Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Matemáticas e Informática

Trabajo Fin de Grado

Diseño de una Aplicación Elemental de Modelado

Autor: José David Ortega Yangua
Tutor: Jonatan Sánchez Hernández

Madrid, Enero 2025

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Matemáticas e Informática

Título: Diseño de una Aplicación Elemental de Modelado

Enero 2025

Autor: José David Ortega Yangua
Tutor: Jonatan Sánchez Hernández
Matemática Aplicada A Las Tecnologías De La Información Y Las Comunicaciones
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

Tabla de contenidos

Resumen

Este proyecto consiste en el diseño y desarrollo de una aplicación elemental de modelado geométrico tridimensional, implementada en C++ con OpenGL. Su objetivo principal es proporcionar un entorno que permita explorar y manipular modelos 3D mediante operaciones básicas de transformación afín y edición geométrica, sirviendo como plataforma práctica para comprender los fundamentos matemáticos y computacionales detrás de los sistemas gráficos modernos.

La aplicación se estructura en torno a un *pipeline* gráfico de renderizado completo que incluye la gestión de espacios (objeto, mundo, cámara, *Clip-Space* y ventana), la representación eficiente de mallas poligonales mediante estructuras de datos especializadas (vértices y caras encapsulados en *buffers* de tipo VAO) y un sistema de iluminación basado en el modelo Blinn-Phong implementado en *shaders* GLSL. La cámara interactiva, controlada mediante cuaterniones, evita problemas como el Bloqueo de Cardán y permite movimientos orbitales y desplazamientos con teclado (WASD).

La interacción con la escena se realiza a través de un Gizmo de transformación, que traduce el movimiento 2D del ratón en operaciones 3D intuitivas, y mediante algoritmos de selección basados en rayos (algoritmo de Möller-Trumbore) para la identificación precisa de vértices, caras y modelos.

Las transformaciones afines que se pueden realizar en la aplicación son: traslación, rotación (con cuaterniones) y escala.

Además, se han implementado herramientas de edición avanzadas como la extrusión de caras y la inserción de vértices en posiciones arbitrarias, así como la generación de primitivas geométricas predefinidas (cubos, prismas, pirámides).

La interfaz gráfica, desarrollada con ImGui, integra controles para la gestión de modos y visualización de información en tiempo real.

Abstract

This project involves the design and development of an elementary three-dimensional geometric modeling application, implemented in C++ with OpenGL. Its primary objective is to provide an environment that allows for the exploration and manipulation of 3D models through basic affine transformation and geometric editing operations, serving as a practical platform for understanding the mathematical and computational foundations behind modern graphics systems.

The application is structured around a complete rendering pipeline that includes the management of spaces (object, world, camera, clip, and window), the efficient representation of polygonal meshes through specialized data structures (vertices and faces encapsulated in VAO buffers), and a lighting system based on the Blinn-Phong model implemented in GLSL shaders. The interactive camera, controlled using quaternions, avoids issues such as gimbal lock and enables orbital movements and keyboard-based navigation (WASD).

Interaction with the scene is achieved through a transformation Gizmo, which translates 2D mouse movement into intuitive 3D operations, and via ray-based selection algorithms (the Möller-Trumbore algorithm) for precise identification of vertices, faces, and models.

The affine transformations that can be performed in the application are: translation, rotation (quaternions), and scaling.

Advanced editing tools have been implemented, such as face extrusion and the insertion of vertices at arbitrary positions, as well as the generation of predefined geometric primitives (cubes, prisms, pyramids).

The graphical interface, developed with ImGui, integrates controls for mode management and real-time information display.

Capítulo 1

Introducción

1.1. Contexto y motivación

El desarrollo de aplicaciones de modelado tridimensional constituye una de las áreas fundamentales dentro de los gráficos por computador. Estas herramientas permiten la creación, manipulación y visualización de modelos geométricos en entornos interactivos, siendo imprescindible en ámbitos como el diseño asistido por computadora.

En este contexto, el presente proyecto tiene como objetivo la implementación de un editor de modelado 3D elemental que permita la manipulación de geometría mediante transformaciones afines (traslación, rotación y escalado) y operaciones comunes en aplicaciones ya existentes como la extrusión o la inserción de caras interiores.

El sistema está desarrollado en C++ utilizando OpenGL, junto con bibliotecas para la gestión de ventanas e interacción de usuario, y GLM para la representación y manipulación matemática de vectores, matrices y cuaterniones.

La motivación principal radica en comprender de forma práctica los principios de la geometría afín y proyectiva aplicados al *pipeline* de gráficos.

La relevancia académica del proyecto se sustenta en la aplicación directa de fundamentos teóricos de álgebra lineal, geometría tridimensional y representación proyectiva al desarrollo de software interactivo.

1.2. Objetivos y alcance

El objetivo general de este proyecto es diseñar y desarrollar una aplicación interactiva para la edición y transformación de modelos tridimensionales, haciendo uso de técnicas de renderizado en tiempo real basadas en OpenGL 3.3.

Para lograrlo, se establecen los siguientes objetivos específicos:

1. Implementar un motor básico de renderizado utilizando OpenGL y shaders escritos en GLSL, incluyendo la gestión de la cámara, la proyección y la

iluminación.

2. Desarrollar herramientas interactivas que permitan la selección y transformación de modelos, vértices y caras mediante un Gizmo (base de direcciones) de manipulación visual.
3. Aplicar transformaciones geométricas (traslación, rotación y escala) fundamentadas en la geometría afín y proyectiva. Además, implementar la extrusión.
4. Incorporar una cámara orbital controlada por el usuario, que permita explorar la escena mediante rotación y desplazamiento en el espacio.

El alcance del proyecto se limita a un entorno de modelado elemental orientado al aprendizaje y visualización de transformaciones geométricas. No se contempla la implementación de funcionalidades más complejas. El propósito de ello es mantener la simplicidad del sistema para centrarse en la relación entre la representación matemática del espacio 3D y su visualización directa en una pantalla 2D.

1.3. Metodología

El desarrollo del proyecto se ha estructurado en torno a un enfoque incremental y modular, siguiendo una metodología de desarrollo iterativo basada en ciclos de diseño, implementación y validación.

En una primera fase, se configuró el entorno de trabajo con las bibliotecas necesarias (FreeGLUT, GLEW y GLM) y se verificó la correcta inicialización del contexto de OpenGL.

Ya con el entorno preparado, se implementó la representación de modelos mediante buffers de vértices (VBO) y arrays de vértices (VAO) e índices de triangulación (EBO), complementados por un sistema básico de shaders para el cálculo de iluminación.

En la fase intermedia, se diseñó la cámara de visión, que combina una proyección en perspectiva y una vista orbital basada en cuaterniones para evitar singularidades. La cámara se controla mediante eventos de ratón y teclado, permitiendo rotar y trasladar la escena de forma intuitiva.

Se implementaron los algoritmos de selección de vértices, caras y modelos. Se estableció una jerarquía y se estudiaron los sucesos posibles en una secuencia de selecciones dependiendo del modo del editor. También se implementó un modo para añadir vértices a un modelo ya existente para comprobar si se había modificado correctamente las estructuras asociadas al modelo.

Posteriormente, se desarrolló la herramienta de edición geométrica (Gizmo), que traduce los desplazamientos del ratón en transformaciones 3D sobre los objetos seleccionados. Tras ello, se implementó la traslación, la rotación y la escala.

Además, se desarrolló la extrusión y, en cada fase, se realizaron pruebas de validación.

Capítulo 2

Espacios: Cómo observar el espacio 3D en 2 dimensiones

El proceso de convertir coordenadas tridimensionales (3D) en píxeles bidimensionales (2D) en una pantalla es fundamental en los gráficos por computadora. Este proceso, conocido como el *pipeline* de renderización, es un procedimiento matemáticamente riguroso implementado por APIs como OpenGL. Comprender cada etapa de este *pipeline*, desde las coordenadas locales de un objeto hasta su representación final en la pantalla, es esencial para el desarrollo de aplicaciones gráficas robustas . Este capítulo analiza en detalle las transformaciones de coordenadas y los espacios intermedios definidos en el estándar OpenGL.

En OpenGL, el *pipeline* de transformación es un proceso secuencial donde la geometría de una escena pasa a través de una serie de espacios de coordenadas. Cada espacio de coordenadas tiene un propósito específico que facilita distintas operaciones o cálculos. El flujo general del *pipeline* puede resumirse de la siguiente manera:

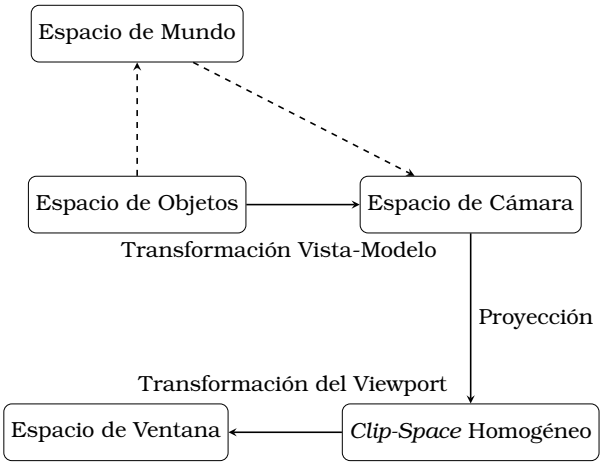

Figura 2.1: Los espacios de coordenadas que aparecen en el *pipeline* de renderizado. Adaptado de [1].

2.1. Espacio de Objetos y Espacio de Mundo

El Espacio de Objetos sería un sistema de coordenadas local usado solo por el propio modelo. Su posición y orientación en el mundo suelen ser guardadas en el Espacio de Mundo [1]. Por simplicidad en la implementación solo se guardan las coordenadas de posición en el mundo real 3D en base usual de cada vértice del modelo.

2.2. Espacio de Cámara

Para facilitar la proyección y el *clipping*, la cámara y todos los modelos se transforman mediante la transformación de vista [2].

La cámara consiste de una posición en el espacio mundial y una dirección, que se utilizan para ubicar y orientar el espacio observable por el usuario. Además, la cámara debe tener una forma de delimitar el espacio observable para reducir el coste computacional en casos extremos.

Para ello, se debe introducir el concepto de *frustum*. El *frustum* de visión es una pirámide truncada de base cuadrada que tiene a la cámara en su vértice superior. Los dos planos que resultan del truncamiento (el superior y la base inferior) se denominan plano cercano y plano lejano, respectivamente. Estos planos definen las distancias mínima y máxima a las cuales los objetos en una escena son visibles para la cámara [1].

