TRABAJOS FIN DE GRADO

Universidad de Jaén

Escuela Politécnica
Superior de Jaén

DISEÑO E IMPLEMENTACIÓN DE UN EDITOR DE SÓLIDOS HETEROGÉNEOS

Autor: Alberto Elorza Rubio

Grado: Ingeniería Informática

Tutores: Ángel Luis García Fernández
Francisco de Asís Conde Rodríguez

Departamento de Informática

Fecha: 16/04/2024

(Página intencionalmente en blanco)

Universidad de Jaén

Departamento de Informática

Don Ángel Luis García Fernández y Don Francisco de Asís Conde Rodríguez, tutores del Trabajo Fin de Grado titulado: '**Diseño e implementación de un editor de sólidos heterogéneos**', que presenta Don Alberto Elorza Rubio, otorgan el visto bueno para su entrega y defensa en la Escuela Politécnica Superior de Jaén.

Jaén, junio de 2024

El alumno:

Firmado por ELORZA RUBIO, ALBERTO (FIRMA) el día 06/06/2024 con un certificado emitido por AC DNIE 005

Alberto Elorza Rubio

Los tutores:

Firmado por GARCIA FERNANDEZ ANGEL LUIS - ***3879** el día 06/06/2024 con un certificado emitido por AC FNMT Usuarios

Ángel Luis García Fernández

CONDE RODRIGUEZ FRANCISCO DE ASIS - 24226298F

Firmado digitalmente por CONDE RODRIGUEZ FRANCISCO DE ASIS - 24226298F
Fecha: 2024.06.06 09:09:41 +02'00'

Francisco de Asís Conde Rodríguez

(Página intencionalmente en blanco)

Agradecimientos

Quiero agradecer a mi familia y amigos todo el apoyo que me han dado así como a mis tutores del TFG por toda la ayuda que me han brindado a lo largo del trabajo, además del aprendizaje obtenido en el transcurso de este.

FICHA DEL TRABAJO FIN DE TÍTULO	
Titulación	Grado en Ingeniería Informática
Modalidad	Proyecto de Ingeniería
Especialidad (solo TFG)	Tecnologías de la Información
Mención (solo TFG)	Sin mención
Idioma	Español
Tipo	Específico
TFT en equipo	No
Autor/a	Alberto Elorza Rubio
Fecha de asignación	16/04/2024
Descripción corta	El trabajo consistirá en el análisis, diseño e implementación de una aplicación de escritorio multiplataforma (al menos Windows/Linux) para la creación y edición de sólidos heterogéneos.

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

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

Contenido

Índice de ilustraciones

Índice de tablas

1 ESPECIFICACIÓN DEL TRABAJO

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

1.1 Introducción

El Software a desarrollar consiste en un entorno de trabajo o **framework** para la edición de modelos 3D de sólidos heterogéneos. Se trata de un software orientado tanto a la edición de los modelos para su uso, manipulación y exportación, así como a la experimentación con nuevos materiales, propiedades y asistencia por parte del software a la edición del usuario.

El motivo de su desarrollo es proporcionar una herramienta para la manipulación de este tipo de objetos que sea **multiplataforma**, la cual sea lo suficientemente versátil para soportar modificaciones futuras y así facilitar el tratamiento y la experimentación con los sólidos heterogéneos de manera cómoda y funcional.

1.2 Objetivos del trabajo

A continuación, se enumeran los objetivos del trabajo:

- Desarrollar una aplicación multiplataforma, incluyendo al menos Windows y Linux.
- Diseñar una arquitectura de software robusta pero flexible que permita su extensión y modificación en un futuro.
- Implementar un entorno gráfico para la manipulación de sólidos heterogéneos, lo cual comprende la escena donde se visualizan los cambios y las diferentes ventanas donde se puedan manipular las diferentes propiedades de los elementos en escena.

- Diseñar un proceso de carga y guardado de las escenas al completo para poder guardar y recuperar los cambios realizados.
- Permitir un control a alto nivel sobre el proceso de renderizado, así como de las propiedades que lo condicionan.
- Diseñar un modelo de sólidos heterogéneos desacoplado de la interfaz y fácil de editar, extender y refactorizar atendiendo a futuras modificaciones sobre el modelo utilizado.

1.3 Antecedentes y estado del arte

Los objetos heterogéneos están compuestos de varios materiales cuyas proporciones a lo largo del solido son variables, beneficiándose por tanto de las mejores características gracias a la mezcla de los materiales que los conforman.

La mezcla de los materiales que componen el sólido heterogéneo se conoce como material heterogéneo o funcionalmente graduado (*functionally graded material*). Los materiales heterogéneos se usan en muchos tipos de objetos reales fabricados, por ejemplo, implantes médicos, partes de motores o neumáticos de competición.

El modelo en particular que usaremos está basado en un hiperparche de Bézier, siendo el hiperparche una función continua de 3 parámetros que define las coordenadas del conjunto de puntos que comprenden tanto el exterior como el interior del sólido. La forma del objeto estará definida por una serie de coeficientes geométricos también llamados puntos de control.

1.3.1 Estructura de los sólidos heterogéneos

Los sólidos heterogéneos están conformados por un conjunto de celdas, las cuales comparten coeficientes, los encargados de manipular su geometría. Cada una de estas celdas puede tener a su vez subceldas, las cuales son modificadas por operaciones de mecanizado, por ejemplo, dividir una subcelda en 2 respecto de un eje, obteniendo así 2 subceldas.

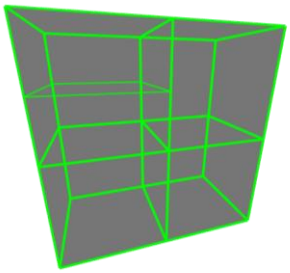


Ilustración 1.1 – Captura de la selección de un sólido heterogeneo

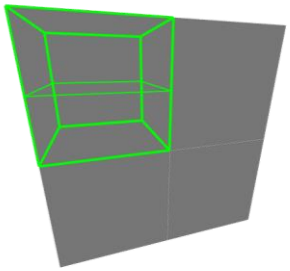


Ilustración 1.2 – Captura de la selección de una celda de un sólido heterogeneo

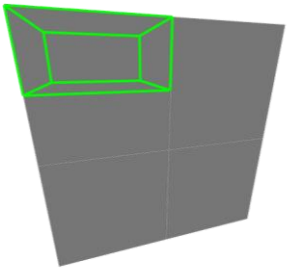


Ilustración 1.3 – Captura de la selección de una subcelda de un sólido heterogeneo

De esta forma es cómo se obtiene el **árbol de edición** con el cual se manipulará el sólido heterogéneo. Un primer nivel que define las propiedades generales del sólido (Ilustración 1.1), bajo el cual se agrupa un segundo nivel con las celdas que delimitan la geometría formada por sus coeficientes (Ilustración 1.2), y debajo de estas un nivel variable de subceldas generadas por operaciones de mecanizado (Ilustración 1.3). En caso de no aplicar ninguna operación tendríamos una subcelda que ocuparía la totalidad del espacio definido por los coeficientes en la celda padre.

1.3.2 Coeficientes

Cada una de las celdas está formada por un total de **64** coeficientes para modelar la geometría de la celda, los cuales pueden estar compartidos entre celdas

vecinas, dando lugar a la **topología** del sólido. Las subceldas a su vez cuentan con sus propios 64 coeficientes, aunque mantienen una serie de diferencias:

- A diferencia de las celdas, estos coeficientes surgen de realizar **operaciones de manufacturado**. Es decir, son el resultado del cálculo de una operación en la subcelda en lugar de una transformación individual sobre cada uno de los coeficientes, tal y como ocurre en las celdas de primer nivel.
- Los coeficientes de las subceldas no contienen **información topológica** del entorno de subceldas vecinas, mientras que los coeficientes con los que trabajan las celdas necesitan de la información topológica para mantener la **continuidad** entre las mismas.
- Únicamente las celdas trabajan con propiedades que afecten directamente a la **distribución del material**. De este modo, los coeficientes de las subceldas solo mantendrán geometría y toda la distribución del material lo determinará la celda y los coeficientes de distribución de material.

Cada uno de los coeficientes de las celdas tiene asignado un coeficiente **a0** comprendido en el rango **[0, 1]** el cual indica la distribución del material.

En cuanto a la topología, existe una jerarquía de coeficientes donde los más importantes son los **corner points**, los cuales se corresponden a las esquinas de la celda. Los corner points son importantes en una serie de operaciones:

- Mantener la continuidad geométrica: los vecinos directos de los corner points además, de sus propias transformaciones, deberán de aplicar las transformaciones de los corner points de los que son vecinos, de manera que no se rompa la **continuidad**.
- Mantener la continuidad del material: los vecinos directos de un corner point comparten la misma proporción de material, a menos que el diseñador decida saltársela y asignarle su propio valor.

1.3.3 Materiales en los sólidos heterogéneos

Los sólidos heterogéneos han sido probados con varios tipos de materiales hasta el momento:

- **Simples**: Son los más sencillos de aplicar, dado que **se utiliza el material primario o secundario** para renderizar el modelo, sin usar ningún tipo de cálculo heterogéneo mediante los coeficientes.
- **Continuos**: Dentro de los materiales heterogéneos es el más sencillo de entender, dado que mezclará ambos materiales según la proporción que haya de cada uno en un punto. Este tipo de materiales **no puede ocurrir en la realidad**, dado que por muy heterogénea que sea la muestra de ambos materiales en un punto solo habrá o bien uno u otro. Igualmente sirve para ver los resultados teóricos que tendría una combinación perfecta de dos materiales.
- **Funcionalmente graduados**: En este caso se busca un resultado más natural que en el anterior, dado que los se combinan, pero no se mezclan, es decir, en un punto solamente tendremos un material. Para obtenerlos se utiliza una función de ruido de **Perlin** ([1]) que determina en que proporción aparecen los materiales y por tanto cual se debe elegir [2]. La aplicación de estos depende de cada celda dado que van parametrizados por el **perlin period** de la celda.
- **Compuestos**: Al igual que en el caso anterior este tipo de materiales determinará cual aplicar según la proporción de uno u otro. En este caso la proporción la calculará por medio de una serie de **puntos de control y cuatro coeficientes**. De cada punto a determinar el material, se buscarán los cuatro puntos de control más cercanos y se les aplicarán los coeficientes.

1.4 Descripción de la situación de partida

La situación de partida consiste en una aplicación creada para la edición de sólidos heterogéneos con el modelo que vamos a utilizar, así como los diversos artículos publicados sobre el modelo en revistas científicas ([3], [2], [4], [5]).

El software de partida (Ilustración 1.4) consiste en un programa escrito en Swift junto con Metal para la programación de shaders. Este software permite visualizar y editar el sólido, realizar capturas con resoluciones personalizadas, así como guardar

y recuperar los modelos utilizando el formato JSON. A continuación, se describirán en mayor detalle las funcionalidades mencionadas.

La visualización permite filtrar si se está mostrando solamente la malla formada por las isocurvas frontera de las celdas o el sólido completo tal y como se vería con los diferentes materiales incluyendo: material compuesto, material funcionalmente graduado, material continuo o solamente un material uniforme.

La edición permite mover los puntos de control modificando la geometría, así como controlar la presencia de un material en torno a un punto de control. Por otro lado, también se pueden realizar operaciones de mecanizado.

Además de la edición, visualización y guardado de sólidos, el software permite el control de la cámara que muestra la escena, las luces que interactúan con los sólidos y el modo en que se visualizan los objetos.

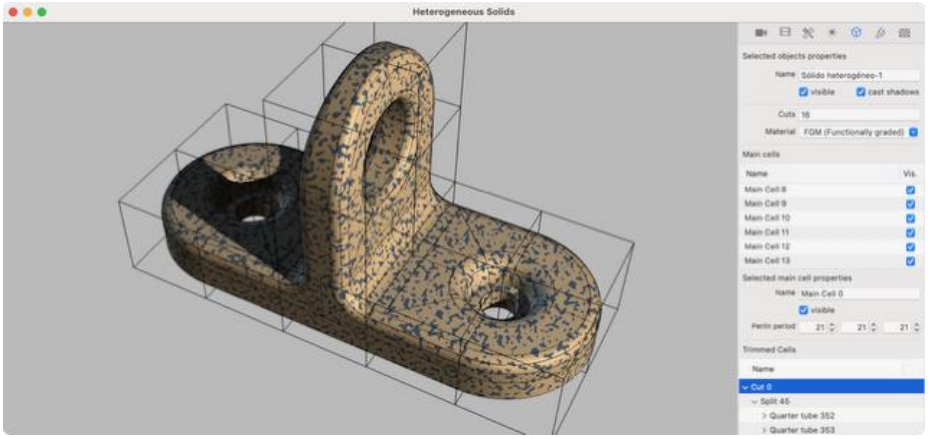


Ilustración 1.4 – Software de partida

1.4.1 Descripción del entorno actual

El software anterior cumple el propósito de permitir la visualización de los sólidos para poder generar imágenes para utilizarlas en documentos y artículos. Se espera que con este nuevo software el usuario tenga la suficiente flexibilidad para mantener las funcionalidades de la situación de partida, así como para extender las posibilidades de los sólidos heterogéneos mediante futuras actualizaciones de la aplicación.

1.4.2 Resumen de las deficiencias y carencias identificadas

El software anterior presenta una deficiencia fundamental relacionada con su portabilidad a plataformas que no sean macOS y varias carencias a la hora de la edición de los sólidos.

Al estar desarrollado en **Swift** para macOS, su portabilidad es limitada y por tanto su uso en otras plataformas como Linux y Windows. Por tanto, la tecnología utilizada para resolver esto deberá permitir compilar el software para la mayor cantidad de sistemas posibles.

El enfoque de dicho software en cuanto a la edición de los sólidos no estaba pulido de cara al usuario final, dando lugar a que los patrones de edición tuviesen que ser repetidos entre modelos y piezas (**no contaba con sistema de macros**), además de presentar un árbol de edición poco dinámico.

Por otro lado, el programa en cuestión no permite la edición simultánea de múltiples sólidos. Por tanto, no es posible trabajar en dos sólidos ni visualizar uno mientras se edita otro.

1.5 Requisitos iniciales

- El sistema debe ser capaz de visualizar un modelo de sólido heterogéneo.
- El sistema debe ser capaz de editar la estructura de los sólidos heterogéneos.
- El sistema debe ser capaz de editar los materiales asignados a sólidos heterogéneos.
- El sistema debe ser capaz de almacenar en disco los modelos de sólidos heterogéneos.
- El sistema debe ser capaz de recuperar del disco los modelos de sólidos heterogéneos.
- El sistema debe ser capaz de modificar las opciones gráficas de renderización.

- El sistema debería ser capaz de almacenar y recuperar los materiales conjuntamente con los modelos de sólidos heterogéneos.
- El sistema debería ser capaz de soportar extensiones en cuanto a sus funcionalidades de edición y guardado de sólidos, por ejemplo: añadir fácilmente un nuevo tipo de nodo en el árbol o modificar las propiedades de un objeto sin afectar a la estructura del proyecto.
- El sistema debería de ser capaz de soportar la edición de múltiples escenas simultáneamente.
- El sistema debería permitir deshacer y rehacer los cambios realizados sobre los modelos y sus materiales.

1.6 Alcance

El trabajo constará al terminar de:

- La **aplicación final**, incluyendo su **código fuente**, una versión **compilada** y su **carpeta de recursos** donde se encuentran los shaders utilizados y la fuente tipográfica de la aplicación.
- **Documentación** la cual contiene tanto la explicación de qué y cómo se ha desarrollado, así como el manual que explica su funcionamiento, cómo utilizarlo y cómo modificarlo.
- Un **vídeo** que muestre la aplicación final en funcionamiento.

1.7 Hipótesis y restricciones

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

1.8 Estudio de alternativas y viabilidad

Para el desarrollo del editor de solidos heterogéneos se han valorado diferentes bibliotecas gráficas para trabajar con **C++** permitiendo un **entorno de edición 3D junto con menús y ventanas gráficas**:

- **QT.** Es una solución ampliamente utilizada y muy robusta además de multiplataforma y extensa documentación. A pesar de todas estas ventajas se ha descartado dado que no se van a aprovechar todas sus características e introduciría una gran sobrecarga en el sistema tanto de recursos como de complejidad en la implementación.
- **wxWidgets.** Esta solución permite, a diferencia de **QT**, el uso de componentes nativos del sistema, y su integración es algo más liviana, pero en nuestro caso produce los mismos inconvenientes de sobrecarga y complejidad no deseada, requiriendo para nuestro sistema una solución más liviana.
- **Dear ImGui.** Es una biblioteca muy liviana con dependencias mínimas diseñada para crear interfaces gráficas de modo inmediato, es decir, renderiza en base a un estado toda la interfaz para cada fotograma, siendo ampliamente utilizada para aplicaciones que requieran actualizaciones muy rápidas y dinámicas de la interfaz, como la nuestra. Además, el enfoque de **Dear ImGui** basado en ventanas nos será muy ventajoso a la hora de organizar la interfaz y permitir al usuario configurarla a sus necesidades durante el uso de la aplicación.

Hasta el momento hemos identificado dos contras de Dear ImGUI. El primero es el uso de **diálogos del sistema**, el cual no lo trabaja dado que solo se encarga de renderizar sobre OpenGL, siendo la solución elegida, recaer sobre otra biblioteca más pequeña que resuelva este problema, la cual se comentará en el apartado 1.10. El otro problema tiene que ver con la creación de **atajos de teclado**, que hasta la fecha se encuentra en desarrollo. Por tanto, si quisiéramos hacer uso de esto en Dear ImGUI, tendríamos que programarlo nosotros mismos más adelante o esperar a que esta característica sea lanzada en una nueva versión.

1.9 Descripción de la solución propuesta

La solución para el editor de sólidos heterogéneos será una aplicación de escritorio desarrollada en **C++**, la cual realizará el rendering mediante la biblioteca OpenGL y dibujará las ventanas de la interfaz de usuario por medio de la biblioteca Dear ImGUI.

Esta solución permite crear una aplicación que pueda ser compilada en los principales sistemas operativos, además de estar basada en Open Source y contar con una sobrecarga mínima por parte del lenguaje de programación y del entorno utilizado, facilitando el desarrollo de una aplicación bien optimizada, rápida y versátil, debido al control sobre el código fuente y el uso de abstracciones mínimas por parte de estas tecnologías.

1.10 Tecnologías utilizadas

El sistema se desarrollará en C++ versión 17, dado que es un lenguaje que nos permite realizar aplicaciones multiplataforma a bajo nivel y con muy buen rendimiento. Las bibliotecas principales sobre las que se apoyará serán **OpenGL** para la comunicación con la tarjeta gráfica y todas las operaciones de renderizado, **GLFW** para la abstracción de la interfaz del sistema operativo subyacente y así poder crear ventanas, y por último **Dear ImGUI** para el dibujado de la interfaz de usuario de forma agnóstica al sistema operativo.

La comunicación con la tarjeta gráfica se realizará por medio del leguaje de shaders GLSL, el cual viene integrado en OpenGL.

Para funcionalidades secundarias nos apoyaremos en otras bibliotecas más pequeñas:

- **GLM**: biblioteca matemática con uso transversal a todo el proyecto.
- **LodePNG**: para poder almacenar imágenes en el sistema.
- **Nlohmann_JSON**: para poder manejar el formato JSON de forma cómoda.
- **SPDLOG**: para poder comunicar y gestionar mensajes de ejecución y error desde cualquier punto del programa

- **ImGuizmo**: para renderizar utilidades 3D sobre Dear ImGUI
- **portable-file-dialogs**: biblioteca multiplataforma de diálogos para poder realizar operaciones de carga y guardado con los diálogos nativos independientemente del sistema operativo.

1.11 Metodología de desarrollo de software

Se utilizará un desarrollo en **cascada**, dado que se seguirán una serie de etapas donde cada una de ellas deberá esperar a la finalización de la anterior. Las fases serán las siguientes: **Análisis**, **Diseño**, **Implementación** y **Pruebas**.

1.12 Análisis de riesgos

Los riesgos que hay que afrontar vienen dados por la variación en los entornos donde se despliegue la aplicación, es decir, la desactualización de las diferentes bibliotecas en las cuales será desarrollado el framework. Entre las bibliotecas más dependientes se encuentra OpenGL, el cual está marcado como obsoleto en macOS. Dado que en un futuro podría ser eliminado, la aplicación no podría utilizarse en este sistema operativo.

Teniendo en cuenta que el diseño que se hará de los módulos de la aplicación será bastante desacoplado entre ellos, surgirían varios casos posibles ante la necesidad de rehacer en un futuro la aplicación:

Si la alternativa a OpenGL que se use tiene ya un adaptador para Dear ImGUI se podrían simplemente reescribir los apartados de renderizado para la nueva plataforma, manteniendo toda la lógica de modelo y las interfaces.

En caso de no contar con soporte para Dear ImGUI se sumaría la necesidad de reescribir la interfaz gráfica.

Es muy raro el caso en el que haya que desechar por completo la aplicación, dado que C++ seguiría siendo soportado en todos los sistemas operativos puesto que se compila a la arquitectura y sistema. Distinto sería un cambio de enfoque y lenguaje, manteniendo a lo sumo en este caso una traducción de la lógica del modelo de los sólidos heterogéneos.

Otra posibilidad es encontrar un error en alguna de las bibliotecas de menor uso, pero cuya funcionalidad es indispensable para cubrir todos los requisitos de la aplicación. La biblioteca *portable-file-dialogs* es la que menor interacción tiene en Github, y la que mayor probabilidad tiene de ser abandonada. En este caso surge otra serie de contingencias según el riesgo.

Si el error es solucionado, se actualizará la biblioteca en la aplicación

Si el error es conocido, será reparado localmente.

En el peor de los casos se programará una alternativa. Para que no sea difícil integrar se hará uso de una interfaz común para los diálogos. De esta forma por medio de polimorfismo simplemente habría que programar una clase nueva que solvente el problema de llamado nativo a diálogos y que responda a las llamadas realizadas por la API definida.

1.13 Estimación del tamaño y esfuerzo

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

1.14 Planificación temporal

Al trabajo se le dedicará un total de 100 días laborales (3 horas por día), sumando las 300 horas, y una duración estimada de 22 semanas. En la Ilustración 1.5 se muestra la distribución de cada una de las etapas del desarrollo en cascada mediante un diagrama de Gantt. Se ha dividido la planificación en 4 etapas: Análisis, Diseño, Implementación y Pruebas, las cuales se han subdividido a su vez en varias tareas, como se muestra en la Tabla 1.1 junto a las fechas de inicio y fin de cada una.

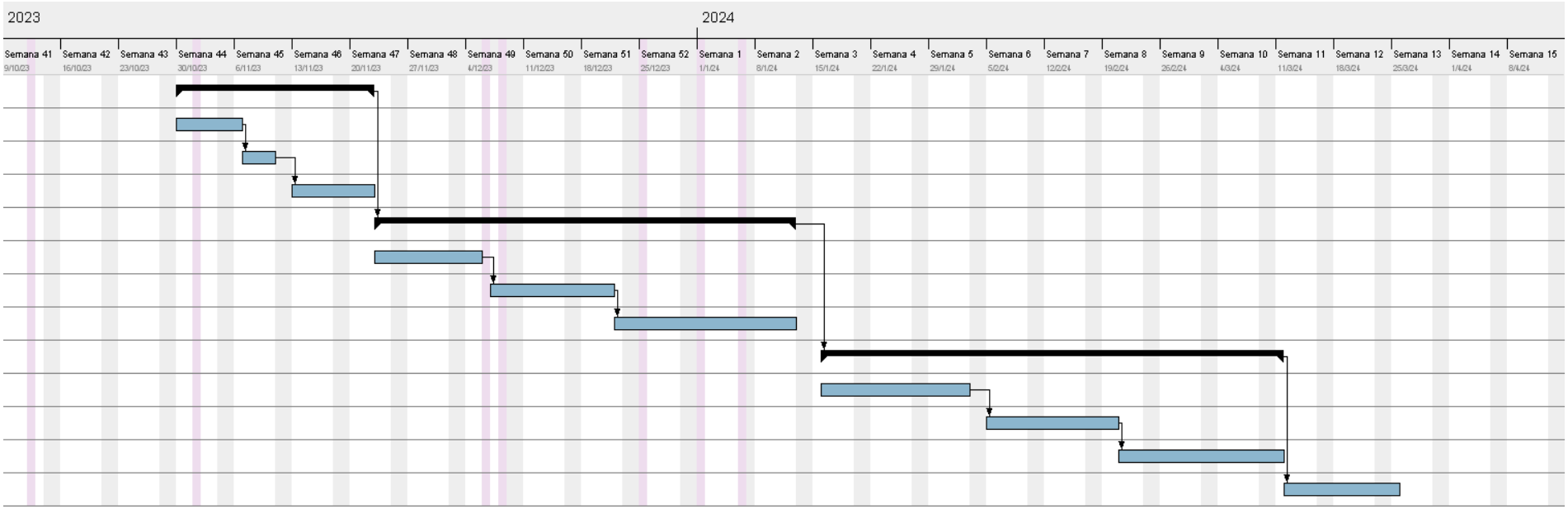


Ilustración 1.5 – Diagrama de Gantt

	Fecha Inicio	Fecha Fin
Análisis	30/10/23	22/11/23
Análisis de requisitos	30/10/23	6/11/23
Análisis de usuario	7/11/23	10/11/23
Análisis de arquitectura	13/11/23	22/11/23
Diseño	23/11/23	12/1/24
Diseño de los casos de uso	23/11/23	5/12/23
Diseño de la interfaz	7/12/23	21/12/23
Diseño de la arquitectura	22/12/23	12/1/24
Implementación	16/1/24	11/3/24
Implementación de la interfaz	16/1/24	2/2/24
Implementación de la escena	5/2/24	20/2/24
Implementación del modelo de sólidos heterogéneos	21/2/24	11/3/24
Pruebas	12/3/24	25/3/24

Tabla 1.1 – Tabla de tiempos del diagrama de Gantt

1.15 Presupuesto

Para el **presupuesto de trabajo** se ha consultado el convenio colectivo de empresas (el cual incluye las tecnologías de la información) [6]:

- Análisis y Diseño (*Área 3, Grupo B, Nivel 2*): 27.147,12 € anuales. Dado que, según el convenio, en un año se trabajan 1800 horas, el cose de la hora sería: 15,08 € y dado que esto es un coste mínimo, en nuestro caso hemos decidido subir el coste hasta: 17 €
- Implementación y Pruebas (*Área 3, Grupo C, Nivel 2*): 24.810,14 €. Dado que según el convenio son 1800 horas, la hora sería: 13,78 € y dado que esto es un mínimo, en nuestro caso nos gustaría cobrar: 15 €
- Se ha elegido el **Área 3** para ambos casos dado que es le área que incluye el desarrollo de software. Para *análisis y diseño* hemos cogido el **Grupo B** dado que tienen en cuenta las funciones de análisis, coordinación y supervisión de tareas. Mientras que, para la

implementación y pruebas el **Grupo C** realizan actividades de tipo técnico y se responsabilizan de programación y supervisión. En ambos casos se ha elegido el **Nivel 2** dado que a pesar de contar con los

- conocimientos necesarios no se tiene tanta experiencia como el **Nivel 1**.

