigue siendo muy utilizada específicamente en ese campo. A diferencia de *OpenGL*, *DirectX* es exclusiva para sistemas de *Microsoft Windows* y las consolas de videojuegos *Xbox*.

En cuanto a alternativas multiplataforma, la que más destaca es *Vulkan*, desarrollada por *Khronos*¹. *Vulkan* ofrece una alternativa a más bajo nivel que *OpenGL* a cambio de una ejecución más potente y optimizada para gráficos de alta calidad. Este lenguaje es mucho más reciente, anunciado en 2013, y se utiliza en productos que necesitan gráficos de alta calidad.

¹**Khronos:** agrupación de empresas que publica y mantiene estándares de visualización 3D. *OpenGL* forma parte de *Khronos* desde 2006.

Para este proyecto *DirectX* queda descartado ya que queremos poder ejecutar la aplicación en *Windows* y en *Linux*. En cuanto a *Vulkan*, aunque es multiplataforma, ofrece una complejidad extra al ser a más bajo nivel. Para mi nivel de conocimiento, comenzar con *OpenGL* ofrece una forma más idónea de aprender sobre la programación en la GPU y sus conexiones con la GPU.

3.1.1. Shaders

Un *shader* es un programa definido por el usuario que está diseñado para ejecutarse sobre la GPU (unidad de procesamiento gráfico)[6]. Originalmente se utilizaban para el sombreado de escenas 3D , de ahí su nombre (sombreador en inglés), ajustando la luz y el color, ahora se pueden utilizar para una gran variedad de campos, incluso para funciones no relacionadas con gráficos.

Los *shaders* están diseñados para encajar dentro de la estructura de dibujo, en la que se definen diferentes secciones que representan un proceso programable. Cada sección tiene un conjunto fijo de entradas y salidas que se reciben y se pasan a las siguientes secciones.

En concreto para *OpenGL* [6] este es el cauce:

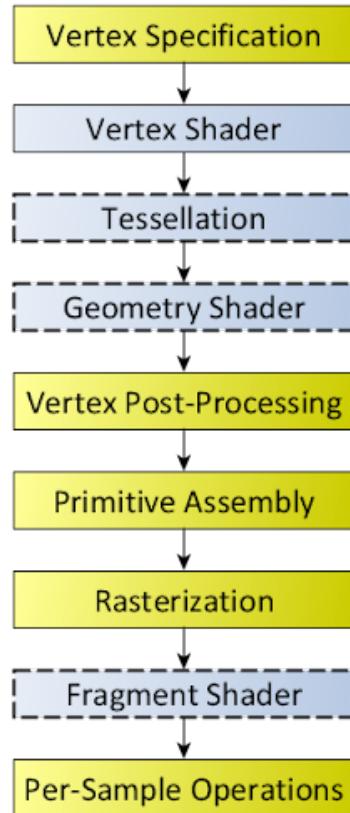


Figura 3.1: Cauce gráfico de *OpenGL*.(Referencia 14)

La aplicación (código en la CPU) envía la información necesaria al pipeline incluidos los vértices que van a componer el objeto 3D. También define qué tipo de primitivas se van a dibujar, triángulos, puntos o líneas.

Vertex Specification

La aplicación crea una lista ordenada de vértices que enviará al cauce. Aquí se forman las primitivas, formas básicas de dibujado como aristas, vértices o triángulos.

Vertex shader

Esta información procede como entrada al *Vertex shader*, este *shader* puede ser definido por el usuario y procesa cada vértice individualmente. Como entrada reciben atributos de vértice (*Vertex Attribute*) con la información necesaria para el procesado y realiza operaciones simples.

Al ser un *shader* programable por el usuario se pueden definir las entradas y salidas del mismo. Hay una salida que es especialmente importante y es la posición del vértice, esta posición es necesaria para los siguientes pasos y el correcto dibujado del modelo.

Tessellation

Esta técnica se utiliza para mejorar ciertos detalles en los modelos 3D, consiste en subdividir las primitivas en primitivas más pequeñas. Se divide en tres partes consecutivas.

1. *Tessellation Control Shader* (TCS): determina cuantas primitivas se van a generar, esta parte es programable.
2. *Tessellation primitive generator*: recibe la información del TCS y genera las primitivas.
3. *Tessellation Evaluation Shader* (TES): también es programable, calcula la posición y otros datos por cada vértice generado.

Esta técnica es completamente opcional, y no se va a utilizar en este proyecto.

Geometry shader

Este *shader* recibe los vértices de una primitiva (triángulo, un solo vértice o arista) y retorna cero o varias primitivas. En este *shader* podemos realizar operaciones para cada primitiva pudiendo incluso convertir la primitiva original en primitivas nuevas, generando nuevos vértices si es necesario. Este *shader* también es opcional, aunque este sí se utilizará en el proyecto.

Vertex Post-Processing

El post procesado de vértices realiza varias operaciones para preparar los datos para el montaje de primitivas y la rasterización.

Clipping: donde se eliminan las primitivas que no aparecerán en la imagen ya que se encuentran fuera del campo de visión de la cámara, este proceso puede acelerar el resto de pasos ya que elimina polígonos que no necesitan ser procesados.

Primitive Assembly: en el montaje de primitivas estas son divididas en primitivas más simples que se envían al rasterizador.

Face Culling: después de todas las transformaciones, los triángulos tienen una dirección a la que encaran. Esto está definido por el orden de sus vértices:

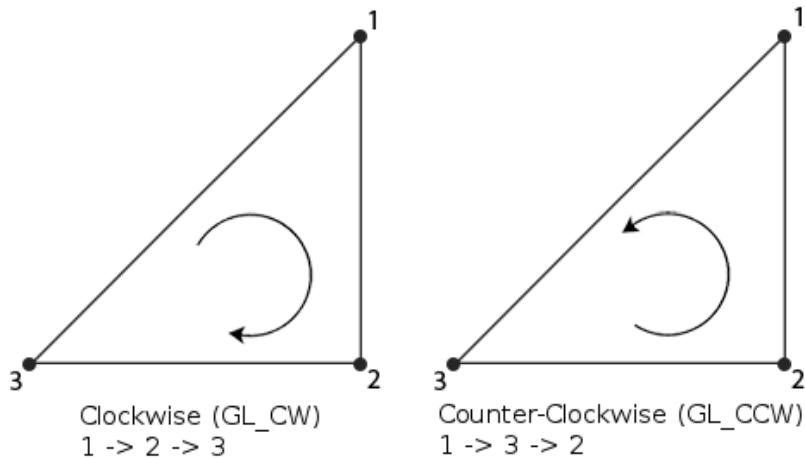


Figura 3.2: La dirección a la que encaran los triángulos depende en el orden de los vértices.(Referencia 15)

Por defecto *OpenGL* utiliza la dirección contraria a las agujas del reloj para definir que un triángulo esta encarado hacia el frente. Y eliminara todos los triángulos que no lo están. Estas reglas se pueden cambiar, incluso forzando a *OpenGL* a mostrar los triángulos por ambas caras.

Rasterización

En esta etapa las primitivas se discretizan formando conjuntos de cuadrados conocidos como fragmentos y se les asigna un color que puede ser definido por una textura o por los vértices más cercanos. Además se eliminan todos los fragmentos que están cubiertos por otros utilizando el buffer Z, una lista en la que se compara la distancia de todos los fragmentos en una posición en específica y solo se guardan los más cercanos a la cámara. Al final de este paso los fragmentos se convierten en los píxeles que conforman la imagen que será mostrada por pantalla o guardada en un archivo.

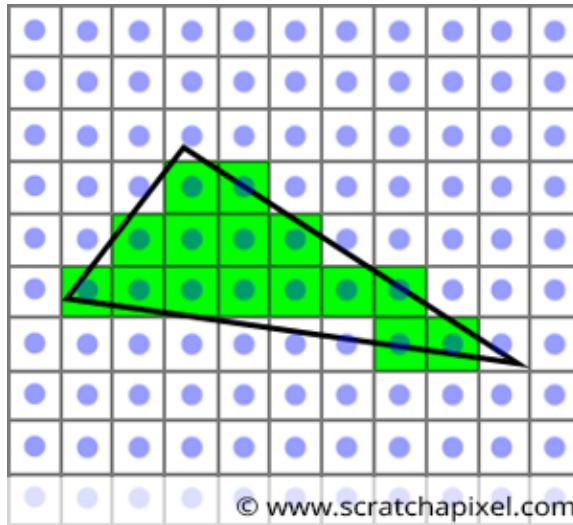


Figura 3.3: Rasterización de un triángulo, transformando la geometría en el conjunto de fragmentos más cercanos a la forma.(Referencia 16)

Los fragmentos representan un segmento de una primitiva, el tamaño de un fragmento está relacionado con un píxel, aunque se pueden producir múltiples fragmentos para un solo píxel.

Fragment shader

Una vez se han rasterizado las primitivas el *Fragment Shader* se encarga de realizar operaciones para cada fragmento. Recibe como entrada las salidas del *Vertex Shader* interpoladas.

Cada fragmento de entrada se transforma en un fragmento simplificado de salida que contiene uno o más colores asociados y con valor de profundidad (la distancia a la que se encuentra de la cámara).

Interpolación en OpenGL: La interpolación en *OpenGL* utiliza coordenadas baricéntricas. Cada fragmento sobre el triángulo recibe unas coordenadas que servirán como pesos para calcular la interpolación de las variables. De manera que si queremos generar la variable x interpolada para el fragmento en la posición (w_1, w_2, w_3) a partir de las salidas de los vértices x_1, x_2 y x_3 , utilizaremos:

$$x = w_1 * x_1 + w_2 * x_2 + w_3 * x_3$$

Per-sample operations

Con los fragmentos generados se realizan varias operaciones para generar una lista de pequeños cuadrados de color que forman una imagen. Estos cuadrados pequeños se llaman píxeles y cada uno almacena un valor de color que se acabará mostrando en su posición en la imagen. En este paso también se realizan para comprobar si los píxeles se encuentran dentro del área visible que *OpenGL* y no interfiriendo con la ventana de otro programa si se encuentran cubiertos por otros píxeles, en ambos casos serán eliminados.

Además podemos aplicar máscaras para escribir o evitar que se escriban píxeles con valores de color o profundidad específicos en las listas pudiendo, por ejemplo, bloquear un todos los píxeles de un color en concreto de la imagen de salida.

3.2. La técnica Mesh Colors

Creada por Cem Yuksel, John Keyser y Donald H. House en 2008 como una alternativa al pintado de textura clásico en la que se elimina la necesidad de crear mapas de UVs, eliminando así la posibilidad de errores de continuidad y permitiendo aplicar detalles sobre regiones específicas dejando el resto con una resolución ² menor. En *Mesh Colors*, los colores se asocian directamente a la geometría de manera similar a [1]



Figura 3.4: Ejemplo de un modelo texturizado utilizando *Mesh Colors*.(Referencia 17)

²En el contexto del proyecto utilizaremos la palabra resolución para referirnos a un valor numérico que representa el nivel de detalle.

Mesh Colors parte de la base de *Vertex Colors* en la que cada vértice en el modelo recibe una muestra de color que se interpola en el fragment shader.

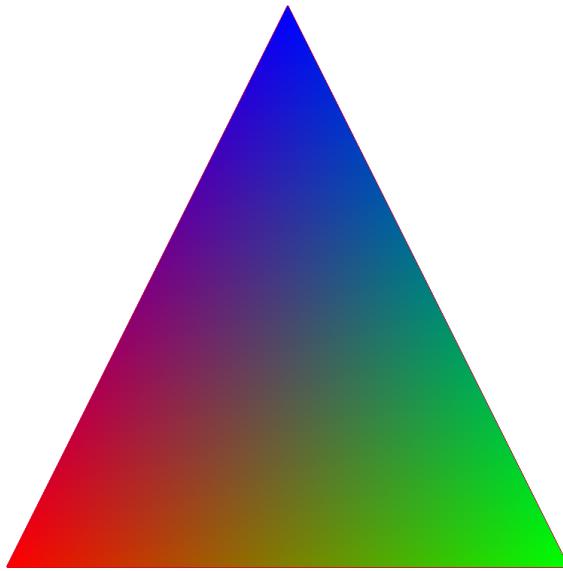


Figura 3.5: Triángulo pintado utilizando *Vertex Colors* y asignando los colores rojo verde y azul en cada vértice y dejando a *OpenGL* encargarse de la interpolación para el resto de la superficie.

Mesh Colors incluye muestras de color definidas para las aristas y para las caras junto con las incluidas en los vértices de *Vertex Colors*, la cantidad de muestras por arista y por cara dependen de la resolución local de cada triángulo.

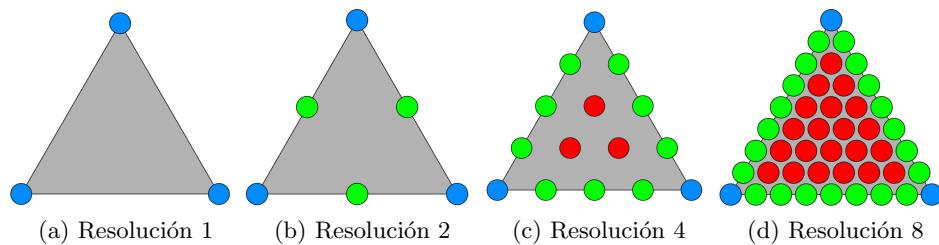


Figura 3.6: Los puntos representan las muestras de color: azul para los vértices, verde para las aristas y rojo para las caras. Partiendo de la resolución 1 (a) en la que el dibujado se comporta como *Vertex Colors*, cada vértice tiene un color, podemos incrementar la resolución para aumentar el número de muestras.(Referencia 17)

Este esquema se puede adaptar fácilmente a modelos no triangulares simplemente dividiendo los cuadrados en triángulos y tratando cada triángulo como su propia cara.

El número de muestras en el triángulo está directamente relacionado con su resolución de forma que:

Siendo R la resolución de la cara.

- Muestras por vértice: 1
- Muestras por arista: $R - 1$
- Muestras por cara: $((R - 1)(R - 2))/2$

Las muestras de color se distribuyen por la cara utilizando una conversión de coordenadas baricéntricas a los índices de una matriz que almacena las muestras. La posición en coordenadas baricéntricas dentro del triángulo de la muestra $C(i,j)$ se calcula mediante:

$$P_{ij} = \left[\frac{i}{R}, \frac{j}{R}, 1 - \frac{i+j}{R} \right]$$

Siendo R la resolución de la cara y los valores i y j número enteros tal que $0 \leq i \leq R$ y $0 \leq j \leq R$.

- Los colores $C(0,0)$, $C(R,0)$ y $C(0,R)$ representan las muestras de color de los vértices.
- $C(0,k)$, $C(k,0)$ y $C(k, (R-k))$ siendo k un entero tal que $0 \leq k \leq R$ para las aristas.
- El resto representan las muestras de color de la cara.