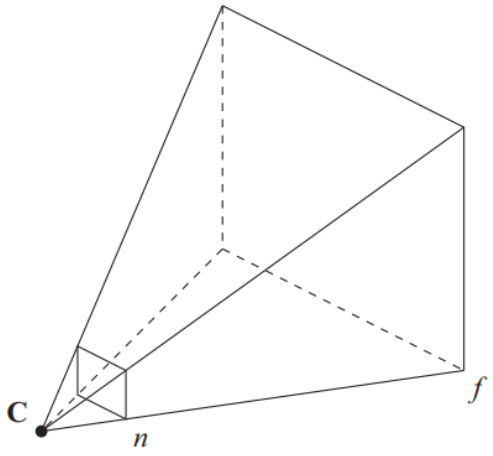

Figura 2.2: Frustum [1].

Existen otras formas de delimitar el espacio observable pero se ha escogido el *frustum*.

El propósito de la transformación de vista es recolocar la cámara junto al espacio en un nuevo origen y reorientarlos para que miren en la dirección del eje z negativo, con el eje y apuntando hacia arriba y el eje x apuntando hacia la

derecha. El espacio así delimitado se denomina Espacio de Cámara o Espacio de Vista.

Para implementar la matriz de vista inicial se utiliza la función `glm::lookAt()` de la biblioteca GLM. A la función se le pasa la posicíon de la cámara y de un punto donde apuntar, además de un vector que apunte hacia arriba [3].

Internamente `glm::lookAt()` hace los siguientes cálculos:

$$M_{view} = \begin{pmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.1}$$

Donde **R** representa el vector hacia la derecha, **U** el vector hacia arriba, **D** el vector hacia delante y **P** la posición de la cámara [4].

En la implementación se guarda en un cuaternión para las rotaciones de cámara, haciendo la conversión cada vez que se necesita la matriz de vista.

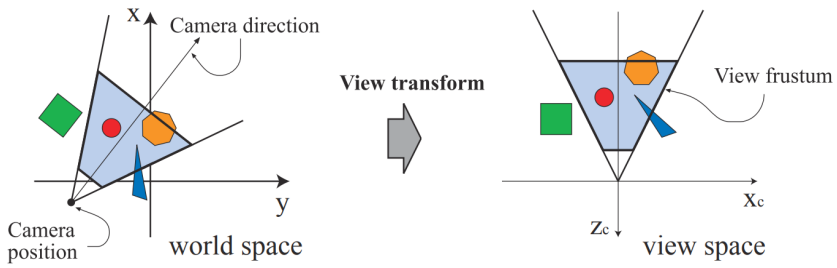

Figura 2.3: La matriz de vista recoloca el Espacio de Mundo [2].

2.3. *Clip-Space*

En el mundo real, los objetos más lejanos se ven más pequeños. Esto es algo que los cerebros humanos esperan. Si renderizáramos el espacio 3D sin perspectiva, todo se vería plano y poco natural.

Dos formas de proyectar el espacio 3D serían la proyección en perspectiva (*frustum*) y la proyección ortogonal.

En la proyección ortogonal, los objetos mantienen su tamaño relativo independientemente de su distancia a la cámara, eliminando cualquier sensación de profundidad visual.

La proyección usando el *frustum* es más natural para el ojo humano ya que los objetos se empequeñecen a medida que se alejan y por eso se ha escogido para la aplicación de modelado.

El *Clip-Space* (espacio de recorte) constituye una fase fundamental en el *pipeline* gráfico, actuando como dominio intermedio donde la geometría 3D se prepara para su transformación final a coordenadas de pantalla.

En este espacio, se tiene que las coordenadas visibles cumplen las siguientes relaciones [5]:

$$|x| \leq w, \quad |y| \leq w, \quad |z| \leq w$$

La transformación al *Clip-Space* se realiza mediante la matriz de proyección, que mapea el *frustum* de vista a un cubo unitario.

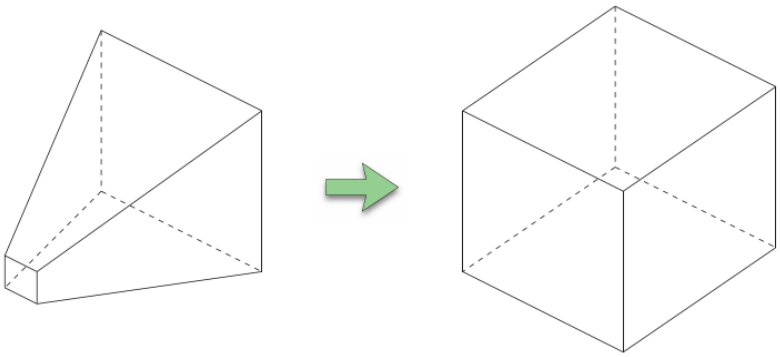

Figura 2.4: Frustum a Cubo unitario mediante la matriz de proyección [1].

La matriz de proyección perspectiva para un *frustum* se define como [2]:

$$M_{\text{proj}} = \begin{pmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.2}$$

donde:

- n, f = planos cercano y lejano ($z_{\text{near}}, z_{\text{far}}$)
- l, r = planos izquierdo y derecho ($x_{\text{min}}, x_{\text{max}}$)
- b, t = planos inferior y superior ($y_{\text{min}}, y_{\text{max}}$)

En la implementación, simplemente se usa `glm::frustum()` de la biblioteca GLM [3].

Posterior al uso de la matriz de proyección, se ejecuta la división perspectiva, que transforma las coordenadas homogéneas a un espacio de dispositivo normalizado (NDC):

$$(x, y, z, w) \rightarrow \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right) \tag{2.3}$$

2.4. Espacio de Ventana

El *viewport* es la región rectangular de la ventana de aplicación donde se renderiza la escena 3D. Es esencialmente el marco por donde usuario ve el mundo virtual.

En OpenGL, se usa la función `glViewport(x, y, width, height)` donde (x, y) es la esquina inferior izquierda en coordenadas de la ventana y `width` y `height` son las dimensiones de la ventana.

Para convertir las coordenadas (x_d, y_d, z_d) del espacio de dispositivo normalizado (NDC) a coordenadas de ventana (x_w, y_w, z_w) se realizan los siguientes cálculos [6]:

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{p_x}{2} x_d + o_x \\ \frac{p_y}{2} y_d + o_y \\ \frac{f-n}{2} z_d + \frac{n+f}{2} \end{pmatrix} \tag{2.4}$$

Aquí, x_w, y_w y z_w son las coordenadas resultantes del vértice en el espacio de ventana, y x_d, y_d y z_d son las coordenadas entrantes del vértice en el espacio de dispositivo normalizado. p_x y p_y son el ancho y alto del *viewport* en píxeles, y n y f son las distancias de los planos cercano y lejano en la coordenada z, (normalmente los rangos son $n = 0$, $f = 1$). Finalmente, o_x y o_y son la esquina inferior izquierda del *viewport*.

Para representar en la pantalla solo son necesarias las coordenadas (x_w, y_w) mientras que z_w se guarda internamente en el *depth buffer*.

Capítulo 3

Estructuras y Organización de Clases

Este capítulo describe las estructuras fundamentales que componen el sistema gráfico implementado. Cada estructura tiene un propósito específico en el *pipeline* de renderizado, desde la representación básica de vértices hasta la organización completa de modelos tridimensionales.

3.1. Vértice

La estructura `Vertex` representa un punto en el espacio tridimensional con atributos adicionales para renderizado. Cada vértice es un `struct` que contiene:

- `pos`: Coordenadas de posición en el espacio 3D (`glm::vec3`)
- `normal`: Vector normal para cálculos de iluminación (`glm::vec3`)
- `color`: Color RGB del vértice (`glm::vec3`)

Se proporcionan tres constructores:

- Constructor por defecto sin parámetros
- Constructor completo con posición y color explícitos
- Constructor simplificado solo con posición (color por defecto gris azulado)

Por defecto, la normal se inicializa apuntando hacia arriba (0.0, 1.0, 0.0), lo que corresponde a una orientación horizontal. Los colores se definen en el rango [0, 1] para cada componente RGB.

El método `calculateNormals()` es utilizado cada vez que se realiza una transformación en el modelo para que la iluminación sea consistente.

Vertex
+pos: glm::vec3 +normal: glm::vec3 +color: glm::vec3
+Vertex(): void +Vertex(float x, float y, float z, float r, float g, float b): void +Vertex(float x, float y, float z): void

Cuadro 3.1: Diagrama de la estructura Vertex

3.2. Cara

La estructura *Face* representa uno o más triángulos mediante índices a la lista de vértices del modelo. Implementa un diseño flexible que permite:

- Almacenar múltiples triángulos por cara
- Manejar listas de índices de diferentes formatos
- Validar que el número de índices sea múltiplo de 3

Face almacena sus datos en un vector de arrays de 3 enteros sin signo (*unsigned int*), donde cada array representa un triángulo. Se proporcionan cuatro constructores:

- Constructor por defecto vacío
- Constructor con lista de inicialización
- Constructor con vector de triángulos
- Constructor con vector plano de índices

Esta estructura es esencial para la representación eficiente de mallas poligonales, permitiendo reutilizar vértices entre múltiples caras reduciendo el coste computacional.

Face
+index: std::vector<std::array<unsigned int, 3»
+Face(): void +Face(std::initializer_list<unsigned int> ind): void +Face(const std::vector<std::array<unsigned int, 3»& triangles): void +Face(const std::vector<unsigned int>& flatTriangles): void

Cuadro 3.2: Diagrama de la estructura Face

3.3. Modelo

La clase *Model* encapsula una malla 3D completa, combinando vértices y caras en una entidad coherente. Sus principales características son:

- Almacena todos los vértices en un vector de objetos *Vertex*.

- Almacena todas las caras en un vector de objetos `Face`.
- Proporciona métodos de acceso para modificar la geometría.
- Ofrece conversión a una única lista de índices para renderizado.

Los métodos principales incluyen:

- `getVertex()`: Obtiene referencia a los vértices.
- `getFaces()`: Obtiene referencia a las caras.
- `addVertex()`: Añade un nuevo vértice.
- `getAllIndex()`: Genera lista plana de índices para OpenGL.

Esta clase actúa como contenedor principal para la geometría 3D, facilitando la manipulación y renderizado de objetos complejos.