Recursos humanos					
	Análisis	Diseño	Implementación	Pruebas	Total
Días	17	33	40	10	100
Horas	51 H	99 H	120 H	30 H	300 H
Coste unitario	17 €	17 €	15 €	15 €	
Coste total	867 €	1683 €	1800 €	450 €	**4800 €**

Tabla 1.2 – Tabla de presupuesto de recursos humanos

Para el **hardware** se contará tanto con el coste del ordenador personal como de los dispositivos de entrada-salida utilizados, ratón, teclado y monitor.

Hardware			
	Ordenador Personal	Dispositivos E/S (Teclado, Ratón, Monitor)	Total
Coste	900 €	250 €	1150 €
Amortización	5 años	5 años	5 años
Periodo de uso	22 semanas	22 semanas	22 semanas
Coste Amortizado	76 €	21 €	**97 €**

Tabla 1.3 – Tabla de presupuesto hardware

Para el **software** se considerará que los productos de pago se usaron con su versión comercial, emulando un proyecto real. El software de libre uso se considerará que tuvo coste nulo. El software de pago es: Microsoft Office y CLion.

Software				
	Microsoft Office	CLion para estudiantes	Software de libre uso	Total
Coste	99 €	119.79 €	0 €	
Amortización	1 año	1 año		1 año
Periodo de uso	22 semanas	22 semanas	22 semanas	22 semanas
Coste Amortizado	42 €	51 €	0 €	**93 €**

Tabla 1.4 – Tabla de presupuesto software

El cálculo de los precios amortizados de hardware y software lo hemos hecho dividiendo entre la amortización y multiplicando por el periodo de uso.

El presupuesto final es el siguiente:

Coste Total	
Sub Total	4990 €
Sobrecoste indirecto (10%)	499 €
Sub Total	5489 €
Margen de beneficio (15%)	823,35 €
Sub Total	6312,35 €
IVA (21%)	1325,59 €
Total	**7637,94 €**

Tabla 1.5 – Tabla de presupuesto total del proyecto

2 DISEÑO

Como ya se ha introducido anteriormente, esta aplicación requiere que se diseñe tanto un sistema de ventanas, la comunicación entre estas, y la capacidad de dibujar y renderizar dentro de estas ventanas, así como el modelo de solidos heterogéneos que será editado en este sistema. Por último, se necesitará de un mecanismo para interactuar con la entrada y salida, pudiendo así almacenar los sólidos y cargarlos más tarde.

2.1 Análisis del usuario

El software en un principio va completamente enfocado a usuarios que conocen los sólidos heterogéneos. Por tanto, en este caso no será necesario realizar ningún tipo de restricción: el sistema, por su naturaleza dividida en ventanas que interactúan en base a la selección no debería dar problemas a estos usuarios. Además, dado que la mayoría de operaciones existen en el sistema anterior, no deberían dar lugar a confusión.

Dado que los sólidos heterogéneos están implementados para resolver automáticamente tanto operaciones sobre sus elementos en el árbol como en materiales y la aplicación de operaciones de mecanizado, aquellos usuarios que no tengan un amplio conocimiento sobre el modelo de sólidos heterogéneos pero que quieran experimentar, se espera que reciban un feedback adecuado.

Para este último caso en particular, donde el usuario aprende por su cuenta, es cuando la interfaz modular tiene especial importancia, evitando que el usuario se sature con los elementos en pantalla y la cantidad de opciones posibles.

2.2 Especificaciones del sistema

En este apartado se especificarán de forma detallada los requisitos funcionales y no funcionales del sistema, acompañados por sus respectivos diagramas de casos de uso. En todos los casos de uso, el único actor que intervendrá será el usuario que quiere editar o visualizar el sólido heterogéneo.

R01 (Ilustración 2.1): El usuario debe poder crear y cerrar escenas, cargarlas desde archivo y guardar las que se encuentren abiertas, así como guardarlas de nuevo bajo otro nombre. Entre las escenas abiertas el usuario debe poder seleccionar aquella con la que quiere trabajar.

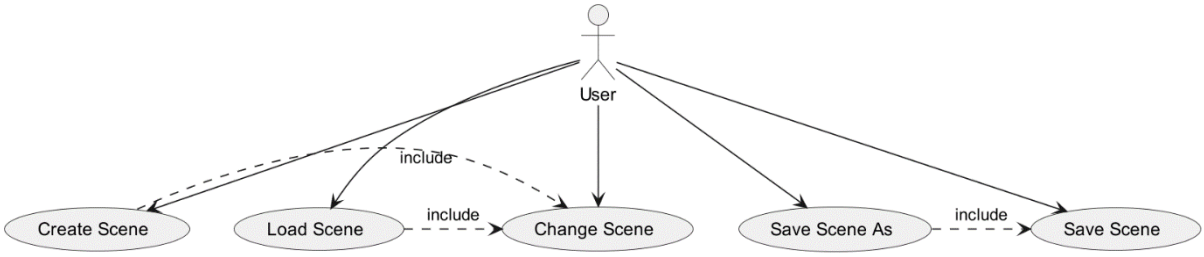
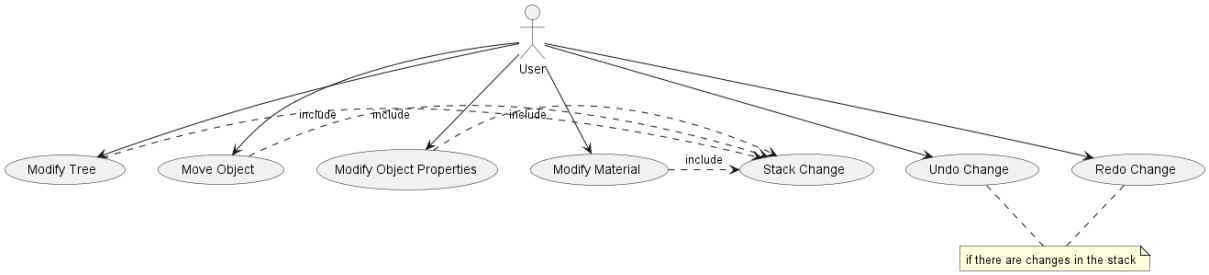


Ilustración 2.1 – Casos de uso. Trabajar con múltiples escenas

R02 (Ilustración 2.2): El usuario debe poder realizar cambios sobre el modelo y los materiales y deshacer esos cambios o rehacerlos en caso de no haber sido



reemplazados. Entre los cambios se encuentra trabajar con los nodos del árbol, mover estos nodos, modificar sus propiedades y realizar operaciones de mecanizado.

Ilustración 2.2 – Casos de uso. Edición del modelo y los materiales

R03 (Ilustración 2.3): El usuario debe poder modificar las propiedades de renderización, así como los shaders con los que se está visualizando la escena y las luces que iluminan la escena. Requisitos no funcionales: los cambios deben verse aplicados en tiempo real.

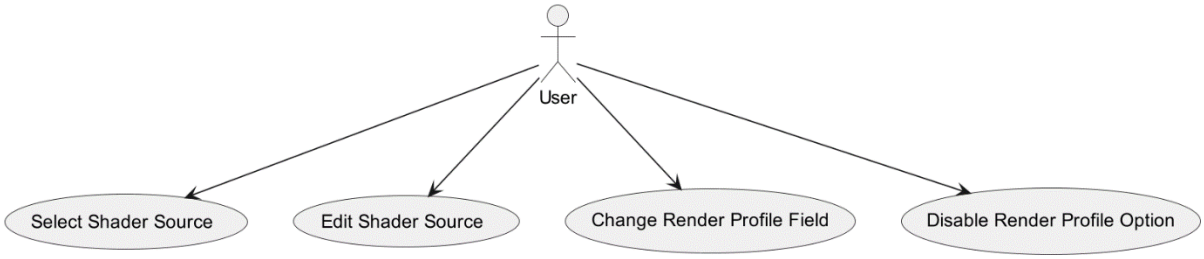


Ilustración 2.3 – Casos de uso. Modificación de las propiedades gráficas

R04 (Ilustración 2.4): El usuario debe poder navegar por la escena y seleccionar los elementos que quiera editar. Entre los movimientos que podrá hacer se encontrarán orbitar sobre un punto, movimiento Dolly en dirección al objetivo y movimiento Truck respecto al objetivo de la cámara.

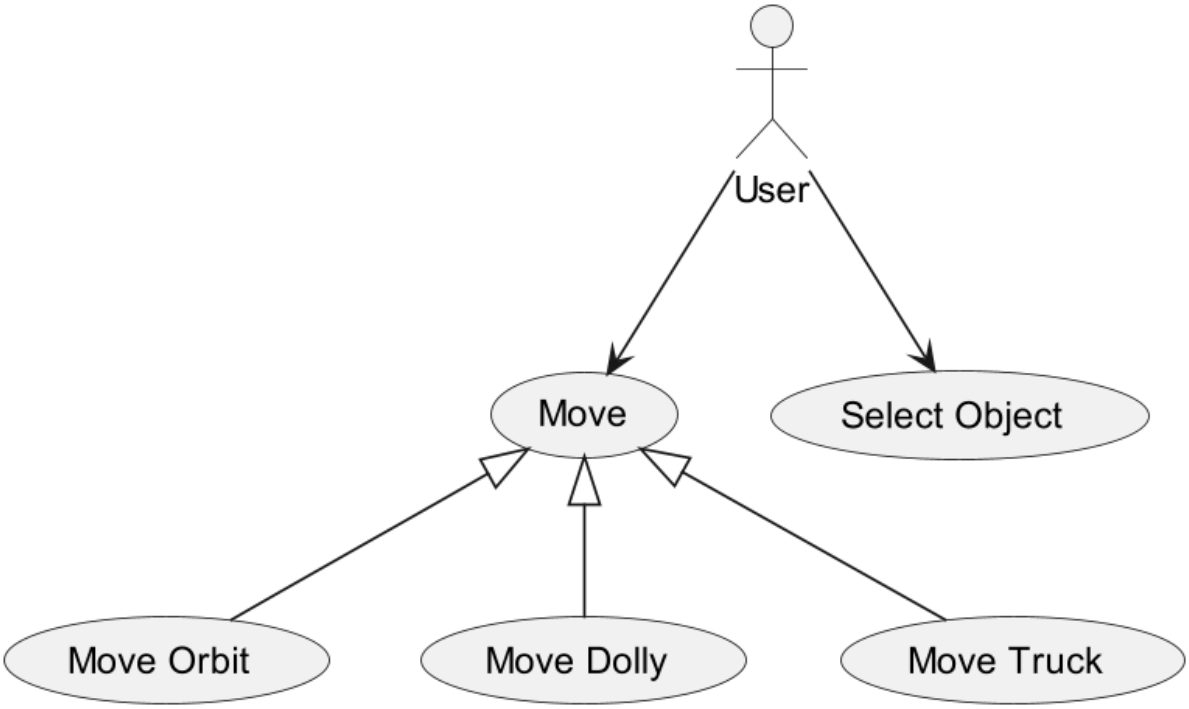


Ilustración 2.4 – Casos de uso. Navegación por la escena

2.3 Análisis y diseño del sistema

Para hacer más robusto el conjunto del software se tomarán una serie de decisiones iniciales en cuanto al diseño, de forma que se encuentre un balance entre comodidad en el desarrollo y facilidad para el mantenimiento y la extensión. A continuación, se comentarán las decisiones tomadas:

La interfaz gráfica se organizará como un elemento independiente del resto del sistema. De forma que analógica a un sistema **MVC** (Modelo Vista Controlador [7]), las vistas serían la interfaz, totalmente desacoplada del modelo y el controlador que más adelante se comentará en el diseño arquitectónico a qué equivaldrían. Por tanto, en caso de cambiar de interfaz, esta estaría totalmente desvinculada del modelo.

El formato **JSON** será usado por defecto para el guardado y será el único implementado. A pesar de esto, en un futuro podrían añadirse más. Por tanto, se creará una abstracción común para el guardado.

La interacción planteada con la escena, obviando la que las bibliotecas de terceros implementen por su cuenta, será **selección por color**. Se asociará a cada objeto un índice de color, cuando se realice la pasada de selección, de forma que al seleccionar el color (renderizado en segundo plano para que el usuario no lo vea) se notifique la selección del objeto a la aplicación.

El modelo de **sólidos heterogéneos** se tomará manteniendo las mismas relaciones entre los elementos que existían en el software original para evitar crear un modelo completamente nuevo. Dado que el entorno será diferente, todos los nodos deberán adaptarse para soportar las operaciones sobre el **árbol**, así como a las nuevas funcionalidades con las que se espera extender el modelo de sólidos heterogéneos.

2.3.1 Patrones de diseño

En este apartado se comentarán los diferentes patrones auxiliares que se utilizarán para mejorar la calidad del software y resolver de forma genérica problemas comunes a lo largo del desarrollo. Se explicará la razón de ser de cada uno de los patrones, de forma que sirva como índice para saber por qué y cuando utilizarlos, así como hacerlos fáciles de reconocer si son mencionados posteriormente.

2.3.1.1 Decorator

El patrón **decorator** [8] permite añadir funcionalidad a un objeto encapsulando su comportamiento. En nuestro caso en concreto, se implementará un patrón decorator que encapsule funciones, de forma que en tiempo de ejecución se pueda modificar el comportamiento aplicando otras funciones que llamen a las anteriores. Por ejemplo, en el proceso de renderizado necesitamos que bajo ciertas circunstancias un nodo no se dibuje, para evitar retirarlo del padre y luego incluirlo de nuevo, lo cual es un proceso muy engorroso. A través del patrón decorator, podemos decorar la función de renderizado con una que no haga nada, y así no se dibujará.

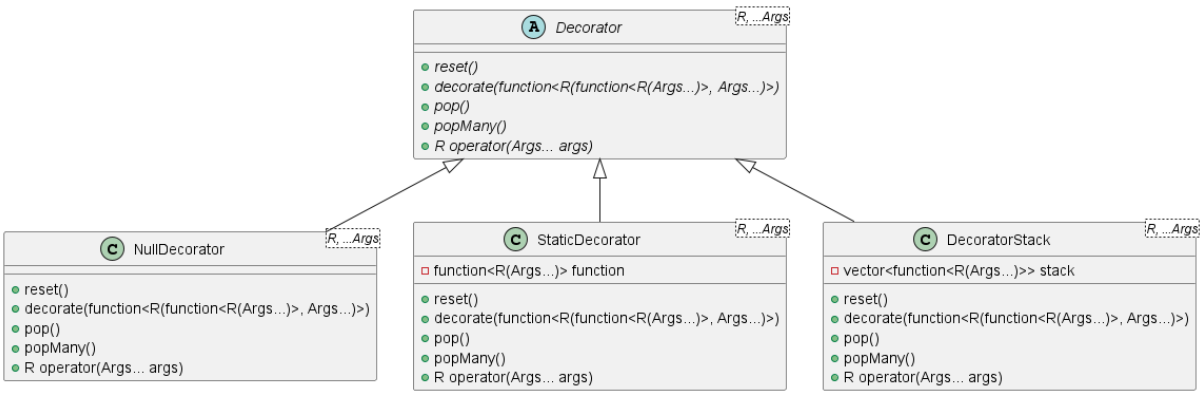


Ilustración 2.5 – Diagrama de clases. Decorator

Como se muestra en la Ilustración 2.5, para facilitar que se implemente con múltiples estructuras de datos se ha diseñado como una clase abstracta pura. Se han hecho 3 implementaciones, el **NullDecorator**, que no hace nada, el **StaticDecorator** que no aplicará decoradores, y el **DecoratorStack**, que es la implementación que mantendrá apiladas las funciones permitiendo eliminar la de más arriba y decorar con la última que se encuentre, así como eliminar las *N* primeras. Los parámetros y salida de la función decorada se indicarán a la plantilla.

2.3.1.2 Observer

El patrón **observer** [9] nos permite asignar dinámicamente elementos a un tema y notificarles de los cambios ocurridos. Por medio de este sistema resulta muy fácil conectar objetos y notificarles cuando un evento ocurre en lugar de realizar preguntas en bucle al tema o que el tema pregunte a todos los candidatos, lo cual es más lento y tedioso.

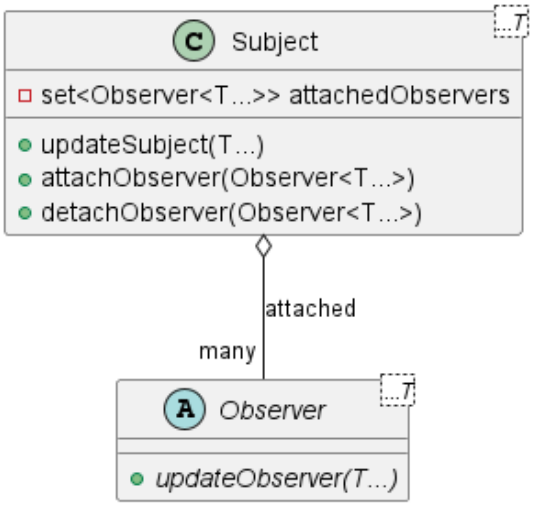


Ilustración 2.6 – Diagrama de clases. Observer

En la Ilustración 2.6 se implementa de forma que el *Subject* sea una clase concreta que emite a todos los observadores la información, mientras que el comportamiento del *Observer* será abstracto y dependerá de la implementación particular que se realice. Ambos se comunicarán por el tipo común de dato que se indique a la plantilla, *T*.

2.3.1.3 Visitor

El patrón **visitor** [10] permite adaptar un algoritmo a los diferentes tipos de clases que implementen una interfaz. Es muy útil cuando se trabaja sobre nodos y no se quiere aplicar una conversión de tipo sobre cada uno de los nodos operados, de forma que se puedan extraer los casos particulares de la lógica del algoritmo.

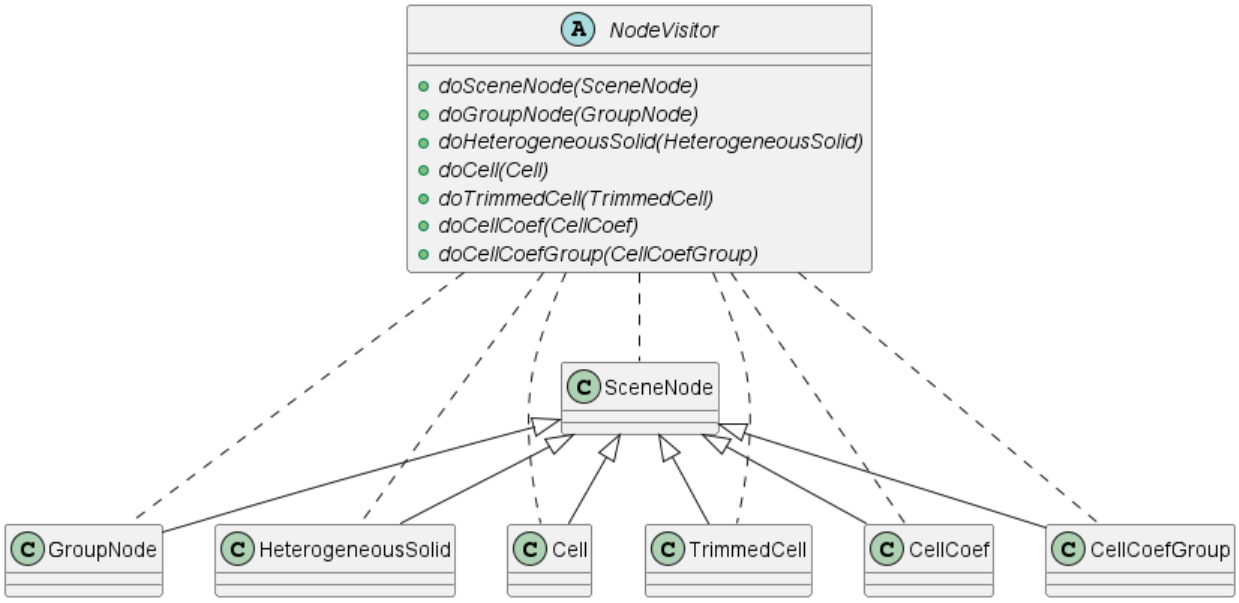


Ilustración 2.7 – Diagrama de clases. Visitor

En la Ilustración 2.7 se puede ver cómo todos los nodos con lo que se va a trabajar tienen un método correspondiente en el **NodeVisitor**. Pese a no ser el patrón más flexible, dado que requiere que cada nuevo nodo que se quiera procesar en el algoritmo se incluya como un nuevo método, se ha optado por el dado que permite fácilmente resolver el problema de trabajar con nodos específicos para ciertas operaciones, así como extraer lógica fuera de los nodos.

Algunas consideraciones en cuanto a su uso para que sea más flexible:

- Si no se va a hacer un uso especial en un nodo, como serialización o creación de ventanas personalizadas, no es necesario añadir un método para ese caso al visitor.
- Si el comportamiento del nodo es genérico, como una selección nada más, se puede llamar a **doSceneNode**, dado que el nodo implementará *SceneNode*, evitándose así añadir un nuevo método a la interfaz.

2.3.1.4 Command History

El **historial de comandos** que hemos utilizado es una mezcla entre un patrón **comando** [11] convencional y un patrón **memento** [12]. Es uno de los patrones más importantes de la aplicación dado que permite al usuario realizar cambios sobre el modelo y deshacerlos, lo cual ocurrirá en numerosas ocasiones en la edición, siendo por tanto una funcionalidad que ha de ser pulida para ser versátil. Además de encapsular la **ejecución** de comandos para hacerlos y deshacerlos, debe permitir **fusionar** aquellos comandos en caso de acontecer dos o más acciones iguales, o **descartar** en caso de realizar una acción que se considere que no tenga efecto.

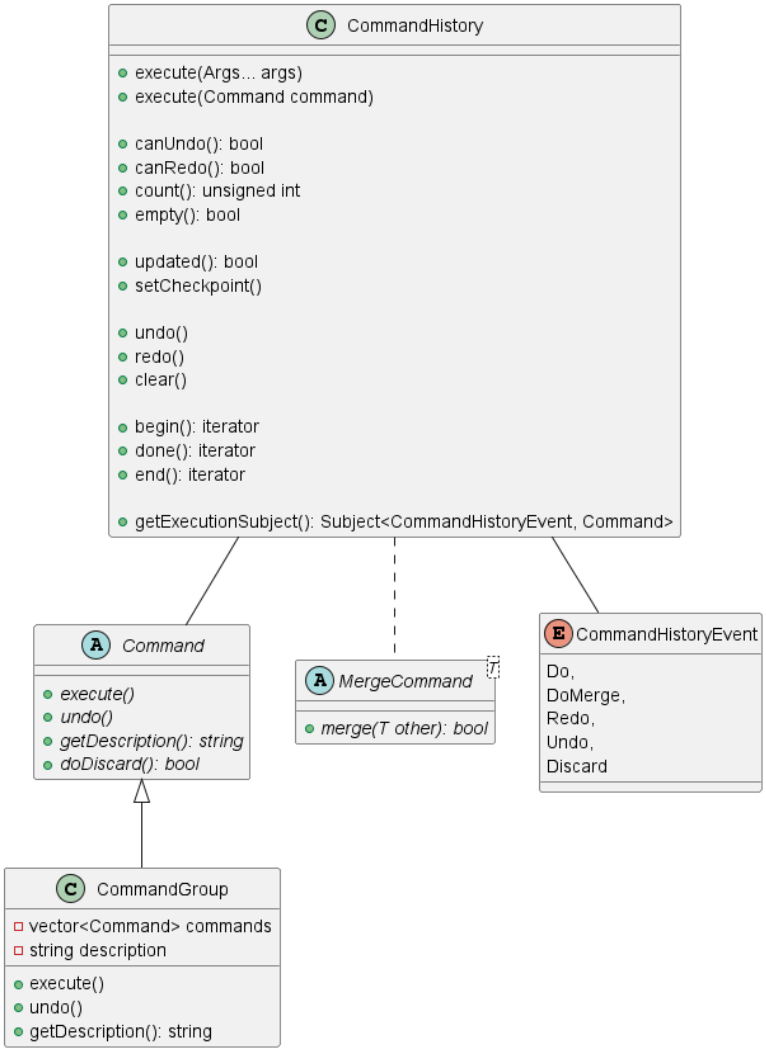


Ilustración 2.8 – Diagrama de clases. CommandHistory

En la Ilustración 2.8 se puede ver cómo el historial de comandos trabaja con la clase abstracta **Command**, la cual tiene una implementación genérica para encapsular grupos de comandos de forma atómica. En el caso de **MergeCommand**, será la clase CommandHistory la que determinará si el comando la implementa y por tanto puede intentar fusionar el comando anterior. Como particularidad, aprovecharemos el patrón Observer para emitir eventos cuando un comando se procese, de forma que se pueda reaccionar ante esto, por ejemplo, para marcar el documento como editado.

2.3.2 Diseño arquitectónico del sistema

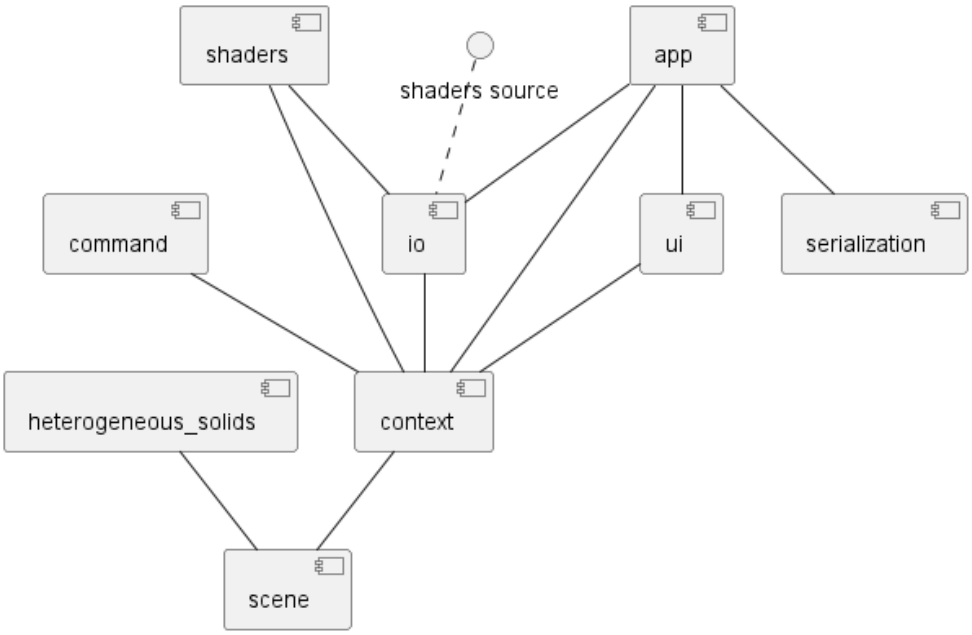


Ilustración 2.9 – Diagrama de componentes. Arquitectura del sistema

Tal y como muestra la Ilustración 2.9, los diferentes componentes que conformarán el sistema tendrán un punto de origen común, la aplicación, el cual contiene la comunicación de más bajo nivel con la ventana y tendrá acceso a los 3 componentes fundamentales: **interfaz gráfica (UI),** gestión de la **entrada-salida (IO)** y **gestor de contextos** (Context Manager). Además de estos 3 elementos que se coordinan para el funcionamiento de la aplicación, existen otros objetos auxiliares como los **serializadores** de escena disponibles o el propio objeto de la ventana para realizar acciones como cambiar su título.