Por el contrario para calcular la muestra correspondiente al punto P , para un triángulo con resolución R , tenemos que realizar las siguientes operaciones:

1. partiendo del punto $P = (i, j, k)$,
2. calculamos la parte entera $B = [i, j, k] = \lfloor RP \rfloor$
3. y la parte fraccionaria $w = [w_i, w_j, w_k] = RP - B$

Una vez tenemos w podemos distinguir tres casos:

- $w = 0$, nos encontramos directamente sobre el centro de la muestra siendo B el índice que representa la muestra.

- $w_i + w_j + w_k = 1$, los colores más cercanos son $C_{(i+1)j}, C_{i(j+1)}$, y C_{ij} y cada elemento de w representa los pesos de cada color respectivamente.
- $w_i + w_j + w_k = 2$, los colores más cercanos son $C_{i(j+1)}, C_{(i+1)j}$ y $C_{(i+1)(j+1)}$ y sus pesos son $w0 = [1, 1, 1] - w$.

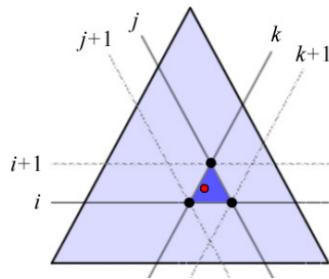


Figura 3.7: Selección de las 3 muestras adyacentes, los puntos oscuros, que influyen el color de en la posición del punto rojo. (Referencia 17)

Con esta fórmula y utilizando los pesos podemos obtener el color interpolando entre las muestras con un filtro lineal. Si utilizamos la muestra con mayor peso para cada punto utilizaremos el filtro “color más cercano” y se formarán patrones de colores planos.

Una vez calculado el color de esa posición lo asignamos, como salida final, al color del píxel correspondiente.

4. Implementación

En este apartado vamos a explicar el proceso de creación de la aplicación y su código, qué tecnologías se han utilizado y cómo se ha implementado *Mesh Colors*. Como ya se ha mencionado antes, *OpenGL* y la programación en la GPU son un apartado muy importante de este proyecto, por lo que gran parte del apartado se centrará en la implementación de los *shaders* y cómo los hemos conectado con el código en *C++*.

4.1. Herramientas

El programa está construido sobre **C++17** con **Qt 5.15** para la estructura, la funcionalidad y la interfaz de usuario y **OpenGL 4.6** para la visualización.

La estructura básica fue proporcionada por el profesor. Estos archivos proporcionan una base en la que se pueden visualizar varios objetos 3D en distintos modos: visualizando sólo los vértices, solo las aristas o las caras rellenas.

Se parte de esta base para desarrollar el proyecto, creando archivos nuevos y modificando o extendiendo los existentes.

4.2. Visualización 3D

Para mostrar los elementos tridimensionales en la pantalla utilizamos varias clases para almacenar la información de los modelos y conectar esa información con los *shaders* de *OpenGL*.

4.2.1. Objetos 3D

Todos los objetos que aparecen en el visualizador 3D tienen una clase en común, `_basic_object3D`, esta clase simplemente contiene una lista de vértices del tipo `QVector3D` y otra lista de colores del tipo `QVector4D`. Esta es la única información necesaria para visualizar un objeto tridimensional de forma directa, por ejemplo la clase `axis` que son los ejes de coordenadas que utilizaremos como punto de referencia para la escena 3D.

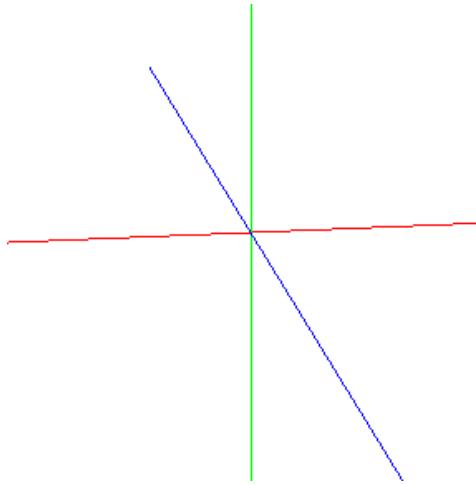


Figura 4.1: Visualización de los ejes de coordenadas.

Este objeto se crea de forma muy simple, inicializando directamente los vértices y los colores en el constructor de la clase `axis` que hereda de `_basic_object3D`.

```
_axis::_axis(float Size)
{
    vertices.resize(6);
    vertices[0] = QVector3D(Size, 0.0, 0.0); // x
    vertices[1] = QVector3D(-Size, 0.0, 0.0);
    vertices[2] = QVector3D(0.0, Size, 0.0); // y
    vertices[3] = QVector3D(0.0, -Size, 0.0);
    vertices[4] = QVector3D(0.0, 0.0, Size); // z
    vertices[5] = QVector3D(0.0, 0.0, -Size);

    colors.resize(6);
    colors[0] = QVector3D(1, 0, 0); // red
    colors[1] = QVector3D(1, 0, 0);
    colors[2] = QVector3D(0, 1, 0); // green
    colors[3] = QVector3D(0, 1, 0);
    colors[4] = QVector3D(0, 0, 1); // blue
    colors[5] = QVector3D(0, 0, 1);
}
```

El proceso para crear modelos 3D debería ser externalizado para poder importar los modelos directamente desde otros programas 3D. Crear manualmente los objetos 3D en código como se hizo con los ejes no es compatible con esta idea por lo que tenemos que crear otra clase que pueda albergar modelos externos.

La clase `object3D` también hereda de `_basic_object3D` e implementa la funcionalidad que permite leer archivos de modelos. Estos modelos deben utilizar el formato `ply`.

Formato ply

PLY(*Polygon File Format*) es un formato de imagen 3D desarrollado en Stanford que ofrece un estándar genérico que puede ser exportado en programas comunes como Maya o Blender. Se utiliza para almacenar una lista de polígonos planos, aunque también puede contener información como color y transparencia. Este formato se puede grabar en dos tipos de archivos: de texto, o binarios. Los archivos de texto se pueden leer con lenguaje natural, mientras que los binarios son mucho menos pesados.

La estructura del archivo ply se divide en tres partes, utilizaremos texttt-Cube.ply, el modelo 3D de un cubo, como ejemplo para describir el formato ply:

```
ply
format ascii 1.0
```

estructura las dos primeras líneas indican que el archivo es del tipo ply y qué versión en concreto se ha utilizado para su creación, en este caso la versión.

```
ascii 1.0.
element vertex 8
property float x
property float y
property float z
element face 12
```

Las siguientes líneas muestran cómo están almacenados los datos, la cantidad de vértices y caras que tiene el modelo y qué tipo se utiliza para almacenar los vértices, en este caso es un número de coma flotante.

```
property list uchar int vertex_indices
```

Esta línea define cómo se componen los datos de las caras, en este caso podemos ver que es cada cara es una lista cuyo primer elemento es un uchar que contiene la cantidad de vértices que tiene cada cara. **int** y **vertex_indices** indica que los siguientes enteros en la línea forman la lista de índices que conforman la cara .

```
end_header
-1 -1 -1
.
.
.
```

```
1 1 -1
3 0 1 3
.
.
.
3 5 0 4
```

Después del fin de la cabecera (`end_header`) encontramos la lista de vértices seguida de la lista de caras con los formatos correspondientes a los definidos en la cabecera[7]. Para leer estos archivos utilizamos la clase `_file_ply` proporcionada por el tutor que permite recibir el flujo de entrada de un archivo y extraer de los datos la lista de vértices y los triángulos formados por esos vértices. A diferencia de la clase base, en `Object3D`, el modelo proporciona información de qué vértices pertenecen a cada triángulo. Partiendo de la lista de vértices y triángulos creamos una nueva lista que contiene para cada triángulo las coordenadas de los vértices que lo forman, esta lista es la que luego se le entrega a *OpenGL* para dibujar el modelo.

```
for (int i = 0; i < triangles.length(); i++) {
    const QVector3D& triangle = triangles[i];
    verticesDrawArrays[i * 3] = vertices[triangle.z()];
    verticesDrawArrays[i * 3 + 1] = vertices[triangle.y()];
    verticesDrawArrays[i * 3 + 2] = vertices[triangle.x()];
}
```

Además de los colores de los vértices esta clase también almacena por cada triángulo un color en sus vértices que sirve para identificarlo después de dibujar el modelo en la pantalla, permitiendo al usuario seleccionar triángulos con el clic del ratón. La selección se explicará en detalle en el apartado sobre `GL_Widget`.

Normales

Por último, esta clase almacena la información de las normales para la iluminación del modelo. En geometría las normales representan un vector perpendicular a un plano, en este caso las normales que se utilizan en la aplicación son las perpendiculares al plano de cada triángulo[8].

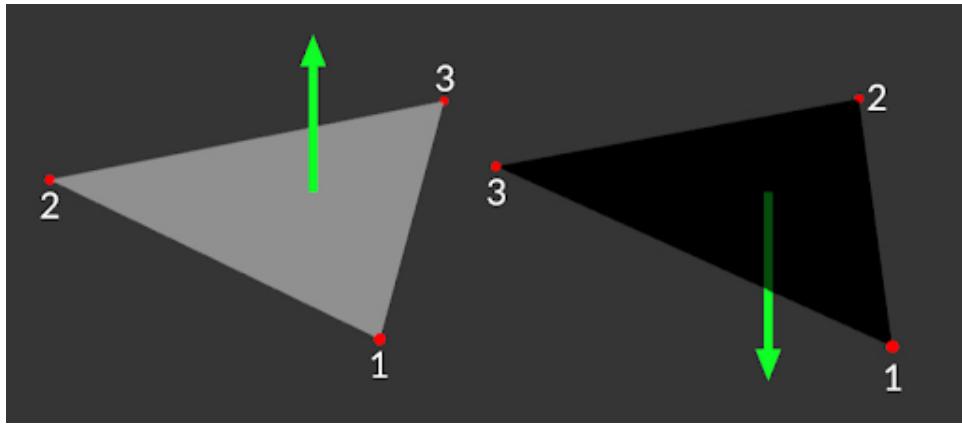


Figura 4.2: Comparación de la dirección de las normales en función del orden de los vértices.(Referencia 18)

Siendo p_1, p_2, p_3 , los vértices del triángulo y dados los vectores $U = p_2 - p_1$ y $V = p_3 - p_1$ la normal se calcula como el producto cruzado de U y V ($N = UXV$):

$$\begin{aligned}N_x &= U_y * V_z - U_z * V_y \\N_y &= U_z * V_x - U_x * V_z \\N_z &= U_x * V_y - U_y * V_x\end{aligned}$$

En código calculamos la normal utilizando directamente la función de producto cruzado que proporciona la clase `QVector3D`.

```
QVector3D& v1 = vertices[triangle.x()] - vertices[triangle.y()];
QVector3D& v2 = vertices[triangle.y()] - vertices[triangle.z()];
QVector3D& Normal = QVector3D::crossProduct(v1, v2).normalized();
```

La información de las normales se utiliza para iluminar el modelo, se explicará este proceso en profundidad en el apartado sobre los shaders.

MeshColorsFace

La técnica MeshColors requiere almacenar y editar información directamente en los modelos, para ello se utilizan la clase `MeshColorsFace` y La clase `MeshColorObject3D`. La primera se maneja la información que necesita cada cara del modelo, contiene la resolución de la cara y los índices de cada muestra de color en el vector de colores que se envía a *OpenGL*. Cuando se actualiza un color o una resolución esta clase se encarga de actualizar el color de cada muestra y su tamaño si es necesario.

Incrementar y decrementar resolución Cuando se incrementa o reduce la resolución de una cara la clase intenta aproximar la forma del dibujo en la cara con la nueva resolución. Esto permite mantener mantener un dibujo aproximado al creado por el artista al aumentar o reducir la resolución. Para incrementar:

1. Partiendo de la resolución actual, todas las muestras que tenemos se utilizarán como base para generar la resolución nueva, las llamamos muestras clave.

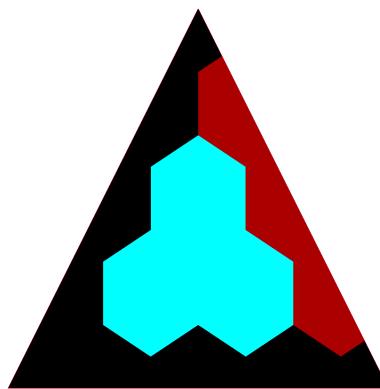


Figura 4.3: Estado inicial.

2. Las muestras clave se reducen de tamaño pero mantienen su posición, dejando huecos entre ellas que se llenan con muestras generadas

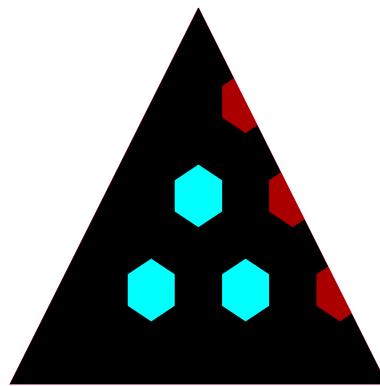


Figura 4.4: Muestras clave con color generadas tras incrementar la resolución.

3. Una vez tenemos las muestras clave podemos colorear las generadas. Cada muestra generada se colorea utilizando el color medio de las muestras clave adyacentes.

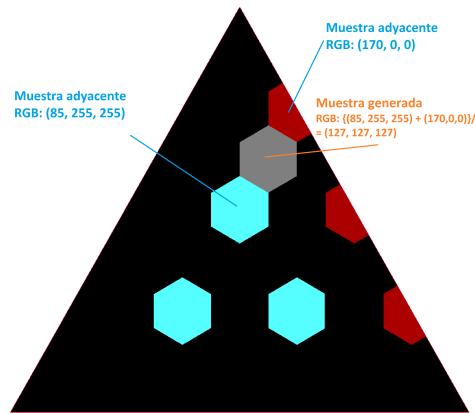


Figura 4.5: Cálculo de una muestra generada en función de las muestras clave adyacentes.

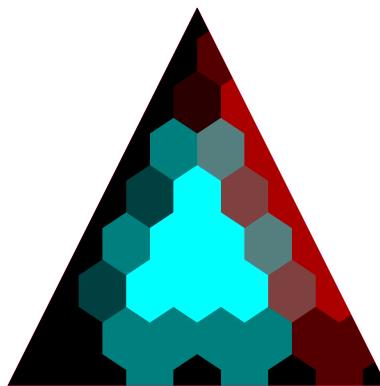


Figura 4.6: Resultado de aplicar el paso 3 a todas las muestras generadas.

Para la reducción de resolución, las muestras clave que tenemos ahora llenarán por completo la cara en la resolución inferior. Para ello:

1. Las muestras clave calculan su nuevo valor utilizando una media del color de todas las muestras adyacentes y el suyo propio.