Model
-faces: std::vector<Face> -vertex: std::vector<Vertex>
+Model(): void +Model(std::vector<Vertex>& vert, std::vector<Face>& fcs): void +getVertex(): std::vector<Vertex>& +getFaces(): std::vector<Face>& +addVertex(const Vertex& v): void +getAllIndex(): std::vector<unsigned int>

Cuadro 3.3: Diagrama de la clase Model

3.4. *MeshBuffers*

La estructura `MeshBuffers` gestiona los recursos de OpenGL necesarios para renderizar una malla. Contiene:

MeshBuffers
+VBO: GLuint +EBO: GLuint +VAO: GLuint +indexCount: GLuint +edgeEBO: GLuint +edgeVAO: GLuint +edgeIndexCount: GLuint
+MeshBuffers(): void

Cuadro 3.4: Diagrama de la estructura MeshBuffers

Esta estructura permite renderizar el mismo modelo tanto en modo sólido como en modo alambre, optimizando el uso de recursos de la GPU [7]. Todos los buffers se inicializan a 0 en el constructor, indicando que no han sido generados aún.

La función `createMesh` se encarga de generar y configurar los *buffers* de OpenGL necesarios para la representación gráfica de un modelo. En primer lugar, se obtienen los vértices y los índices del modelo, que se almacenan en un *Vertex Buffer Object* (VBO) y un *Element Buffer Object* (EBO), respectivamente. Estos buffers se asocian a un *Vertex Array Object* (VAO) principal destinado al renderizado de triángulos.

Los atributos de vértice se configuran de manera explícita siguiendo una estructura intercalada: la posición de un vértice ocupa los tres primeros valores de tipo `float`, la normal los tres siguientes y el color los tres finales, dando lugar a un *stride* total de nueve valores `float` por vértice. Esta disposición permite un acceso eficiente a los datos desde el *Vertex Shader* sin necesidad de realineamientos adicionales en memoria.

Además del VAO principal, la función construye un segundo VAO destinado a la visualización de las aristas del modelo. Para ello, se extraen los bordes del perímetro de cada cara y se almacenan en una estructura de tipo `std::set`, lo que garantiza la eliminación de duplicados independientemente del orden de los vértices. Una vez recopiladas, las aristas se transforman en un vector de índices que se carga en un `EBO` específico para el renderizado de líneas.

Ambos `VAO` comparten el mismo `VBO` de vértices, asegurando coherencia entre la geometría renderizada como triángulos y como aristas. Finalmente, se almacenan los contadores de índices correspondientes a cada `VAO` y se devuelve la estructura `MeshBuffers`, que encapsula todos los identificadores necesarios para el renderizado eficiente del modelo.

Tanto los modelos como los *MeshBuffers* se guardan en listas globales accesibles para todo el programa principal.

3.5. Cámara

La clase `Camara` implementa un sistema completo de visualización 3D, manejando la perspectiva como la vista del mundo virtual y el uso de cuaterniones para una representación estable de la orientación.

La cámara almacena tres componentes esenciales: posición en el espacio 3D, orientación como cuaternión, y matriz de proyección que define el frustum de visualización descrito en el apartado 2.3.

La orientación con cuaterniones evita problemas como el *gimbal lock* y permite interpolaciones suaves. Su uso en la rotación será descrita en el apartado 7.3.

La matriz de vista se genera como la inversa de la transformación de cámara en el mundo, combinando rotación (del conjugado del cuaternión) y traslación inversa. El método `getViewAndProjectionMatrix()` entrega una matriz combinada ya preparada.

El constructor calcula automáticamente la orientación desde una matriz lookAt, asegurando consistencia. `updateProjection()` ajusta dinámicamente la pro-

yección perspectiva, recalculando los planos los planos del *frustum* para adaptarse cambios de tamaño de la ventana de la aplicación.

$$right/top = width/height \tag{3.1}$$

$$aspectRatio := width'/height' \tag{3.2}$$

$$right := top * aspectRatio \tag{3.3}$$

La cámara expone una interfaz completa para consultar sus vectores fundamentales: frontal, derecho y superior. Estos vectores se derivan de la orientación actual y son esenciales para cálculos de movimiento relativo. El método move permite desplazar la cámara en su sistema de coordenadas local.

La clase también gestiona el estado de entrada del ratón con variables para guardar el tipo click y rastrear posiciones previas, facilitando operaciones interactivas.

Con las teclas *WASD* se puede mover la cámara en las direcciones determinadas por el cuaternión de orientación.

Camara
-projection: glm::mat4 -position: glm::vec3 -orientation: glm::quat -leftButtonPressed: bool -rightButtonPressed: bool -lastMousePos: glm::vec3 -lastMouseX: int -lastMouseY: int -left: float -right: float -bottom: float -top: float -zNear: float -zFar: float
+Camara(): void +Camara(const glm::vec3& pos, const glm::vec3& target, const glm::vec3& up, float left, float right, float bottom, float top, float zNear, float zFar): void +getProjection(): glm::mat4 const +getPosition(): glm::vec3 const +getViewMatrix(): glm::mat4 const +getViewAndProjectionMatrix(): glm::mat4 const +move(const glm::vec3& direction): void +isLeftButtonPressed(): bool const +setLeftButtonPressed(bool pressed): void +isRightButtonPressed(): bool const +setRightButtonPressed(bool pressed): void +getLastMousePos(): glm::vec3 const +setLastMousePos(const glm::vec3& pos): void +getOrientation(): glm::quat const +setOrientation(const glm::quat& newOrientation): void +getFront(): glm::vec3 const +getRight(): glm::vec3 const +getUp(): glm::vec3 const +updateProjection(float width, float height): void +getLastMouseX(): int +getLastMouseY(): int +setLastMouseX(int x): void +setLastMouseY(int y): void

Cuadro 3.5: Diagrama de la clase Camara

Capítulo 4

Renderizado en OpenGL

El sistema de renderizado implementado en esta aplicación se basa en el pipeline renderización de OpenGL en C++, adaptado para ofrecer herramientas de manipulación geométrica. El núcleo del sistema se organiza alrededor del bucle principal de GLUT [8], que gestiona los eventos de entrada, la lógica de actualización y el renderizado visual.

La función *main()* comienza con la configuración del contexto gráfico mediante GLUT, estableciendo un *double buffer* (frontal y posterior) con soporte para color RGBA y test de profundidad.

```
glutInit(&argc, argv);
glutInitDisplayMode(GL_DOUBLE | GL_RGBA | GLUT_DEPTH);
```

Esta configuración es fundamental para garantizar un renderizado sin parpadeos y una correcta visualización tridimensional. También se crea y se inicializa la ventana de la aplicación.

```
glutInitWindowSize(width, height);
int x = 200, y = 100;
glutInitWindowPosition(x, y);
glutCreateWindow("Editor 3D");
```

Posteriormente, GLEW [9] se encarga de cargar las extensiones modernas de OpenGL, permitiendo el uso de *shaders* programables en GLSL versión 330, esenciales para implementar técnicas de iluminación avanzadas.

```
GLenum res = glewInit();
if (res != GLEW_OK) {
    fprintf(stderr, "Error: '%s'\n", glewGetErrorString(res));
    return 1;
}
```

Tras ello, se inicializa la interfaz de ImGui, que se renderiza directamente sobre la escena 3D. Esta se explicará en el capítulo 8.

Lo siguiente es especificar que un fragmento pasará el test si su profundidad es menor o igual (*LEQUAL*) a la existente.

```
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LEQUAL);
glDepthMask(GL_TRUE);
```

Se define que los vértices ordenados en sentido antihorario (*glFrontFace(GL_CCW)*) representan la cara frontal del polígono.

Tras ello, se define el *polygon offset* para líneas. Esto resuelve un problema visual conocido como *z-fighting*, donde los bordes de las caras y los modelos sólidos compiten por la misma posición de profundidad, causando errores visuales.

```
glEnable(GL_POLYGON_OFFSET_LINE);
glPolygonOffset(1.0f, 1.0f);
```

Se sigue con la compilación de los *shaders* y la asignación de funciones *callback* de ratón, teclado, redimensión de ventana y renderizado.

```
compileShaders(VERTEXSHADERFILENAME, FRAGMENTSHADERFILENAME, &viewProjectionLo
 glutReshapeFunc(reshapeCB);
 glutDisplayFunc(renderSceneCB);
 glutKeyboardFunc(keyboardCB);
 glutMouseFunc(mouseCB);
 glutMotionFunc(mouseMotionCB);
```

4.1. Función de Renderizado y Envío de Datos al *Shader*

La función *renderSceneCB()* constituye el núcleo del *pipeline* de renderizado de la aplicación de modelado. Este *callback* se ejecuta en cada frame del bucle principal, garantizando una actualización continua y fluida de la escena.

Se comienza con la limpieza sistemática de los *buffers* de color y profundidad mediante *glClear()*, eliminando cualquier información residual del frame anterior. La posición de la luz se define en coordenadas mundiales $(2{,}0, 2{,}0, 2{,}0)$, situándose en el primer octante para iluminación diagonal que realce la percepción de volumen y profundidad. La posición de la cámara (*viewPos*) se obtiene para cálculos de iluminación dependientes del observador.

El envío de parámetros al *shader* sigue una secuencia lógica: primero las propiedades de iluminación (posición de luz y cámara) mediante *glUniform3f()*, luego las matrices combinadas de vista y proyección. El uso de *glUniformMatrix4fv()* con el parámetro *GL_FALSE* indica que la matriz ya está en el formato columna-mayor requerido por OpenGL.

En primer lugar, se renderizan las superficies sólidas de todos los modelos. La variable uniforme *drawEdges* se establece en 0, indicando al *fragment shader* que aplique el modelo de iluminación Blinn-Phong (descrito en el capítulo 5). El bucle recorre la lista de *MeshBuffers*, activando cada VAO sólido mediante *glBindVertexArray()* y ejecutando *glDrawElements()* en modo *GL_TRIANGLES*.

En segundo lugar, se renderizan los bordes perimetrales de las caras. Cambiando `drawEdges` a 1, el *fragment shader* simplifica su cálculo a un color constante (negro). `glLineWidth(2.0f)` establece un grosor visualmente prominente que facilita su visualización. El bucle recorre la lista de `MeshBuffers`, activando cada VAO de bordes (`edgeVAO`) mediante `glBindVertexArray()` y ejecutando `glDrawElements()` en modo `GL_LINES`.

En último lugar, el Gizmo de transformación requiere tratamiento especial ya que su manipulación debe ser posible para el usuario sin importar el posicionamiento de los modelos una vez ya seleccionados. `glDisable(GL_DEPTH_TEST)` y `glDepthMask( GL_FALSE)` desactivan temporalmente el test y escritura de profundidad, haciendo que el Gizmo se renderice sobre toda la geometría existente. Cada eje (X, Y, Z) se dibuja independientemente, permitiendo diferentes colores y lógica de selección. La restauración del estado de profundidad al final asegura que las siguientes escenas no se vean afectadas.