El **gestor de contextos** permitirá establecer qué contexto estará activo. De esta forma, si una ventana quiere cambiar el contexto podrá hacerlo fácilmente, y el resto de ventanas en consecuencia utilizarán el nuevo contexto para sus interfaces. El contexto será utilizado tanto por las ventanas como por menús y diálogos. Por otro lado, el contexto en sí mismo solo es un contenedor con las diferentes clases que conforman el modelo. El modelo contará con un **identificador** que permitirá identificarlo únicamente en caso de ser necesario. Los elementos que componen el contexto son los siguientes:

- **Scene**. Contenido de la escena
- **Renderer**. Objeto encargado de renderizar la escena.

- **RenderProfile**. Perfil con la configuración del renderizador. Se mantiene por separado en caso de necesitar varios más adelante.
- **CommandHistory**. Gestor del historial de comandos.
- **MacroSet**. Contenedor con las macros.
- **Snapshot**. Configuración para las capturas de pantalla.
- **MaterialEditor**. Contiene información de los materiales, así como de material que se está editando.
- **SceneStatus**. Estado de la escena. Se mantiene información de error en caso de ocurrir algún problema renderizando. Al estar en el contexto, puede limpiarse el error tanto desde la escena como desde los shaders en caso de realizar cambios.
- **SceneTools**. Contiene la transformación que se está realizando gráficamente sobre el modelo. Si se necesitase alguna otra herramienta además de la transformación matricial, se incluiría como otro campo en esta clase.
- **Selector**. Clase encargada de la selección e identificación de los objetos en la escena. Las ventanas que requieran información sobre los objetos seleccionados se la preguntarán a esta clase.
- **EventManager**. Clase que distribuirá todos los eventos conocidos. Será un punto de encuentro y distribución de los eventos de la escena a los componentes que los consuman.
- **Document**. Documento opcional que se establecerá cuando al contexto se le suministre para realizar las operaciones de guardado.
- **IO**. Se mantendrá en todos los contextos una referencia a la entrada-salida común para que pueda ser consumida por las vistas fácilmente.

2.3.3 Arquitectura de la escena y la abstracción gráfica

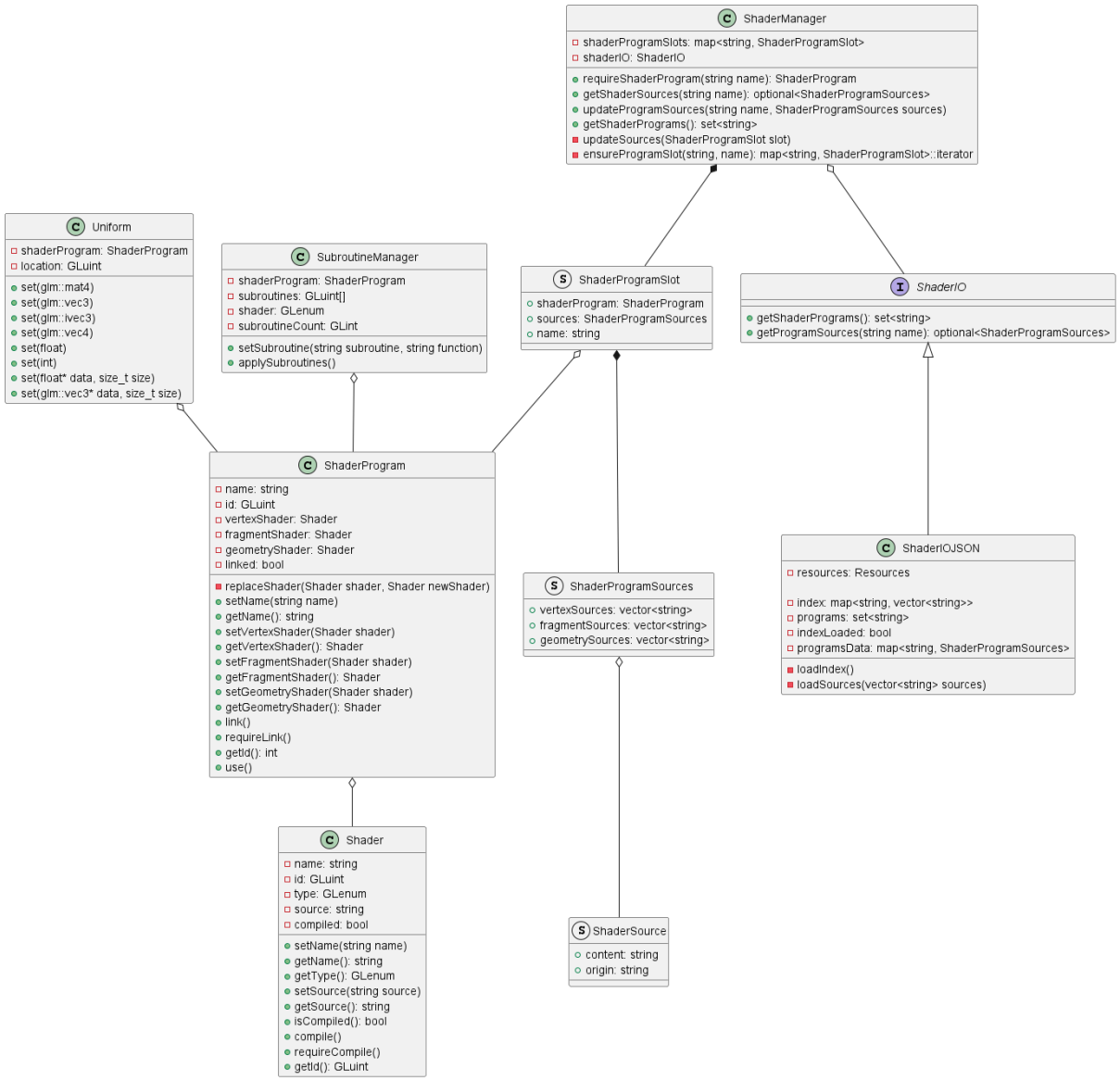


Ilustración 2.10 – Diagrama de clases. Gestor de shaders

Para comenzar con la abstracción gráfica tendremos que plantear la gestión de los shaders, los cuales nos permiten comunicarnos con la tarjeta gráfica y dibujar así la escena en pantalla (Ilustración 2.10). Dado que queremos poder separar en diferentes archivos un mismo shader para tenerlo más organizado, necesitaremos diseñar nuestro propio mecanismo de carga de shaders. Para ello, utilizaremos una interfaz, la cual nos provee con los fuentes desde el disco, y le crearemos una especialización que utilizará el formato JSON para definir qué fuentes componen cada shader de manera cómoda.

El *shader manager* será el encargado de proveer los **shader program** compilados, pudiendo ser alterados y recompilados también a través de este. Los objetos de clase **shader**, por tanto, solo servirán para manipular el código fuente y recompilar el programa que los use, dado que, de cara al exterior del gestor, se utilizará el shader program para todas las operaciones.

El **shader program** puede ser usado para crear **uniforms** (abstracción del uniform de GLSL) y así compartir valores con la CPU (matrices, vectores, etc.). Además, también se puede determinar la forma de trabajar del shader program asignando subrutinas mediante el **SubroutineManager**. Cada gestor de subrutinas se asocia al tipo de subrutinas que maneja (en nuestro caso, solamente serán del fragment shader), para por ejemplo, determinar el tipo de luz que se va a utilizar, o el tipo de material que se va a usar.

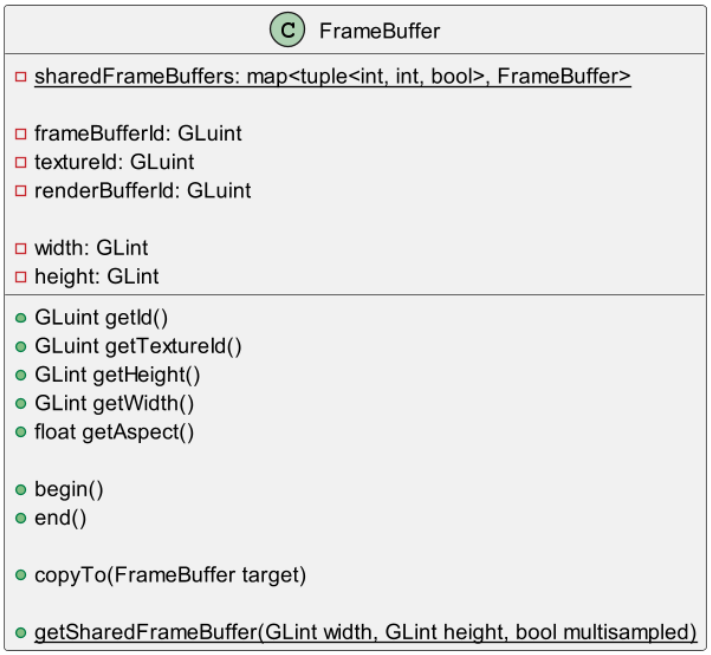


Ilustración 2.11 – Diagrama de clases. FrameBuffer

Para poder realizar múltiples renderizados, uno de cada escena visible en pantalla, así como para la selección por color, se creará una abstracción del **FrameBuffer** (Ilustración 2.11), el cual podrá definirse como *multisampled* para permitir *antialiasing* y obtener mejores resultados en la renderización de la escena. En

el caso de las capturas puntuales de pantalla, se mantendrá un mapa de FrameBuffers compartidos, de forma que si no existen, se cree uno con la resolución requerida.

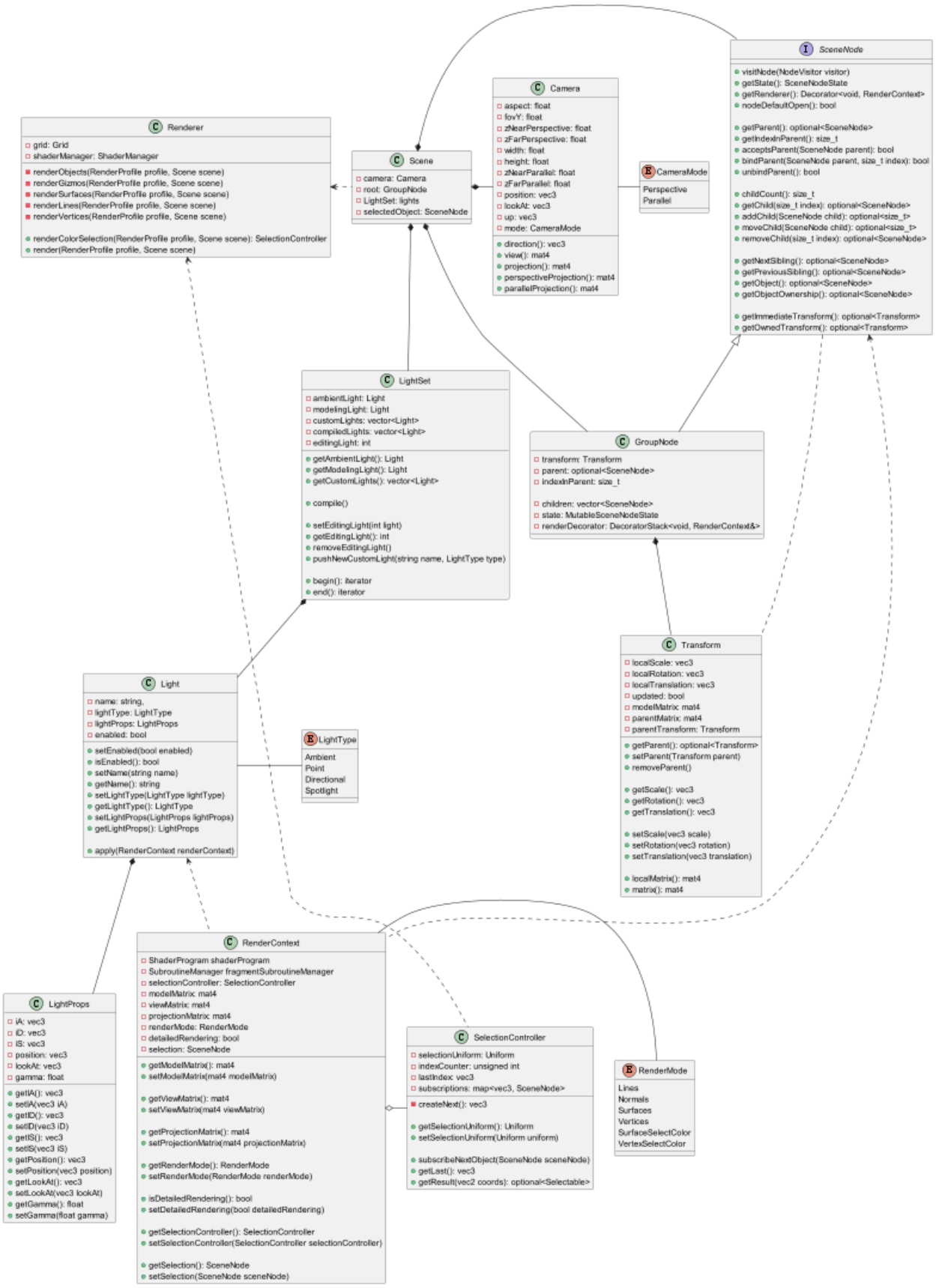


Ilustración 2.12 – Diagrama de clases. Escena

La escena (Ilustración 2.12) será el objeto que determina lo que se está viendo en pantalla, qué se ve (**el árbol de escena**), cómo se ve (**las luces** que lo iluminan) y desde dónde (**cámara**). Esta información junto a la configuración de renderizado (*Perfil de renderizado*) la utiliza el renderer para dibujar la escena con la ayuda de los **shaders** cargados desde el disco.

Las luces tendrán la siguiente distribución: una *luz ambiente opcional**,*** y o bien una *luz de modelado* o un *conjunto de luces*, pudiendo ser las luces de los siguientes tipos: **foco**, **direccional** y **puntual**. Cada luz viene definida por su nombre, su tipo, las propiedades que tiene y si está encendida o no.

La cámara permitirá establecer o bien proyección paralela o proyección perspectiva, además de modificar tanto el punto donde se sitúa como el punto que observa, permitiendo realizar movimientos de cámara.

El proceso de renderizado se realizará a través del **RenderContext**, el cual se compartirá por todo el árbol, de forma que a través de este se acceda al **shader program** y se pueda dibujar en la escena, así como saber si un objeto se debe renderizar, si los hijos de este se deben renderizar, o cómo se han de renderizar (líneas, superficies, vértices), según cómo se haya parametrizado el contexto.

El árbol de escena se construirá a partir de una interfaz compartida por todos los nodos del árbol, **SceneNode**, que contendrá las primitivas a las cuales debe responder cada nodo para el buen funcionamiento del sistema, permitiendo de esta forma la comunicación entre los nodos, el movimiento de unos nodos a otros y por consiguiente una manipulación completa y extensible de la escena. Entre estas primitivas estarán los métodos para conocer y asignar el nodo padre, así como los nodos hijo y la obtención del **transformador**, que será el encargado de dar al objeto su matriz de transformación.

Además de las primitivas para la interacción entre ellos, los nodos contarán con métodos para obtener un **estado** (descripción y visibilidad), obtener el **icono** y su comportamiento por defecto en la renderización del árbol de escena, un método para

interactuar con el patrón **visitor**, y otro que devolverá un **decorador** para la renderización del nodo.

El único nodo genérico es el **grupo de nodos**, el cual delega su comportamiento a una secuencia de hijos, siendo el mecanismo por defecto para mantener una **estructura jerárquica** abstracta en el árbol. Este nodo permitirá interactuar con los hijos solamente a través de la herencia del **transformador**. El nodo **raíz** de la escena, del cual cuelgan todos los demás, será un nodo de este tipo.

En cuanto al proceso de renderizado habrá dos tipos:

- **Renderizado de la escena**: Realiza una iteración por cada luz para las superficies, así como una por líneas y otra por vértices (utilizan diferentes *shader program*) para obtener la escena que se verá en pantalla.
- **Renderizado para selección**: Dado que solo se pueden seleccionar objetos (superficies) y coeficientes (vértices) y no se utilizarán luces, dado que se usará un color para definir de forma única a cada objeto, se realizarán 2 pasadas únicamente.

Para mandar la geometría calculada a la **GPU**, se han creado abstracciones tales como **Mesh**, **NormalField**, y **Poliline**, las cuales evitan tratar directamente con los IBOs y los VAOs de OpenGL, para crear **mallas**, **normales** y **polilíneas**, respectivamente.

2.3.4 Arquitectura del modelo de sólidos heterogéneos

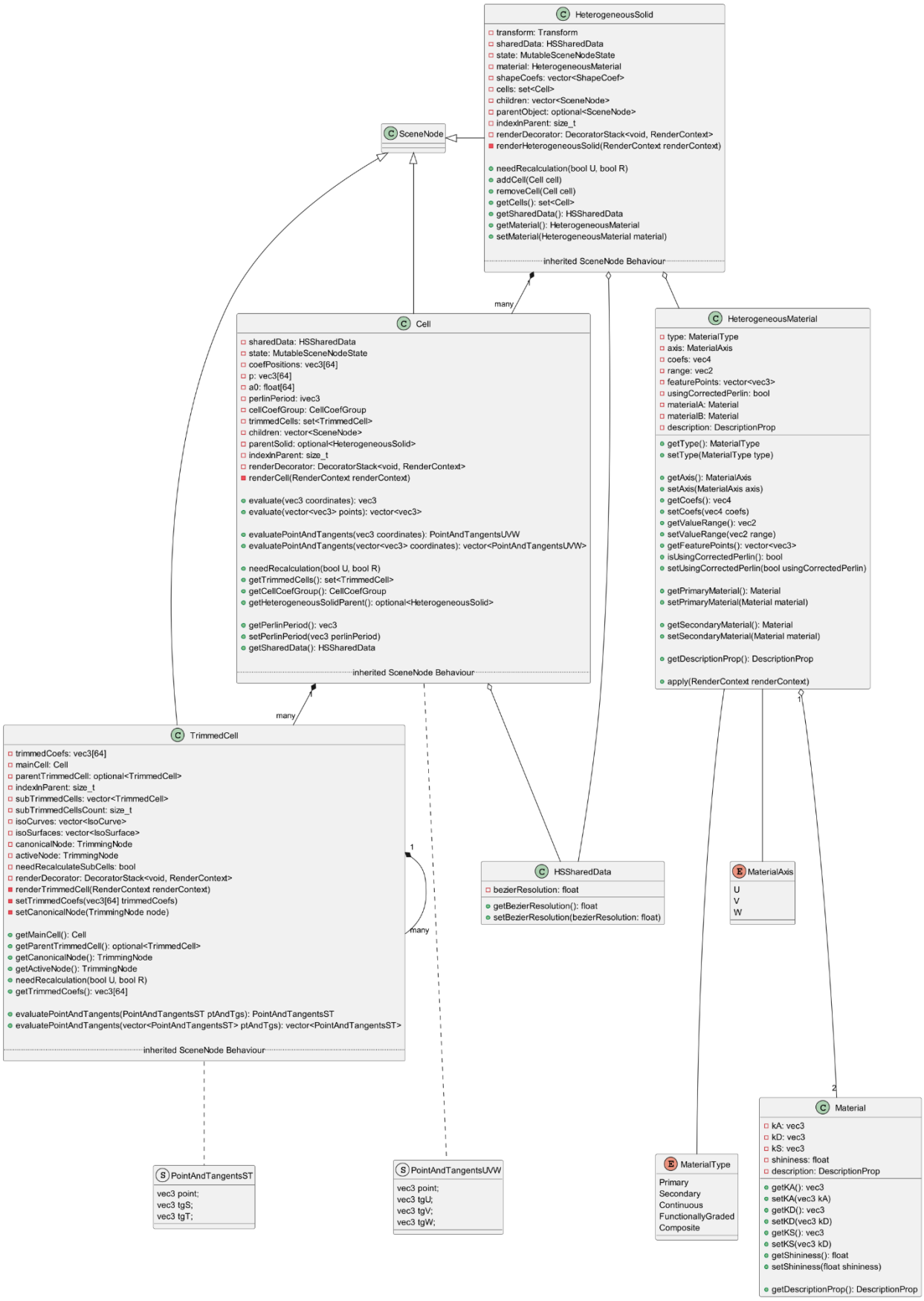


Ilustración 2.13 – Diagrama de clases. Estructura de los sólidos heterogeneos

La Ilustración 2.13 muestra cómo se organizan los sólidos heterogéneos a más alto nivel, identificando el **sólido heterogéneo**, las **celdas**, las **subceldas**, y los **materiales** como los elementos principales. El sólido heterogéneo está compuesto por un número variable de celdas las cuales gestiona mediante los coeficientes compartidos entre ellas, cuya estructura se comentará más adelante. Las celdas contienen subceldas, las cuales a su vez contienen también subceldas, pudiendo crear un número indeterminado de subniveles. En el caso de las subceldas, su geometría es manipulada a través de las **operaciones de manufacturado** y de la utilización de macros, la cual se mostrará también más adelante.

Siguiendo con la estructura general de los sólidos heterogéneos, entre el sólido y las celdas se mantendrá una clase la cual permite establecer propiedades generales sobre el modelo, **HSSharedData**. De momento, solo se trabaja con la resolución que se va a utilizar para calcular los modelos.

El sólido tendrá asignado el **material heterogéneo** que utilizará, a su vez conformado por 2 **materiales simples**. El material dispondrá de todas las propiedades independientemente del tipo utilizado: primario, secundario, compuesto, continuo, etc.

La disposición en árbol de la estructura de los sólidos heterogéneos permite:

- Facilitar el cálculo de la geometría, dado que cada **subcelda** comunicará sus cálculos al dominio de su celda superior, y todas conocerán quien es su **celda**.
- Gestionar de forma cómoda el modelo, pudiendo eliminar una celda y todo el árbol colgante mediante una sola operación, o cambiar todas las subceldas colgantes a una subcelda mediante un cambio de operación de manufacturado.
- Renderizar de forma jerárquica el modelo, separando las obligaciones de cada objeto. El sólido le indica al material que aplique sus propiedades, después cada celda establece sus propiedades particulares como el **perlin period**, y las subceldas delegan hasta llegar al último nivel, el cual contiene la geometría.

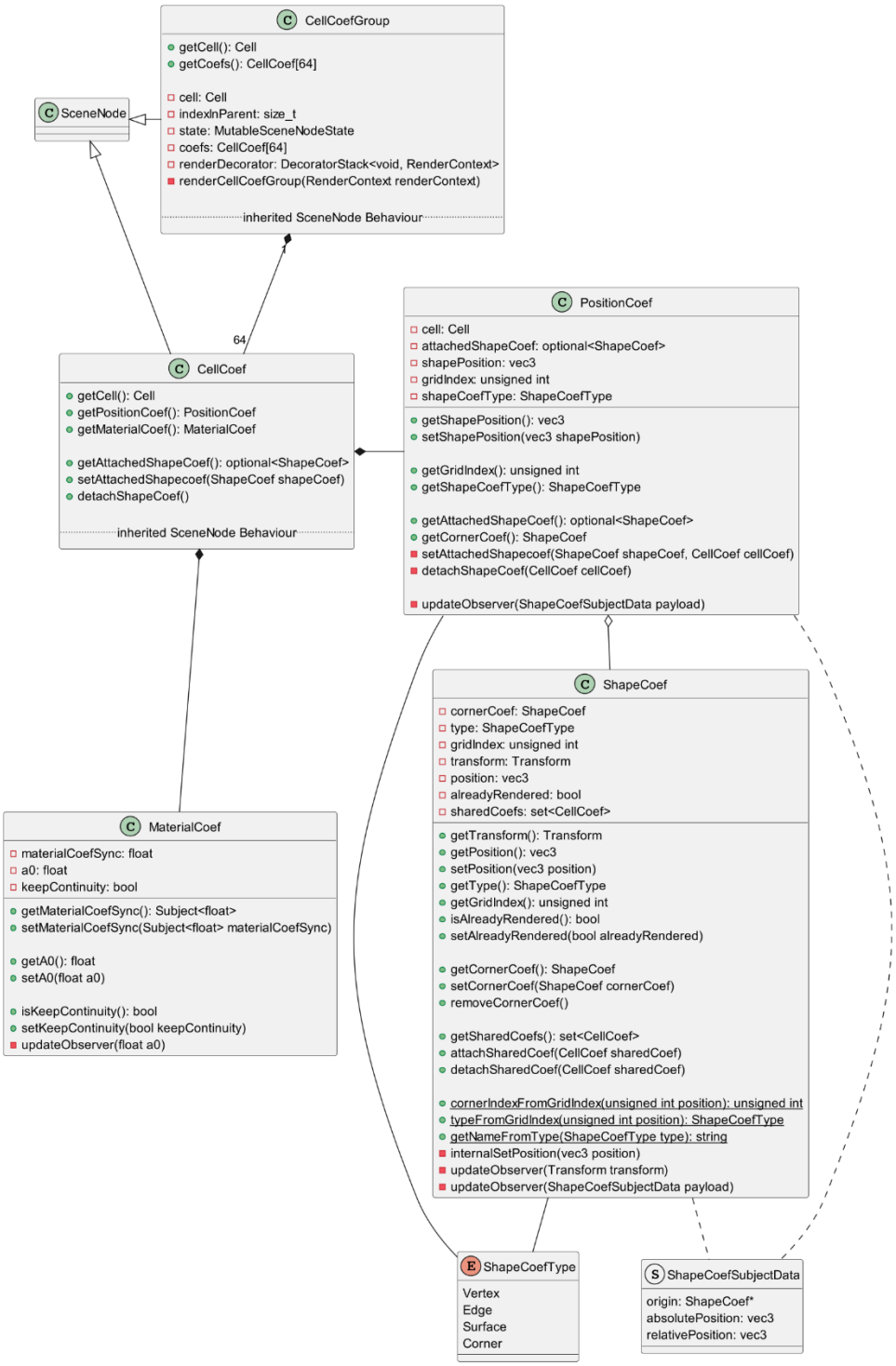


Ilustración 2.14 – Diagrama de clases. Estructura de los coeficientes

La Ilustración 2.14 muestra cómo se organizan los coeficientes. Agrupados bajo el **CellCoefGroup** se encuentra los **CellCoef**, teniendo cada uno su propio **PositionCoef** para la geometría (**posición** del coeficiente) y **MaterialCoef** para la

disposición del material (propiedad **a0**). El **CellCoefGroup** es un objeto que tienen todas las celdas, permitiendo disponer de los coeficientes y de sus propiedades aun sin estar incluidas en ningún sólido heterogéneo. Tanto este, como sus **CellCoef** heredan del **SceneNode**, de forma que puedan ser tratados como otro objeto de la escena y por tanto manipulados con un **transformador**, así como visualizados en el esquema del **árbol de la escena**. Además, cada coeficiente de celda se encargará de dibujarse como un punto en la escena.