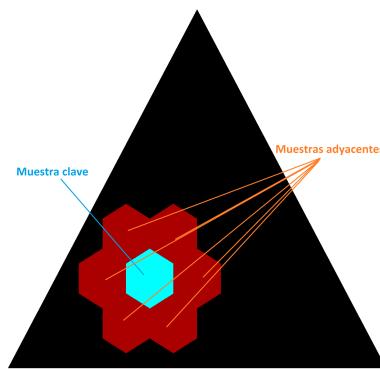


Figura 4.7: Cálculo del color de la muestra clave.

2. La muestra clave aumenta de tamaño reemplazando a las muestras adyacentes.

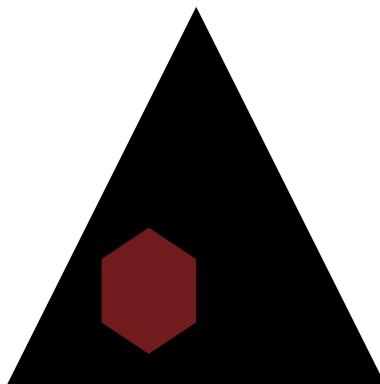


Figura 4.8: Cálculo de una muestra para la resolución inferior.

3. Realizamos esta operación para todas las muestras clave.

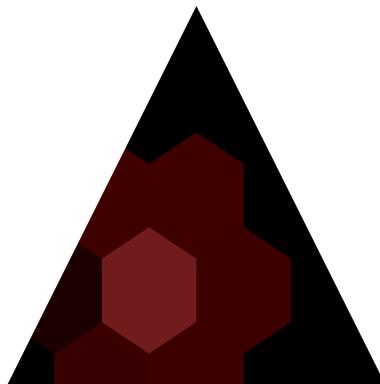


Figura 4.9: Resultado de la reducción de resolución.

Con esta clase podemos gestionar la información de MeshColors en cada cara. Para el objeto completo utilizamos `MeshColorObject3D`.

MeshColorObject3D

`MeshColorObject3D` hereda de la clase `Object3D` y expande su funcionalidad añadiendo una lista en la que cada elemento representa el `MeshColorFace` de un triángulo.

Esta clase implementa los elementos que se enviarán al *shader* para dibujar los colores del modelo, estos elementos se encuentran dentro de una estructura a la que llamamos SSBO.

El SSBO (*Shader Storage Buffer Object*) es un tipo especial de *Buffer Object*, un objeto de *OpenGL* que permite almacenar *arrays*. Lo que hace especial a SSBO y la razón para usarlo en este proyecto es que permite transmitir al shader cantidades de información mucho mayores. Mientras que los Buffer Objects básicos como UBOs(*Uniform Buffer Object*) tienen un límite de 16kb, SSBO permite utilizar la memoria necesaria hasta alcanzar el límite de la GPU lo que nos da la posibilidad de almacenar un array que tenga las muestras de color de un modelo con miles de caras.

La GPU es muy rápida a la hora de procesar muchos datos, por eso los SSBO permiten en pocas instrucciones almacenar una gran cantidad de datos de forma local que la GPU podrá acceder durante la ejecución de los *shaders*.

Para crear el SSBO necesitamos crear una estructura en C++ con los datos que queremos enviar a los *shaders*.

```
typedef struct ssbo_data
{
    int Resolution[MAX_TRIANGLES];
    QVector4D Colors[MAX_TRIANGLES][MAX_SAMPLES][MAX_SAMPLES];
} ssbo_type;
```

El tamaño de la estructura tiene que ser predefinido para poder conectarlo con el *shader*, por lo que existe un límite de tamaño máximo que tenemos que definir para los modelos 3D que cargamos, al igual que un límite de la resolución que el usuario puede aplicar a las caras.

El límite lo definimos con las constantes `MAX_TRIANGLES` y `MAX_SAMPLES` que permiten definir el tamaño (cantidad de caras) del modelo y la resolución máxima de cada cara. Las listas pueden albergar, como máximo, un modelo con `MAX_TRIANGLES` caras en las que cada una tiene la resolución `MAX_SAMPLES`.

4.2.2. Gl Widget

La clase `GLWidget` se encarga de conectar el código de C++ y Qt con *OpenGL*, en lugar de utilizar las llamadas de *OpenGL* directamente utilizamos la clase `QOpenGLWidget`, que es una interfaz que ofrece Qt para manejar las llamadas a *OpenGL* [9].

`QOpenGLWidget` proporciona funciones virtuales básicas para implementar en las clases que la heredan.

- `initializeGL`: se llama primero al iniciar el programa es y se encarga de preparar la configuración y datos para el dibujado.
- `resizeGL` cada vez que la ventana cambia de tamaño se invoca para actualizar el *viewport*, que es la ventana en la que *OpenGL* está dibujando el mundo tridimensional.
- `paintGL` se encarga de dibujar el modelo, tiene las llamadas que envían la información del modelo a los *shaders* y se llama cada vez que el programa se actualiza.

Partiendo de esta base, `GLWidget` es la clase más compleja del proyecto y presenta gran parte de la funcionalidad que forma el programa. En concreto la funcionalidad para dibujar requiere que realicemos una serie de pasos para poder visualizar y editar los datos. Primero necesitamos crear un contexto.

Contexto de OpenGL

El contexto de *OpenGL* es una estructura de datos que agrupa todos los estados que definen la conexión entre el sistema y *OpenGL*, contiene referencia a *buffers*, texturas, *shaders*, etc. Es necesario que exista un contexto para poder ejecutar comandos de *OpenGL*[10].

Para crear el contexto utilizamos `QOpenGLContext`, que permite crear y manejar un contexto nativo de *OpenGL*. Esto nos permite acceder a funciones específicas del contexto actual que necesitaremos para poder enviar información a los *shaders*.

Lo siguiente que tenemos que obtener son los *shaders*. Los *shaders* son archivos de código que tenemos en el proyecto pero no se compilan durante la compilación del resto del programa, sino que se cargan y compilan durante la ejecución utilizando `QOpenGLShaderProgram`. Esta clase permite al programador compilar y enlazar programas de *OpenGL* de forma simple, en este caso:

```
void _gl_widget::loadProgram()
```

```

{
    context = new QOpenGLContext(this);

    const QString& path = QString(PROJECT_PATH) + "src/shaders/";

    meshColorsShader = new QOpenGLShaderProgram(context);

    meshColorsShader->addCacheableShaderFromSourceFile(QOpenGLShader::Vertex, path + "MeshColorsVertex.vsh");
    meshColorsShader->addCacheableShaderFromSourceFile(QOpenGLShader::Geometry, path + "MeshColorsGeometry.gsh");
    meshColorsShader->addCacheableShaderFromSourceFile(QOpenGLShader::Fragment, path + "MeshColorsFragment.fsh");
    meshColorsShader->link();

    basicRenderingShader = new QOpenGLShaderProgram(context);

    basicRenderingShader->addCacheableShaderFromSourceFile(QOpenGLShader::Vertex, path + "BaseVertex.vsh");
    basicRenderingShader->addCacheableShaderFromSourceFile(QOpenGLShader::Fragment, path + "BaseFragment.fsh");
    basicRenderingShader->link();
}

```

Una vez hemos creado el contexto podemos crear los *shaders*, después solo tenemos que añadirlos al programa. Existen dos funciones dentro de la clase `QOpenGLShader` que permiten añadir los *shaders* `addShaderFromSourceFile` y `addCacheableShaderFromSourceFile`, la única diferencia es que la primera también los compila al añadirlos, mientras que la segunda delega eso para la función `link`, donde la compilación no es obligatoria en cada ejecución ya que guarda la compilación anterior y comprueba si es necesario actualizarla. Por lo tanto la segunda opción puede ser una optimización frente a la primera si no cambiamos mucho el *shader*.

En este proyecto se han utilizado dos `QOpenGLShaderPrograms`, uno que contiene los *shaders* para visualizar las muestras de *MeshColors* y el segundo es un *shader* básico de visualización para mostrar objetos que no utilizan la técnica *MeshColors*, como los ejes de coordenadas.

VAO y VBO

Los *Buffer Objects* de *OpenGL* son *arrays* de bloques de memoria sin formato que son asignado por el contexto de *OpenGL*, en estos bloques se puede enviar información como datos de los vértices. Un tipo de *Buffer Objects* son los *Vertex Buffer Object*(VBO) que se refiere en específico a cuando estos bloques de memoria contienen listas de vértices [?].

Vertex Array Object(VAO) es un objeto que contiene uno o más VBOs y está diseñado para agrupar la información de un objeto completamente visualizable. En el caso de este proyecto vamos a utilizar un VAO para cada objeto que se dibuja en pantalla:

- El modelo 3D en el que vamos a pintar.

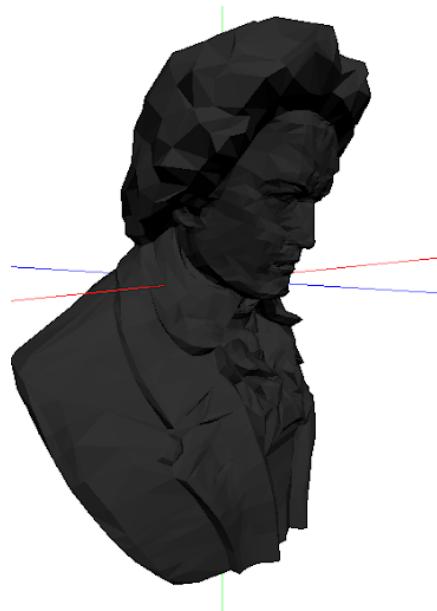


Figura 4.10: Modelo 3D.

- El *wireframe* del modelo 3d: es el mismo modelo pero solo se dibujan las aristas.

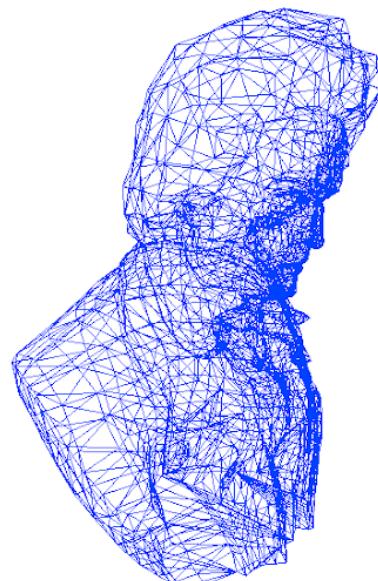


Figura 4.11: Aristas del modelo 3D.

- Una versión del modelo 3D donde cada cara recibe color que representa un índice, se utiliza para seleccionar los triángulos a los que editaremos la resolución.

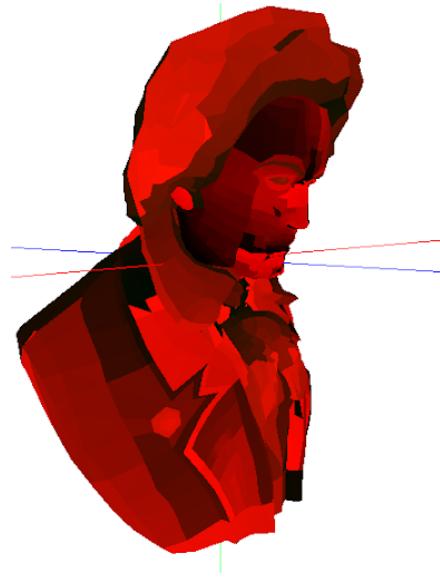
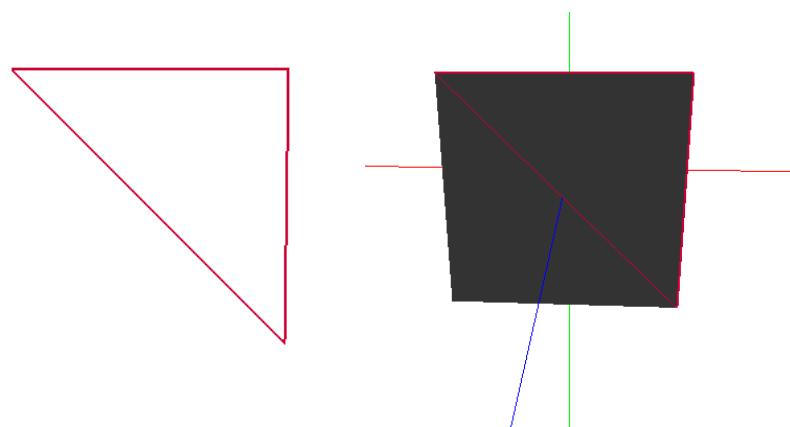


Figura 4.12: Representación de color de los índices de las caras

- Las aristas del triángulo seleccionado.



(a) Aristas del triángulo seleccionado. (b) Aristas del triángulo seleccionado sobre el modelo 3D.

Figura 4.13: Triángulo seleccionado.

- Los ejes de coordenadas.

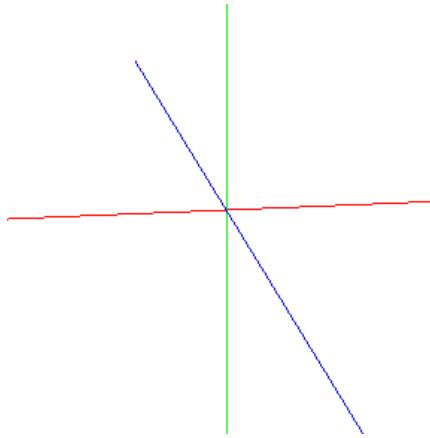


Figura 4.14: Ejes de coordenadas

Para implementar VAO y VBO necesitamos tener definido un contexto y el programa sobre el que se aplican, por ejemplo, en el caso de los ejes de coordenadas utilizamos el programa que contiene los *shaders* de visualización básico. Los VAO y su contenido pueden ser utilizados por distintos programas pero siempre dentro del mismo contexto y teniendo en cuenta que los atributos que el *shader* espera recibir tienen el mismo nombre en ambos programas.

Creación del VAO de los ejes:

```
axisVAO = new QOpenGLVertexArrayObject();
if (axisVAO->create()) {
    axisVAO->bind();

    initializeBuffer(basicRenderingShader, Axis.vertices.data(), ←
        Axis.vertices.size() * sizeof(QVector3D), "vertex", GL_FLOAT, ←
        0, 3);
    initializeBuffer(basicRenderingShader, Axis.colors.data(), Axis.←
        colors.size() * sizeof(QVector4D), "color", GL_FLOAT, 0, 4);

    axisVAO->release();
}
```

Este contiene dos VBO:

- Para los vértices, llamado `vertex` en el shader..
- Para los colores llamado `color` en el shader.

Si queremos utilizar este VAO en otro programa tendría que recibir como entrada los atributos `vec3 vertex` y `vec4 color`. Todos los buffers creados entre el enlace del VAO (`VAO->bind()`) y la liberación (`VAO->release()`) pertenecerán a ese VAO en concreto.

El proceso se puede resumir como, crear un contexto que sirve de puente de conexión con *OpenGL* dentro de ese puente se cargan los programas de la gpu, para cada objeto de la escena 3D se creará un VAO que contiene un conjunto de VBO con información sobre los vértices del modelo.