Tras el renderizado de modelos y Gizmo, se renderiza la interfaz de usuario de ImGui sobre la escena 3D, sin interferir con el *pipeline* de OpenGL. `glutPostRedisplay()` marca la ventana para redibujado continuo, asegurando la fluidez. Finalmente, `glutSwapBuffers()` intercambia los *buffers* frontal y posterior, pertenecientes al *double buffering*, lo cual elimina el parpadeo durante las actualizaciones visuales.

Capítulo 5

Iluminación mediante *Shaders* en OpenGL

Previamente, para cada vértice, se calcula la normal como el promedio normalizado de las normales de todas las caras adyacentes.

El sistema de iluminación y renderización emplea dos *shaders* principales que operan en secuencia dentro del *pipeline* de OpenGL. El *shader* de vértices (`vertex.vert`) se encarga exclusivamente de transformaciones espaciales, recibiendo las posiciones de los vértices en coordenadas locales y aplicando la matriz combinada de vista y proyección convertiéndolas al *Clip-Space* para la rasterización en fragmentos. Simultáneamente, transmite atributos adicionales como normales y colores al siguiente *shader* sin procesamiento adicional, minimizando la complejidad computacional en esta etapa temprana.

El *shader* de fragmentos (`vertex.frag`) implementa la lógica central de iluminación y determinación de color. Se alterna entre modos de renderizado mediante una variable uniforme simple. Cuando se activa el modo de visualización de bordes, el *shader* devuelve inmediatamente un color negro constante, evitando todos los cálculos de iluminación y optimizando significativamente el rendimiento durante esta operación frecuente.

5.1. Modelo de Iluminación Blinn-Phong

Para el sombreado de superficies, se implementa el modelo de iluminación Blinn-Phong, una variante optimizada del modelo Phong clásico. Este modelo combina tres componentes de iluminación que simulan diferentes fenómenos físicos de interacción luz-materia.

Los teoría en la que se basa la implementación reside en [10], y el código que se ha usado de referencia en [11].

La componente ambiental representa la iluminación indirecta que llega a todas las superficies por igual, proporcionando un nivel base de visibilidad incluso en

áreas no directamente expuestas a fuentes luminosas.

$$I_{amb} = k_a \cdot I_{light} \tag{5.1}$$

$$C_{amb} = I_{amb} \cdot C_{mat} \tag{5.2}$$

donde:

- $k_a = 0{,}3$ es el coeficiente ambiental,
- I_{light} es la intensidad de la fuente luminosa,
- C_{mat} es el color del material,
- C_{amb} es el color ambiental resultante.

La componente difusa dota a los objetos un mayor brillo cuanto más cerca se encuentran de la dirección de luz.

$$L = \frac{(P_{light} - P_{frag})}{||(P_{light} - P_{frag})||} \tag{5.3}$$

$$\theta = \text{máx}(N \cdot L, 0) \tag{5.4}$$

$$I_{diff} = k_d \cdot \theta \cdot I_{light} \tag{5.5}$$

$$C_{diff} = I_{diff} \cdot C_{mat} \tag{5.6}$$

donde:

- L es el vector dirección hacia la luz,
- P_{light} es la posición de la luz en espacio mundial,
- P_{frag} es la posición del fragmento en espacio mundial,
- N es la normal del vértice,
- $k_d = 1{,}0$ es el coeficiente difuso.

La componente especular utiliza la aproximación de Blinn, que calcula un vector intermedio entre la dirección de vista y la dirección de la luz.

$$V = \frac{P_{eye} - P_{frag}}{||P_{eye} - P_{frag}||} \tag{5.7}$$

$$H = \frac{L + V}{||L + V||} \tag{5.8}$$

$$\alpha = \text{máx}(N \cdot H, 0) \tag{5.9}$$

$$I_{spec} = k_s \cdot \alpha^n \cdot I_{light} \tag{5.10}$$

$$C_{spec} = I_{spec} \cdot C_{light} \tag{5.11}$$

donde:

- V es el vector dirección hacia la cámara,
- P_{eye} es la posición de la cámara,

- H es el vector intermedio,
- $k_s = 0{,}2$ es el coeficiente especular,
- $n = 32$ es el factor de brillo,
- C_{light} es el color de la luz.

La iluminación final es la suma de las tres componentes:

$$C_{final} = C_{amb} + C_{diff} + C_{spec} \tag{5.12}$$

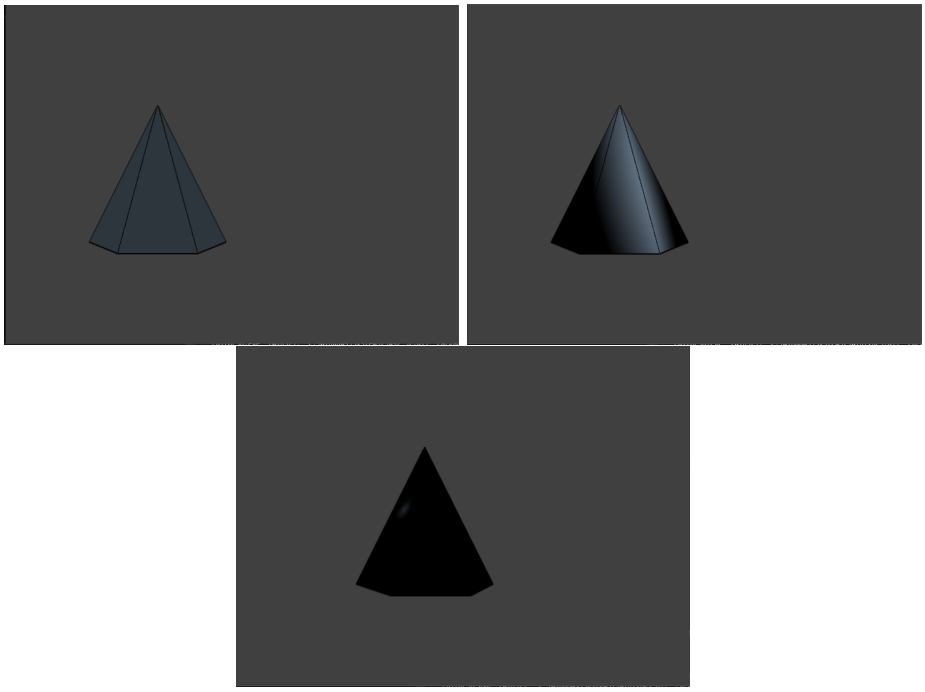

Figura 5.1: Resultado de utilizar cada una de la componentes por separado. En orden: luz ambiental, luz difusa y luz especular (solo una pequeña zona es iluminada).

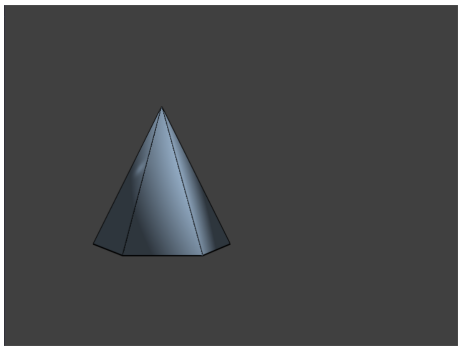

Figura 5.2: Resultado de unir las tres componentes que forman la iluminación.

Capítulo 6

Selección e Interacción

6.1. Algoritmos de selección

Los sistemas de modelado 3D requieren herramientas precisas y eficientes para la selección de elementos geométricos (vértices, caras, modelos completos).

Un rayo es una línea que tiene un único punto extremo O y se extiende hasta el infinito en una dirección dada $\vec{D}$. Los rayos se expresan típicamente mediante la ecuación paramétrica [1]:

$$R(t) = O + t \cdot \vec{D} \tag{6.1}$$

donde:

- O es el punto de origen,
- $\vec{D}$ es el vector de dirección,
- t es un parámetro escalar con $t \geq 0$.

Este concepto es en el que se basarán los algoritmos de selección para detectar intersecciones.

Antes de todo, para seleccionar un elemento mediante el uso del ratón por el usuario, es necesario emplear un rayo proyectado con origen en la posición de la cámara y con una dirección calculada con la función `screenToWorld()` y las coordenadas de ventana x_w e y_w.

En la función `screenToWorld()`, lo primero que se hace es convertir las coordenadas de ventana a coordenadas NDC modificando la ecuación 2.4:

$$\begin{pmatrix} x_d \\ y_d \end{pmatrix} = \begin{pmatrix} \frac{2}{p_x}(x_w - o_x) - 1 \\ 1 - \frac{2}{p_y}(y_w - o_y) \end{pmatrix} \tag{6.2}$$

Siendo p_x y p_y, el ancho y alto del viewport en píxeles, y o_x y o_y, la esquina inferior izquierda del viewport.

Tras ello se crean los puntos near $= (x_d, y_d, -1{,}0, 1{,}0)$ y far $= (x_d, y_d, 1{,}0, 1{,}0)$ en el *Clip-Space*. La tercera coordenada indica que que están en el plano cercano y lejano, respectivamente. [2]

Para obtener los puntos en el espacio de mundo hay que multiplicar por la inversa de $M_{proj} \cdot M_{view}$. Con estos puntos se puede calcular directamente el vector que representa la dirección del rayo creado por el ratón al hacer click.

Se prioriza la selección de vértices en vez de la selección de caras para establecer una jerarquía.

6.1.1. Selección de vértices

El algoritmo de intersección rayo-vértice determina si un rayo en el espacio 3D pasa suficientemente cerca de un vértice específico, dentro de una tolerancia predefinida.

Se revisan todos los vértices de los modelos existentes. Se llama a la función `rayIntersectsVertex()` que comprueba si hay intersección y también devuelve la distancia real. Con esto se puede seleccionar el más cercano al usuario entre los detectados.

La función `rayIntersectsVertex()` calcula el vector desde el origen del rayo hasta el vértice y lo compara con la dirección normalizada del rayo mediante el producto escalar.

$$t = (p_{vertice} - origen_{rayo}) \cdot direccion_{rayo} \tag{6.3}$$

Como el rayo se ha creado al hacer click en la pantalla, es evidente que el rayo siempre estará dentro del espacio visible de la cámara. Por lo tanto, se deduce que, si t es negativo, el vértice no es visible para el usuario y no debe tenerse en cuenta.