A diferencia de los coeficientes de celda, los coeficientes de forma, **ShapeCoef** se mantienen en el sólido, permitiendo ser compartidos por aquellos coeficientes de celda con posiciones idénticas, obteniendo así la **topología** del sólido heterogéneo formada por celdas unidas a través de sus coeficientes.

Por medio del patrón Observer, los coeficientes de celda delegarán la escucha de los cambios en los coeficientes de forma a los coeficientes de posición. De forma parecida, todos los coeficientes de celda que tengan el mismo **corner** sincronizaran sus coeficientes de material en caso de querer mantener la continuidad. En este último caso, la continuidad es opcional y puede ser desactivada si se quiere una distribución más específica.

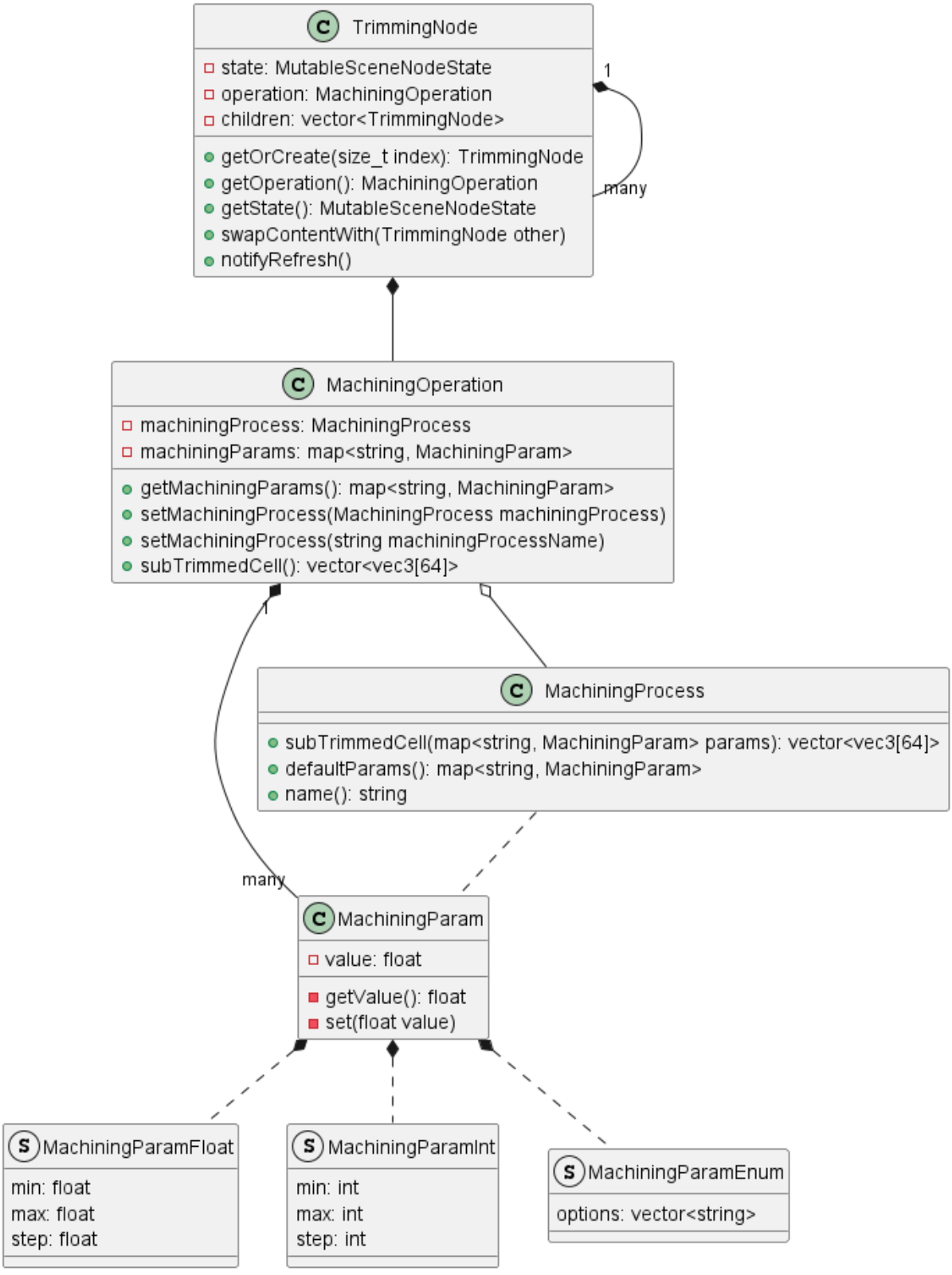


Ilustración 2.15 – Diagrama de clases. Estructura de las operaciones de manufacturado

Dado que las subceldas requieren de una estructura que seguir, la cual puede ser compartida mediante el uso de macros, se introducirán los **TrimmingNode** los

cuales mantienen su propia estructura jerárquica paralela como muestra la Ilustración 2.15. Cada uno de estos nodos tendrá una **MachiningOperation** la cual le indicará como fabricar los nodos hijos en caso de tener aplicado un **MachiningProcess** que dé lugar a un vector de al menos 1 array de coeficientes. En caso contrario será un nodo hoja.

Cada **MachiningOperation** mantiene un mapa de **MachiningParam**, el cual se pasará a cada proceso para que utilice los parámetros que necesite en sus cálculos. Los **MachiningParam** podrán parametrizarse con un flotante, un entero o un enumerado, que le indica el rango de valores válido y los intervalos en los cuales aumentar el valor.

Cada vez que se cambie el árbol que sigue una subcelda (**TrimmedCell**), este le preguntará al **TrimmingNode** su estructura y la clonará, por ejemplo, al cambiar o eliminar la macro. Cualquier cambio en el **TrimmingNode** comunicado a través de **notifyAll**, provocará que las **TrimmingCell** asociadas requieran recalcular sus nodos. Para que las subceldas sean lo más fieles posibles al esquema que sigue, el **TrimmingNode** proporcionará también el estado del **SceneNode** que se mostrará en el árbol de escena, dando lugar a que compartan **descripción**, muy útil para ver desde el árbol todos los nodos que siguen la misma macro.

Por último, todos los **TrimmingCell**, tendrán 2 **TrimmingNode** asociados, la **macro**, que puede ser nulo, y el **canónico** con el que trabajará por defecto en caso de no tener ninguna macro asociada. Esto no solo da un manejo muy versátil de los nodos, sino que combinando varias macros puede tener un efecto muy potente.

Dado que la elección de la macro es en la celda y no en el nodo que se le aplica, nos evitamos **ciclos infinitos**, en situaciones como: una macro que se aplique a su padre o a sí misma como macro. Además, si representamos la jerarquía de celdas con una macro, y **a una subcelda le cambiamos el canónico** por una macro (el canónico que estaba siguiendo era el generado por la primera macro), dejará de seguir el canónico de la macro y tendrá una **jerarquía diferente** a la del resto de nodos que sigan la misma macro.

2.3.5 Arquitectura del serializador

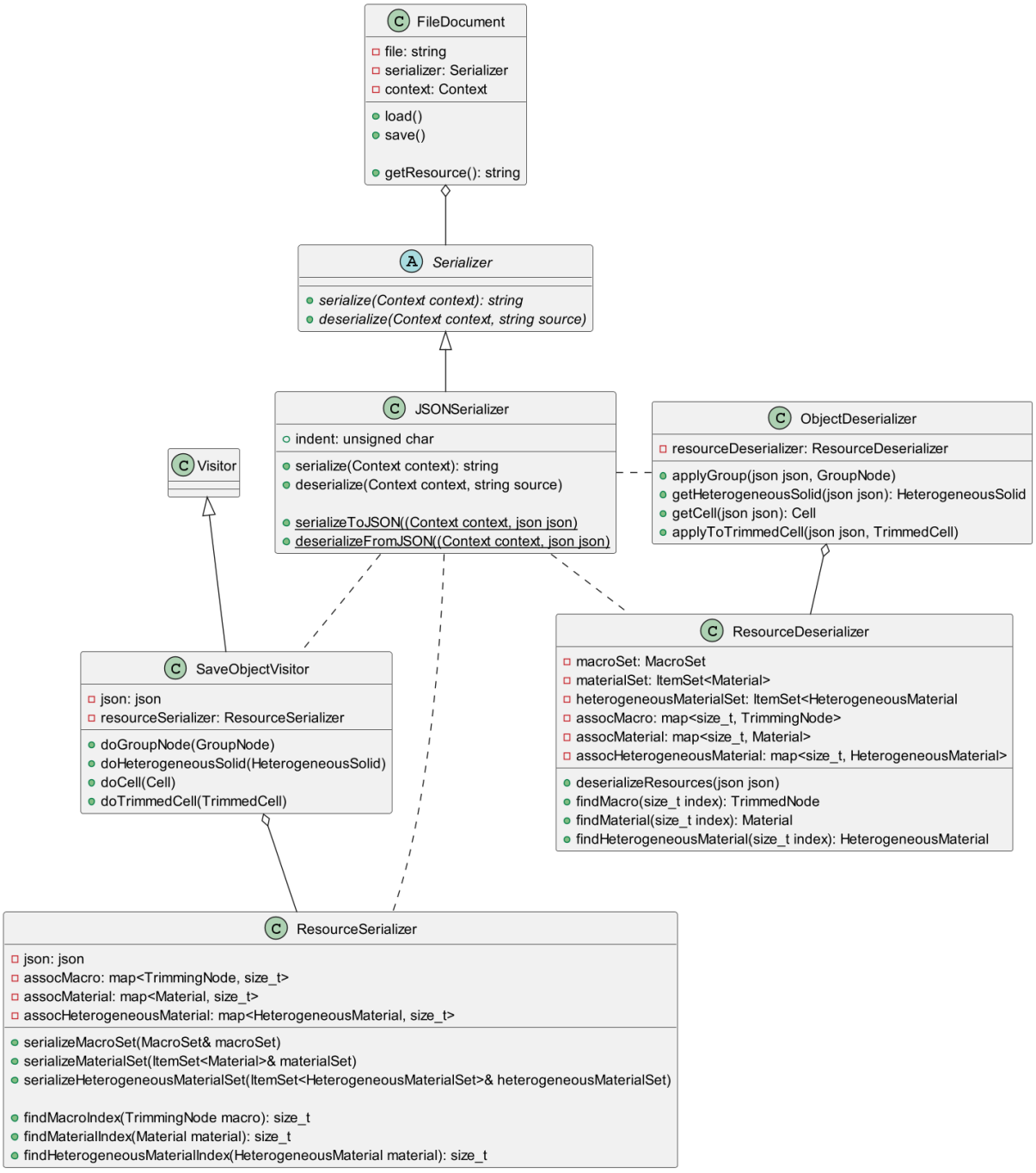


Ilustración 2.16 – Diagrama de clases. Estructura del serializador

Para trabajar con la entrada y salida (***load*** and ***save*** en la clase respectivamente) de documentos que provienen de un archivo, se utilizará la clase **FileDocument**, parametrizada con el **Serializer** que se desee utilizar (Ilustración 2.16). En este caso solamente se ha implementado un serializador para el formato **JSON**, pero se podrían implementar otros utilizando la misma interfaz.

Ante la posibilidad de añadir otro **serializador**, o bien se crea uno nuevo para toda la escena, o bien si el nuevo formato mantiene una estructura de claves y listas similar a la de JSON, se puede hacer perfectamente un Adaptador sobre el serializador de JSON. Esta última solución permite ser flexible y práctico, preocupándonos solo de tener un serializador JSON robusto, el cual como se menciona en las tecnologías utilizadas (sección 1.10) contará con una de las bibliotecas de JSON más utilizadas para C++.

En cuanto a la estructura interna del serializador implementado para JSON, queda totalmente desacoplado del modelo de sólidos heterogéneos, ya que almacenará y creará los objetos sin que estos le faciliten ningún tipo de comportamiento en particular para este proceso, siendo esto posible gracias al patrón Visitor.

El serializador utilizará una serie de clases auxiliares para las operaciones de carga y guardado, diferenciando entre el **árbol de escena** conformado por los objetos de la escena organizados de forma jerárquica, y los **recursos** (materiales y macros) que utilizan estos objetos:

- **ResourceSerializer.** Se encarga de almacenar los recursos dentro del objeto JSON y proveer los índices en los que se encuentran almacenados.
- **ResourceDeserializer.** Se encarga de crear los recursos utilizando la información cargada del objeto JSON, así como de almacenarlos en sus colecciones pertinentes.
- **SaveObjectVisitor.** Itera el árbol de escena y mediante el patrón visitor se encarga de gestionar cada caso en particular para almacenarlo en el objeto JSON. Se ayuda del **ResourceSerializer** para asociar los índices de los recursos en los objetos finales.
- **ObjectDeserializer.** Construye el árbol de escena desde los objetos JSON. Se ayuda del **ResourceDeserializer** para traducir los índices de los recursos en los objetos ya cargados.

2.3.6 Diseño de la interfaz gráfica

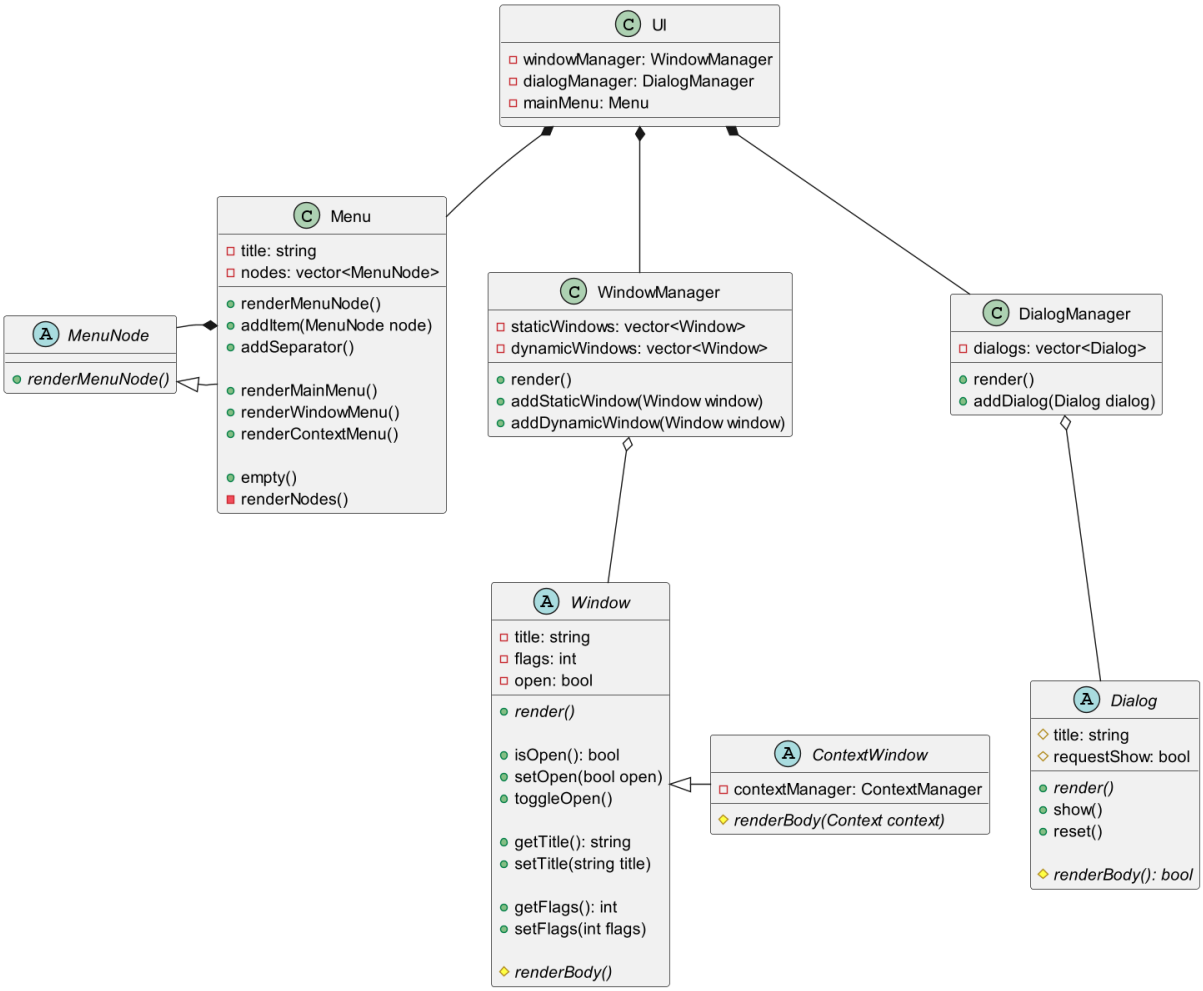


Ilustración 2.17 – Diagrama de clases. Interfaz gráfica

La interfaz abstraerá el sistema de ventanas de **Dear ImGui** mediante 3 objetos: el controlador de ventanas, el controlador de diálogos y el menú principal, como muestra la Ilustración 2.17.

El más simple es el **controlador de diálogos**, al cual se le indicará qué diálogos debe tener en cuenta, de forma que si a un dialogo se le solicita que se muestre el gestor de diálogos se encargará de mostrarlo.

El **controlador de ventanas** tendrá un comportamiento similar al de diálogos, salvo que tendrá dos tipos de ventanas, estáticas y dinámicas. Las ventanas estáticas nunca serán eliminadas si se cierran, mientras que las dinámicas sí. En la práctica

solo tendremos un tipo de ventana dinámica, la escena, y el resto de ventanas serán estáticas. Por ejemplo: la ventana que gestiona el perfil de rendering se podrá volver a abrir después de cerrarse.

La ventana de la escena mantendrá el contexto. De esta forma, cuando se seleccione esta establecerá en el gestor de contextos su contexto, y el resto de ventanas que dependen del contexto podrán trabajar con esa escena.

Las ventanas estáticas, todas menos la escena, trabajarán o no con el contexto. Por ejemplo: la ventana que muestra el control de los *fotogramas por segund*o en una gráfica puede ser agnóstica a cualquier contexto. Para facilitar que se trabaje con el contexto en las ventanas está la clase **ContextWindow**, tal como aparece en la Ilustración 2.17.

En cuanto a los menús, estos se han diseñado partiendo de que todo menú puede ser un nodo. Por tanto, el menú es a su vez un nodo y contiene otros nodos, pudiendo anidarlos en la jerarquía. La clase **UI** mantendrá el menú principal de la aplicación, el cual puede ser modificado añadiéndole nuevos subnodos.

2.3.6.1 Wireframe Global

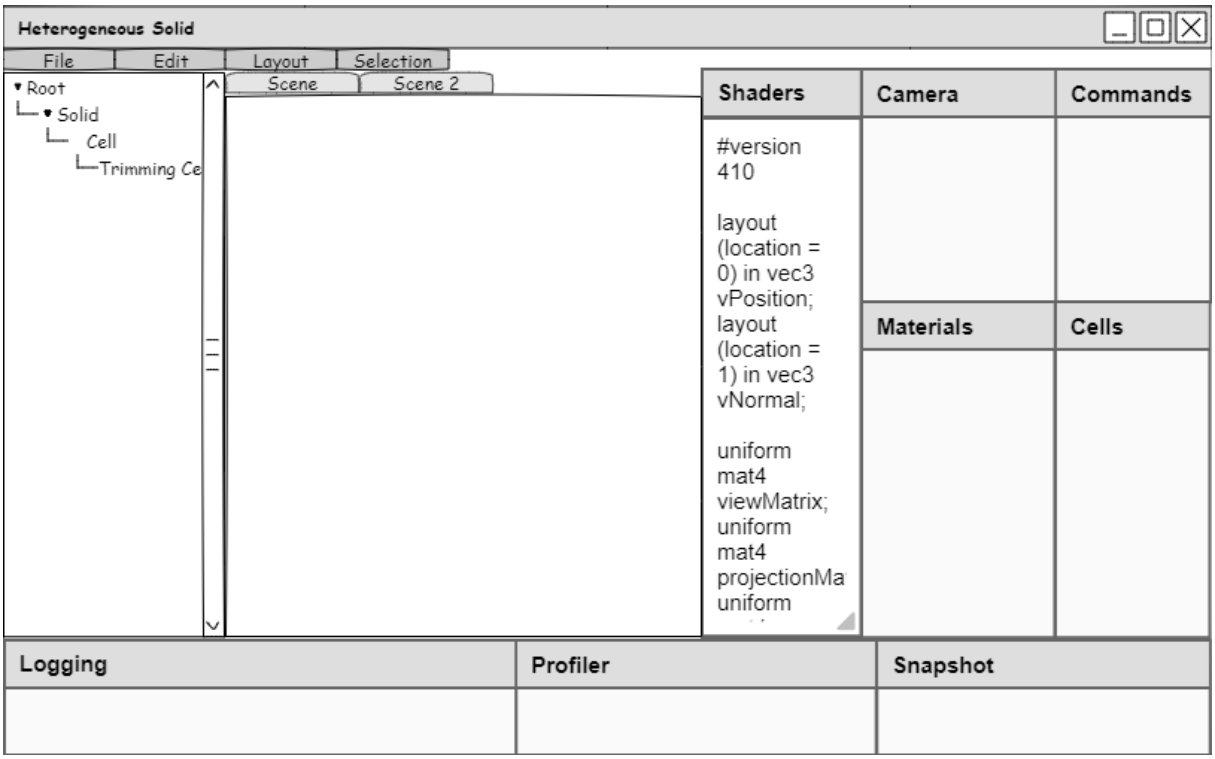


Ilustración 2.18 – Wireframe General de la aplicación

La interfaz se dividirá en dos partes: el menú principal, que contará con opciones sobre el archivo (guardado y carga), edición, layout y la selección, y el cuerpo de la aplicación, donde se encontraran distribuidas las ventanas (Ilustración 2.18).

Dado que el layout tiene una distribución dinámica, las posibilidades con las que el usuario pueda configurarlo no son predecibles. Por tanto, se mantendrá un layout inicial que se entregará con el compilado final, el cual se tendrá en cuenta a la hora de diseñar las ventanas.

Partiendo de la idea anterior, habrá dos ventanas que requerirán un mayor espacio con una forma más rectangular: el editor de shaders y por supuesto la escena. Dado que hay más elementos en pantalla, se mantendrán las escenas en el centro siempre agrupadas bajo diferentes pestañas, y a su lado los shaders, dado que se usan menos y pueden ser o bien escondidos para ocupar más espacio por la escena o bien expandidos cuando se requiera su uso.

En la parte inferior se colocarán ventanas que tienen una distribución más horizontal, como la ventana de logs, la ventana que asiste la configuración de las capturas de pantalla y el gráfico con los fotogramas de la aplicación. Las ventanas que son más alargadas verticalmente son el árbol de escena y la configuración de renderizado, dado que es una lista de opciones. Como no son casos de uso que suelan realizarse simultáneamente, ocuparán el mismo espacio al igual que las escenas están apiladas, cambiando de ventana si es necesario.

Los demás elementos consisten en ventanas más pequeñas y más cuadradas además de ser específicas de las propiedades de un determinado objeto, por lo cual se distribuirán en la parte derecha de la aplicación ocupando el espacio restante, destacando siempre que esta distribución solo se utilizaría en caso de necesitar todas las herramientas simultáneamente.

2.3.6.2 Wireframe de Ventanas: Profiler

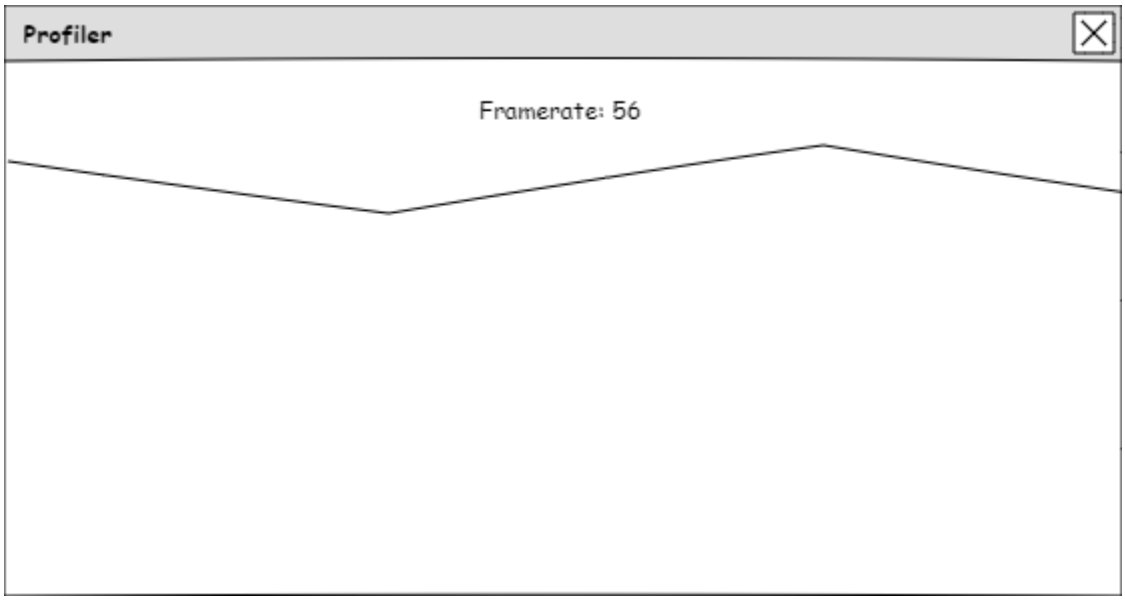


Ilustración 2.19 – Wireframe de Ventanas: Profiler

La ventana del **profiler** (Ilustración 2.19**)** mostrará en tiempo real los fotogramas por segundo del editor. De esta forma se podrá ver mediante el gráfico qué operaciones producen una ralentización mayor a la hora de trabajar con el modelo, y deberían ser optimizadas.