Una vez hemos completado estos pasos podemos dibujar el objeto enlazando el programa y el VAO dentro del contexto y llamando a `glDrawArrays()` que dibujará la primitiva indicada para todos los elementos de los *arrays* de vértices que hemos definido antes, en este caso dibujarán líneas (`GL_LINES`) para todos los vértices del objeto Axis.

```
void _gl_widget::drawAxis()
{
    if (basicRenderingShader != nullptr && axisVAO != nullptr) {
        basicRenderingShader->bind();
        axisVAO->bind();

        glLineWidth(1.0f);

        basicRenderingShader->setUniformValue("matrix", camera.getProjectedTransform());
        basicRenderingShader->setUniformValue("singleLineColor", false);
        basicRenderingShader->setUniformValue("lineColor", QVector4D(0.0f, 0.0f, 0.0f, 0.0f));

        glDrawArrays(GL_LINES, 0, Axis.vertices.size());

        axisVAO->release();
        basicRenderingShader->release();
    }
}
```

Los otros elementos en esta función sirven para definir parámetros que se utilizan sobre todos los vértices del modelo y mantienen su valor, se conocen como valores `uniform`. En este caso tenemos la matriz que define la posición y rotación del modelo, el color de la línea y si ese color se utiliza para todas las líneas del modelo, para los ejes esto no es necesario ya que cada línea tiene un color distinto.

También utilizamos `glLineWidth(1.0f)` para definir el grosor de la línea, este es otro valor `uniform` que *OpenGL* maneja internamente.

Iluminación y cámara

`GlWidget` también controla la posición y rotación de la cámara y la fuente de luz. Ambos son una instancia de la clase `MovableObject` creada para este proyecto con la finalidad de tener objetos en la escena que no tienen modelo 3D y que se pueden mover por ella. La clase contiene la matriz de transformación y su versión proyectada en 2D, además de la posición en el mundo 3D. En la cámara utilizamos la matriz de transformación proyec-

tada para el parámetro `uniform matrix` que se utilizará como matriz de proyección dentro de los *shaders*.

Mientras que para la iluminación utilizamos la posición en el *shader* para saber en qué dirección incide la luz.

Entrada de usuario

El usuario puede controlar la posición de la luz y la cámara, además de otros parámetros utilizando la entrada del teclado y del ratón. `GLWidget` es un `QWidget`, y por lo tanto puede heredar eventos que reaccionan a la entrada del teclado o el ratón del usuario.

```
protected:
    /** Start QOpenGLWidget Interface ***/
    void resizeGL(int Width1, int Height1) Q_DECL_OVERRIDE;
    void paintGL() Q_DECL_OVERRIDE;
    void initializeGL() Q_DECL_OVERRIDE;
    void keyPressEvent(QKeyEvent *Keyevent) Q_DECL_OVERRIDE;
    void keyReleaseEvent(QKeyEvent *Keyevent) Q_DECL_OVERRIDE;
    void mouseMoveEvent(QMouseEvent *e) Q_DECL_OVERRIDE;
    void mousePressEvent(QMouseEvent *e) Q_DECL_OVERRIDE;
    void mouseReleaseEvent(QMouseEvent *event) Q_DECL_OVERRIDE;
    void wheelEvent(QWheelEvent *event) Q_DECL_OVERRIDE;
/** End QOpenGLWidget Interface **/
```

Podemos definir reacciones para eventos como pulsar o dejar de pulsar una tecla, mover el ratón, girar la rueda del ratón e incluso cambiar el tamaño de la ventana. Este último se utiliza para ajustar el *viewport* de la escena para mantener las proporciones.

En el caso de los eventos de teclado, un *switch* recibe la tecla pulsada y en función de su valor dentro de una lista que proporciona Qt podemos saber qué tecla se ha pulsado:

```
switch(Keyevent->key()){
    case Qt::Key_Left:lightAngleY += 10.0f * ANGLE_STEP; break;
    case Qt::Key_Right:lightAngleY -= 10.0f * ANGLE_STEP; break;
    case Qt::Key_Up:lightAngleX += 10.0f * ANGLE_STEP; break;
    case Qt::Key_Down:lightAngleX -= 10.0f * ANGLE_STEP; break;
    case Qt::Key_PageUp:lightDistance *= 1.2; break;
    case Qt::Key_PageDown:lightDistance /= 1.2; break;
```

Mientras que en la del ratón recibimos un evento distinto que, también tiene información de las teclas pulsadas además de la posición actual del cursor:

```
void _gl_widget::mousePressEvent(QMouseEvent *e)
{
    if (e != nullptr && underMouse()) {
```

```
    const float x = e->pos().x();
    const float y = height() - e->pos().y();

    if (e->buttons() & Qt::LeftButton && !ControlPressed) {
        selectSample(x, y);
    }
    else if (e->buttons() & Qt::MiddleButton) {
        selectTriangle(x, y);
    }
}
```

La información de la posición devuelve el píxel en concreto en el que estaba en cursor cuando se produjo el evento. En la coordenada Y la dirección es la opuesta a la que *OpenGL* utiliza por lo que tenemos que invertir restando la altura total de la ventana para obtener la correcta.

Selección de elementos

El evento de pulsar la tecla del ratón es especialmente importante para el proyecto ya que es la acción fundamental para poder pintar sobre el modelo. La selección de triángulos y la selección de muestras funcionan en su mayor parte de forma similar. A ambos se les proporciona un identificador entero durante la creación del objeto 3D, este identificador luego se convierte en un color utilizando la siguiente conversión:

```
const float r = ((i & 0x000000FF) >> 0) / 255.0f;
const float g = ((i & 0x0000FF00) >> 8) / 255.0f;
const float b = ((i & 0x00FF0000) >> 16) / 255.0f;

QVector4D TriangleID = QVector4D(r, g, b, 1.0f);
```

Aquí convertimos el número entero i , en formato RGBA. Explicaremos este proceso con un ejemplo:

Queremos codificar el triángulo con índice 1368 que en binario es 10101011000,

- Empezando por el valor de la componente roja, realizamos la operación binaria & con el valor hexadecimal 0x000000FF. Esta operación compara los dos valores bit a bit y retorna 1 si los dos son 1 y 0 en otro caso. Por lo tanto para el valor hexadecimal 0x000000FF, que en binario es 0000000011111111:

`0000000011111111 & 10101011000 = 1011000`

- Al convertir 1011000 a decimal obtenemos 88. Este sistema codifica cada color con un entero de 2 bytes, por lo que el valor máximo posible es 255. Por lo tanto tenemos que dividir el valor obtenido por 255 para

conseguir un valor entre 0 y 1. En este caso el valor rojo en el color será $88/255 = 0,345$.

- Pasamos ahora al verde, esta vez utilizamos la máscara 0x0000FF00 que en binario es 1111111100000000:

$$1111111100000000 \& 10101011000 = 10100000000$$

- A este resultado se le aplica una operación de desplazamiento a la derecha de 8 bits:

$$\textcolor{red}{1010}00000000 \gg 8 = 00000000\textcolor{red}{1010}$$

- El resultado, 1010, en decimal es 10. Por lo que el valor de la componente verde del color sería $10/255 = 0,0392$.
- La componente azul en este caso no es relevante ya que necesitaríamos un número que fuera relevante al multiplicar con la máscara 0x00FF0000 que requiere un valor mayor de 16711680. Por lo que el valor de este componente, para este ejemplo, es 0.
- El componente alfa se utiliza para diferenciar entre el modelo 3D y el resto de elementos. Como la escena no tiene en cuenta la transparencia a la hora de visualizar, este valor se utiliza para marcar cuando un objeto es relevante para la selección, cuando el valor de la componente alfa es 1.

Por lo tanto la codificación RGBA del índice 1368 sería (0.345, 0.0392, 0.0, 1.0)

Una vez tenemos el código RGBA para todos los elementos del modelo podemos mostrarlos por pantalla y seleccionar píxeles de la imagen para obtener el identificador de los elementos que hay detrás. Este es el resultado al asignar índices a cada muestra de color en un modelo:

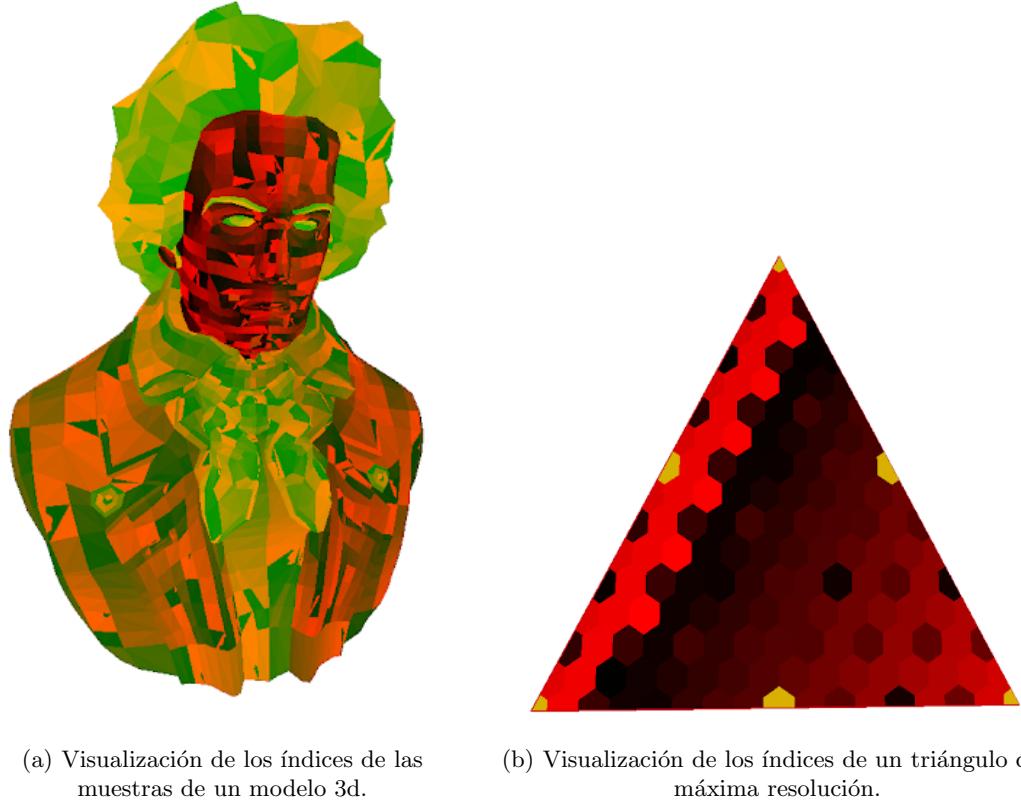
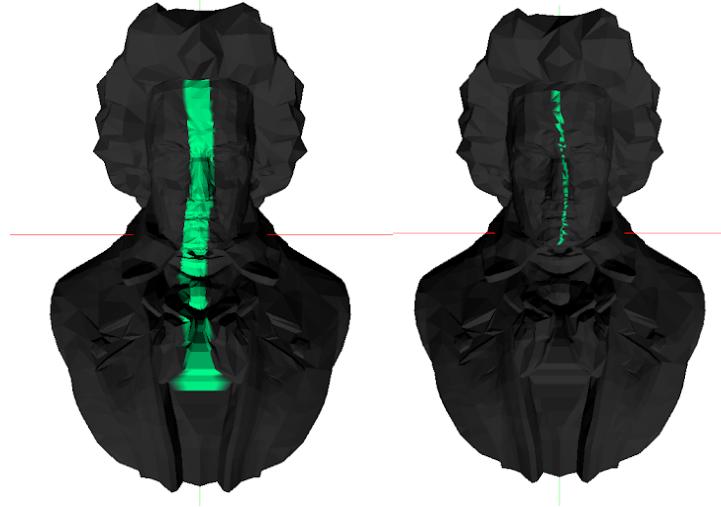


Figura 4.15: Visualización de índices de las caras y las muestras.

Una vez tenemos esta información podemos utilizar las coordenadas del ratón para recibir los colores de los píxeles y obtener los índices de las muestras o los triángulos correspondientes.

Con la posición del ratón utilizamos `glReadPixels()`, esta función nos permite leer la cantidad de píxeles que queramos en un cuadrado empezando desde la posición indicada. En el caso de la selección de triángulos queremos mantener siempre la precisión máxima por lo que leeremos solo un píxel. Para la selección de muestras, en cambio, el usuario puede querer pintar áreas más grandes en una sola pasada del cursor, por eso utilizamos un variable accesible desde la interfaz de usuario que edita el tamaño de ese cuadrado haciendo que se lean más píxeles.



(a) Tamaño de pincel grande. (b) Tamaño de pincel pequeño.

Figura 4.16: Diferencia de tamaño de pincel al pintar sobre el modelo.

El usuario también puede controlar la opacidad del pincel desde la interfaz. Esto afecta una fórmula a la hora de definir los colores en el dibujado:

$$\text{Color} = (1 - \text{Opacidad}) * \text{colorAntiguo} + \text{Opacidad} * \text{colorNuevo}$$

Es decir, aplica una mezcla en la que el color nuevo representa un porcentaje definido por la opacidad y el color anterior el porcentaje opuesto.

Una vez tenemos los colores actualizados, el objeto 3D actualiza sus muestras de colores correspondientes y se re-dibuja la ventana para que muestre los colores nuevos.

4.2.3. Shaders

Vertex Shader

El **vertex shader** en este proyecto no realiza ningún cálculo, sólo recibe información de los vértices y la transmite al **geometry shader**. El **shader** solo recibe la posición de los vértices y la matriz de transformación y calcula la posición de cada vértice.

```
#version 430
uniform mat4 matrix;
in vec3 vertex;
```

```
in vec3 normals;  
  
out vec3 VertexNormal;  
  
void main(void)  
{  
    gl_Position = matrix*vec4(vertex, 1.0f);  
    VertexNormal = normals;  
}
```

Geometry Shader

Este *shader* se ejecuta una vez por primitiva, nosotros aprovechamos eso para inicializar el índice de la primitiva, que utilizaremos para identificar la cara en el fragment shader, con una variable de *OpenGL* `gl_PrimitiveID` = `gl_PrimitiveIDIn`, es necesario inicializarla así en el vertex de geometría.

Aquí tambien podemos alterar la información de los tres vértices, vamos a crear una capa de color que utilizaremos para simular coordenadas baricéntricas. Asignamos a un vértice el color rojo (1.0,0.0,0.0), a otro el verde (0.0,1.0,0.0) y otro en azul (0.0,0.0,1.0). Con estos colores podemos utilizar las primeras dos componentes de cada color para identificar las posiciones de cada fragmento dentro del triángulo simulando las coordenadas baricéntricas ya que *OpenGL* interpola automáticamente los elementos de los vértices si no se niega explícitamente.