En otro caso, se calcula el siguiente punto:

$$p = origen_{rayo} + t \cdot direccion_{rayo} \tag{6.4}$$

Este punto es el punto contenido en el rayo que es más cercano al vértice a comprobar. Finalmente, se calcula la distancia entre el vértice y el punto p y, si entra dentro de los valores permitidos por la tolerancia, se devuelve `True`.

6.1.2. Selección de caras por triángulos

Se revisan todos los triángulos de las caras de los modelos existentes. Se llama a la función `rayIntersectsFace()` que comprueba si hay intersección y también devuelve la distancia real. Con esto se puede seleccionar la cara o modelo más cercano al usuario entre los detectados.

Para definir la función `rayIntersectsFace()` es necesario introducir el algoritmo de Möller–Trumbore.

Algoritmo de Möller–Trumbore

El algoritmo de Moller–Trumbore aborda el problema de determinar la intersección entre un rayo y un triángulo en el espacio tridimensional.

El marco teórico del algoritmo se ha extraído de [12].

Sea un rayo $R(t) = O + t \cdot D, t \geq 0$ con O punto de origen y D vector dirección del rayo y sea un triángulo tal que todo punto definido dentro de él se puede expresar como:

$$T(u, v) = (1 - u, v)V_0 + uV_1 + vV_2, u \geq 0, v \geq 0, u + v \leq 1 \tag{6.5}$$

donde:

- V_0, V_1 y V_2 son los vértices del triángulo,
- (u, v) son las coordenadas baricéntricas del punto.

Igualando ambas ecuaciones se llega a:

$$O + tD = (1 - u - v)V_0 + uV_1 + vV_2 \tag{6.6}$$

Reorganizando la ecuación:

$$\begin{pmatrix} -D, & V_1 - V_0, & V_2 - V_0 \end{pmatrix} \begin{pmatrix} t \\ u \\ v \end{pmatrix} = V_2 - V_0 \tag{6.7}$$

Denotando $E_1 = V_1 - V_0$, $E_2 = V_2 - V_0$ y $T = V_2 - V_0$, y aplicando la regla de Cramer, se obtiene la siguiente ecuación:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{|-D,\ E_1,\ E_2|} \begin{pmatrix} |T,\ E_1,\ E_2| \\ |-D,\ T,\ E_2| \\ |-D,\ E_1,\ T| \end{pmatrix} \tag{6.8}$$

Finalmente, se utiliza la propiedad del álgebra lineal que indica que $|A, B, C| = -(A \times C) \cdot B = -(C \times B) \cdot A$, resultando en:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{pmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{pmatrix}, \tag{6.9}$$

La implementación de `rayIntersectsFace()` se ha basado en el código hallado en [13]. Se debe comprobar que $(D \times E_2) \cdot E_1 \neq 0$ y que u y v están en todo momento dentro del triángulo.

Si todas esas comprobaciones se validan, se obtiene un t resolviendo el sistema. Si ese t es mayor que 0, se devuelve el punto de intersección, usando la ecuación del rayo.

Además, se ha incorporado una fase temprana de descarte de caras traseras no visibles en OpenGL (*backface culling*) mediante el cálculo del producto escalar entre la normal del triángulo y la dirección del rayo.

6.2. Interacción mediante Gizmo

6.2.1. Referencia cartesiana

En A, espacio afín sobre K, sea $R_c = \{O; B\}$ donde $O \in A$ es el origen de referencia y B es una base del espacio vectorial asociado a A.

R_c es una referencia cartesiana si $\forall Q \in A$ se determina un $\vec{v} = \vec{OQ}$ que se puede expresar de forma única como $\vec{v} = \lambda_1\vec{v}_1 + ... + \lambda_n\vec{v}_n$ tal que $\lambda_i \in K$ y $\vec{v}_i \in B, \forall i \in \{1, ...n\}$.

6.2.2. Gizmo de Transformación

Es imprescindible saber como traducir la interacción del usuario para poder aplicar una transformación o conjunto de transformaciones al modelo o elemento seleccionado.

Para ello, se introduce el concepto de Gizmo, una herramienta común en aplicaciones de modelado que sirve para visualizar el punto de origen de la transformación y los ejes que forman la base codificados por colores (rojo, verde, azul).

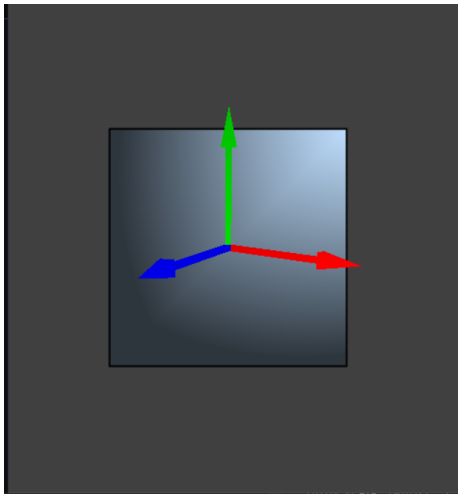

Figura 6.1: Gizmo sobre un cubo.

Este concepto es claramente una aplicación de la noción de referencia de la geometría afín. En la implementación, el centro de referencia es el centroide (punto resultado de realizar una media aritmética a las componentes de un conjunto de puntos) ya sea del modelo entero o de una cara.

Al hacer click izquierdo en uno de los ejes del Gizmo y arrastrar en una dirección, se generará un vector en el mundo real que será la representación del movimiento y que será interpretado por distintas funcionalidades para modificar el modelo asociado. También, con el click derecho y arrastre, se pueden rotar los vectores que forman la base manteniendo siempre la ortogonalidad.

Gizmo
+position: glm::vec3 +rotation: glm::quat +modelX: Model +modelY: Model +modelZ: Model +currentNormal: glm::vec3
+Gizmo() +Gizmo(const glm::vec3& pos) +setPosition(const glm::vec3& pos): void +setRotation(const glm::quat& newRotation): void +rotateGizmo(const glm::quat& deltaRotation): void +setNormal(const glm::vec3& normal): void +getCurrentNormal() const: glm::vec3
-updateGizmoModels(): void

Cuadro 6.1: Diagrama de la clase Gizmo

En la implementación, se inicializa con un parámetro posición y se crean 3 modelos que no están en la lista de modelos a editar. Estos serán flechas orientadas a los ejes X,Y y Z en la base usual del espacio mundial. Se creará inicialmente un cuaternión identidad que se usará para la rotación del Gizmo.

Además, se ha incluido la función `gizmoLookAtNormal()` que orienta el Gizmo para que tenga un eje orientado a la normal acumulada de la cara seleccionada (`calculateFaceNormal()`). Esto permite que el usuario pueda manipular con mayor facilidad el Gizmo y así realizar transformaciones más comodamente.

6.2.3. Algoritmo de Interacción mediante Gizmo

En el momento del click, mediante `mouseCB(button, state, x, y)`, se guardan los valores de las coordenadas de ventana (x, y) y si se ha hecho click con el botón izquierdo o derecho. También, se llama a la función `select()`.

En la fase de selección, si se está en los modos de traslación, escala o extrusión y hay un modelo o cara ya seleccionado, se añaden los ejes del Gizmo como un elemento a priorizar por encima de los modelos corrientes.

Se usa la función `detectGizmoClick()` (con los parámetros punto de origen y dirección del rayo generado por el click) para determinar que eje del Gizmo se está interactuando. Para ello se usa la función `rayIntersectsFace()` ya descrita y una enumeración global con las constantes `NONE_AXIS`, `X_AXIS`, `Y_AXIS` y `Z_AXIS`.

En cada frame, al arrastrar el ratón con el botón pulsado con un eje seleccionado, mediante `mouseMotionCB(x, y)` se llama a `screenToWorld()`, previamente definido, se invoca a la funcionalidad a realizar, dependiendo del modo de la aplicación, y se actualizan las coordenadas de ventana del click.

OpenGL se encarga de entregar los parámetros correctamente a `mouseCB()` y `mouseMotionCB()` al asociarlos en el `main()` mediante las líneas `glutMouseFunc(mouseCB)` y `glutMotionFunc(mouseMotionCB)`.

Dentro de las funcionalidades de traslación, escala y extrusión, se utiliza el método `movementGizmo()` con los parámetros deltaX y deltaY. Estos se calculan restando las coordenadas (x, y) a las coordenadas de ventana guardadas anteriormente.

En `movementGizmo()` se calcula el vector de movimiento en espacio mundial para la manipulación de objetos 3D usando un Gizmo de transformación.

Se comienza estableciendo los vectores fundamentales necesarios para el cálculo. Primero, extrae el cuaternión de rotación del Gizmo y la convierte en una matriz de base 3x3 (`mat3_cast()`). Los vectores gizmoX, gizmoY y gizmoZ representan las direcciones de los ejes del Gizmo transformadas según su orientación actual. Simultáneamente, se obtienen los vectores de orientación de la cámara (frontal, derecho y superior), que son cruciales para determinar la relación espacial entre la vista del usuario y el Gizmo.

Dependiendo del eje activamente seleccionado por el usuario (X, Y o Z), el algoritmo asigna la dirección correspondiente a `selectedAxisDir`. Posteriormente, verifica si la cámara se encuentra demasiado cerca del extremo del eje del Gizmo. Esta validación previene comportamientos erráticos o numéricamente inestables cuando la distancia entre la cámara y el punto de manipulación es mínima.

El algoritmo incorpora una forma de detectar casos límite donde la proyección del eje en la pantalla colapsaría esencialmente a un punto. Primero, calcula el vector desde la cámara hacia el Gizmo y evalúa qué tan directamente centrado está el Gizmo en la vista mediante el producto escalar con la dirección frontal de la cámara. Luego, verifica si el eje seleccionado es casi paralelo al vector frontal de la cámara.

Cuando ambos condiciones se cumplen simultáneamente, se trata de forma especial. En lugar de intentar calcular una proyección significativa, utiliza directamente el movimiento vertical del ratón para controlar el desplazamiento a lo largo del eje seleccionado. Se devuelve el vector del eje seleccionado multiplicado por `deltaY` y por 0.01 (ajuste de sensibilidad).