2.3.6.3 Wireframe de Ventanas: Logging

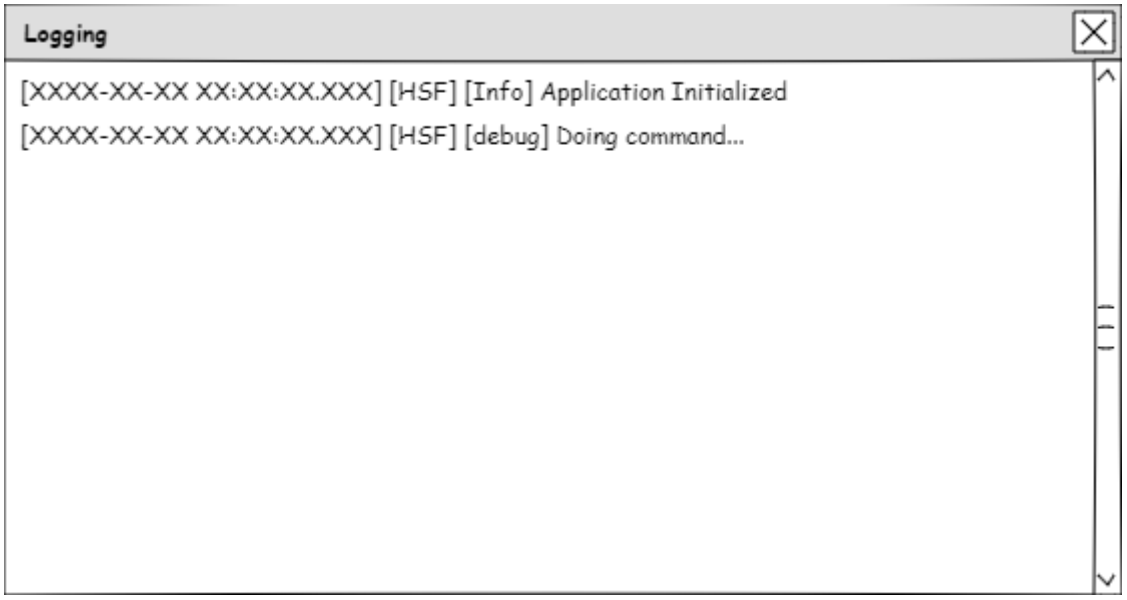


Ilustración 2.20 – Wireframe de Ventanas: Logging

La ventana de **logging** (Ilustración 2.20**)** será la mejor herramienta de depuración, dado que visualizará todos los eventos que se emitan en la aplicación. Funcionará por medio de la biblioteca *spdlog* mediante un adaptador, y desde cualquier parte del código se podrá emitir un mensaje con el nivel deseado, ya sea *información*, *depuración*, *advertencia* o *error*, entre otros.

2.3.6.4 Wireframe de Ventanas: Snapshot (Captura de Pantalla)

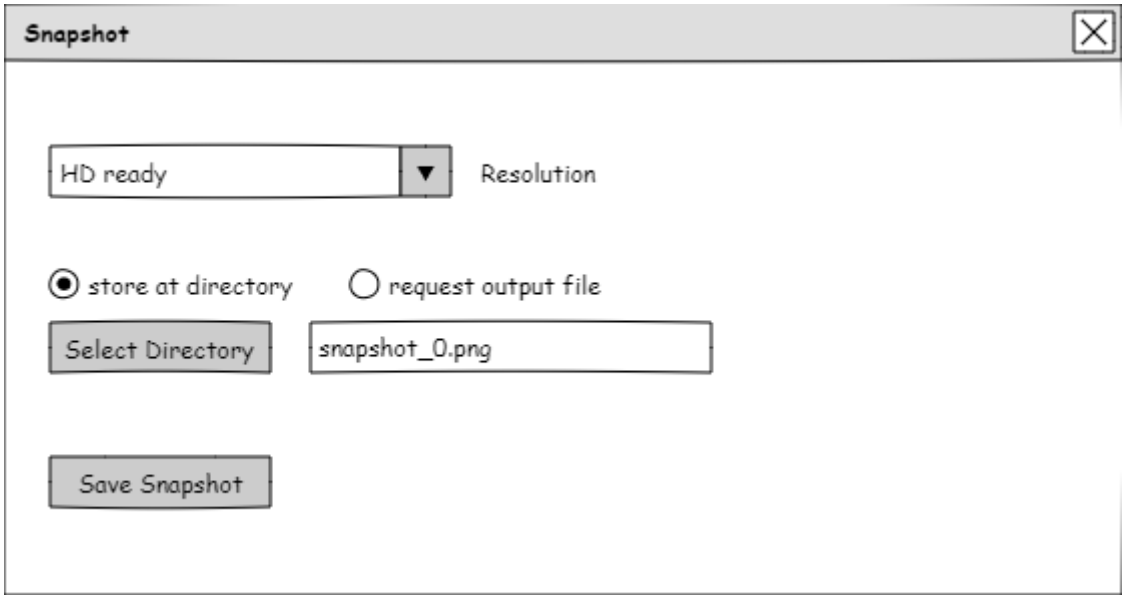


Ilustración 2.21 – Wireframe de Ventanas: Snapshot

La ventana del snapshot (Ilustración 2.21) permitirá:

- Elegir la **resolución** de la captura
- Elegir un **directorio** o solicitar el **archivo** de salida
- **Almacenar** la captura en disco con la configuración elegida

Las propiedades gráficas vendrán determinadas por las opciones seleccionadas en la ventana *Render Process.* La más importante de las que afectan al guardado es el color de fondo, dado que indicará la transparencia de la imagen capturada.

2.3.6.5 Wireframe de Ventanas: Camera

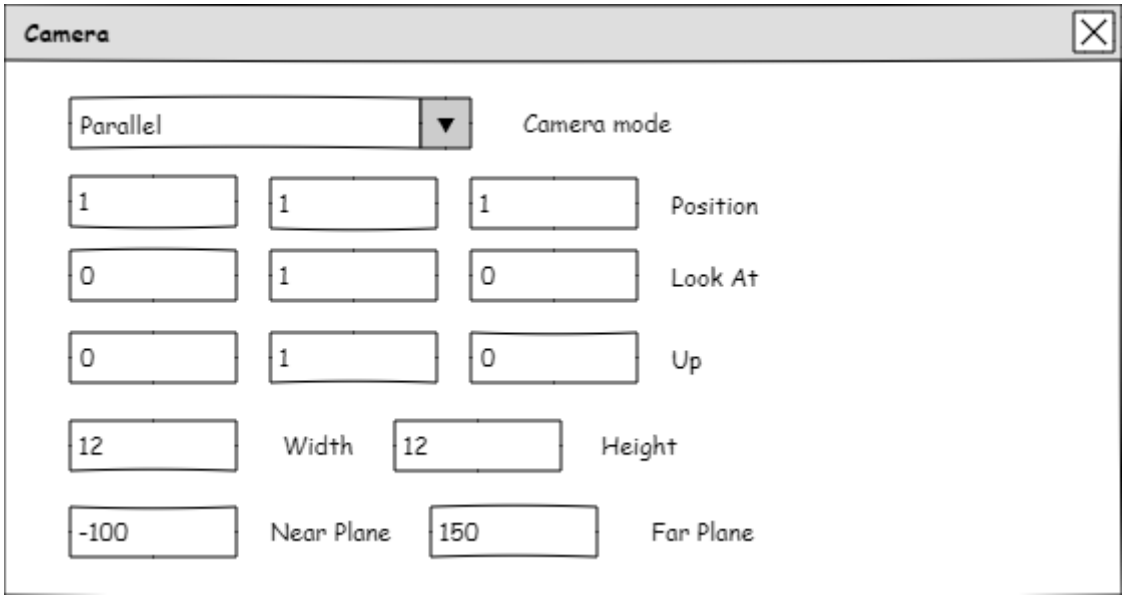


Ilustración 2.22 – Wireframe de Ventanas: Camera Parallel

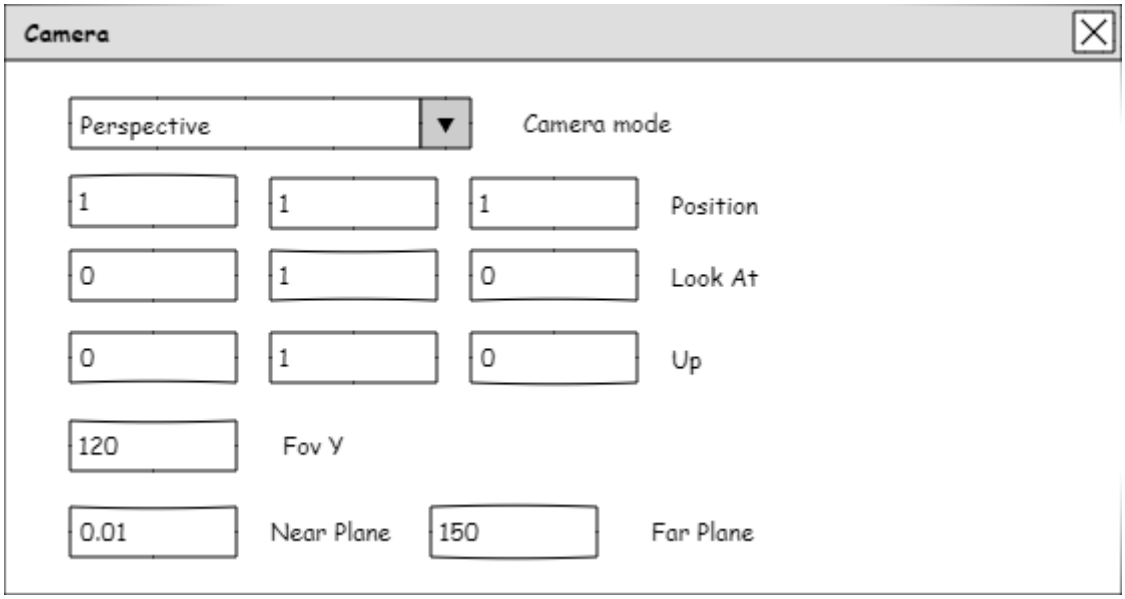


Ilustración 2.23 – Wireframe de Ventanas: Camera Perspective

La cámara tendrá modo **Paralelo** (Ilustración 2.22) y modo **Perspectiva** (Ilustración 2.23), contando en ambos con los valores de posición, punto observado y vector arriba (este último se calcula automáticamente). Ambos modos tendrán plano **cercano** y **lejano**, pero se mantendrán internamente en variables diferentes para evitar actualizarlos constantemente al cambiar el modo. La propiedad **fovY** es exclusiva de la cámara en perspectiva, así como el **ancho** y el **alto** son exclusivos de la cámara en paralelo.

2.3.6.6 Wireframe de Ventanas: Command History

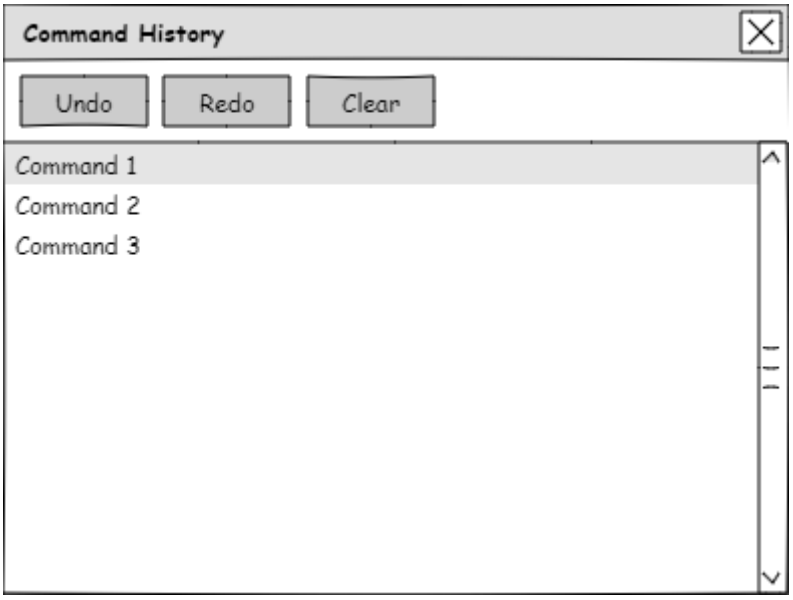


Ilustración 2.24 – Wireframe de Ventanas: Command History

Esta ventana (Ilustración 2.24) permite tener control sobre el historial de comandos:

- **Visualizar** en una lista los comandos, siendo el primero que aparece el último en aplicarse. Se diferenciarán en 2 colores (activo e inactivo) para saber cuáles se han deshecho y cuales siguen aplicados.
- **Deshacer** el último comando aplicado.
- **Rehacer** el último comando deshecho.
- **Limpiar** el historial de comandos completo, liberando los recursos que tuviese retenidos.

2.3.6.7 Wireframe de Ventanas: Tree

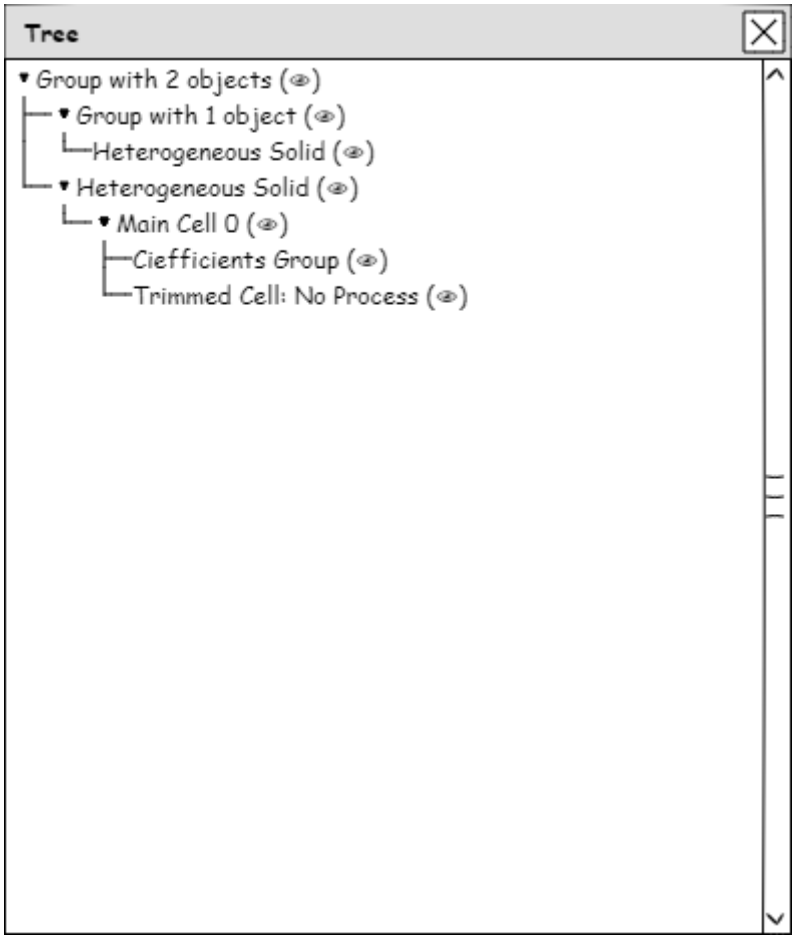


Ilustración 2.25 – Wireframe de Ventanas: Tree

La ventana del árbol de la escena (Ilustración 2.25) contiene la jerarquía de todos los elementos que parten desde la **raíz**. Desde esta ventana se puede tanto por medio del botón derecho del ratón como del icono del ojo, modificar la visibilidad (activa o inactiva) de los objetos. Además, por medio del botón derecho del ratón puede cambiarse también el nombre del objeto.

2.3.6.8 Wireframe de Ventanas: Render Process

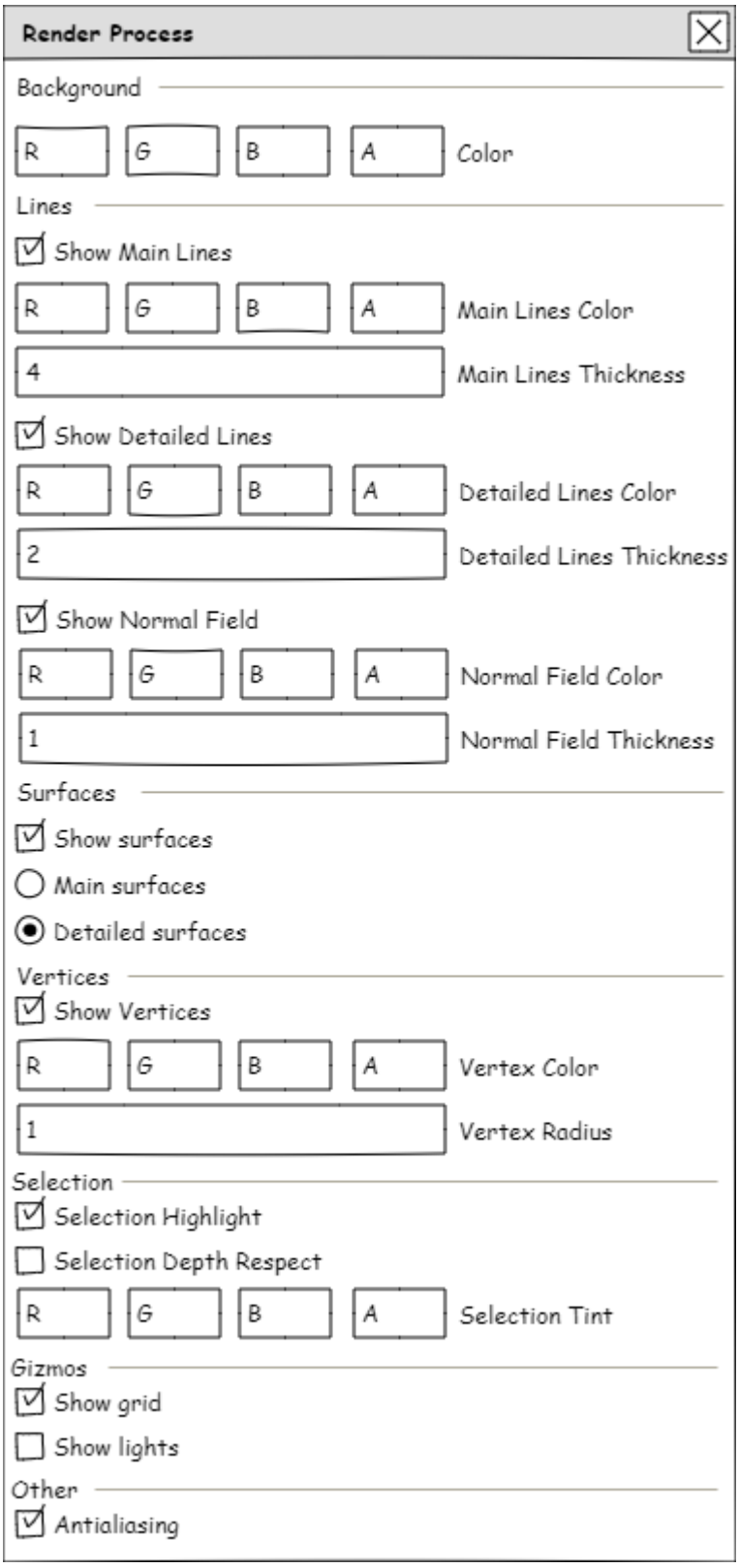


Ilustración 2.26 – Wireframe de Ventanas: Render Process

En *render process* (Ilustración 2.26) se controlará el **render profile** por defecto, dividiendo los parámetros en los siguientes apartados:

- **Fondo.** Permite configurar el color de fondo al renderizar la escena.
- **Líneas.** Se puede elegir la anchura y el color de los 3 tipos de líneas de la escena: las principales (las de la celda), las detalladas (las de la subcelda) y las del campo de normales.
- **Superficies**. Se puede elegir si se muestran o no las superficies y el nivel de detalle, siendo las superficies principales solamente las de la celda y las detalladas las de las subceldas.
- **Vértices.** Configuración (color y radio), de los vértices.
- **Selección.** Permite configurar si se quiere seleccionar elementos, el color y el comportamiento respecto a la profundidad de renderizado.
- **Gizmos.** Se trata de elementos auxiliares. En este caso las luces y la malla de referencia.
- **Otros.** En este apartado hemos incluido la opción de activar/desactivar el *antialiasing*.

2.3.6.9 Wireframe de Ventanas: Shaders

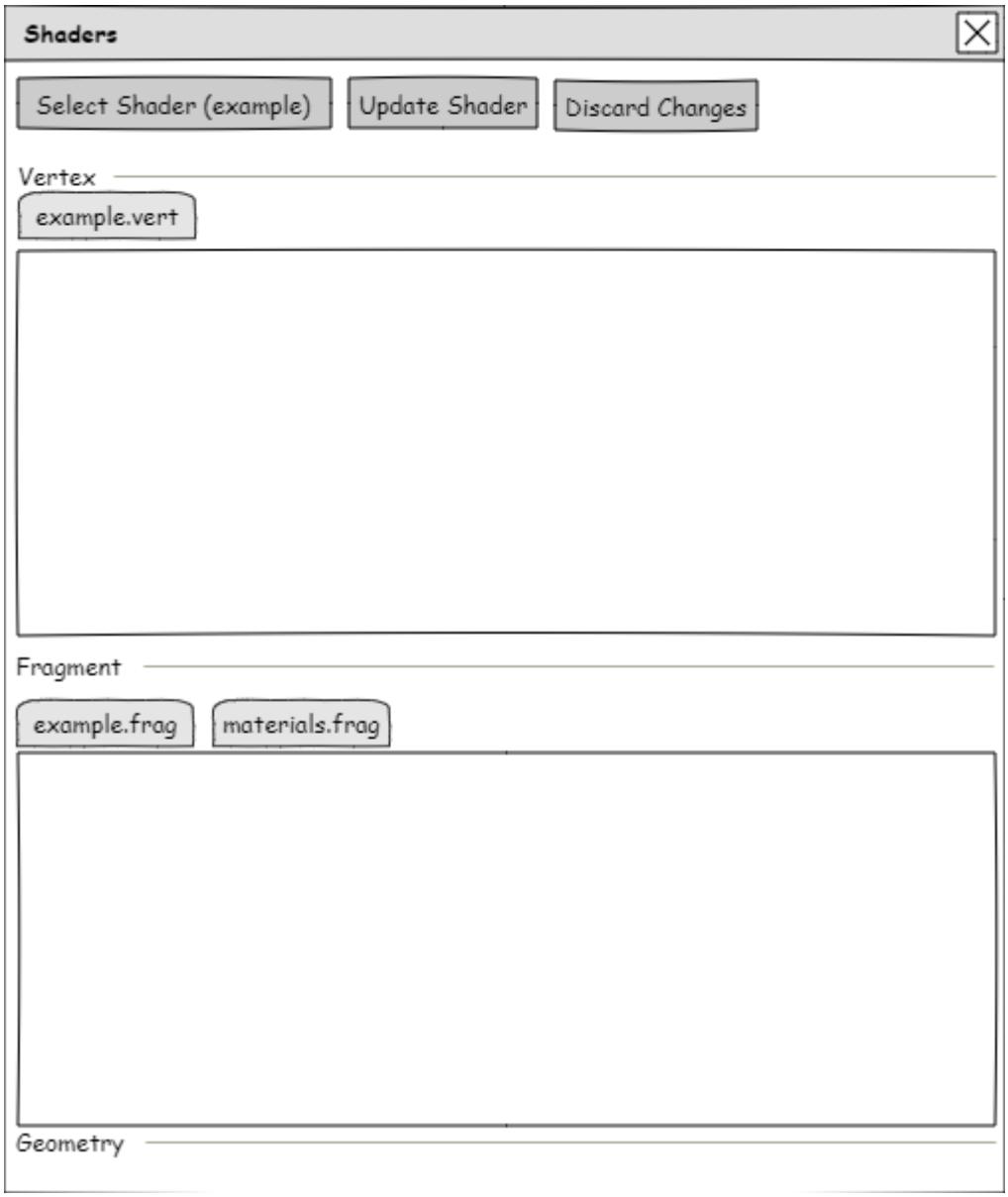


Ilustración 2.27 – Wireframe de Ventanas: Shaders

La ventana de shaders (Ilustración 2.27) tiene tres controles básicos para el código fuente de los shaders: **seleccionar**, **actualizar**, y **descartar**. De esta forma se puede elegir cualquier shader y o bien modificar y actualizar para que se aplique a la escena que se está visualizando, o si no se desea actualizar, se pueden descartar los cambios.

El shader cargado se dividirá en 3 secciones que se ajustaran al tamaño de la ventana: **vertex**, **fragment**, y **geometry**. El geometry aparece cerrado en el wireframe

dado que no usaremos ninguno a lo largo del trabajo, pero en caso de crearse lo detectaría y fraccionaría en 3 la pantalla (en lugar de solo 2 paneles).

Cada panel que tiene el tipo de shader cuenta con las diversas fuentes que lo conforman, por ejemplo, en el wireframe se han incluido dos fuentes en el **fragment**.