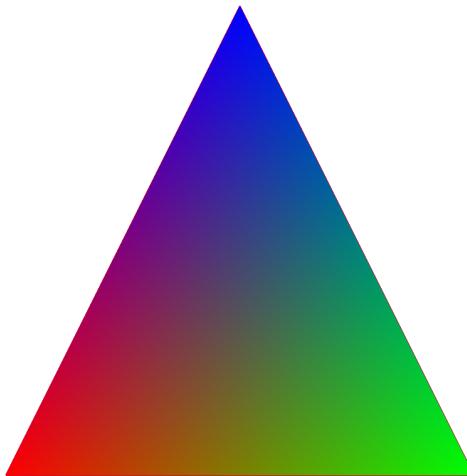


Figura 4.17: Colores aplicados aplicados en el `geometry shader` e interpolados para generar un mapa de coordenadas baricéntricas.

Para evitar la interpolación automática podemos utilizar la palabra clave `flat` al declarar las variables de entrada y salida.

Fragment Shader

Este es el paso más importante para *MeshColors* en este *shader* obtenemos las muestras de color del **ssbo** y decidimos cuáles son las más relevantes para el fragmento actual.

```
#version 430

layout(std430, binding = 3) buffer MeshColorsData
{
    int Resolution[9000];
    vec4 Colors[9000][17][17];
};

uniform bool ColorLerpEnabled;
uniform bool LightingEnabled;
uniform vec3 LightPos;

flat in vec3 Normal[3];
in vec4 fragment_Color;

out vec4 out_Color;
```

Este *shader* es el que conecta con el **ssbo** que creamos en *MeshColorObject3D*. Aquí recibimos la resoluciones de todos los triángulos del modelo junto con todas sus muestras, *OpenGL* obliga a que estos elementos tengan tamaño estático, el tamaño escogido para los *arrays* es suficiente para poder visualizar todos los triángulos de los modelos que utilizamos como ejemplo y que se encuentran en el repositorio del proyecto dibujando las caras con resolución 32. En caso de querer utilizar modelos mayores o con resoluciones mayores basta con dar incrementar esos valores, siempre y cuando respeten el límite máximo de memoria de la GPU.

Además este *shader* recibe varios **uniforms** que definen variables que usaremos para todos los fragmentos. Las variables deciden si queremos interpolación entre las muestras o no, si queremos activar la iluminación y la posición actual de la fuente de luz.

Como entrada desde el vertex shader recibe *fragmentColor* que representa la coordenada baricéntrica del fragmento actual.

Y como salida retorna el color del píxel que se mostrará en pantalla.

El cuerpo del *shader* parece complejo, pero en realidad basta con seguir la fórmula de *MeshColors* explicada anteriormente para la selección de las muestras más cercanas para cada fragmento:

- La versión interpolada utiliza la fórmula 3.2 para calcular el color de la muestras utilizando los colores más cercanos y sus pesos:

```
if(ColorLerpEnabled)
```

```

{
    if( round(w.r+w.g+w.b) == 0)
    {
        c = Colors[ SampleIndex ][ int(B[0]) ][ int(B[1]) ];
    }
    else if( round(w.r+w.g+w.b) == 1)
    {
        vec4 c1 = w.r * Colors[ SampleIndex ][ int(B.r+1) ][ int(B.g←
            )];
        vec4 c2 = w.g * Colors[ SampleIndex ][ int(B.r) ][ int(B.g←
            +1) ];
        vec4 c3 = w.b * Colors[ SampleIndex ][ int(B.r) ][ int(B.g)←
            ];
        c = c1+c2+c3;
    }
    else if( round(w.r+w.g+w.b) == 2)
    {
        vec4 c1 = (1-w.r)*Colors[ SampleIndex ][ int(B.r) ][ int(B.g←
            +1) ];
        vec4 c2 = (1-w.g)*Colors[ SampleIndex ][ int(B.r+1) ][ int(B←
            .g) ];
        vec4 c3 = (1-w.b)*Colors[ SampleIndex ][ int(B.r+1) ][ int(B←
            .g+1) ];
        c = c1+c2+c3;
    }
}

```

- La versión sin interpolación también utiliza la fórmula 3.2 pero solo se queda con el color cuyo peso sea mayor para asignarlo a la muestra:

```

if( round(w.r+w.g+w.b) == 0)
{
    c = Colors[ SampleIndex ][ int(B[0]) ][ int(B[1]) ];
}
else if( round(w.r+w.g+w.b) == 1)
{
    float maxc = max(max(w.r, w.g), w.b);

    if(maxc == w.r)
    {
        c = Colors[ SampleIndex ][ int(B.r+1) ][ int(B.g) ];
    }
    else if(maxc == w.g)
    {
        c = Colors[ SampleIndex ][ int(B.r) ][ int(B.g+1) ];
    }
    else if(maxc == w.b)
    {
        c = Colors[ SampleIndex ][ int(B.r) ][ int(B.g) ];
    }
}
else if( round(w.r+w.g+w.b) == 2)
{
    float maxc = max(max((1-w.r), (1-w.g)), (1-w.b));

    if(maxc == (1-w.r))
    {
        c = Colors[ SampleIndex ][ int(B.r) ][ int(B.g+1) ];
    }
}

```

```
    else if(maxc == (1-w.g))
    {
        c = Colors[SampleIndex][int(B.r+1)][int(B.g)];
    }
    else if(maxc == (1-w.b))
    {
        c = Colors[SampleIndex][int(B.r+1)][int(B.g+1)];
    }
}
```

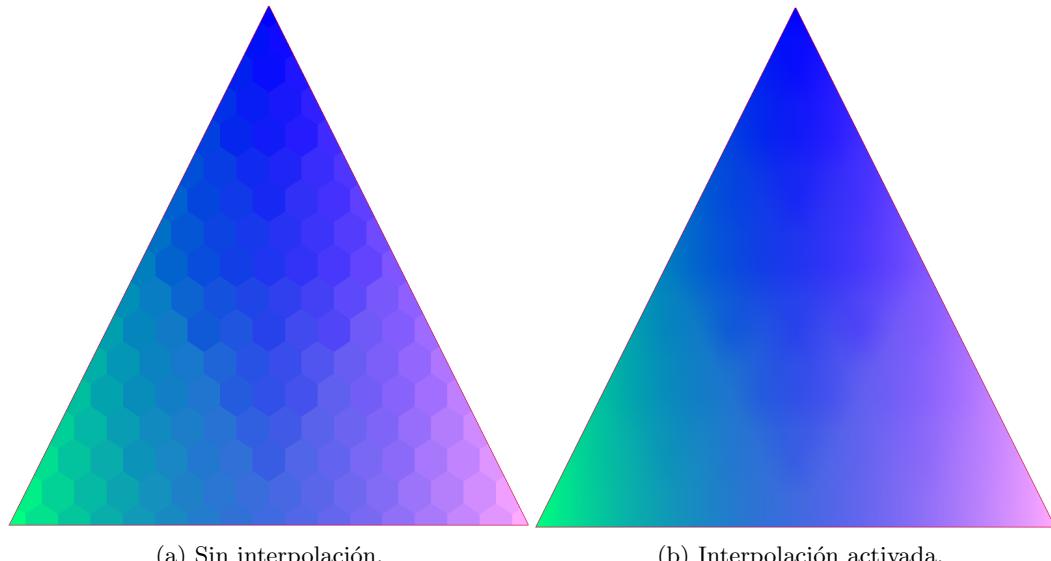


Figura 4.18: Muestras de un triángulo dibujadas con y sin interpolación.

Iluminación: Despues de aplicar la fórmula de *MeshColors* obtenemos un color si mostramos el color directamente por pantalla el resultado será el modelo 3D sin tener en cuenta la iluminación.

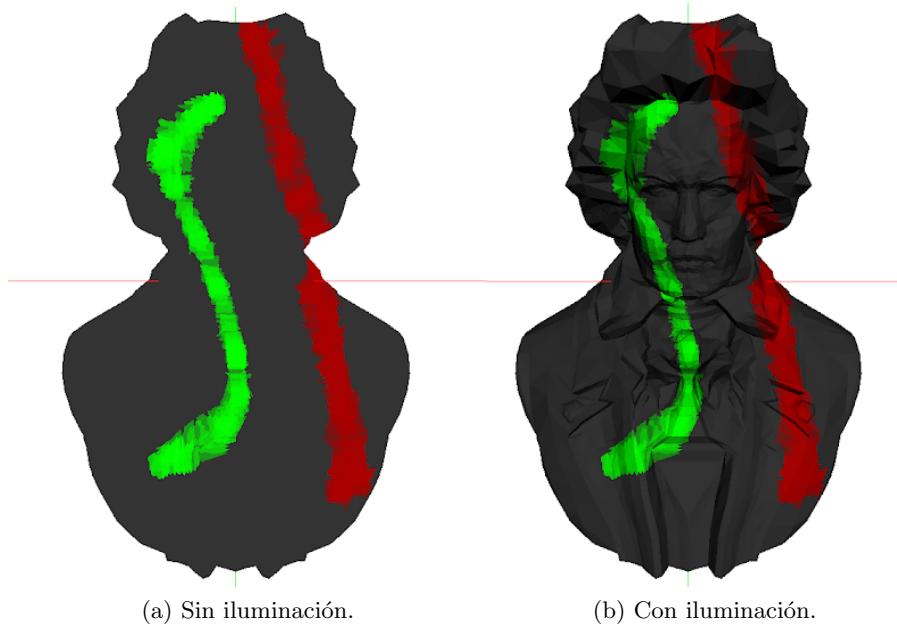


Figura 4.19: Modelo 3D visualizado con y sin iluminación.

Cuando la iluminación está activada utilizamos el modelo de reflexión Phong para calcular el color. Este modelo permite calcular la iluminación local de los puntos de una superficie utilizando la siguiente fórmula [11]:

$$SP = \text{ambient}() * K_a + \text{diffuse}() * K_d + \text{specular}() * K_s$$

- K_a , K_d y K_s son variables numéricas que sirven para definir el peso de las componentes a las que multiplican.
- La componente ambiental `ambient()` es un valor numérico que simula la luz que es proyectada en los objetos alrededor de la escena, esta componente sirve como un valor de iluminación base para que la escena no sea perfectamente oscura en ausencia de luz.
- El componente especular define las propiedades reflectantes del objeto y simula la luz realizando una reflexión perfecta sobre el objeto.

Esta componente se calcula como

$$(2(L \cdot N)N - L) \cdot V$$

Siendo L el vector direccional desde la superficie hasta la fuente de luz, N la normal de la superficie y V el vector direccional desde el punto de la superficie a la cámara.

- Por último el componente difuso simula la reflexión sobre una superficie imperfecta en la que los rayos son emitidos en todas direcciones, las superficies producen menos reflexión de luz. Este componente se calcula como el producto escalar entre el vector direccional desde el punto de la superficie hasta la fuente de luz L y la normal de la superficie N . Si L y N tienen una diferencia mayor de 90 grados el valor del producto escalar sería negativo, por lo que escogemos el máximo entre el resultado y 0 para asegurarnos de no recibir números negativos.

En la implementación del *shader* del proyecto el sistema de iluminación es lo más simple posible ya que solo se utiliza para visualizar el modelo en situaciones muy concretas, por lo que el *shader* solo tiene en cuenta la componente difusa para obtener una iluminación básica pero funcional[11]:

```
vec3 LightColor = vec3(1.0f, 1.0f, 1.0f);
float diff = max(dot(normalize(Normal[0]), normalize(LightPos)), ←
    0.0);
vec3 diffuse = diff * LightColor;

out_Color = vec4(diffuse * c.xyz, 1.0f);
```

4.3. Interfaz de usuario

Las aplicaciones de Qt tienen una estructura específica a la hora de programar y organizar las interfaz de la aplicación y su funcionalidad, los *widgets*. Son el elemento principal para crear interfaces en Qt, pueden mostrar y recibir información directamente del usuario. Los *widgets* pueden contener más *widgets* dentro de ellos para formar varias capas de interfaz de usuario.

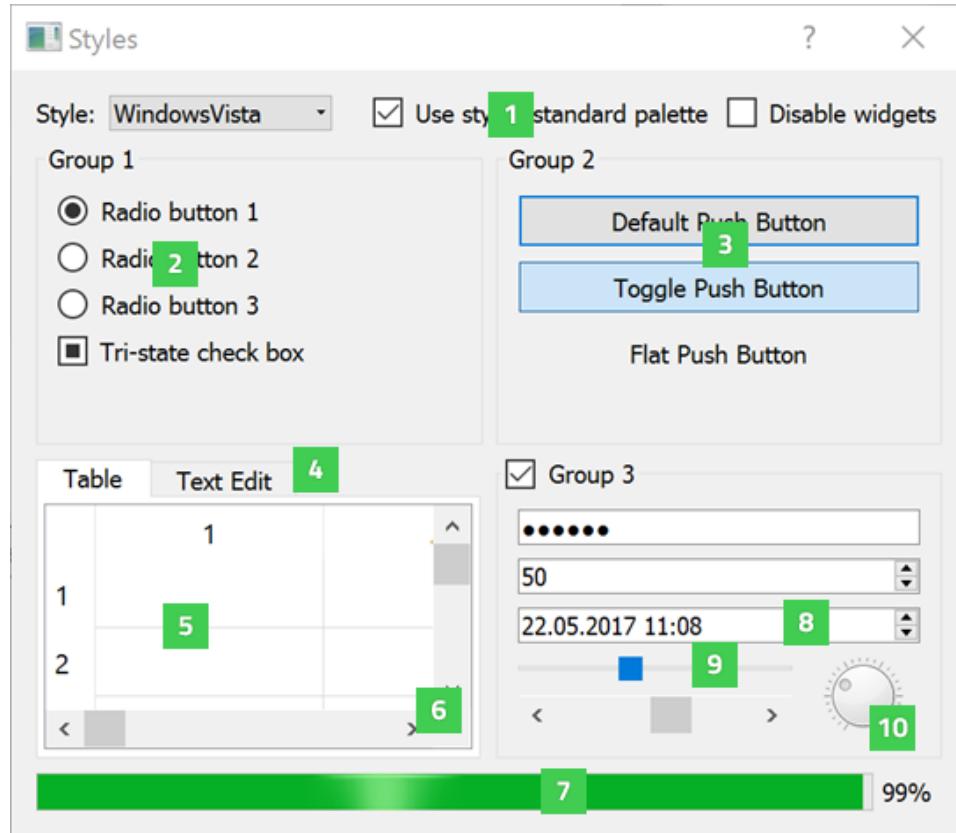


Figura 4.20: Ejemplo de distintos tipos de *widget*, cada número representa una clase distinta. (Referencia 19)

Si un *widget* no está incluido dentro de otro *widget* padre se considera que es una ventana. La ventana proporciona el espacio donde la interfaz será construida, normalmente estas ventanas se generan directamente en el entorno del escritorio y suelen estar integradas en el sistema de gestión de ventanas del sistema operativo correspondiente[9].

Todos los elementos de interfaz de usuario que Qt proporciona son subclases de `QWidget`, que es la clase básica para la interfaz de usuario en Qt y que proporciona métodos clave como la habilidad para escuchar eventos del teclado, del ratón y del gestor de ventanas del sistema operativo.