En cualquier otro caso, el algoritmo proyecta el centro del Gizmo y la punta del eje seleccionado en el espacio 2D de la pantalla utilizando las matrices de vista y proyección de la cámara.

$$centro_{2D} = M_{proj} \cdot M_{view} \cdot (x_{centro}, y_{centro}, z_{centro}, 1) \tag{6.10}$$

$$extremo_{2D} = M_{proj} \cdot M_{view} \cdot ((x_{centro}, y_{centro}, z_{centro}, 1) + (x_{dir}, y_{dir}, z_{dir}, 0)) \tag{6.11}$$

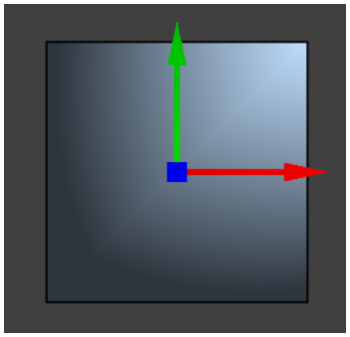

Figura 6.2: Caso especial en el que un eje del Gizmo se visualiza como un punto.

donde:

- $(x_{centro}, y_{centro}, z_{centro})$ es la posición del Gizmo en el Espacio de Mundo,
- M_{proj} y M_{view} son las matrices de vista y proyección, respectivamente,
- $(x_{dir}, y_{dir}, z_{dir})$ es el vector dirección del eje seleccionado (`selectedAxisDir`).

Estas coordenadas se normalizan al *Clip-Space* dividiendo por el componente w. A partir de estas proyecciones, se calcula la dirección aparente del eje en la pantalla (`axisScreenDir`).

$$centro_{clip} = \frac{centro_{2D}}{centro_{2D}.w}, extremo_{clip} = \frac{extremo_{2D}}{extremo_{2D}.w} \tag{6.12}$$

$$axisScreenDir = \frac{extremo_{clip} - centro_{clip}}{\|extremo_{clip} - centro_{clip}\|} \tag{6.13}$$

El desplazamiento del ratón (`deltaX`, `deltaY`) se convierte a coordenadas normalizadas de dispositivo (NDC) escalando apropiadamente según las dimensiones de la ventana. Esta conversión permite comparar directamente el movimiento del ratón con la dirección proyectada del eje. Se normaliza el vector de movimiento del ratón y calcula su alineación con la dirección del eje en pantalla mediante un producto escalar. Esta alineación determina si el movimiento del usuario es en la dirección positiva o negativa del eje.

$$alineacion = axisScreenDir \cdot \frac{(NDCdeltaX, NDCdeltaY)}{\|(NDCdeltaX, NDCdeltaY)\|} \tag{6.14}$$

Finalmente, se calcula la magnitud del movimiento a partir de la longitud del vector de desplazamiento del ratón y se aplica el signo determinado por la alineación.

$$movimientoConSigno = \begin{cases} \|(NDCdeltaX, NDCdeltaY)\| & \text{si } alineacion \geq 0 \\ -\|(NDCdeltaX, NDCdeltaY)\| & \text{si } alineacion < 0 \end{cases} \tag{6.15}$$

Se devuelve el vector `selectedAxisDir` multiplicado por `movimientoConSigno`.

El algoritmo para rotar los ejes del Gizmo es el mismo que el utilizado para rotar modelos en general. Se verá en el apartado 7.3.

6.3. Cuaternión

Para registrar el movimiento del ratón y usarlo en la rotación tanto de la cámara como de los modelos (apartado 7.3), es necesario emplear otro tipo de métodos.

Según Dunn y Parberry [5], existen 3 formas de representar el desplazamiento angular y cambios de orientación:

- **Forma matricial 3x3**: se expresa la orientación relativa de dos espacios de coordenadas proporcionando la matriz de rotación para transformar vectores de un espacio de coordenadas al otro. Sin embargo, su uso es computacionalmente ineficiente, es muy propenso a crear matrices incorrectas y ocupan demasiado en la memoria.
- **Ángulos de Euler**: se descompone el desplazamiento angular en tres rotaciones, cada una sobre uno de los 3 ejes perpendiculares. Es un método muy eficiente en cuanto a memoria, sin embargo, existen varios problemas [14]:
  - Al combinar dos o más rotaciones, resulta difícil visualizar y predecir cómo se comportará la orientación final.
  - Bajo ciertas condiciones (generalmente un giro de ±90°)[5] se pierde el acceso a uno de los ejes de rotación del objeto. Este problema recibe el nombre de Bloqueo de Cardán (*Gimbal Lock*).
- **Cuaterniones**: la representación empleada en esta aplicación de modelado.

6.3.1. Definición y Ventajas del Cuaternión

Un cuaternión q es el par ordenado:

$$q = [w, (x, y, z)], \quad w \in \mathbb{R}, (x, y, z) \in \mathbb{R}^3 \tag{6.16}$$

El cuaternión está estrechamente relacionado con el escalar θ y el eje de rotación en forma de vector, (n_x, n_y, n_z), que define un desplazamiento angular. Para calcular el par $(\theta, (n_x, n_y, n_z))$ a partir de un cuaternión [5]:

$$\begin{bmatrix} w, & (x, y, z) \end{bmatrix} = [\cos(\theta/2), (\sin(\theta/2) \cdot n_x, \sin(\theta/2) \cdot n_y, \sin(\theta/2) \cdot n_z)] \,. \tag{6.17}$$

Esto es más óptimo de almacenar que una matriz 3x3 y evita el Bloqueo de Cardán producido por los ángulos de Euler.

Habrá momentos puntuales en los que se quiera utilizar un matriz de giro. Para ello, se utilizan las funciones `glm::mat3_cast()` y `glm::mat4_cast()` de la biblioteca GLM [15]:

$$qToM_{3x3} = \begin{pmatrix} 1 - 2y^2 - 2z^2 & 2xy + 2wz & 2xz - 2wy \\ 2xy - 2wz & 1 - 2x^2 - 2z^2 & 2yz + 2wx \\ 2xz + 2wy & 2yz - 2wx & 1 - 2x^2 - 2y^2 \end{pmatrix} \tag{6.18}$$

$$qToM_{4x4} = \begin{pmatrix} 1-2y^2-2z^2 & 2xy+2wz & 2xz-2wy & 0 \\ 2xy-2wz & 1-2x^2-2z^2 & 2yz+2wx & 0 \\ 2xz+2wy & 2yz-2wx & 1-2x^2-2y^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{6.19}$$

6.4. Algoritmo de Interacción mediante Cuaterniones

Una vez ya explicado que en detalle los cuaterniones, se debe describir su relevancia en concreto en la interacción.

Al hacer click y en el arrastre, se llama a la función `mapToSphere()`. Esta transforma las coordenadas (x, y) del ratón a coordenadas NDC ((x_{NDC}, y_{NDC})) y se proyectan sobre la esfera unitaria tridimensional:

$$l = x_{NDC}^2 + y_{NDC}^2 \tag{6.20}$$

$$p_{esfera} = \begin{cases} (x_{NDC}, y_{NDC}, \sqrt{1-l}) & \text{si } l \leq 1 \\ (\frac{x_{NDC}}{\sqrt{l}}, \frac{y_{NDC}}{\sqrt{l}}, 0) & \text{si } l > 1 \end{cases} \tag{6.21}$$

Se invoca a `quatFromVectors()` teniendo como parámetros al anterior punto proyectado que se ha guardado (`from`) y el actual sobre la esfera (`to`).

En esa función, se calcula el componente escalar del cuaternión y el producto vectorial de `from` y `to` como resultado de desarrollar la ecuación 6.17:

$$w = ||from|| \cdot ||to|| + from \cdot to \tag{6.22}$$

$$(x_v, y_v, z_v) = from \times to \tag{6.23}$$

Tras ello, la función `quatFromVectors()` devuelve el cuaternión $[w, (x_v, y_v, z_v)]$ que será utilizado en la rotación de modelos o de la cámara.

Capítulo 7

Transformaciones Geométricas y Herramientas

7.1. Traslación

La funcionalidad de traslación `translate()` implementa un sistema de manipulación espacial que permite mover elementos dentro de una escena 3D mediante interacción con el usuario. Este mecanismo es fundamental en aplicaciones de modelado y edición tridimensional, donde la capacidad de reposicionar modelos y vértices de manera precisa e intuitiva es esencial.

Se comienza verificando si hay elementos seleccionados (vértices o modelos) y si se está moviendo mediante un eje del Gizmo de transformación.

7.1.1. Traslación de un único vértice

Si se está en modo de selección de vértices, se realiza una traslación específica sobre ese punto geométrico. La función `movementGizmo()` devuelve un vector en 3 dimensiones, considerando la orientación del eje seleccionado.

Este vector se aplica directamente a la posición del vértice seleccionado, actualizando tanto su representación interna como su posición mundial calculada.

$$V' = V + translation \tag{7.1}$$

$$pos'_{gizmo} = pos_{gizmo} + translation \tag{7.2}$$

donde:

- $translation$ es el vector devuelto por `movementGizmo()`.
- V es la posición del vértice a trasladar,
- pos_{gizmo} es el origen del Gizmo.

La actualización inmediata de los `MeshBuffers` asociados mediante `updateModelMesh()` garantiza que los cambios se reflejen visualmente en el siguiente ciclo de renderizado.

7.1.2. Traslación de un modelo

En cambio, si se está en modo de selección de modelos, se calcula la matriz de traslación mediante `glm::translate(glm::mat4(1.0f), translation)`, siendo *translation*, nuevamente, el vector devuelto por `movementGizmo()`.

Esta función de la biblioteca GLM internamente hace los siguientes cálculos:

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7.3}$$

donde (t_x, t_y, t_z) es el vector $translation$.

Esto equivaldría a la forma matricial de la ecuación 7.1 y se aplicaría a cada vértice.

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} \tag{7.4}$$

donde $(x, y, z) = V$.

Tanto la posición del Gizmo como los `MeshBuffers` se actualizarían como en el apartado anterior.

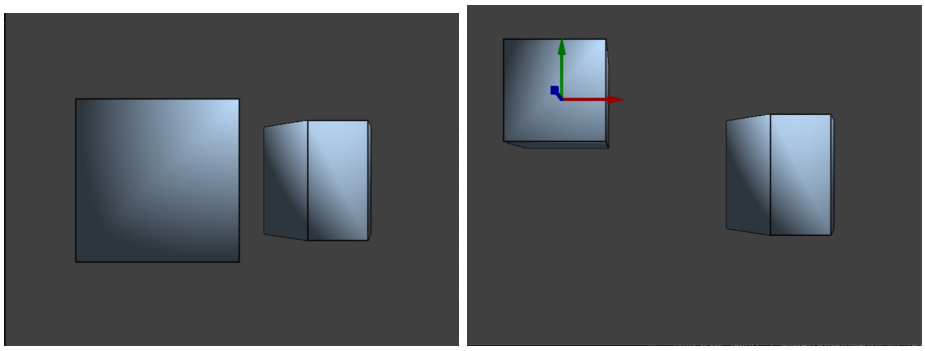

Figura 7.1: Situación inicial y final tras aplicar una traslación al cubo.