2.3.6.10 Wireframe de Ventanas: Cell

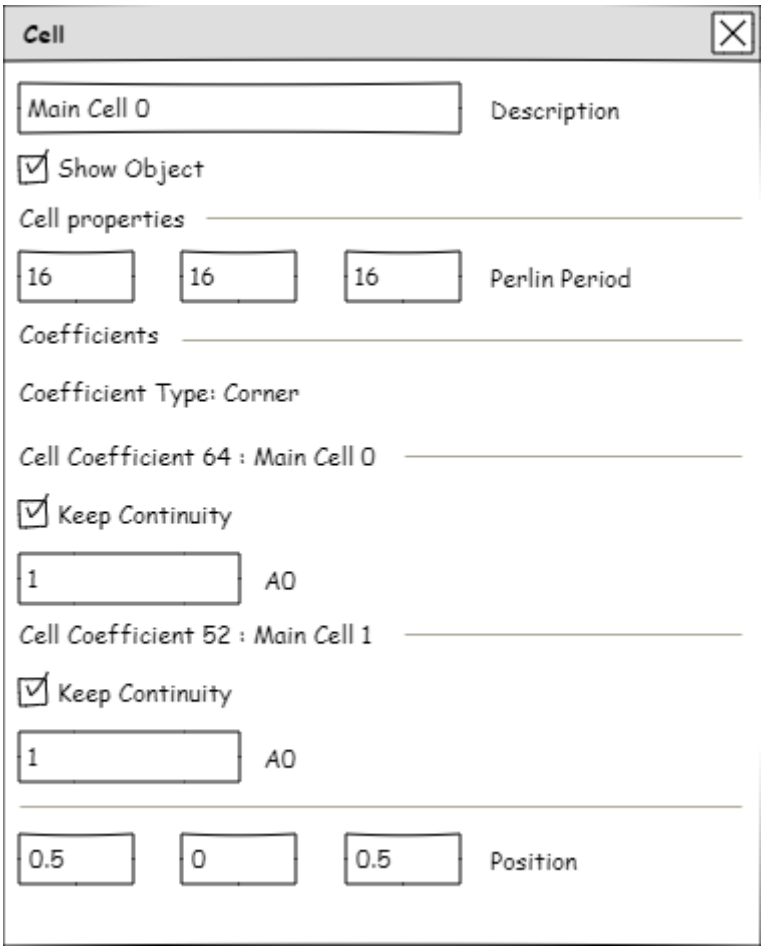


Ilustración 2.28 – Wireframe de Ventanas: Cell

La ventana de la celda (Ilustración 2.28) se encarga de trabajar con los objetos seleccionados de tipo celda. Permite cambiarles la descripción, la visibilidad y las propiedades, en este caso el **perlin period** de la celda.

En esta ventana aparece también el apartado de coeficientes en caso de ser seleccionado alguno. Se indica el tipo del coeficiente y los **coeficientes de materiales** y de posición. En el caso del **coeficiente de posición**, este es único para el

coeficiente a pesar de estar compartido por múltiples celdas, mientras que los coeficientes de material se enumerarán para cada celda junto con un parámetro para especificar si sigue o no la **continuidad** del material a las celdas vecinas.

2.3.6.11 Wireframe de Ventanas: Trimming

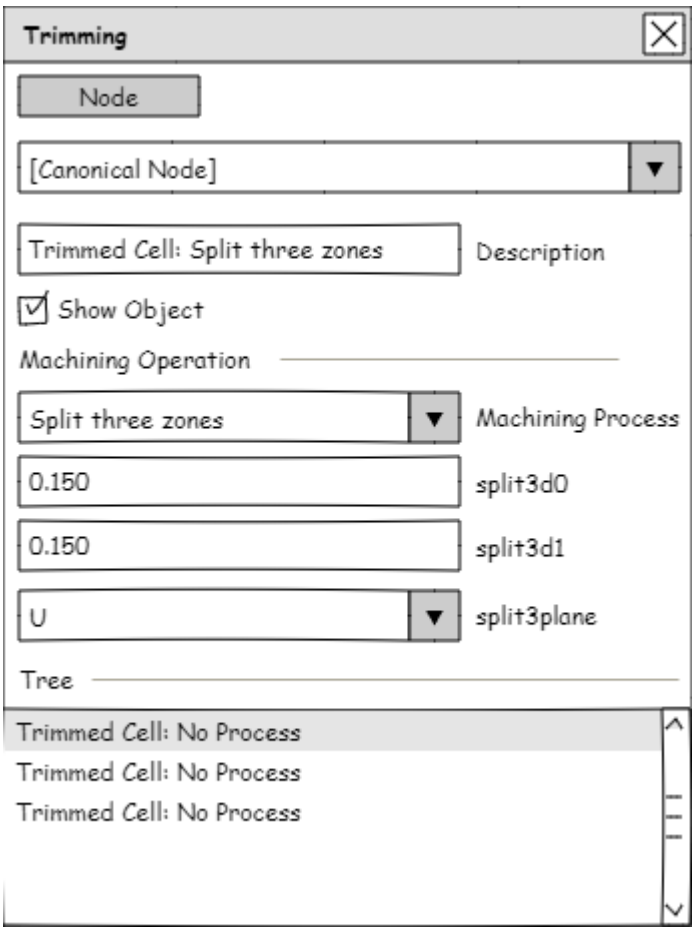


Ilustración 2.29 – Wireframe de Ventanas: Trimming

La ventana de Trimming (Ilustración 2.29) permite trabajar con los **nodos** de manufacturado sobre las **subceldas de recorte**, de forma que se pueda compartir una macro o trabajar sobre el nodo canónico.

Bajo el menú node se pueden crear nuevas **macros**, que después se pueden asignar en el combo box para trabajar en el resto de opciones, mostrando tanto la descripción como la visibilidad asociada al nodo.

Por último, se puede elegir un proceso de manufacturado y parametrizarlo, el cual se encargará de generar los hijos, a los cuales se da la posibilidad de seleccionarlos en el apartado *Tree* al final de la ventana.

2.3.6.12 Wireframe de Ventanas: Object

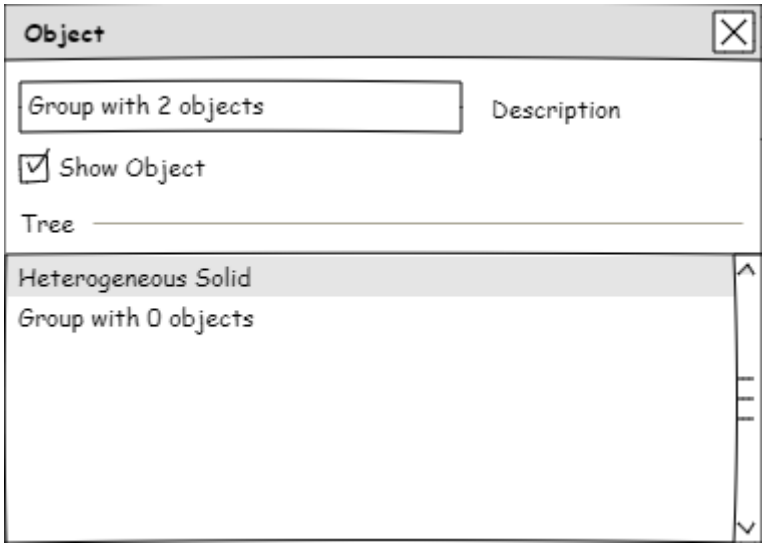


Ilustración 2.30 – Wireframe de Ventanas: Object, Group

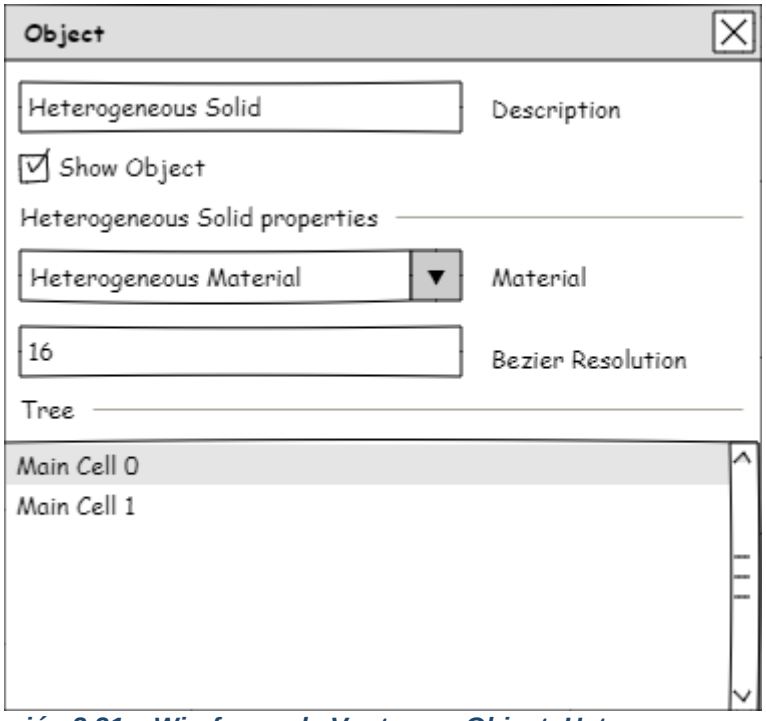


Ilustración 2.31 – Wireframe de Ventanas: Object, Heterogeneous Solid

La ventana de objetos se encarga de gestionar **los objetos de alto nivel**, tales como el **grupo de objetos genérico** (Ilustración 2.30) o el **sólido heterogéneo** (Ilustración 2.31). Esta ventana permite un comportamiento muy genérico, como el cambio de la descripción, la visibilidad y la selección de un hijo.

En el caso de los sólidos heterogéneos permite configurar 2 propiedades: el **material** con el que se trabaja y la **resolución** con la que operará el sólido, por ejemplo, para renderizarlo con mayor o menor geometría.

2.3.6.13 Wireframe de Ventanas: Lights

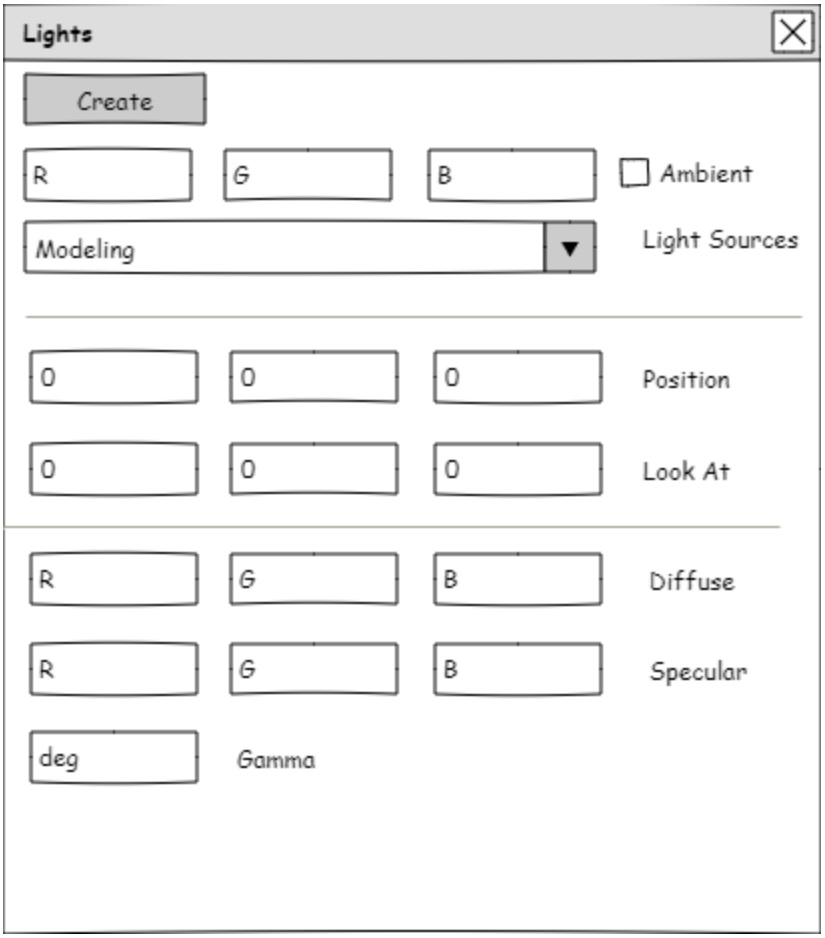


Ilustración 2.32 – Wireframe de Ventanas: Modeling Light

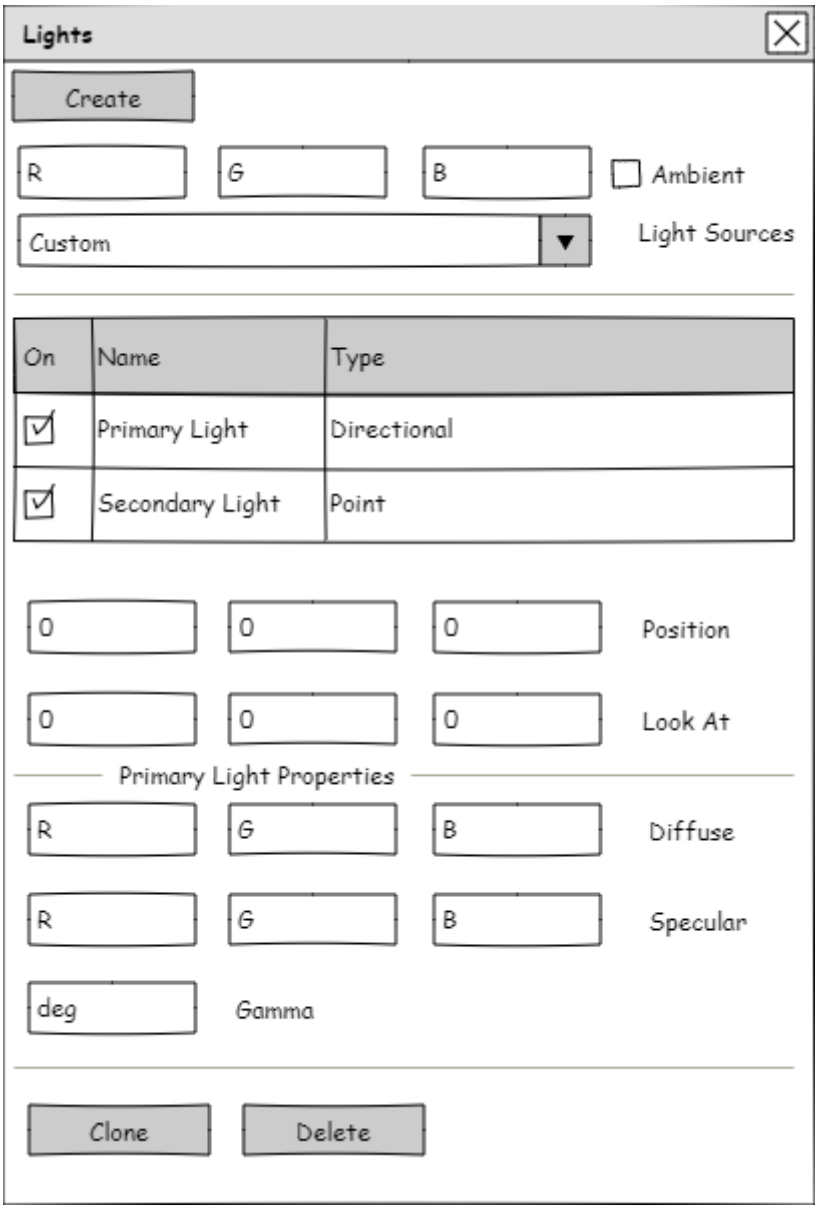


Ilustración 2.33 – Wireframe de Ventanas: Custom Lights

Dado que hay 2 modos de luces, de **modelado** (Ilustración 2.32) y **luces personalizadas** (Ilustración 2.33), se utilizará un combo box para cambiar de modo, pudiendo en caso de seleccionar la de modelado, cambiar sus propiedades, y en caso de seleccionar luces personalizadas, poder elegirlas y encenderlas o apagarlas desde una tabla.

Las propiedades que se podrán cambiar serán: el **tipo de luz**, la **posición**, **dónde apunta**, el **color difuso** y el **especular**, además del ángulo ***Gamma***, el cual lo utiliza la luz **focal** como ángulo de apertura del foco.

Para la **luz de modelado** no se podrán modificar ni la posición ni dónde apunta. Solamente servirá para inspeccionar el valor de sus atributos, dado que la luz de modelado se actualiza con la **posición de la cámara**.

2.3.6.14 Wireframe de Ventanas: Transform

Ilustración 2.34 – Wireframe de Ventanas: Transform

La ventana de *Transform* (Ilustración 2.34) cumple con dos objetivos:

- Seleccionar el **guizmo** de manipulación de la transformación que se va a usar en la escena.
- Modificar la transformación del objeto activo en caso de tenerlo, pudiendo cambiar tanto la **posición**, como la **rotación** y la **escala** de la transformación del objeto.

2.3.6.15 Wireframe de Ventanas: Materials

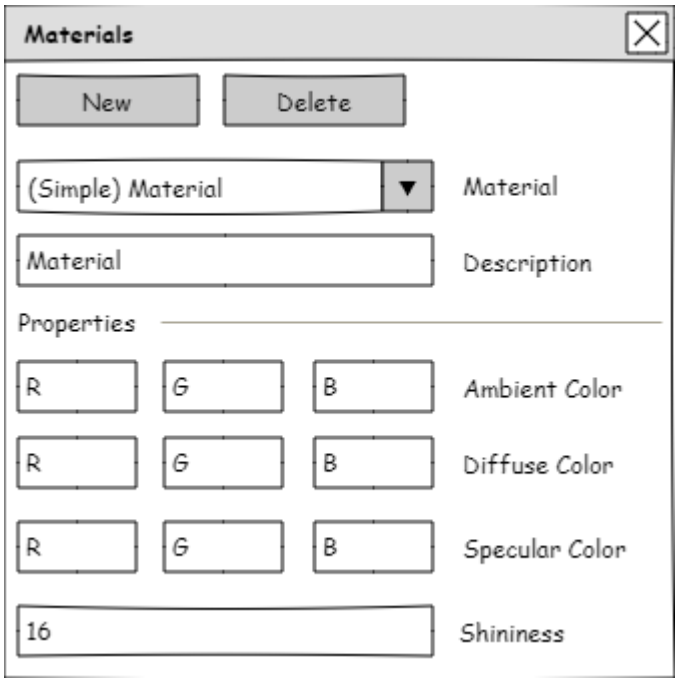


Ilustración 2.35 – Wireframe de Ventanas: Materials seleccionando un material

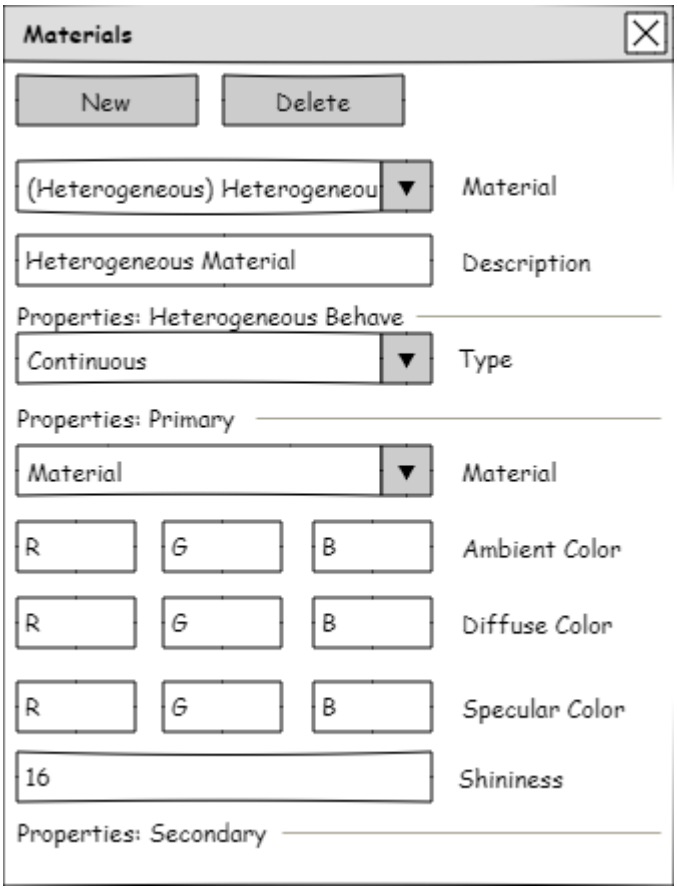


Ilustración 2.36 – Wireframe de Ventanas: Materials seleccionando un material heterogéneo

La ventana de edición de materiales permitirá elegir entre todos los materiales existentes en el contexto actual, tanto **simples** como **heterogéneos**. En el caso de los materiales simples (Ilustración 2.35), permitirá cambiar sus propiedades, mientras que en el de los heterogéneos (Ilustración 2.36), permitirá cambiar tanto el tipo de material heterogéneo, como los materiales primario y secundario, además de sus propiedades.

3 DESARROLLO

En este apartado se ahondará en la implementación de los elementos del sistema que tenga un especial interés.

3.1 Gestión de memoria

Dado que C++ por su naturaleza permite un control de la memoria tanto potente como versátil, pero también complejo y potencialmente propenso a fugas de memoria. A la hora de escribir el código es necesario especificar una serie de **reglas** para **prevenir fallos** y evitar el colapso de la aplicación, o en el peor de los casos identificar fácilmente el origen de estos.

En el caso de este proyecto se ha optado por trabajar con **punteros inteligentes** para la gestión de la **memoria dinámica**, optando por dejar los punteros básicos para contados casos. Para la mayoría de relaciones de agregación entre objetos con tiempos de vida similares se intentará utilizar referencias de C++, facilitando la API, y la comunicación entre clases.

```cpp
[[nodiscard]] virtual size_t childCount() const {
    return 0;
}
[[nodiscard]] virtual std::optional<std::reference_wrapper<SceneNode>> getChild(size_t index) const {
    return std::nullopt;
}
virtual std::optional<size_t> addChild(std::unique_ptr<SceneNode>& child) {
    return std::nullopt;
}
virtual std::optional<size_t> moveChild(SceneNode& child) {
    return std::nullopt;
}
/// @brief removes a child object an returns its ownership
/// @post under no circumstance the pointer might get out of scope
///         ensure once implemented to update every child index
/// @return an optional that might hold the unique object pointer
[[nodiscard]] virtual std::optional<std::unique_ptr<SceneNode>> removeChild(size_t index) {
    return std::nullopt;
}

[[nodiscard]] std::optional<std::reference_wrapper<SceneNode>> getNextSibling() const;
[[nodiscard]] std::optional<std::reference_wrapper<SceneNode>> getPreviousSibling() const;
[[nodiscard]] std::optional<std::reference_wrapper<SceneNode>> getObject() const;

/// @brief tries to retrieve the unique pointer holding the object ownership
/// @pre the object must have a parent object in the node tree
/// @post the pointer must not get out of scope under no circumstance
/// @return an optional that might hold the unique object pointer
[[nodiscard]] std::optional<std::unique_ptr<SceneNode>> getObjectOwnership();

/// @brief the node closest transform,
/// if the object has a transform it produces the same output as getOwnedTransform
[[nodiscard]] virtual std::optional<std::reference_wrapper<Transform>> getImmediateTransform();

/// @brief the node owned transform if it has one
[[nodiscard]] virtual std::optional<std::reference_wrapper<Transform>> getOwnedTransform() {
    return std::nullopt;
}
```

Ilustración 3.1 – Ejemplo de punteros inteligentes en la API del SceneNode

El **SceneNode** es un muy buen ejemplo del uso de punteros inteligentes como se ve en la Ilustración 3.1 dado que únicamente aquellos métodos como ***getObjectOwnership*** o ***removeChild*** que van a devolver el control del objeto, usan *unique_ptr*, mientras que los otros devuelven referencias dado que no retornan la posesión del objeto. Además, se utiliza el tipo **optional** para que sea más semántico y no se realicen comprobaciones de puntero nulo.

```cpp
void RemoveObjectCommand::execute() {
    if (auto obj = parent.removeChild(index)) {
        if (!obj.value()->unbindParent())
            spdlog::warn("RemoveObjectCommand : Unable to unbind parent from object");

        object = std::move(obj.value());
    } else
        spdlog::warn("Unable to execute RemoveObjectCommand : Index found no child");
}

void RemoveObjectCommand::undo() {
    if (object == nullptr) {
        spdlog::warn("Unable to undo RemoveObjectCommand : No object");
        return;
    }

    if (auto idx = parent.addChild(object))
        index = idx.value();
    else
        spdlog::warn("Abnormal AddObjectCommand undo : Insertion returned no index");
}
```

Ilustración 3.2 – Ejemplo de gestion del ciclo de vida de un objeto por un comando

La Ilustración 3.2 muestra como una clase que implementa la interfaz comando (**clase abstracta pura**), está encargado de eliminar objetos de la escena.

Funciona eliminando el objeto del padre, para lo cual toma la posesión del objeto, y en caso de ser deshecho, devuelve el hijo al padre, perdiendo la posesión de este. Mediante este sencillo mecanismo se mantiene siempre el objeto o bien en el **árbol de escena** o bien en algún otro **comando** (no solamente el comando de eliminar, el de crear tiene un comportamiento similar). Por tanto, para eliminar los recursos solo habría que limpiar la pila del **historial de comandos**, eliminando así el comando y los recursos que mantiene.

3.2 Comandos

Mediante comandos se han satisfecho todas las operaciones sobre los sólidos y el árbol de la escena incluyendo: modificaciones sobre el transformador, movimiento

de nodos, cambios en la visibilidad de los objetos, cambio de las descripciones, edición de los coeficientes, y operaciones sobre los nodos de mecanizado.

```
template <class C>
void CommandHistory::execute(std::unique_ptr<C> command) {
    static_assert(std::is_convertible<C*, Command*>::value);
    if (offset) {
        history.resize(history.size() - offset);
        offset = 0;
    }
    if (!history.empty()) {
        auto* previous = dynamic_cast<MergeCommand<C>*>(history.back().get());
        if (previous && previous->merge(*command)) {
            spdlog::debug("Doing merged command: {}", command->getDescription());
            history.back()->execute();
            if (!history.back()->doDiscard())
                executionSubject.updateSubject(CommandHistoryEvent::DoMerge, *history.back());
            else {
                auto item = std::move(history.back());
                history.pop_back();
                executionSubject.updateSubject(CommandHistoryEvent::Discard, *item);
            }
            return;
        }
    }
    spdlog::debug("Doing command: {}", command->getDescription());
    command->execute();
    if (!command->doDiscard()) {
        history.push_back(std::move(command));
        executionSubject.updateSubject(CommandHistoryEvent::Do, *history.back());
    } else executionSubject.updateSubject(CommandHistoryEvent::Discard, *command);
}
```

Ilustración 3.3 – Captura de la ejecución de un comando en el historial

En la Ilustración 3.3 se puede ver cómo funciona la ejecución de un comando, la cual comprueba si el historial tiene elementos deshechos (**offset**), los cual los descartaría, también comprueba si hay algún comando anterior, y si este puede **fusionarse** con el comando actual. Una vez se ejecute el comando, sea por fusión o no, se le preguntará si se puede descartar o no, en caso de ser una operación que tenga nulo efecto. Cabe destacar que por medio del patrón Observer se notifican todos estos cambios con el **CommandHistoryEvent** correspondiente.