Para organizar la estructura visual de los widgets podemos utilizar los *layouts*, clases que distribuyen el espacio disponible para cada widget contenido en ellos en función a sus requisitos.

Aunque existen varias clases de *layouts*, en concreto para este proyecto se utilizan `QVBoxLayout` y `QHBoxLayout` que permiten organizar los *layouts* de forma vertical y horizontal respectivamente.

Partiendo desde el widget de la ventana, al que llamamos window. Esta clase sirve como widget base para todos los demás y es donde creamos e inicializamos la jerarquía de widgets y layouts que utilizaremos para toda la interfaz. Esta jerarquía se separa en tres zonas diferentes que se agrupan por funcionalidad: la barra de herramientas, el área de dibujado y el área de opciones.

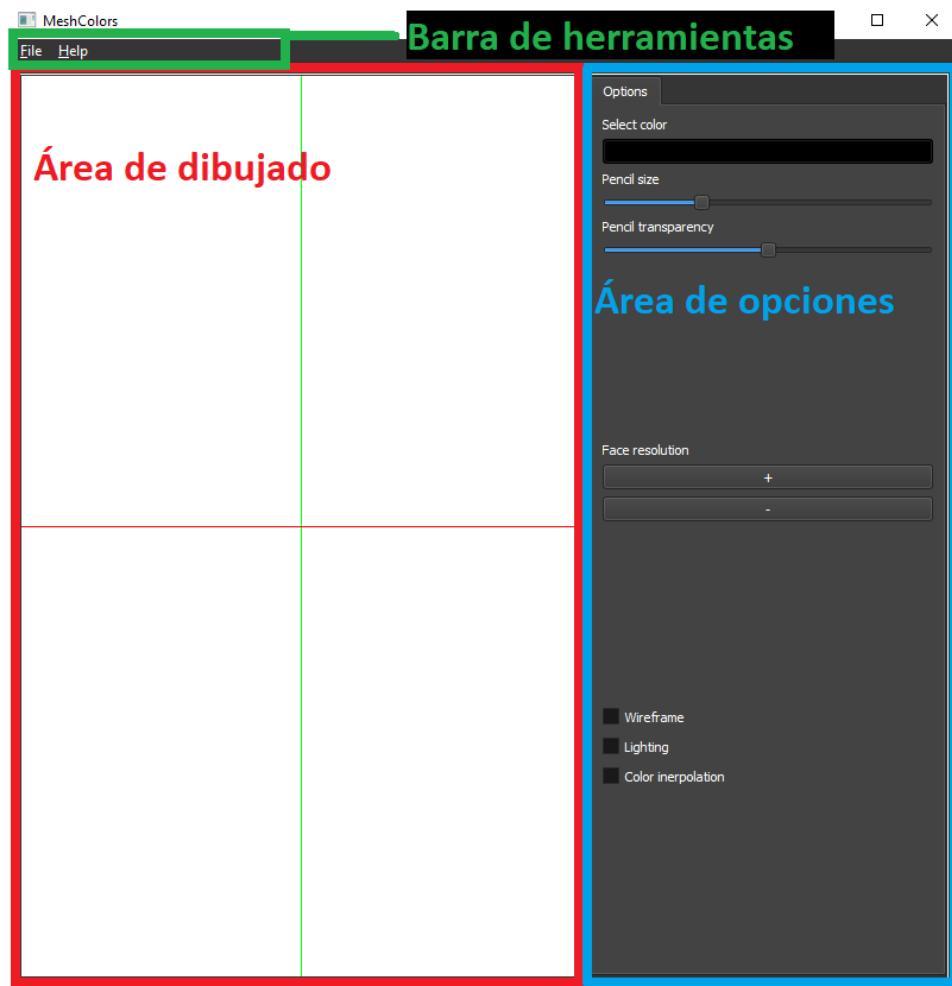


Figura 4.21: Áreas principales de la interfaz de la aplicación

4.3.1. Barra de herramientas

Partiendo desde las opciones en la parte superior izquierda podemos ver dos listas que se despliegan al pasar el cursor por encima mostrando distintas opciones, estos tipos de menús se llaman *Pull-Down menu*. El menú muestra una lista de acciones junto con un atajo de teclado que el usuario puede

utilizar para acceder directamente a esas acciones.

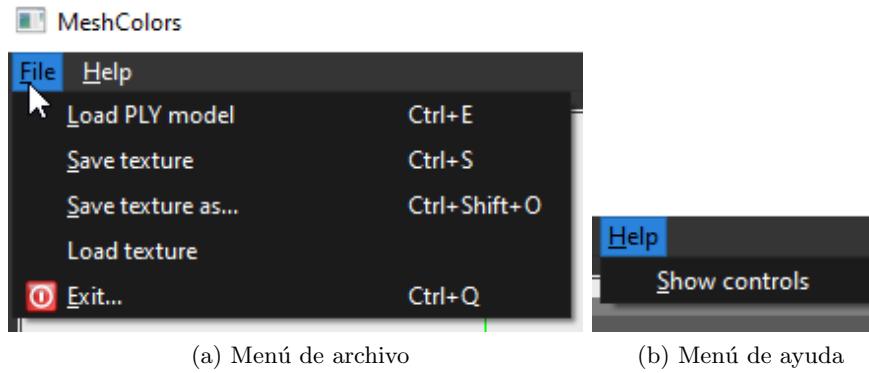


Figura 4.22: Menús accesibles desde la barra de herramientas

Las opciones de la lista son instancias de la clase `QAction`. Esta clase funciona como una interfaz abstracta para crear comandos que pueden ser insertados en los menús y barras de herramientas. Pueden contener iconos, textos explicativos y textos para mostrar los atajos de teclado, aunque estos atajos tienen que ser implementados manualmente por el programador.

4.3.2. Menú de ayuda al usuario

El menú *Help* está ideado para añadir opciones para allanar la curva de aprendizaje del usuario, actualmente contiene solo un elemento, pero el trabajo base está implementado para poder añadir fácilmente otras opciones con secciones como tutoriales o preguntas frecuentes.

La acción *Show controls* crea una pequeña ventana de ayuda que permite al usuario consultar los controles del ratón en cualquier momento de forma rápida.

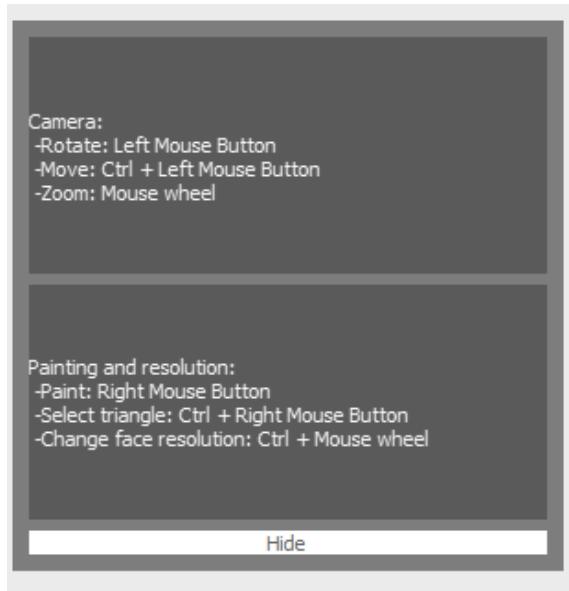


Figura 4.23: Ventana de ayuda para explicar los controles

Esta ventana es un widget del tipo `QDialog` que permite crear este tipo de ventanas sobre la aplicación para comunicaciones con el usuario y notificaciones cortas. `QDialog` no necesita un widget o layout padre para adaptarse a la ventana, y siempre se mostrará por encima de los otros elementos. En este caso se encuentra bajo el layout en el área de dibujado para poder recolocar esa ventana y mantenerla dentro de las fronteras del área de dibujo.

4.3.3. Menú de acciones de archivo

En el menú *File* encontramos acciones que interactúan directamente con el sistema de archivos, cuando el usuario selecciona cualquier opción del menú, excluyendo Exit, una ventana aparece que le permite seleccionar un archivo del tipo indicado para cargar. Estos tipos de acciones contienen un widget llamado `QFileDialog`, este widget permite al usuario moverse por la jerarquía de archivos del sistema operativo en el que la aplicación está siendo ejecutada. Implementar este widget es una simple línea de código con algunos parámetros personalizables.

```
QFileDialog::getOpenFileName(this,  
    tr("Choose mesh color texture file"), "",  
    tr("Mesh colors map (*.mcm);All Files (*)"), 0,  
    QFileDialog::DontUseNativeDialog);
```

Esta línea abre la ventana para seleccionar los archivos.

En la primera implementación del lector de archivos aparecieron problemas de compatibilidad en las ventanas que `QFileDialog` creaba, el problema parecía venir del sistema de selección de archivos de Windows 10, por lo que decidí añadir el parámetro `QFileDialog::DontUseNativeDialog` que fuerza a Qt a utilizar su sistema de widgets en lugar del explorador nativo del sistema operativo. <https://doc.qt.io/qt-5/qfiledialog.html#details>

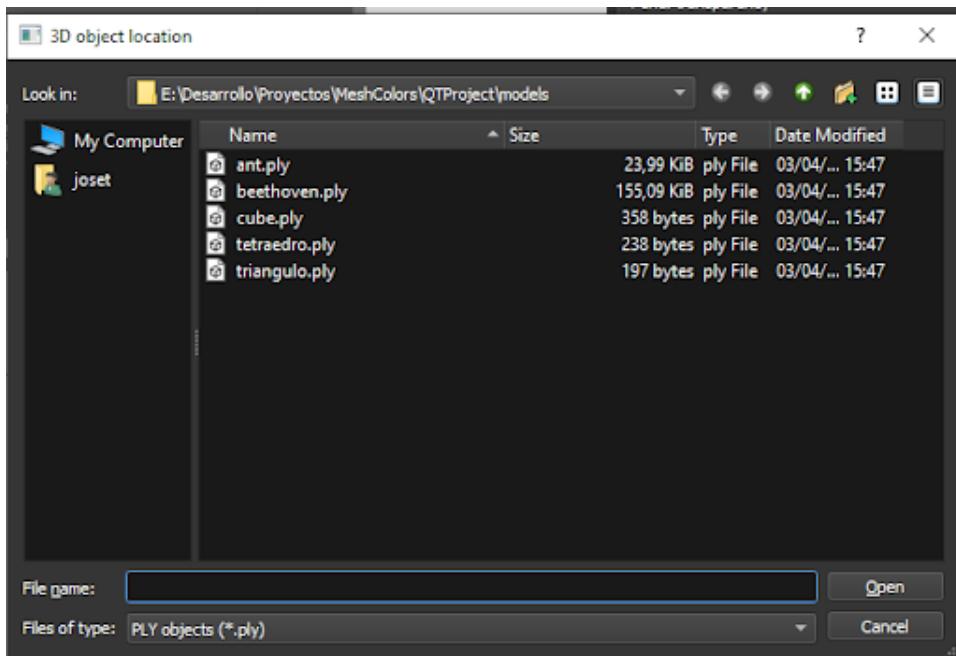


Figura 4.24: Ventana de selección de archivos para cargar el modelo en formato ply.

Una vez seleccionado la función retorna un string que contiene la ruta al archivo, si no se selecciona nada el string estará vacío. Una vez tenemos esta información podemos utilizar la interfaz `QFile` que permite abrir y leer o escribir archivos de texto y binarios. Aunque `QFile` puede leer y escribir los datos, nosotros utilizaremos `QDataStream` que permite crear un canal de codificación binario con el que leer y escribir archivos en cualquier sistema operativo y de forma independiente a la codificación binaria. En el caso de este proyecto solo hay dos acciones que leen de archivos.

Los archivos que utilizamos en el proyecto son de dos formatos distintos:

Formato mcm

El formato mcm(*Mesh Colors Map*) ha sido creado como parte de este proyecto para poder almacenar los datos de la aplicación. Este formato es

muy simple, primero almacena la resolución de todas las caras y luego la lista de colores de todo el modelo. Con estos dos datos es suficiente para almacenar las texturas generadas, hay que tener en cuenta que para que funcionen correctamente hay que aplicarlas sobre el mismo modelo que se utilizó para crear la textura.

4.3.4. Área de opciones

En la parte derecha de la ventana de la aplicación hay una sección en la que se encuentran todas las opciones de dibujado y de resolución. La base para esta sección está creada utilizando `QTabWidget`, un `QWidget` que proporciona una barra de pestañas y un área para cada página donde añadir *widgets* y *layouts*. Inicialmente este widgets tenía varias páginas, una para los colores, otra para las opciones de visualización(*Wireframe*, *Interpolation*, *Lighting*) e Incrementar/Decrementar resolución. Por recomendación del tutor moví todas las opciones a la página inicial, haciéndolas más visibles y accesibles para el usuario. Aun así decidí mantener el `QTabWidget` ya que el resultado gráfico proporciona una separación elegante entre la zona de dibujado y las opciones y también proporciona fácil acceso para crear pestañas nuevas en caso de querer extender la aplicación con elementos como capas de dibujado u opciones de iluminación.

Dentro de esta sección podemos encontrar varios elementos que el usuario puede utilizar para dibujar, cambiar la visualización del modelo y alterar las resoluciones. Todas estas opciones están organizadas mediante un *layout* llamado `vertical_options`.

Opciones de dibujado

Este área está dedicada para cambiar el “pincel” que el usuario utiliza alterando el color y transparencia de los colores aplicados y el tamaño del área que se modifica al hacer click sobre el modelo.