7.2. Escala

La funcionalidad de escalado (`scale`) implementa un sistema completo de escalado tridimensional que permite modificar el tamaño de modelos mediante interacción directa con el usuario si se esta en el modo escala.

7.2.1. Vector de escala en caso uniforme

Para operaciones de escalado uniforme (cuando no hay eje específico seleccionado), se calcula `scaleVector` utilizando el movimiento vertical del ratón como control.

$$scaleFactor = 1 - deltaY \cdot 0{,}01 \tag{7.5}$$

donde:

- `-deltaY` es el desplazamiento vertical convertido,
- 0.01 es un factor de sensibilidad adecuado según las pruebas realizadas.

$$scaleVector = \begin{cases} \begin{pmatrix} 0{,}01 \\ 0{,}01 \\ 0{,}01 \end{pmatrix} & \text{si } scaleFactor \leq 0{,}01 \\ \begin{pmatrix} scaleFactor \\ scaleFactor \\ scaleFactor \end{pmatrix} & \text{si } scaleFactor > 0{,}01 \end{cases} \tag{7.6}$$

Esta vez, el valor mínimo de 0.01 previene que los modelos se vuelvan excesivamente pequeños.

7.2.2. Vector de escala por eje

En el caso de que se este moviendo un eje, entonces, para calcular `scaleVector` se establece el `float` delta. Este valor es el producto escalar entre el vector devuelto por `movementGizmo()` y la dirección del eje. En esta situación el `scaleFactor` pasa a ser:

$$scaleFactor = 1 + delta \cdot 0{,}05 \tag{7.7}$$

donde 0.05 es un factor de sensibilidad adecuado para este caso en concreto.

También se realiza la substitución de `scaleFactor` si es inferior a 0.01. El `scaleVector` será diferente dependiendo del eje seleccionado del Gizmo:

$$scaleVector = \begin{cases} \begin{pmatrix} scaleFactor \\ 1 \\ 1 \end{pmatrix} & \text{si el eje seleccionado es el eje X} \\ \begin{pmatrix} 1 \\ scaleFactor \\ 1 \end{pmatrix} & \text{si el eje seleccionado es el eje Y} \\ \begin{pmatrix} 1 \\ 1 \\ scaleFactor \end{pmatrix} & \text{si el eje seleccionado es el eje Z} \end{cases} \tag{7.8}$$

7.2.3. Transformación Compuesta de Escala

Una vez ya calculado el vector de escala, se realiza una traslación al origen, se rota el espacio hacia la orientación del gizmo de transformación, se escala, se hace la rotación inversa y se traslada de regreso.

$$T_f = T(centroide) \cdot R_g \cdot \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot R_g^{-1} \cdot T(-centroide) \tag{7.9}$$

donde:

- $T(centroide)$: Traslación al centroide del modelo,
- R_g: Matriz de rotación calculada a partir del cuaternión de orientación del Gizmo (`mat4_cast()`),
- $scaleVec = (s_x, s_y, s_y)$.

Los `MeshBuffers` se actualizarían como en la traslación mientras que la posición del Gizmo es el resultado de recalcular el centroide del modelo.

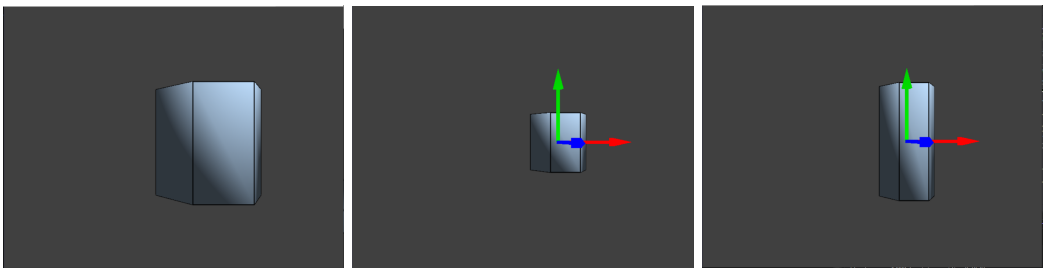

Figura 7.2: Situación inicial, tras hacer una escala uniforme y tras hacer una escala en el eje Y (verde) del Gizmo.

7.3. Rotación

La funcionalidad de rotación permite rotar modelos y la cámara mediante la interacción del usuario. Si se hace click derecho y se arrastra, la cámara rotará. En cambio, si se está en modo rotación y se hace click izquierdo (derecho para el Gizmo si está visible), se rotará el modelo seleccionado.

7.3.1. Rotación de Cámara

Dado el cuaternión de rotación ($q_1 = [w_1, v_1]$) conseguido mediante el proceso descrito en el apartado 6.4 y el cuaternión de orientación ($q_2 = [w_2, v_2)]$) de la cámara, se calcula un nuevo cuaternión (q'):

$$q = q_1 q_2 = [w_1 \cdot w_2 - v_1 \cdot v_2, w_1 \cdot v_2 + x_2 \cdot v_1 + v_1 \times v_2] = [w, v] \tag{7.10}$$

$$m = ||q|| = \sqrt{w^2 + ||v||^2} \tag{7.11}$$

$$q' = \frac{q}{m} = [\frac{w}{m}, \frac{v}{m}] \tag{7.12}$$

Estas operaciones sobre cuaterniones se definen en [5].

El cuaternión de la cámara se actualiza para pasar a ser el cuaternión calculado. La matriz de vista también se actualizará.

Figura 7.3: Situación inicial y final tras rotar la cámara.

7.3.2. Rotación de Modelos

Dado el cuaternión de rotación ($q = [w, v]$) conseguido mediante el proceso descrito en el apartado 6.4 y el centroide ($c = (x_c, y_c, z_c)$) del modelo, se realizan las siguientes operaciones a cada vértice ($p = (x_p, y_p, z_p)$):

$$p' = q \cdot \begin{pmatrix} x_p - x_c \\ y_p - y_c \\ z_p - z_c \\ 0 \end{pmatrix} = (-v \cdot (p - c), w \cdot (p - c) + v \times (p - c)) \tag{7.13}$$

$$p'' = p' + c \tag{7.14}$$

La posición del vértice pasa a ser p'' y se actualiza el `MeshBuffer` correspondiente.

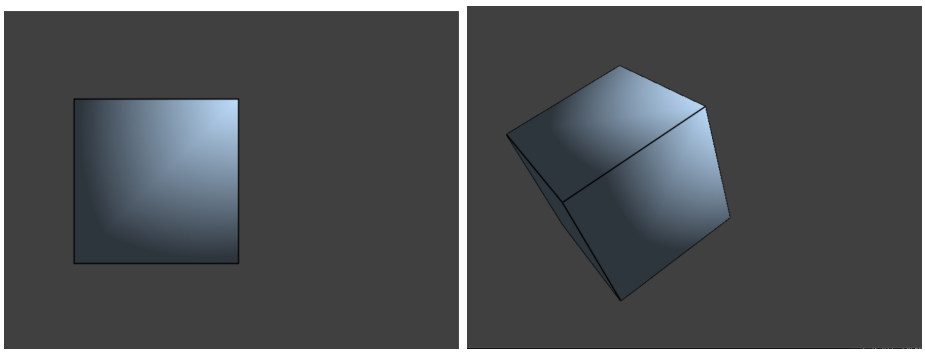

Figura 7.4: Situación inicial y final tras rotar el cubo.

7.4. Creación de vértices

Para añadir un vértice a una cara existente en el modo de creación de vértices, se usa el punto de intersección (*N*) guardado tras la invocación de `select()` y el triángulo *ABC* de la cara en el que se ubica.

Se crean los triángulos *ABN*, *BCN* y *CAN*, se elimina el triángulo *ABC* y se insertan los triángulos creados.

Tras ello se actualiza el `MeshBuffer` correspondiente y se recalculan las normales de los vértices para la iluminación.

7.5. Extrusión de una cara

La extrusión de una cara (`extrudeFace()`) consiste en la copia de la cara seleccionada y creación de las caras laterales que unen la copia con los vértices de la original. El resultado seria un "prisma" tridimensional con la cara seleccionada como base.

Primero se consigue el vector de movimiento mediante `movementGizmo()`y se revisa la variable `createdVertexExtruded`, si es falsa entonces:

Se guardan los vértices de la caras en `uniqueOldVertexIndices()` y se crean los nuevos vértices trasladados con el vector de movimiento. Se asocian los vértices viejos con los nuevos mediante el mapa `oldToNewVertexMap`, el cual es una variable global en el programa.

Se usa la función `perimeterIndex()` para conseguir todas las aristas exteriores (v_0^{old}, v_1^{old}) de la cara original. Estas se consiguen comprobando si su número de apariciones en los triángulos de una misma cara es mayor que uno.

Se tiene que (v_0^{new}, v_1^{new}) son los vértices asociados a cada (v_0^{old}, v_1^{old}) mediante el mapa `oldToNewVertexMap`).

Las caras laterales del "prisma" resultado se crean con los triángulos $[v_0^{old}, v_1^{old}, v_1^{new}]$ y $[v_0^{old}, v_1^{new}, v_0^{new}]$ para cada arista.

El mapa `oldToNewVertexMap` también se utiliza para modificar los índices de los triángulos de la cara original para que se usen los índices que representan los vértices creados y, así, formar la nueva cara. Finalmente, la variable `createdVertexExtruded` pasa a ser `True`.

En el caso de que `createdVertexExtruded` sea cierta al inicio de `extrudeFace()`, simplemente se trasladan, con el vector de movimiento del Gizmo, los vértices nuevos guardados en el mapa global `oldToNewVertexMap`.

La variable `createdVertexExtruded` y el mapa `oldToNewVertexMap` se reinician al cambiar de modo o seleccionar una cara distinta.

Si la extrusión de la cara se realiza hacia el interior del volumen del modelo se visualizará una "caja" abierta en vez de un "prisma". Se deja a responsabilidad

del usuario los posibles errores visuales producidos al atravesar todo el interior del modelo con la extrusión.

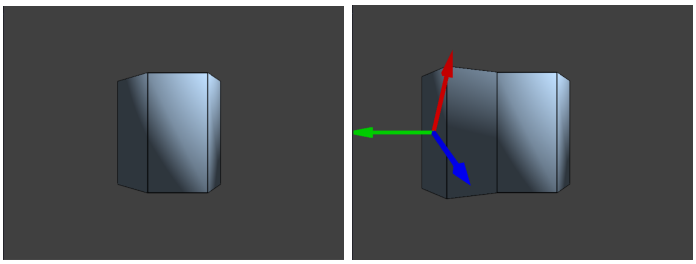

Figura 7.5: Situación inicial y final tras realizar la extrusión de una cara del prisma.