3.3 Creación de contextos

```
unsigned int Context::ContextIdCounter = 0;

Context::Context(IO& io) :
    contextId(ContextIdCounter++),
    scene(std::make_unique<Scene>()),
    renderer(std::make_unique<Renderer>(io.getShaderManager())),
    renderProfile(std::make_unique<RenderProfile>()),
    commandHistory(std::make_unique<CommandHistory>()),
    macroSet(std::make_unique<MacroSet>()),
    snapshot(std::make_unique<Snapshot>(*Context::renderer, HDReady, io)),
    materialEditor(std::make_unique<MaterialEditor>()),
    sceneStatus(std::make_unique<SceneStatus>()),
    sceneTools(std::make_unique<SceneTools>()),
    selector(std::make_unique<Selector>(*Context::scene, *Context::materialEditor, *Context::commandHistory)),
    eventManager(std::make_unique<EventManager>(*Context::scene, *Context::selector, *Context::commandHistory)),
    document(std::nullopt),
    io(io)
```

Ilustración 3.4 – Captura del constructor del contexto

En la Ilustración 3.4 se muestra cómo se construye el **contexto**, el cual da a los elementos con los que se interactúa el acceso a todo el modelo de datos con el que se está trabajando. El contexto solamente necesita en su constructor que se le pase el objeto **IO**, que compartirá con el resto de contextos para trabajar con la entrada/salida. La totalidad de los objetos que se almacenen en el contexto tendrán un **tiempo de vida superior o igual al contexto**, diseñado así para que se puedan **referenciar entre ellos** sin problema, permitiendo un sistema versátil a la hora de añadir nuevos elementos al **modelo** que dependan de otros.

3.4 Renderizado

```
void Renderer::renderSurfaces(const RenderProfile& profile, const Scene &scene) {
    if (!profile.isShowSurfaces()) return;

    RenderContext renderContext(*shaderManager.requireShaderProgram("surface"));
    renderContext.setSelection(&scene.getSelectedObject());
    renderContext.setViewMatrix(scene.getCamera().view());
    renderContext.setProjectionMatrix(scene.getCamera().projection());

    renderContext.setRenderMode(RenderMode::Surfaces);
    renderContext.setDetailedRendering(profile.isDetailedSurfaces());

    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    auto it = scene.getLights().begin();
    if (it < scene.getLights().end()) {
        it->get().apply(renderContext);
        scene.getRoot().getRenderer()(renderContext);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE);
        for (; it < scene.getLights().end(); it++) {
            it->get().apply(renderContext);
            scene.getRoot().getRenderer()(renderContext);
        }
    }
}
```

Ilustración 3.5 – Captura del proceso de renderizado: renderizado de superficies

Dado que el código del **Renderer** es bastante extenso, en la Ilustración 3.5 se muestra uno de los métodos más pequeños, el cual muestra cómo se renderizan las superficies de los objetos. Existen además de este, otros métodos encargados del renderizado de líneas y vértices, pero este es el más particular dado que utiliza las **luces** de la escena para renderizar el **árbol** por cada luz.

Se puede ver como recibe la **escena** y el **perfil de renderizado**, y una vez comprobado que se quieren mostrar las superficies, busca el shader para crear el contexto, el cual rellenará con las matrices de la cámara, el modo de renderizado (superficies) y otros parámetros como el objeto seleccionado y si quiere un renderizado detallado.

Por medio de la jerarquía de objetos, se les pasará el contexto y sabrán como han de dibujarse. Por último, se iteran todas las luces, se **aplica cada luz** al contexto y se **dibuja** todo el árbol.

3.5 Capturas de pantalla

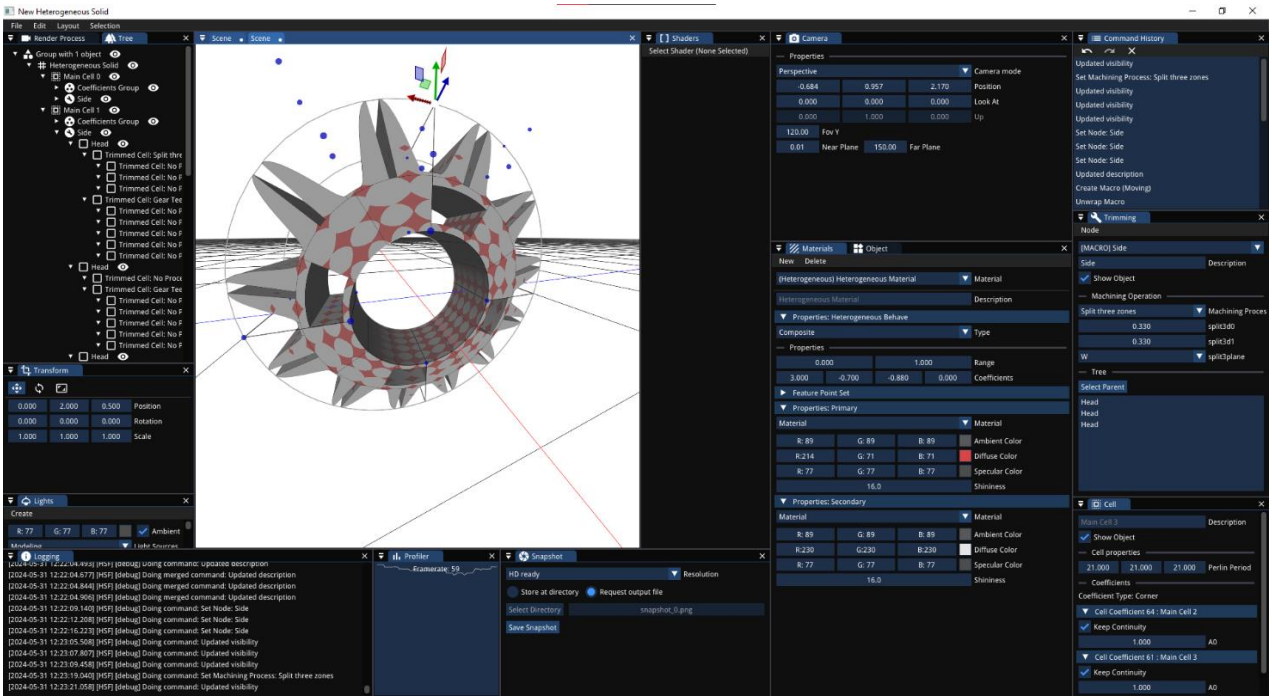


Ilustración 3.6 – Captura de la interfaz

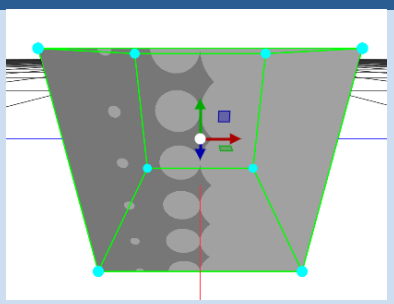
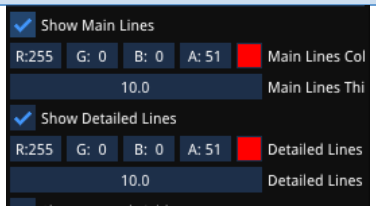
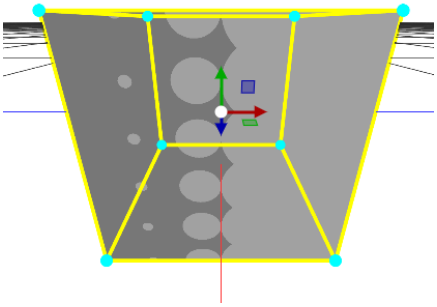
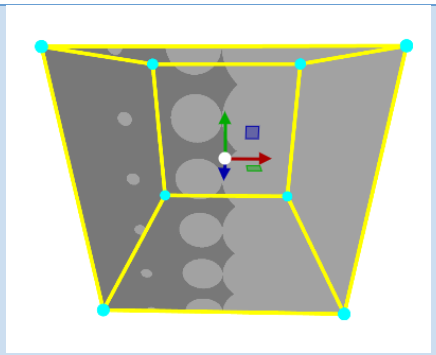
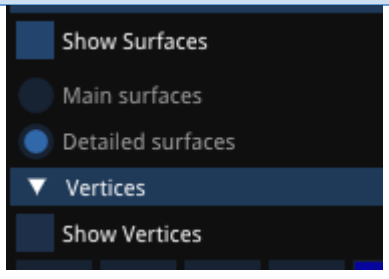
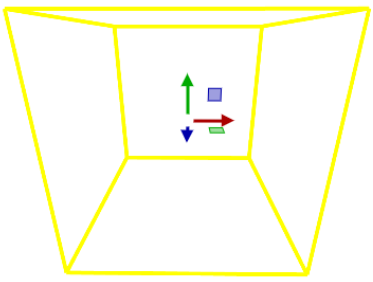
En la Ilustración 3.6 se puede ver el resultado final de la interfaz mientras se edita un sólido.

4 PRUEBAS FINALES

Las pruebas se harán directamente sobre la aplicación final para asegurarnos de que las operaciones funcionan como se esperan, así como que no hay ningún error inesperado ni ningún fallo a nivel de memoria. Para la inclusión en este documento se seleccionará una serie de pruebas respecto a las funcionalidades principales evitando así extender demasiado la memoria, tales como el renderizado, el historial de comandos, el guardado o el tratamiento de los sólidos heterogéneos.

4.1 Pruebas de edición de las propiedades de renderizado

Para esta prueba modificaremos algunas propiedades y veremos si los cambios se ven aplicados en tiempo real sobre el modelo.

Cambio	Propiedades	Visualización
Por defecto	No se ha cambiado nada aun	
Cambiar color y tamaño de líneas. Comportamiento esperado: Se verán amarillas dado que están seleccionadas y por tanto se mezclan ambos colores (color de línea y color de selección).		
Deshabilitar la malla. Comportamiento esperado: deja de verse la malla que indica la altura.		
No mostrar vértices ni superficies. Comportamiento esperado: dejan de verse las caras y los vértices del hiperparche.		

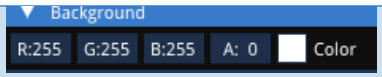
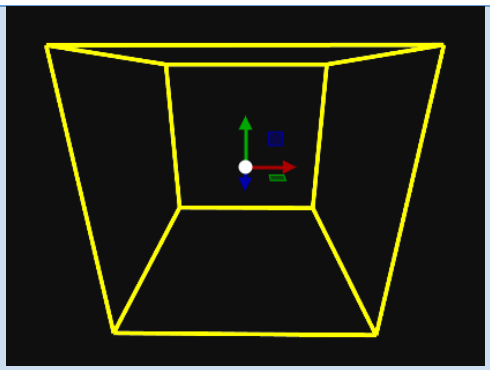
Fondo transparente. Comportamiento esperado: se ve un fondo oscuro (no se puede renderizar un color de fondo sin transparencia, pero es útil para las capturas).		

Tabla 4.1 – Tabla de modificaciones en las propiedades de renderizado

Ilustración 4.1 – Captura de un cubo en modo malla

Después de realizar los cambios mostrados en la Tabla 4.1, se puede realizar una captura para obtener el resultado en modo malla que se ve en la Ilustración 4.1.

4.2 Pruebas de edición de los shaders

Para comprobar el funcionamiento de los shaders se editará un shader de forma errada y de forma correcta. Es decir, forzaremos tanto la situación en la que el cambio produzca un error en la compilación, como otra en la que el resultado de la modificación del shader sea correcto.

4.2.1 Pruebas de shader que no compila

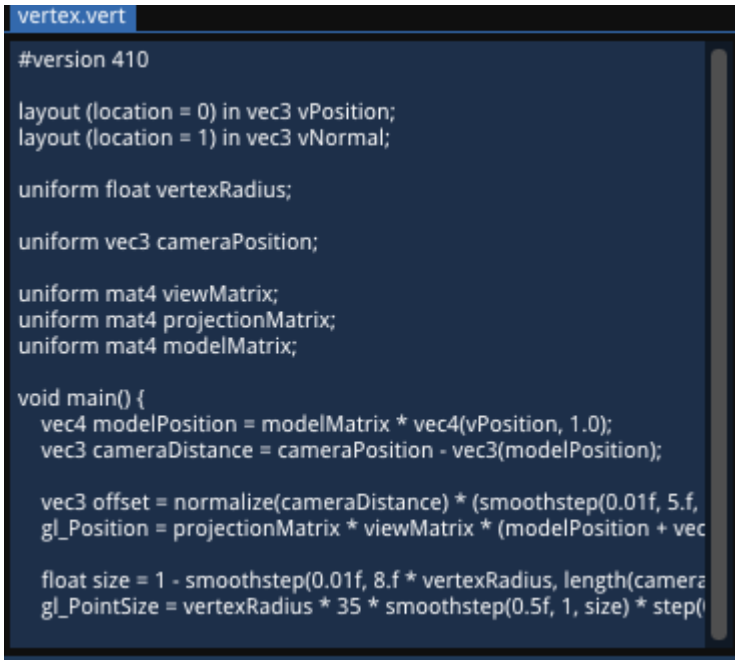


Ilustración 4.2 – Captura editor de shaders

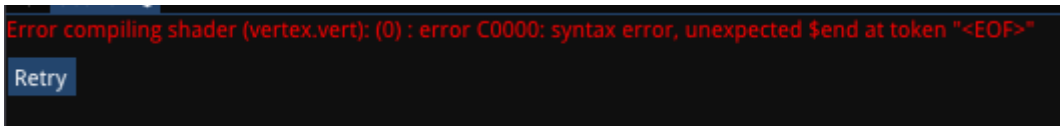


Ilustración 4.3 – Captura error al renderizar la escena

En la Ilustración 4.2 se modifica el código fuente eliminando un corchete para forzar un error de compilación, el cual se puede ver indicado como es de esperar en la Ilustración 4.3.

4.2.2 Pruebas de shader que compila correctamente

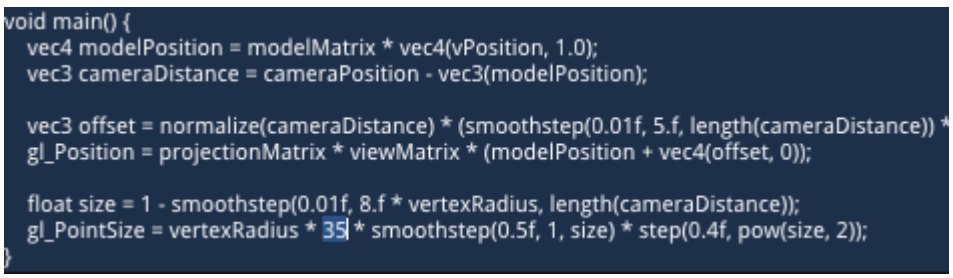


Ilustración 4.4 – Captura editor de shaders

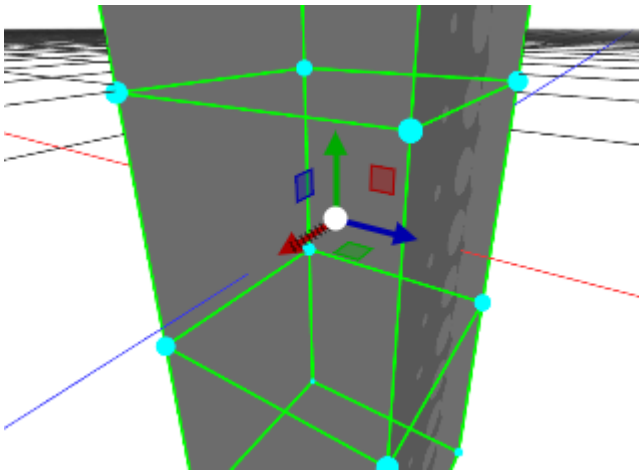


Ilustración 4.5 – Captura de la escena antes de editar el tamaño de los vértices

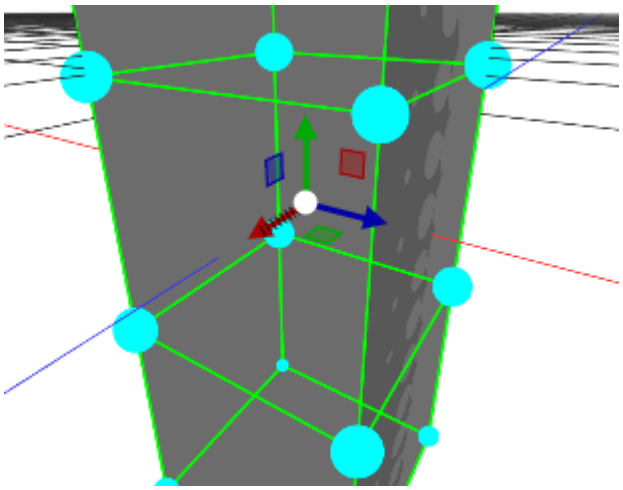


Ilustración 4.6 – Captura de la escena después de editar el tamaño de los vértices

Como se puede apreciar en la Ilustración 4.4, modificamos la constante que multiplica el radio del vértice, de forma que los puntos se vean más grandes. La Ilustración 4.5 muestra cómo se ven los vértices antes de editar el código fuente y la Ilustración 4.6 muestra como se ha actualizado correctamente el tamaño.

4.3 Pruebas de operaciones sobre el árbol

En este apartado comprobaremos si los nodos se pueden mover correctamente entre ellos, así como si las transformaciones se anidan correctamente.

4.3.1 Movimiento de los nodos del árbol

En primer lugar, crearemos bajo el nodo raíz dos sólidos, uno con un material gris y 3 celdas (Ilustración 4.7) y otro vacío y con un material rojo. Si movemos una celda de un sólido a otro debería cambiar su color, pero no su posición.

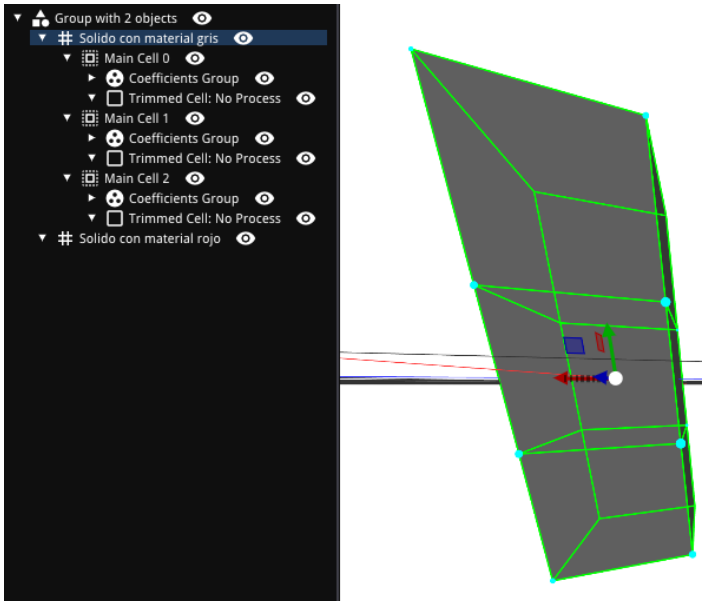


Ilustración 4.7 – Captura de la escena antes de mover una celda

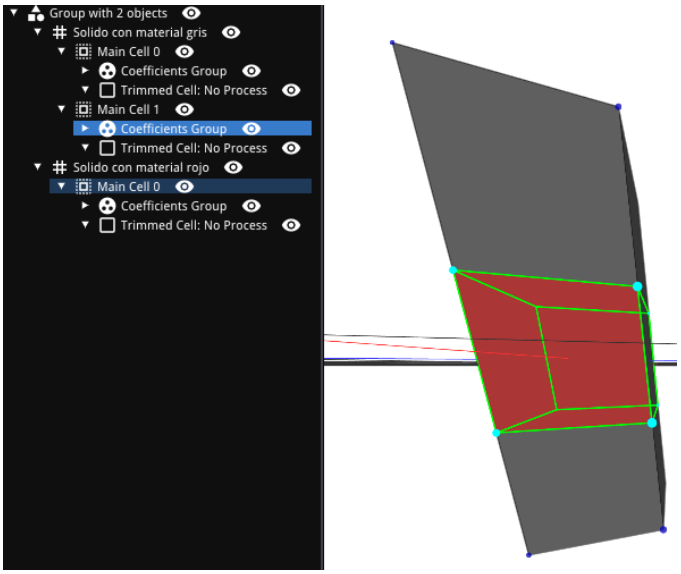


Ilustración 4.8 – Captura de la escena después de mover una celda

Como se puede comprobar, la celda mantiene su posición, pero su color ha cambiado, dado que ahora se está renderizando desde el sólido con el material rojo (Ilustración 4.8).

4.3.2 Aplicación de transformaciones anidadas

Este caso es similar al anterior, dado que vamos a trabajar con el árbol, pero en este caso trabajaremos con las transformaciones.

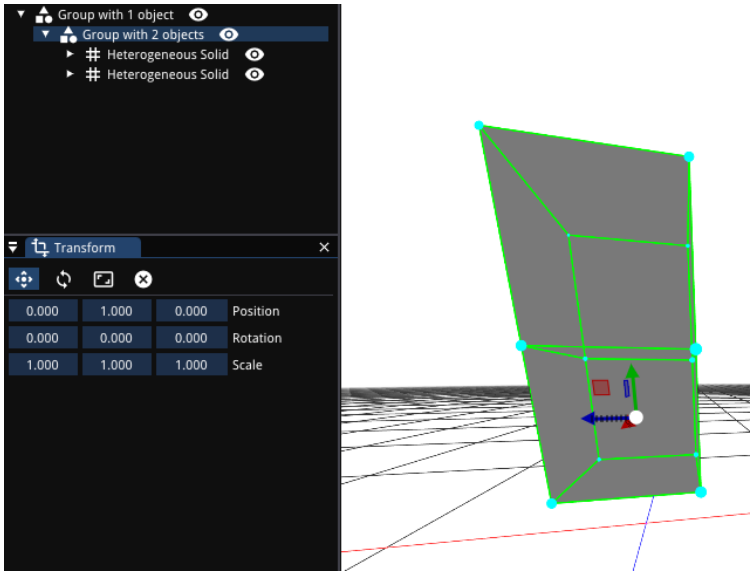


Ilustración 4.9 – Captura de la escena con estructura de transformaciones

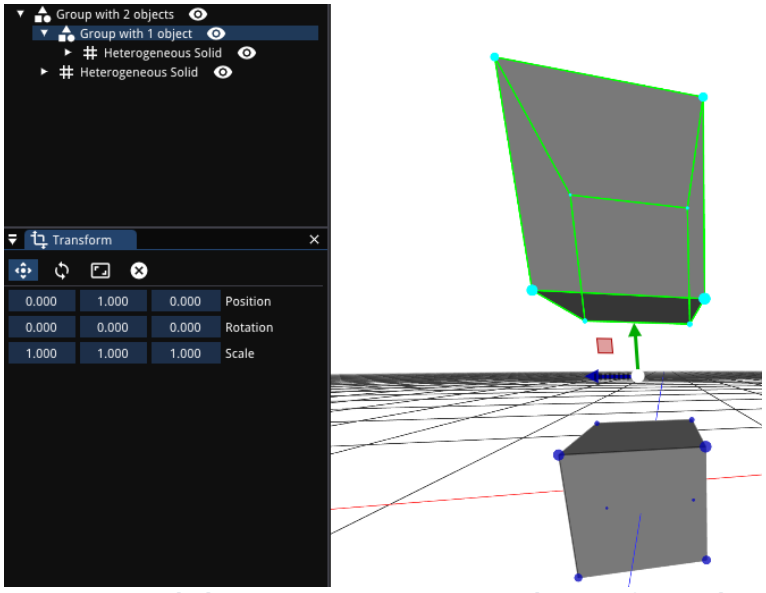
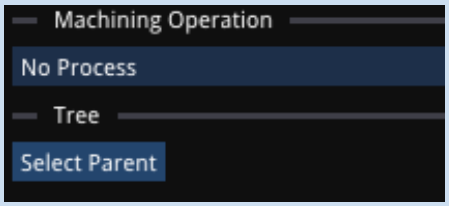
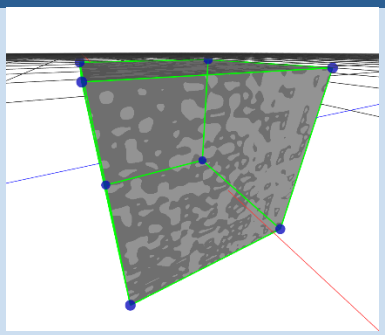
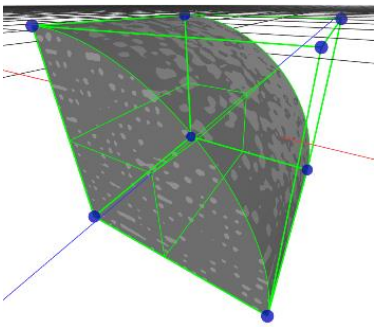
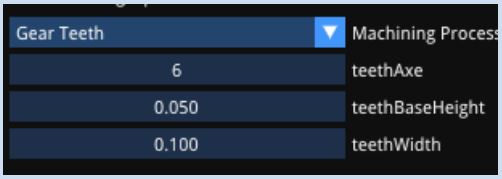
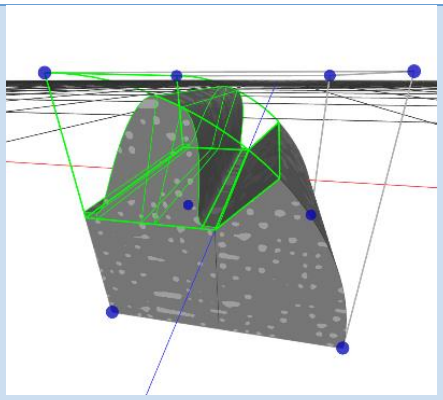


Ilustración 4.10 – Captura de la escena con estructura de transformaciones modificada

En la Ilustración 4.9 se puede ver cómo hay dos solidos (cada uno con una celda) los cuales se encuentran dentro de un grupo con un desplazamiento de 1 en la coordenada **Y**. Por otra parte, en la Ilustración 4.10 se ha extraído el sólido inferior fuera del grupo, **dejando de aplicar ese desplazamiento**, y por tanto colocando el mismo en el origen de coordenadas.

4.4 Pruebas de operaciones de manufacturado

En este caso realizaremos una serie de operaciones de manufacturado y las anidaremos las unas en las otras para ver si se comportan correctamente. Por último, crearemos una macro y la aplicaremos a otra celda (Tabla 4.2).