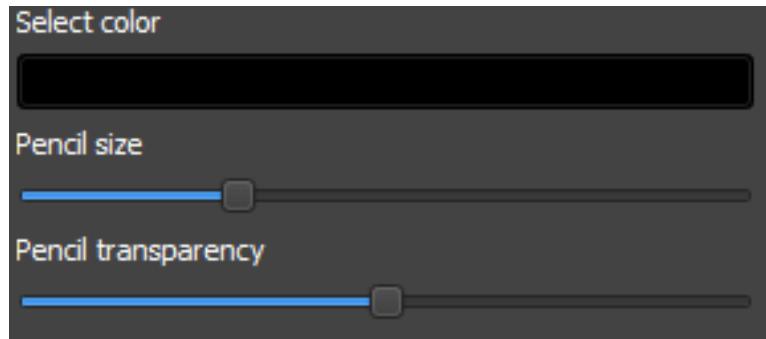


Figura 4.25: Opciones de pincel de dibujado: selección de color y configuración de tamaño y opacidad del pincel

Los textos que aparecen en la captura son implementaciones de la clase `QLabel` que permite mostrar texto o imágenes sobre los widgets de forma muy simple:

```
QLabel *Label1 = new QLabel("Select color");
```

El botón para cambiar el color es un widget de la clase `QPushButton` permite crear un botón que llama a la función que el programador defina al ser pulsado. En este caso llama a una función que utiliza `QColorDialog::getColor`, creando una ventana en la que podemos seleccionar colores:

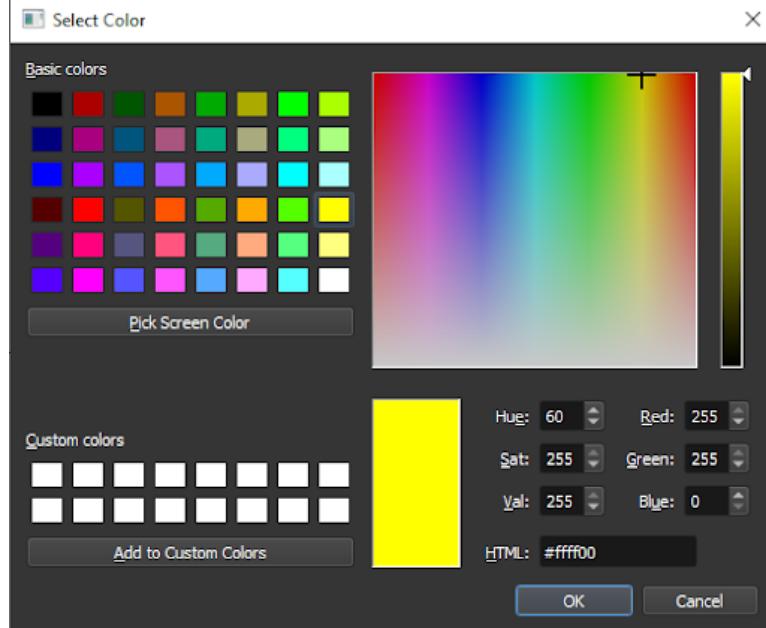


Figura 4.26: Ventana de selección de color.

En esta pestaña podemos seleccionar de una lista de colores predeterminados, seleccionar un color de cualquier zona mostrada en la pantalla, elegir manualmente el valor HSV, RGB o hexadecimal del color y guardar colores personalizados. Una vez seleccionado un color, al hacer clic en OK, la función retorna un valor de la clase `QColor` que utilizaremos como color para dibujar hasta que se seleccione otro. El color seleccionado también se utiliza en el propio botón de seleccionar color para mostrar el color que se está utilizando actualmente.



(a) Color negro seleccionado.

(b) Color amarillo seleccionado. 1

Figura 4.27: Botón de selección de color, cambia de color para mostrar al usuario el que ha seleccionado.

Para seleccionar el tamaño y la transparencia del pincel utilizamos `QSlider`, otro tipo de *widget* que permite crear una barra con un deslizador que el usuario puede mover sobre la barra, la barra puede ser horizontal y vertical y permite elegir un valor para un parámetro dentro de un rango predefinido. En este caso ambos son horizontales para mantener el espacio de las opciones del pincel compacto.

Incrementar/Decrementar resolución

Esta funcionalidad son dos botones simples que llaman a la funciones `IncrementResolution` y `DecrementResolution` respectivamente. Estas funciones comprueban si los nuevos valores de resoluciones están dentro de los límites. estos valores aumentan multiplicando por dos y se reducen dividiendo entre dos, para mantener la consistencia entre las fronteras de caras como se ha explicado anteriormente en este documento.

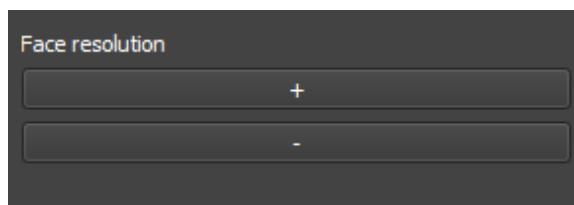


Figura 4.28: Botones para incrementar y decrementar resolución

El límite superior de resolución sirve para proteger al usuario de seleccionar una resolución mayor de la que la lista de muestras de colores puede

utilizar, el tamaño de esta lista depende de un parámetro constante que puede ser alterado para permitir resoluciones mayores en GPUs con mayor potencia.

Opciones de visualización

Esta sección permite al usuario decidir sobre algunos aspectos de cómo la aplicación muestra el modelo tridimensional.



Figura 4.29: Casillas para activar y desactivar las opciones de visualización.

Está compuesto de tres *QCheckBox widgets* que permiten al usuario marcar o desmarcar un cuadrado editando un parámetro booleano que utilizamos para definir las opciones mostradas. El usuario puede desactivar o activar el dibujado de las aristas del modelo, si el modelo utiliza iluminación o la ignora completamente y si el modelo muestra los colores interpolados o dibuja cada muestra con su color específico.

En el caso de la iluminación y la interpolación de color estos parámetros afectan directamente al código del *shader* y han sido explicados previamente.

4.3.5. Área de dibujado

El área de dibujado es la zona en la que la escena tridimensional se muestra, en él podemos ver los ejes de coordenadas y el modelo 3D. Este área contiene exclusivamente un *widget* de la clase `GL_Widget` una clase que hemos creado para este proyecto y que contiene la funcionalidad para dibujar el objeto y los ejes y seleccionar píxeles en la pantalla, esta clase se explica en profundidad en el siguiente apartado de implementación.

5. Metodología para el desarrollo del software

5.1. Requisitos funcionales

Para el diseño de la aplicación se partió desde la extracción de los requisitos funcionales, este proceso se realizó probando distintos programas de pintado de texturas y viendo qué utilidades básicas ofrecían.

Cargar modelos exportados de otros programas en un formato específico. Visualizar los modelos, moviendo la cámara alrededor, acercándose y alejándose. Poder visualizar la geometría del modelo (wireframe). Crear una fuente de luz para visualizar el modelo dentro de un sistema de iluminación. Mover la fuente de luz para iluminar el modelo desde distintos ángulos. Utilizar el cursor como pincel para pintar sobre el modelo. Poder definir el color con el que se pinta. Poder definir la transparencia del color seleccionado. Poder definir el tamaño del pincel. Guardar los colores pintados sobre la superficie del modelo en un archivo externo. Cargar los colores de la superficie del modelo desde un archivo externo.

Estudiando la técnica MeshColors también podemos obtener otros requisitos funcionales:

Poder dibujar sobre subdivisiones dentro de las caras. Ofrecer distintos niveles de subdivisiones. (Resolución) Poder utilizar diferentes niveles de subdivisiones en caras distintas dentro del modelo, incluyendo caras adyacentes. Seleccionar un triángulo en específico dentro del modelo. Aumentar o reducir el número de subdivisiones del triángulo seleccionado. Mostrar el color de cada muestra sin mezclar con las adyacentes Mostrar el color de la muestra interpolado con las muestras adyacentes

5.2. Requisitos no funcionales

El programa se ejecuta de manera fluida sin problemas de rendimiento en su uso habitual. Los archivos creados para guardar los colores del modelo tienen un tamaño aceptable. La interfaz de usuario es simple y fácil de entender.

Partiendo de estos requisitos y del código proporcionado por el profesor el objetivo es expandir la funcionalidad hasta cubrir los requisitos funcionales. Por eso, la metodología escogida para desarrollar el proyecto es un modelo iterativo en el que intentamos desde el principio tener un producto mínimo viable y en cada iteración intentamos implementar más requisitos funcionales junto con el feedback del profesor sobre la iteración anterior. Partiendo desde un visualizador básico en el que poder cargar cualquier modelo y mover la cámara hasta la implementación final donde todos los requisitos funcionales se encuentran implementados.

En cada iteración se realiza una comprobación para asegurar que los requisitos no funcionales se siguen manteniendo, especialmente en el requisito sobre la interfaz de usuario el feedback del tutor fue clave para poder conseguir una interfaz funcional y simple.

6. Resultados

La aplicación obtenida ofrece la funcionalidad clave que se querían alcanzar con los requisitos funcionales: El programa permite cargar un modelo 3d desde un archivo .PLY

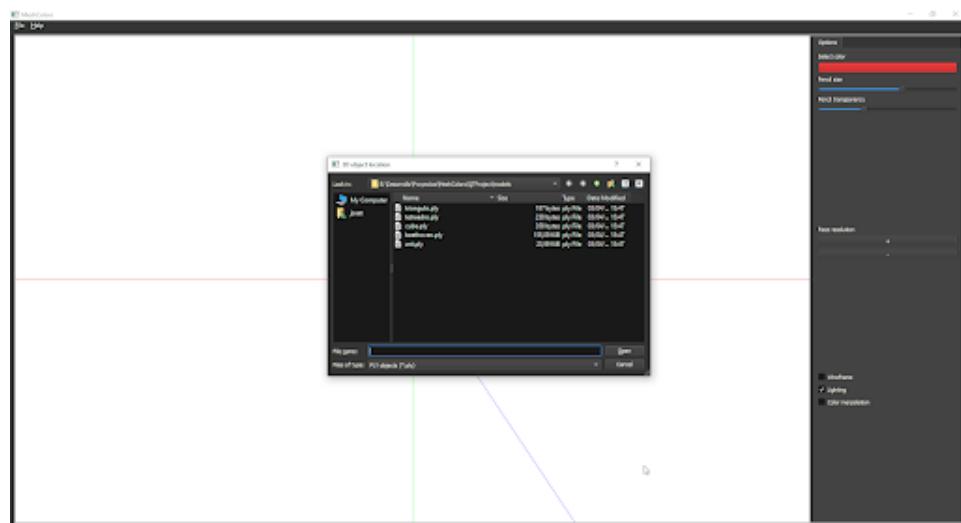


Figura 6.1: Pantalla de selección de modelos.

Se puede visualizar el modelo con o sin iluminación controlando la posición y distancia de la cámara y la posición de la fuente de luz.

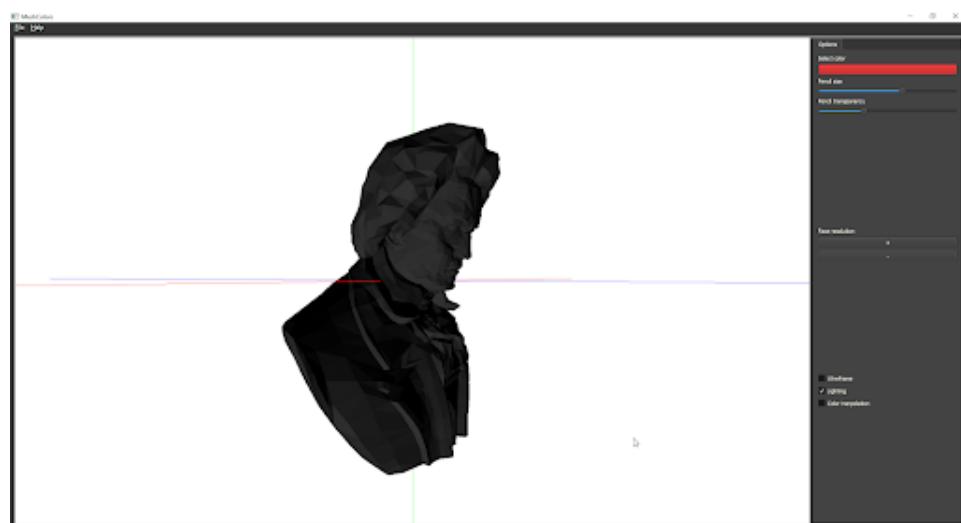


Figura 6.2: Visualización de modelo iluminado.

Sobre el modelo podemos dibujar, definiendo las propiedades del pincel y alterar la resolución de las caras.

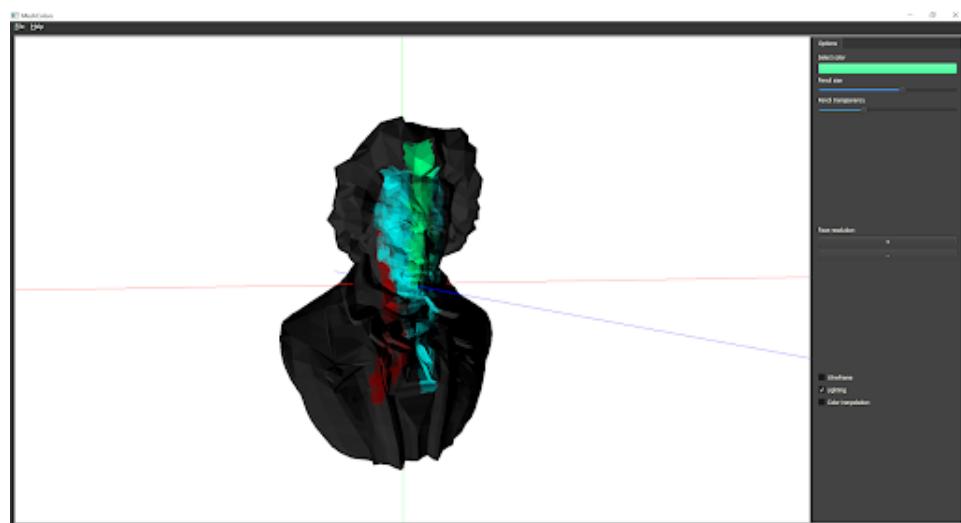


Figura 6.3: Pintado sobre el modelo.

Se pueden guardar los colores pintados en la superficie del modelo en un archivo externo que podemos usar para cargar los colores sobre el modelo.

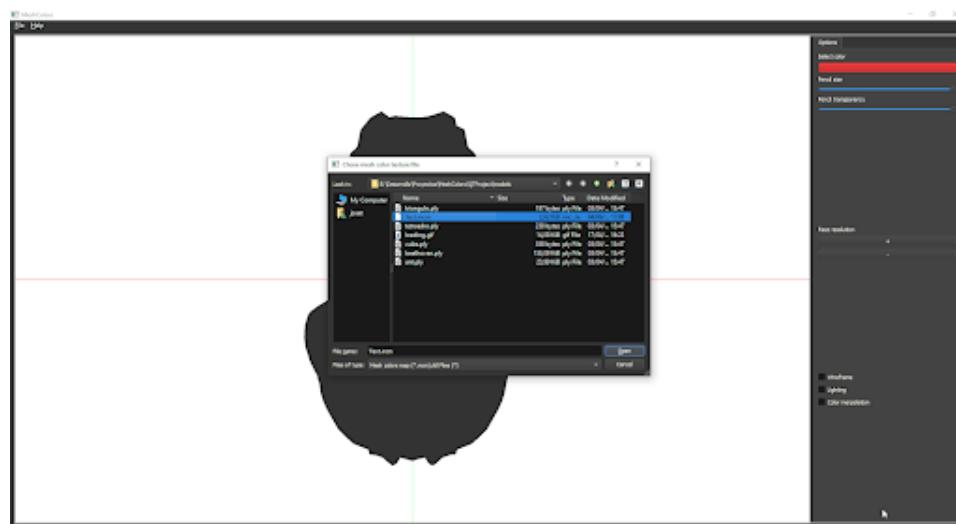


Figura 6.4: Pantalla de selección de archivos para guardar los colores pintados.

La funcionalidad del programa cubre los requisitos funcionales que habíamos definido. En cuanto a los requisitos no funcionales, para la experiencia de usuario se ha mantenido una interfaz simple y con pocos botones, además de implementar un menú de ayuda donde el usuario puede consultar los controles del programa.