Capítulo 8

Interfaz Gráfica de Usuario

La interfaz gráfica de la aplicación de modelado se ha implementado utilizando la biblioteca ImGui (Dear ImGui) [16], un sistema de interfaz gráfica inmediata diseñado específicamente para aplicaciones gráficas. La integración de ImGui proporciona una interfaz flexible y altamente personalizable que permite la interacción con las funcionalidades del editor sin interferir con la visualización 3D principal.

La implementación sigue una arquitectura de doble buffer donde la interfaz ImGui se renderiza sobre el contexto OpenGL existente. Se utilizan los backends específicos para GLUT (`imgui_impl_glut`) y OpenGL 3 (`imgui_impl_opengl3`), lo que permite una integración transparente con el bucle principal de GLUT. La inicialización se realiza en la función `main()`, donde se configuran los contextos de ImGui y se modifican los callbacks necesarios para la gestión aislada de entrada de ratón y teclado mediante `io = ImGui::GetIO()` y `io.WantCaptureMouse`. El sistema de docking está habilitado mediante `ImGuiConfigFlags_DockingEnable`, permitiendo la organización flexible de ventanas dentro del espacio de trabajo.

El núcleo de la interfaz reside en la función `createImGUIWindow()`, que construye el panel de control principal. Este panel se organiza en secciones temáticas mediante separadores y grupos visuales. La interfaz incluye:

- **Creación de modelos básicos**: Controles para generar primitivas 3D (cubo, prisma, pirámide) con el número configurable de lados de la base. Cada botón de creación activa automáticamente el modo de traslación y selecciona el nuevo modelo.
- **Gestión de selección**: Visualización del estado actual (modelo, vértice o cara seleccionados) con información detallada como índices, cantidades y coordenadas. Incluye un botón para eliminar el modelo seleccionado con limpieza completa de recursos.
- **Control de modos de edición**: Botones especializados que gestionan los diferentes modos de operación (crear vértice, rotar, trasladar, escalar, extruir). Estos botones emplean colores diferenciados para indicar el modo activo, proporcionando feedback visual inmediato al usuario. Al presionar

un botón ya activo, se desactiva el modo y se restablece la selección. La función `drawModeButton()` se encarga de todo ello.

- **Información de cámara**: Visualización en tiempo real de los vectores direccionales de la cámara (*front, right, up*), útil para depuración y comprensión de la orientación del *viewport*.

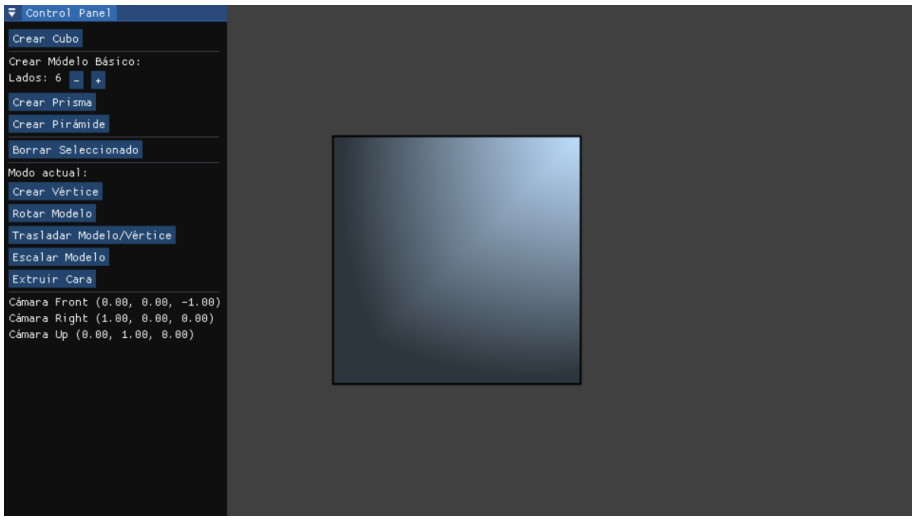

Figura 8.1: La interfaz de la aplicación de modelado.

La renderización de ImGui se realiza en cada frame dentro de `renderSceneCB()`, después de dibujar la escena 3D pero antes del *swap* de buffers. Esto garantiza que la interfaz se superponga correctamente sobre el contenido 3D. Se utiliza el sistema de renderizado de ImGui (`ImGui_ImplOpenGL3_RenderDrawData`) que genera comandos OpenGL optimizados, minimizando el impacto en el rendimiento.

Capítulo 9

Conclusiones, limitaciones y trabajo futuro

Se ha implementado exitosamente una aplicación de modelado 3D interactivo con las siguientes capacidades demostradas:

- Renderizado en tiempo real de geometría poligonal mediante un *pipeline* gráfico y con iluminación Blinn-Phong.
- Visualización mediante *shaders* GLSL de superficies sólidas y bordes de contorno superpuestos.
- Sistema de cámara interactiva con navegación orbital y movimiento libre mediante WASD.
- Mecánica de interacción intuitiva que traduce el movimiento bidimensional del ratón en operaciones tridimensionales significativas mediante un Gizmo de transformación o el uso de cuaterniones.
- Conjunto de transformaciones básicas incluyendo traslación, escalado, implementadas.
- Herramientas de edición geométrica como extrusión de caras con generación de geometría lateral, inserción de vértices en posiciones arbitrarias de las caras, y creación de modelos predefinidos como cubos, prismas y pirámides.

Sin embargo, la aplicación actual tiene severas limitaciones como la inexistencia de un sistema de guardado y cargado de archivos (imposibilita el trabajo en sesiones extendidas) o las funcionalidades de rehacer y deshacer. Además, faltaría por implementar una mayor cantidad de herramientas para que el usuario se pudiera expresar con comodidad.

Las limitaciones funcionales resultantes representan no tanto deficiencias del desarrollo, sino decisiones conscientes de enfoque que priorizaron la comprensión conceptual.

Desde la concepción inicial de este proyecto, se mantuvo un entendimiento realista sobre el alcance elemental que tendría el resultado final en comparación con soluciones profesionales establecidas. El objetivo principal nunca fue desarrollar un competidor directo para aplicaciones como Blender o ZBrush, sino más bien realizar una investigación profunda sobre los conceptos matemáticos, algoritmos computacionales y arquitecturas de software que hacen posible tales aplicaciones de modelado 3D. Este objetivo investigativo se ha cumplido satisfactoriamente.

Considerando el panorama actual del software de modelado 3D, la viabilidad comercial de continuar el desarrollo de esta aplicación resulta cuestionable. La competencia de aplicaciones ya establecidas como Blender (gratuita y de código abierto) o ZBrush (especializada en escultura digital), junto con su enorme cantidad de funcionalidades y amplia adopción en la industria, representan barreras significativas para cualquier intento de posicionamiento en el mercado.

Aun así, este desarrollo ha sido una experiencia muy educativa y se han adquirido conocimientos técnicos muy valiosos. Todo el código resultado se puede ver en `https://github.com/jd-144/Editor-Elemental-de-Modelado`.

Bibliografía

[1] E. Lengyel, *Mathematics for 3D Game Programming and Computer Graphics*, 3.ª ed. Boston, MA, USA: Cengage Learning, 2012, ISBN: 978-1-4354-5886-4.

[2] T. Akenine-Möller, E. Haines y N. Hoffman, *Real-Time Rendering*, 4.ª ed. Taylor & Francis Group, 2024, ISBN: 978-1-138-6270-0. visitado 2 de nov. de 2025. dirección: http://mutantstargoat.com/~nuclear/tmp/realtime_rendering_4ed.pdf

[3] G.-T. Creation. «OpenGL Mathematics (GLM) 0.9.9 API Documentation», visitado 2 de nov. de 2025. dirección: https://glm.g-truc.net/0.9.9/api/index.html

[4] J. de Vries. «Camera – Getting started», visitado 2 de nov. de 2025. dirección: https://learnopengl.com/Getting-started/Camera

[5] F. Dunn e I. Parberry, *3D Math Primer for Graphics and Game Development*, 2.ª ed. A K Peters/CRC Press, 2011, ISBN: 978-1568817231. visitado 2 de nov. de 2025. dirección: https://gamemath.com

[6] G. Sellers, J. Richard S. Wright y N. Haemel, *OpenGL SuperBible: Comprehensive Tutorial and Reference*, 7.ª ed. Addison-Wesley Professional, 2016, ISBN: 978-0-672-33747-5.

[7] J. de Vries. «Hello Triangle», visitado 5 de dic. de 2025. dirección: https://learnopengl.com/Getting-started/Hello-Triangle

[8] P. W. Olszta, A. Umbach y S. Baker. «FreeGLUT: The Free OpenGL Utility Toolkit». J. F. Fay, J. Tsiombikas y D. C. Niehorster, eds., visitado 5 de dic. de 2025. dirección: https://freeglut.sourceforge.net

[9] M. Ikits y M. Magallon. «GLEW: The OpenGL Extension Wrangler Library». N. Stewart, ed., visitado 5 de dic. de 2025. dirección: https://glew.sourceforge.net

[10] J. F. Blinn, «Models of light reflection for computer synthesized pictures», en *Seminal Graphics: Pioneering Efforts That Shaped the Field, Volume 1*. New York, NY, USA: Association for Computing Machinery, 1998, págs. 103-109, ISBN: 158113052X. dirección: https://doi.org/10.1145/280811.280981

[11] J. de Vries. «Basic Lighting», visitado 5 de dic. de 2025. dirección: https://learnopengl.com/Lighting/Basic-Lighting

[12] T. Möller y B. Trumbore, «Fast, Minimum Storage Ray-Triangle Intersection», *Journal of Graphics Tools*, vol. 2, n.º 1, págs. 21-28, 1997. DOI: 10.1080/10867651.1997.10487468

[13] Wikipedia. «Möller–Trumbore intersection algorithm», visitado 5 de dic. de 2025. dirección: https://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm

[14] J. Vince, *Quaternions for Computer Graphics*. Springer-Verlag London Limited, 2011, cap. 6,7, ISBN: 978-1-4471-7508-7. DOI: 10.1007/978-1-4471-7509-4

[15] icaven. «OpenGL Mathematics (GLM)», visitado 5 de dic. de 2025. dirección: https://github.com/icaven/glm

[16] O. Cornut y contributors. «Dear ImGui: Bloat-free Graphical User Interface for C++», visitado 5 de dic. de 2025. dirección: https://github.com/ocornut/imgui

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Tue Jan 13 15:01:43 CET 2026
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)