Interfaz subcelda	Explicación	Resultado
	Se crea una celda	
	Se aplica una operación de redondeado sobre un lado del hiperparche	
	Se aplica una operación de diente de sierra a una celda hija	

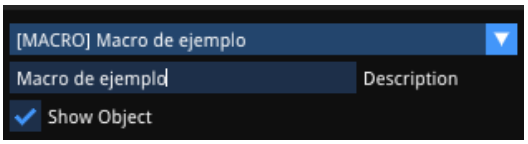
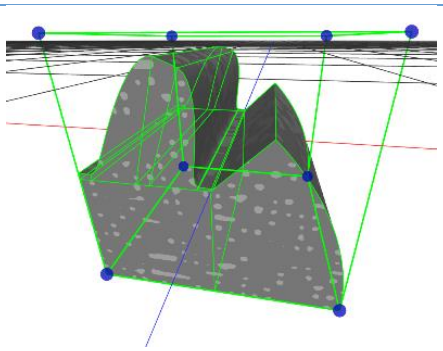
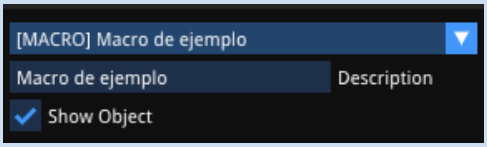
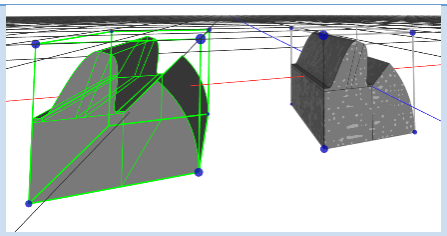
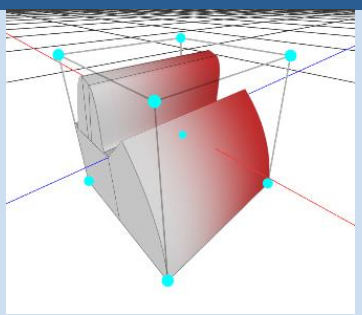
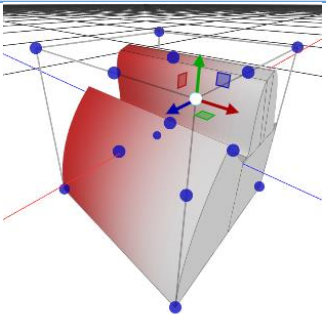
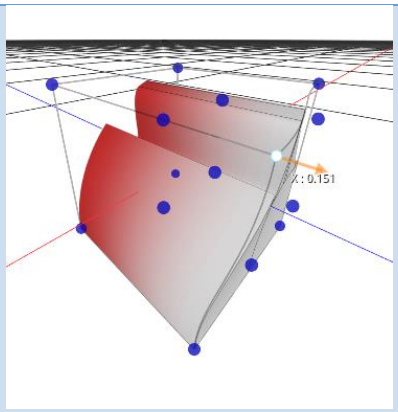
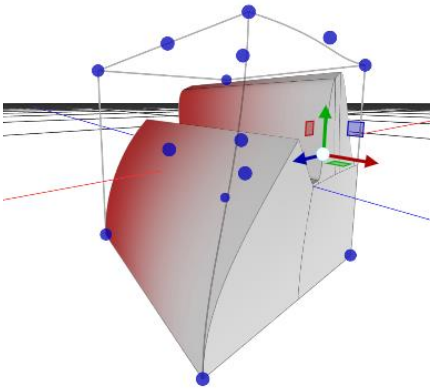
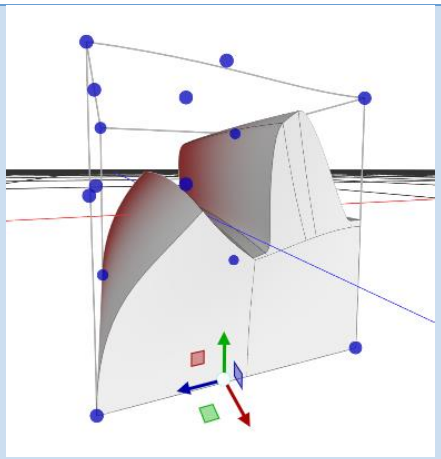
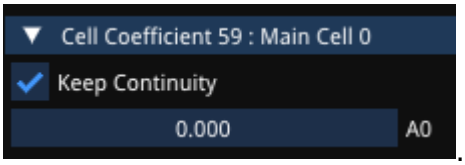
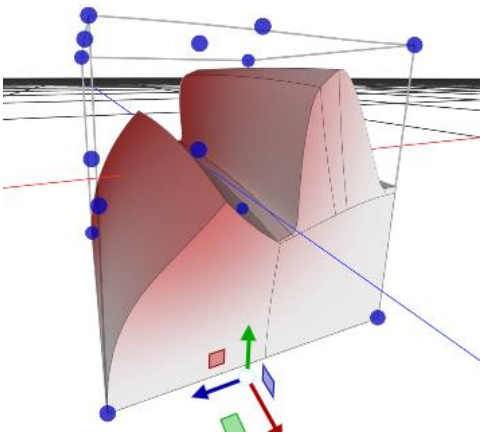
	Se convierte la primera celda en una macro	
	Aplicamos a otro nodo la macro creada	

Tabla 4.2 – Tabla de operaciones de manufacturado

4.5 Pruebas de modificación de coeficientes

Partiendo del primer objeto modelado en el apartado anterior, modificaremos sus coeficientes para ver los resultados que se obtienen. En este caso utilizaremos el material **continuo** para apreciar mejor los cambios. Los cambios que ser realizarán serán: **modificar la posición de un corner point**, **de su vecino**, modificar los coeficientes de material y cambiar **la continuidad** del material (Tabla 4.3).

Explicación	Resultado
Punto de partida	
Selección de un corner point	

Desplazamos el corner point	
Seleccionamos un vecino del corner point	
Desplazamos el coeficiente seleccionado hasta abajo para curvar la región	
Cambiamos la proporción de material, antes estaba a 1, y ahora la establecemos a 0.  . Dado que esta sincronizado cambian todos los vecinos del corner point.	

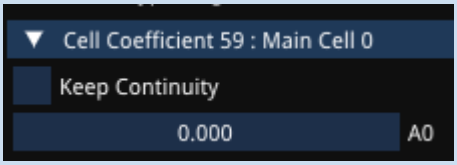
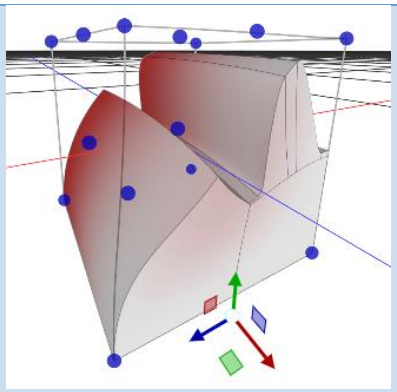
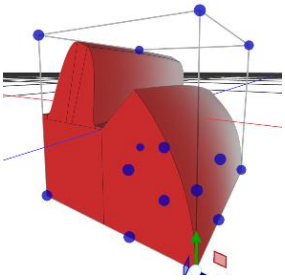
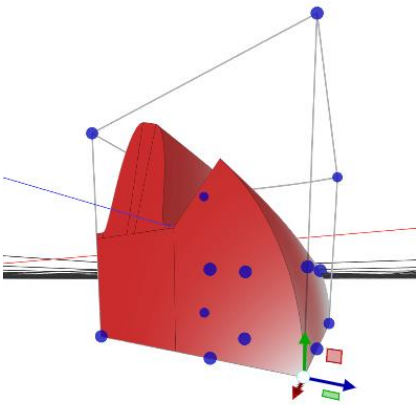
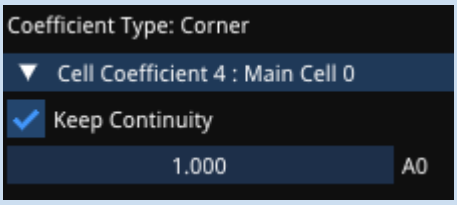
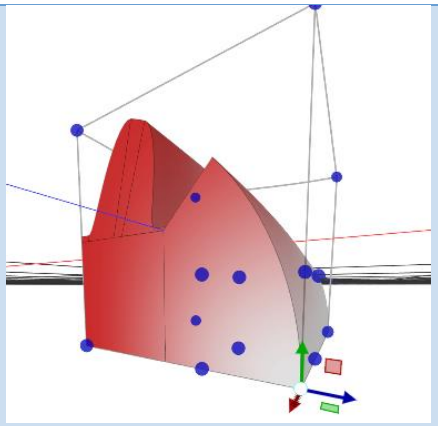
Repetimos el proceso desactivando la continuidad del vecino del corner point que estamos editando. 	
Cambio de la continuidad en un corner point diferente. 	
Por último, aplicamos de nuevo la continuidad en el corner point. 	

Tabla 4.3 – Tabla de modificación de coeficientes

4.6 Pruebas del historial de cambios

En este caso se intentará aplicar la mayor cantidad de operaciones posibles, las cuales, por supuesto tengan soporte para el historial de comandos, de forma que se apilen en el historial y puedan deshacerse y rehacerse. El objetivo es comprobar que haciendo/deshaciendo los comandos de manera arbitraria el historial funciona correctamente, los cambios se ven reflejados en tiempo real y de manera correcta, así como no ocurre ningún error en la aplicación a nivel de memoria.

Después de numerosas pruebas y haber trabajado intensivamente con el software utilizando el historial, parece ser tan **robusto y predecible** como se esperaba.

4.7 Pruebas de carga y guardado

Para probar que la carga y el guardado funcionan correctamente, realizaremos todas las modificaciones posibles sobre un sólido, de forma que, al guardarlo y posteriormente cargarlo, podamos comprobar que todos estos cambios se han mantenido.

Los resultados de esta última prueba son bastante satisfactorios, dado que todos los elementos configurados en el **serializador** se guardan **correctamente**. Por tanto, si en un futuro se desea guardar más información, es perfectamente viable **extenderlo a más objetos**.

5 CONCLUSIONES Y TRABAJOS FUTUROS

5.1 Conclusiones

Después de realizar este TFG, puedo concluir que los resultados han sido bastante buenos, teniendo un producto software realmente flexible a cambios en la interfaz, como se esperaba, y muy potente en cuando a variedad de opciones y transformaciones a poderse realizar sobre los sólidos heterogéneos y sus materiales.

En lo personal, he aprendido a trabajar con un volumen de código que anteriormente no había manejado, teniendo que hallar la solución a diversos retos en cuanto al diseño e implementación del proyecto para conseguir un resultado que cumpliese con las expectativas. Además, he trabajado con el modelo de sólidos heterogéneos, el cual me ha gustado bastante en cuanto a arquitectura y posibilidades, teniendo un gran futuro y habiendo aprendido de las razones detrás de su diseño tanto de forma teórica como práctica.

5.2 Mejoras propuestas

A continuación, se enumerarán los elementos que en mi opinión pueden ser mejorados dentro del sistema para que el software alcance un estado más maduro y resiliente al paso del tiempo.

5.2.1 Continuidad de los sólidos heterogéneos

La continuidad a nivel de material es bastante flexible. Sin embargo, a nivel de forma solo fuerza el movimiento de los corner points sobre sus vecinos. Además de no implementar todos los posibles fallos de continuidad, no tiene ninguna forma de desactivar la corrección de la continuidad por parte del software, a diferencia de los coeficientes de materiales, que sí permiten no seguir la continuidad.

5.2.2 Renderización de los vértices

La configuración actual no permite colorear los vértices de forma diferente según su tipo, así como no permite dibujarlos según alguna política especifica.

El dibujado actual solo permite variar el tamaño según la distancia a la cámara, y el criterio para dibujarlos es mostrar los corner point y en caso de seleccionar uno de ellos, mostrar sus vecinos. Por tanto, resultaría conveniente implementar más opciones a la hora de dibujar los vértices.

5.2.3 Luces

Las luces han sido diseñadas de forma similar a como estaban implementadas en el antiguo software. Por tanto, no se benefician de la estructura de árbol y transformadores. Una posible mejora en un futuro sería crear un nodo para las luces y así poder editarlas gráficamente siguiendo el mismo sistema que el resto de elementos, aplicando transformadores.

5.2.4 Uso de múltiples perfiles de renderizado

La modificación de las opciones de rendering resulta muy cómoda a los usuarios para adaptar la forma en que ven la escena en uso. Sin embargo, puede quedarse corta la edición individual de cada una de las propiedades cada vez que se quiera ver de otro modo la escena.

La solución a esto sería dar la posibilidad de elegir entre diferentes perfiles a la hora de visualizar, pudiendo modificar los unos independientemente de los otros. De esta forma se podría asignar inclusive un perfil a la captura de pantalla de forma que, si por ejemplo se quiere visualizar un fondo pero que al realizar una captura sea transparente y eliminando vértices y líneas, no se tengan que manipular todas las opciones en el momento anterior a realizar la captura, sino simplemente hacer la captura con el perfil ya configurado anteriormente.

Por último, en lo que a perfiles de renderizado se refiere, para mayor comodidad deberían de poder ser almacenados en disco, y dado que son preferencias de usuario, ser almacenados de manera independiente al modelo, para poder ser reutilizados las veces que sean necesarios.

5.2.5 Mejora de los atajos de teclado

Para mejorar la experiencia de usuario, los atajos de teclado deberían de poder ser configurados por este, así como ser almacenados como preferencias. En cuanto al uso de menús, a día de hoy **Dear ImGUI** no permite escuchar eventos de teclado de forma nativa. Por tanto, o bien puede ser programada la escucha de los atajos de los menús o esperar a que esta característica se publique, la cual se encuentra en desarrollo [13].

5.2.6 Exportar los sólidos heterogéneos

Además de permitir el guardado para el posterior uso dentro del mismo programa, en caso de necesitar compatibilizar los sólidos heterogéneos con otro software sería necesario trabajar en una API para exportarlos en diferentes formatos, tales como una nube de puntos, o un .obj, además de formatos más específicos, dependiendo de las aplicaciones concretas futuras que se les den.

5.3 Líneas de trabajo futuras

5.3.1 Optimización del modelo de sólidos heterogéneos

El renderizado y la manipulación de los sólidos heterogéneos es una tarea que requiere el cálculo constante de la geometría de los mismos. Este es un proceso

costoso que requiere calcular todos los puntos en el dominio de toda la jerarquía de subceldas y la celda, realizando gran cantidad de operaciones por fotograma en el que se modifique, desde el movimiento de un coeficiente, al ajuste de un parámetro.

Existen varias opciones para realizar esto, como, por ejemplo:

- **Guardar en cache** las **coordenadas** en la jerarquía, parando el cálculo de las funciones de Bézier lo antes posible en la jerarquía de celdas. Esta solución aprovecharía que al manipular un parámetro las coordenadas se moverían en una **vecindad** similar a la del instante anterior.
- **Paralelizar** el cálculo de las celdas, de forma que el cálculo se haga sin afectar al hilo principal y una vez listo se **sincronice** para mandarlo a la GPU y que se renderice. Esta solución es algo más compleja, dado que si no se realiza una buena concurrencia, los resultados podrían empeorar.
- La última opción planteada consiste en **saltar ciclos**, evitando mandar a la CPU el cálculo de toda la geometría para el instante siguiente **descartarla** y volver a calcularla. Este método puede ser el más sencillo de implementar, pero si se abusa puede provocar **saltos bruscos** al mostrar el objeto en pantalla.

La selección, adaptación e implementación de estas opciones sigue siendo una línea de trabajo abierta, la cual permitiría trabajar en tiempo real con modelos más complejos, manteniendo intacta la experiencia de usuario.

5.3.2 Implementación de modificadores uniformes

Los modificadores uniformes van en la línea del apartado anterior dado que su propósito es en parte reducir el cálculo. La idea consiste en, en lugar de aplicar sucesivas operaciones de **manufacturado** para obtener el resultado esperado, aplicar una operación y el resultado multiplicarlo por una **matriz** que represente las deformaciones deseadas.

Se denomina de esta manera dado que, gracias a las propiedades de las matrices, se podrían apilar varias operaciones parametrizadas como la **multiplicación de matrices**, que después se multiplicarían sobre todos los coeficientes generados **de forma uniforme**, es decir, sería realizar **64** multiplicaciones, lo cual es menos

costoso que la mayoría de operaciones de mecanizado, además de que quedaría todo en un nivel. Por tanto, no habría que calcular todas las coordenadas finales de la geometría a través del dominio de sus celdas superiores.

La idea no es reemplazar las operaciones de manufacturado, sino ser una **herramienta más** para modelar sólidos heterogéneos, que usada correctamente facilitaría el modelado y haría el cálculo más rápido.

5.3.3 Cálculo de propiedades físicas de los materiales

En este proyecto no se ha incluido ni tratado el cálculo de **propiedades físicas** resultantes de la combinación de las propiedades de los varios materiales básicos combinados en los materiales heterogéneos. Es por ello que para trabajarlas dentro del editor sería necesario incluir una nueva ventana, las estructuras convenientes el modelo y diseñar la forma en la cual se visualizarían. Por tanto, esta queda como una línea de trabajo abierta.

6 APÉNDICES

6.1 Guía original del Trabajo Fin de Título

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior de Jaén

PROPUESTA DE TRABAJO DE FIN DE GRADO

GRADO EN:
Grado en Ingeniería Informática

ESPECIALIDAD:
Tecnologías de la Información

MENCIÓN:
General

TÍTULO DEL TRABAJO:
Diseño e implementación de un editor de sólidos heterogéneos

Idioma:
Castellano

DESCRIPCION CORTA DEL TFG:
El trabajo consistirá en el análisis, diseño e implementación de una aplicación de escritorio multiplataforma (al menos Windows/Linux) para la creación y edición de sólidos heterogéneos.

DATOS DEL TRABAJO FIN DE GRADO:

Modalidad del TFG

Proyecto de Ingeniería

Tipo de TFG

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

TFG en equipo

- ☐ **TFG en Equipo**

(Debe ajuntarse memoria justificativa)

Número máximo de estudiantes

1

TUTOR DEL TRABAJO FIN DE GRADO:

Tutor/a

Nombre:
ANGEL LUIS GARCIA FERNANDEZ

Departamento:
Informática

A. Conocimiento:
LENGUAJES Y SISTEMAS INFORMÁTICOS

☑ **Este TFG tiene un segundo tutor**

- ◉ **El tutor es de la Escuela**
- ○ **El tutor es ajeno al Centro**

Segundo Tutor

FRANCISCO DE ASIS CONDE RODRIGUEZ

Datos del Alumno Asignado

Alumno/a asignado/a

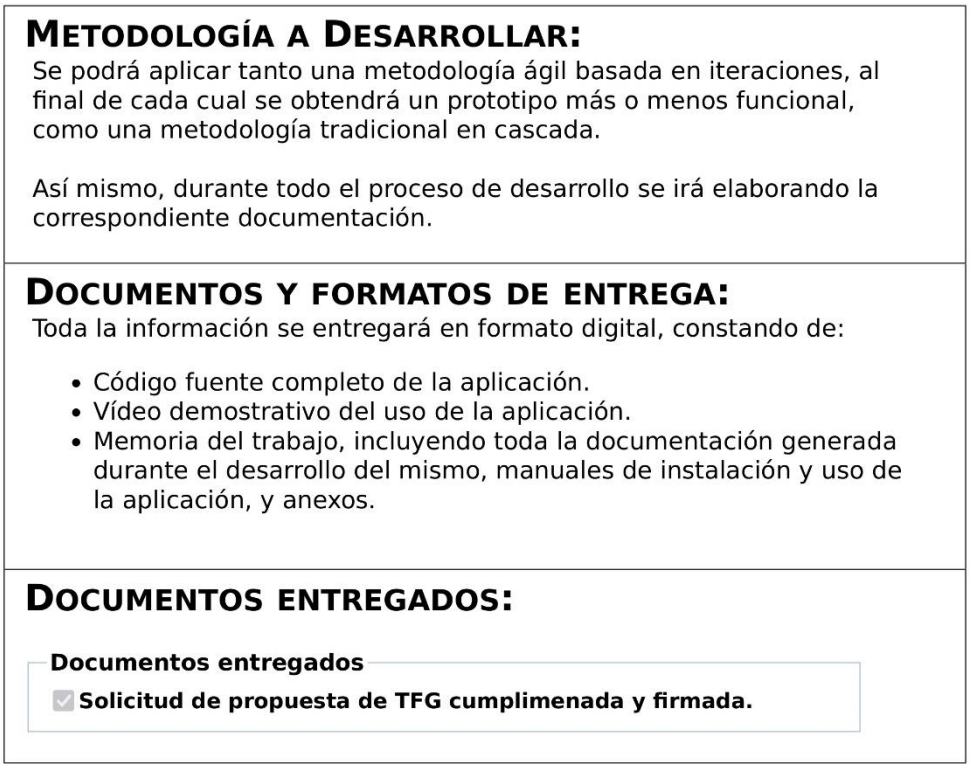
Nombre:
Alberto Elorza Rubio
DNI:
26523985V
Dirección:
C/ Agua 7, 3. 23710 Bailén
Teléfono:
654741625
Correo-E:
aer00015@red.ujaen.es

Conocimientos/Requisitos Previos Recomendados

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

Objetivos del TFG:

- Poner en práctica los conocimientos y destrezas adquiridos durante los estudios de grado.
- Demostrar el nivel de madurez para el diseño e implementación de software.
- Producir un software funcional y eficiente, que pueda instalarse y usarse en computadoras de usuarios finales.
- Obtener un software para asistir al diseño de sólidos heterogéneos, en sus distintas variantes (FGM, composites,...).
- Crear un framework extensible, que permita ampliar la aplicación de acuerdo a futuros requisitos.



METODOLOGÍA A DESARROLLAR:

Se podrá aplicar tanto una metodología ágil basada en iteraciones, al final de cada cual se obtendrá un prototipo más o menos funcional, como una metodología tradicional en cascada.

Así mismo, durante todo el proceso de desarrollo se irá elaborando la correspondiente documentación.

DOCUMENTOS Y FORMATOS DE ENTREGA:

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

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

DOCUMENTOS ENTREGADOS:

Documentos entregados

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

6.2 Instalación y configuración del sistema

El sistema consta de un directorio con 3 elementos:

1. **Ejecutable** del programa.
2. Archivo **.ini** que indica la configuración inicial de ImGUI (si se elimina las ventanas aparecerán desordenadas).
3. **Directorio** con los recursos del proyecto, sin estos no se podría hacer nada dado que contiene los shaders para visualizar la escena.

Una vez se tiene instalado el directorio con todos los elementos (copiar la carpeta en el lugar que se desee), se puede ejecutar el programa sin problema. En caso de **Linux** requeriría de dar permisos al ejecutable, pero por lo demás funcionaría igual.

6.3 Manual de usuario

6.3.1 Menú principal

El menú está dividido en los siguientes apartados:

- **File.** Trabaja con archivos.

  - **New.** Nuevo archivo.
  - **Open.** Abrir archivo.
  - **Save.** Guardar.
  - **Save as.** Guardar como nuevo archivo.
- **Edit.** Se trata de atajos a las acciones del historial de comandos.
  - **Undo.** Deshacer.
  - **Redo.** Rehacer.
  - **Clear History.** Limpiar historial.
- **Layout.** Permite trabajar con las ventanas.
  - **Toggle.** Despliega a su vez un submenú de ventanas para activar/desactivar su visibilidad.
- **Selection.** Trabaja con el objeto seleccionado en la escena.
  - **Visible.** Permite activar/desactivar su visibilidad.
  - **Add …** Dependiendo del objeto, deja añadir nuevos nodos hijos.

6.3.2 Control de la escena

La escena cuenta con los 3 botones principales del ratón para realizar el movimiento de la cámara y la selección, además de diversos atajos de teclado para completar el conjunto de herramientas en cuanto a navegación grafica por la escena se refiere. Los controles son:

- **Botón izquierdo del ratón**. Movimiento orbital de la cámara.
- **Botón derecho del ratón**. Selección de objetos.
- **Botón central del ratón**. Movimiento de tipo *Truck* de la cámara
- **Movimiento de la rueda del ratón**. Movimiento de tipo *Dolly* con la cámara
- **Tecla 0**. Resetear la cámara mirando al origen.
- **Flecha Arriba**. Seleccionar objeto padre.
- **Flecha Abajo**. Seleccionar primer objeto hijo.

- **Flecha Izquierda**. Seleccionar objeto hermano anterior.
- **Flecha Derecha**. Seleccionar objeto hermano siguiente

7 BIBLIOGRAFÍA

[1] «Ruido Perlin - Wikipedia, la enciclopedia libre». Accedido: 29 de mayo de 2024. [En línea]. Disponible en: https://es.wikipedia.org/wiki/Ruido_Perlin
[2] «Modelling Material Microstructure Using the Perlin Noise Function», doi: 10.1111/cgf.14182.
[3] «Modeling the Internal Architecture of Composites», *Comput.-Aided Des.*, vol. 129, p. 102930, dic. 2020, doi: 10.1016/j.cad.2020.102930.
[4] F. Conde, J. Torres, Á.-L. García-Fernández, y F. Feito, «A comprehensive framework for modeling heterogeneous objects», *Vis. Comput.*, vol. 33, ago. 2015, doi: 10.1007/s00371-015-1149-0.
[5] F. de A. Conde Rodriguez y J. C. Torres Cantero, «A Simple Validity Condition for B-Spline Hyperpatches», 2001, doi: 10.2312/egs.20011010.
[6] Ministerio de Trabajo y Economía Social, *Resolución de 13 de julio de 2023, de la Dirección General de Trabajo, por la que se registra y publica el XVIII Convenio colectivo estatal de empresas de consultoría, tecnologías de la información y estudios de mercado y de la opinión pública*, vol. BOE-A-2023-17238. 2023, pp. 108962-109002. Accedido: 18 de abril de 2024. [En línea]. Disponible en: https://www.boe.es/eli/es/res/2023/07/13/(5)
[7] «MVC - Glosario de MDN Web Docs: Definiciones de términos relacionados con la Web | MDN». Accedido: 31 de mayo de 2024. [En línea]. Disponible en: https://developer.mozilla.org/es/docs/Glossary/MVC
[8] «Decorator». Accedido: 6 de mayo de 2024. [En línea]. Disponible en: https://refactoring.guru/es/design-patterns/decorator
[9] «Observer». Accedido: 6 de mayo de 2024. [En línea]. Disponible en: https://refactoring.guru/es/design-patterns/observer
[10] «Visitor». Accedido: 6 de mayo de 2024. [En línea]. Disponible en: https://refactoring.guru/es/design-patterns/visitor
[11] «Command». Accedido: 6 de mayo de 2024. [En línea]. Disponible en: https://refactoring.guru/es/design-patterns/command
[12] «Memento». Accedido: 6 de mayo de 2024. [En línea]. Disponible en: https://refactoring.guru/es/design-patterns/memento
[13] «Shortcuts API · Issue #456 · ocornut/imgui», GitHub. Accedido: 8 de mayo de 2024. [En línea]. Disponible en: https://github.com/ocornut/imgui/issues/456