7. Conclusiones

7.1. Sobre la técnica MeshColors

Tras investigar e implementar esta técnica he podido entender el potencial pero también las flaquezas que conlleva.

La técnica ofrece la posibilidad de crear modelos con gran nivel de detalle, especialmente en modelos con pocos polígonos “low poly” ya que el artista puede centrar los recursos en las caras más visibles o con más necesidad de detalle. Eliminando también la necesidad de crear mapas de UV.

En esta implementación en concreto, debido a que implementamos una generación automática de los colores todas las resoluciones posibles de cada cara se podría conectar esta funcionalidad con motores de visualización 3D en tiempo real como Unity para generar los niveles de detalle de forma automática durante la ejecución en lugar de tener que crear versiones de texturas de menor calidad.

Por otro lado, el proceso de creación de una textura en el pintado de textura clásico puede ser más simple y directo para un artista ya que, una vez que se ha realizado el mapa UV, el artista sólo tiene que dibujar sobre el modelo sin necesidad de preocuparse por la resolución de cada cara. El texturizado clásico genera archivos que son procesables para un humano, un archivo de textura puede ser analizado e incluso optimizado sin necesidad de tener que visualizar el modelo.

Con las herramientas necesarias, como integración con motores de videojuegos, esta técnica tiene potencial para optimizar ciertos proyectos en los se trabajan con modelos más simples y en visualización en tiempo real, aunque requiere una adaptación por parte de los artistas y en el software que utilizan que puede ser un punto negativo para muchos estudios de productos 3D.

7.2. Sobre este proyecto

Durante el desarrollo del proyecto he tenido que investigar la técnica buscando referencias, casos de uso y otras implementaciones, revisando el documento en busca de información y encontrando por el camino algunos

huecos que tuve que llenar con documentación extra o con conocimiento del grado. Durante la implementación del programa he tenido que seguir investigando sobre OpenGL, las ventajas y desventajas que tiene y cómo ha evolucionado durante sus distintas versiones. Creo que el proyecto ha cumplido su objetivo principal de servir como puerta de entrada al mundo de la investigación.

El desarrollo me ha permitido crear una gran cantidad de código de programas para la CPU, obligándome a entender cómo transmitir información entre la CPU y la GPU y profundizando en el campo de los shaders y la iluminación.

Una gran parte de este proyecto ha sido crear una interfaz de usuario, para ello he tenido que aprender a manejar Qt y el sistema de señales y slots, cómo crear y agrupar widgets y layouts y, en general, cómo funciona el sistema de interfaz de un programa a nivel general.

En cuanto a la implementación, el proyecto ha cumplido los requisitos que planteé al inicio, se ha desarrollado siempre teniendo en cuenta la extensibilidad y hacer el código lo más simple posible. Creo que hay espacio para la mejora con tiempo de desarrollo extra en campos como Experiencia de usuario: implementar una funcionalidad de rehacer y deshacer. Dibujado: poder crear materiales con distintas propiedades de iluminación y dibujar con ellos en lugar de usar solo colores básicos. Código: los arrays estáticos que se han creado para manejar los colores deberían ser transformados en una versión dinámica para poder ahorrar memoria durante el proceso de dibujado.

A pesar de esos fallos, me encuentro satisfecho con el trabajo y creo que ha cumplido los objetivos marcados. Espero que este proyecto sirva de ayuda para otras personas intentando aprender más sobre esta técnica, sobre las interfaces de Qt o sobre OpenGL al menos una fracción de lo que me ha servido a mi.

Bibliografía

- [1] Cem Yuksel, John Keyser, Donald H. House, Department of Computer Science, Texas A&M University, 2008. Mesh Colors
- [2] The Institute for Digital Archaeology, 2019. The Science of 3D Rendering
- [3] Ohio State University 2017. Fundamentals of Rendering - Reflectance Functions
- [4] Pontifícia Universidade Católica do Rio Grande do Sul. Programming with OpenGL: Advanced Rendering
- [5] Khronos OpenGL Wiki
- [6] Khronos OpenGL Shader
- [7] Greg Turk 1998. The PLY Polygon File Format
- [8] Weisstein, Eric W, MathWorld. Normal Vector
- [9] Qt documentation. Qt 5.15
- [10] Microsoft documentation, 2019 OpenGL, Rendering Context
- [11] learnopengl.com Basic Lighting

Referencias

1. https://en.wikipedia.org/wiki/Gilles_Tran
2. <https://lucasfalcao3d.com/blog/the-versatility-of-vertex-paint/>
3. <https://free3d.com/3d-models/human-head>
4. <https://www.semanticscholar.org/topic/Texture-synthesis/32638>
5. <https://www.escuelavideojuegosmasterd.es/curso-texturizado>
6. <https://www.youtube.com/watch?v=aq25CWtsLBY>
7. <https://www.youtube.com/watch?v=TZBstRVcQFM>
8. https://it.wikipedia.org/wiki/Mappatura_UV
9. <https://docs.blender.org/manual/en/latest/modeling/meshes/editing/uv.html>
10. <https://www.inf.pucrs.br/flash/tcg/aulas/texture/texmap.pdf>
11. https://en.wikipedia.org/wiki/UV_mapping
12. https://www.researchgate.net/publication/327417063_To_think_structure_to_feel_space_An_alternative_to_teaching_construction_in_architecture
13. <https://blog.imaginationtech.com/why-you-really-should-be-using-mipmapping-in-3d-rendering>
14. https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview
15. https://www.khronos.org/opengl/wiki/Vertex_Specification
16. <https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practicals/rasterization-stage>
17. http://www.cemyuksel.com/research/meshcolors/meshcolors_techreport.pdf
18. <https://www.racoon-artworks.de/cgbasics/normals.php>
19. <https://doc-snapshots.qt.io/qt6-dev/gallery.html>

Apéndice

Manual de usuario

Control de cámara

- Rotar: mover el ratón con el botón izquierdo pulsado.
- Mover: mover el ratón con el botón izquierdo pulsado mientras se pulsa **Ctrl**.
- Zoom: rotar la rueda del ratón.
- Centrar cámara: pulsar tecla F.

Control de dibujado

- Pintar: pulsar el botón derecho del ratón sobre el modelo.
- Seleccionar triángulo: pulsar el botón derecho del ratón sobre el modelo mientras se pulsa **Ctrl**.
- Cambiar resolución del modelo:
 - Rotar la rueda del ratón mientras se pulsa control.
 - Tecla + para incrementar resolución.
 - Tecla - para decrementar resolución.

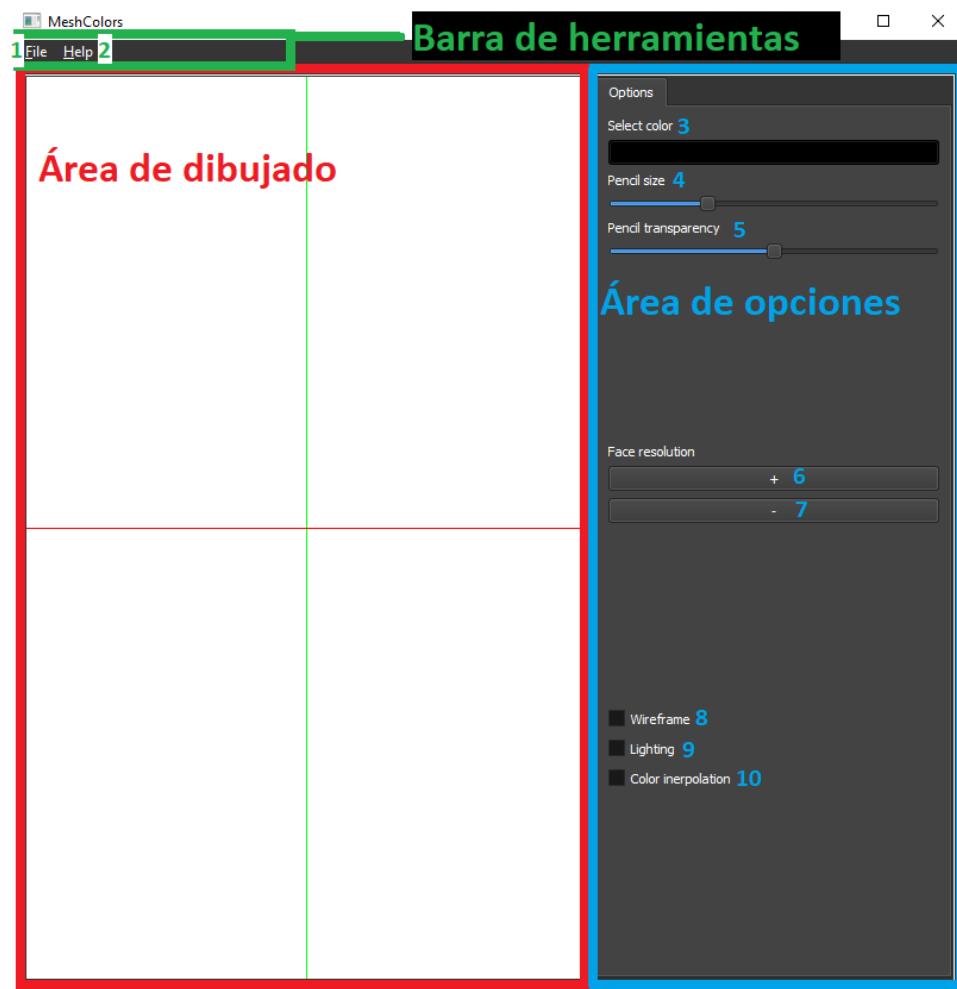


Figura 7.1: Ventana de la aplicación con los controles numerados.

Acciones en la interfaz de usuario

1. Menú de acciones de archivo, donde el usuario puede guardar y cargar pintados de *Mesh Colors* y cargar modelos PLY.
2. Menú de ayuda al usuario, contiene una ventana con los controles de la cámara y dibujado.
3. Selección de color.
4. Deslizador para seleccionar el tamaño del pincel.
5. Deslizador para seleccionar la opacidad del pincel.
6. Incrementar resolución.

7. Decrementar resolución
8. Activar o desactivar modo rejilla (*wireframe*), este modo mostrará las aristas del modelo.

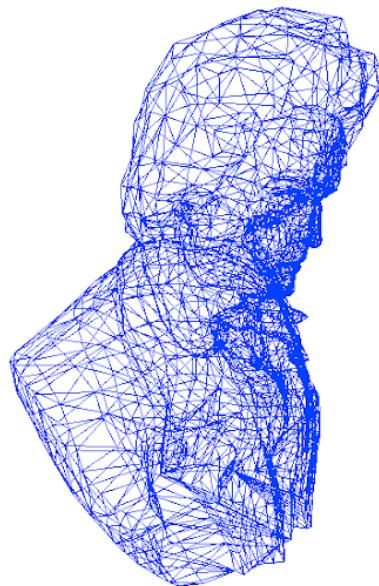


Figura 7.2: Aristas del modelo 3D.

9. Activar o desactivar la iluminación.

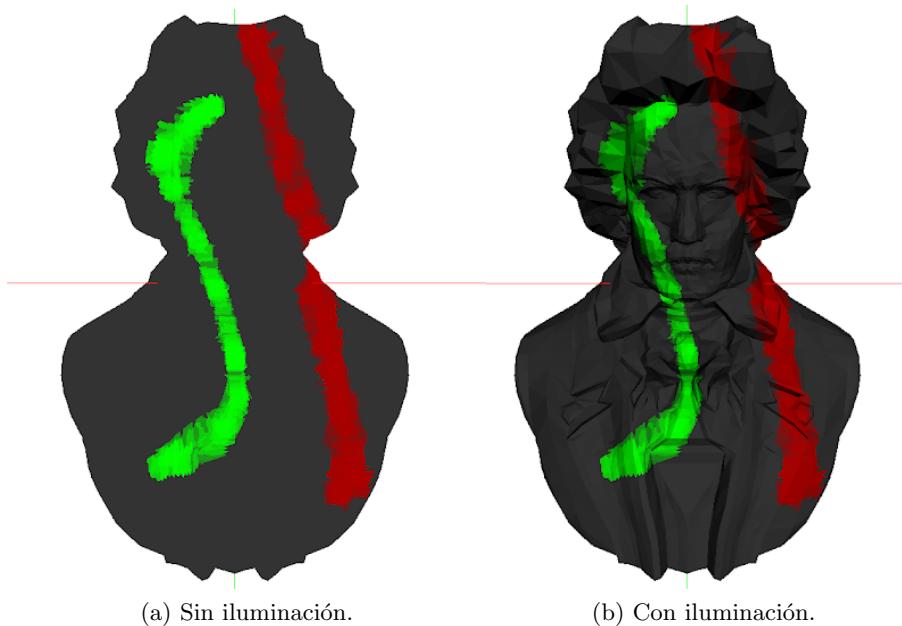


Figura 7.3: Modelo 3D visualizado con y sin iluminación.

10. Activar o desactivar la interpolación de color.

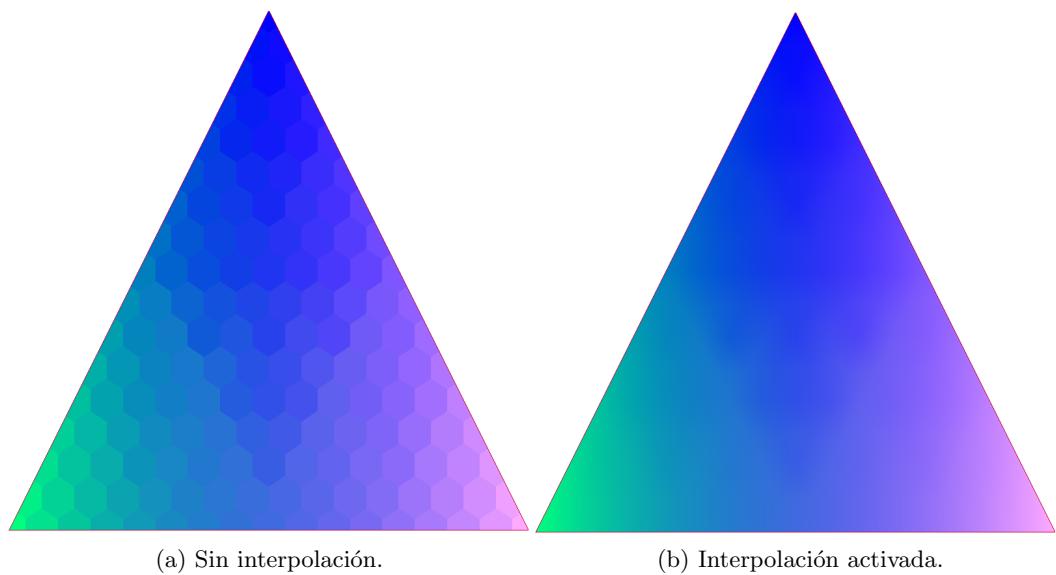


Figura 7.4: Muestras de un triángulo dibujadas con y sin interpolación.

