UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

Extensión de los cuerpos 3D representables en el lenguaje MateFun

Informe de Proyecto de Grado presentado por

Dara Leslie Silvera Martínez y Nicolás Cámera López

en cumplimiento parcial de los requerimientos para la graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisores

Federico Gómez Frois
Sylvia Rita da Rosa Zipitría

Montevideo, 30 de julio de 2023

Índice

Agradecimientos

Como estudiantes de Ingeniería en Computación de la Facultad de Ingeniería de la UdelaR y futuros ingenieros, queremos agradecer por un lado a nuestra institución que ha hecho posible el llegar a esta instancia. Gracias a la continua ayuda hemos podido crecer en todos nuestros ámbitos, tanto en lo profesional como en el humano.

Gracias además a todos los docentes que conforman la institución por habernos brindado su apoyo, tiempo y dedicación a lo largo de toda nuestra carrera. Ellos fueron el pilar principal que hicieron que hoy en día estemos acá.

Un especial agradecimiento a nuestros tutores, Federico Gómez Frois y Sylvia Rita da Rosa Zipitría que gracias a su empatía, experiencia y paciencia nos han instruido y acompañado en cada momento en el desarrollo de este proyecto de grado. Además, queremos agradecer a nuestro cliente Marcos Viera por toda la ayuda que nos ha dado en este proyecto, y a las profesoras Sylvia Borbonet y Teresa Pérez por su especial interés en MateFun, que gracias a ellas este proyecto pudo tener iniciativa.

Finalmente, queremos agradecer a nuestras familias y amigos que siempre nos han estado apoyando y acompañando incondicionalmente, desde el primer momento; a nuestras mascotas, Simona y Lilith; y a todos nuestros compañeros que nos hemos cruzado a lo largo de la carrera con los cuales hemos compartido horas de aprendizaje en esta gran institución.

Gracias a todos y todas.

Resumen

En el presente informe se documenta parte del proceso realizado para el mantenimiento y mejora del lenguaje de programación funcional MateFun, en el marco de la asignatura de Proyecto de Grado de la carrera Ingeniería en Computación. MateFun es un lenguaje de programación funcional, diseñado con el objetivo de introducir la programación en cursos de matemáticas de enseñanza secundaria. Al ser un lenguaje funcional, se espera que fortalezca el aprendizaje del concepto de función matemática. La aplicación está compuesta por la integración de dos componentes; un graficador de funciones 2D y 3D, y una aplicación de consola desarrollada en Haskell que permite ejecutar funciones escritas en un lenguaje de programación funcional muy simple.

En el marco de la presente investigación, se abordó la extensión del conjunto de figuras geométricas en 3D de MateFun, con el objetivo de mejorar las herramientas disponibles para la enseñanza de la geometría espacial en el nivel secundario. Como resultado, se logró la programación y visualización de poliedros, lo cual amplía significativamente el alcance de la plataforma y su utilidad para la comprensión de conceptos matemáticos complejos. Además, se incorporaron nuevas funcionalidades que permiten mejorar la visualización de todas las figuras, como la visualización de sus aristas y la señalización de sus vértices. La propuesta surgió a partir de las sugerencias realizadas por profesoras de matemática de enseñanza secundaria, quienes notaron deficiencias en las herramientas disponibles para trabajar con figuras en el espacio.

Palabras clave

Angular, Delaunator, Earcut, ECMAScript 2015, ES6, Haskell, Java, JavaScript, Three.js, TypeScript, WebGL, Wildfly.

1. Introducción

MateFun es un lenguaje de programación funcional, diseñado con el objetivo de introducir la programación en cursos de matemáticas de enseñanza secundaria. Al ser un lenguaje funcional, se espera que fortalezca el aprendizaje del concepto de función matemática. Con el fin de optimizar la experiencia de aprendizaje, se diseñó una sintaxis minimalista y cercana a la notación utilizada en matemáticas, lo que permite una fácil asimilación del lenguaje y su relación con los conceptos subyacentes. De esta forma, se busca que los usuarios puedan reconocer rápidamente los vínculos entre la programación y las matemáticas [1]. Esta aplicación está compuesta por la integración de dos componentes; un graficador de funciones 2D y 3D, y una aplicación de consola desarrollada en Haskell que permite ejecutar funciones escritas en un lenguaje de programación funcional muy simple.

Este proyecto surge a sugerencia de las profesoras de matemática de enseñanza secundaria Teresa Pérez y Sylvia Borbonet, que constataron dificultades en MateFun al trabajar en geometría espacial. Dichas profesoras remarcaban que sería útil que en MateFun se pudieran representar poliedros regulares, y que se pudiera configurar la transparencia de los cuerpos 3D. Asimismo, el cliente de este Proyecto y a su vez creador del lenguaje MateFun, Marcos Viera, estaba interesado en incorporar una serie de nuevas funcionalidades y mejoras para el lenguaje, tales como ajustar el sistema de iluminación de las figuras 3D, mejorar la definición de ciertas funciones primitivas del lenguaje, agregar una nueva función primitiva para representar más figuras, entre otras.

El presente documento describe brevemente las características y estado del arte de las aplicaciones más populares para representación de funciones 2D y cuerpos 3D en comparación con MateFun, y presenta cómo fue el proceso de desarrollo para extender y ajustar las siguientes funcionalidades de MateFun: graficar nuevos poliedros, tanto regulares como irregulares; configurar la transparencia de todos los cuerpos; visualizar sus aristas, sus vértices, y en caso de que sea posible, visualizar el radio y la altura de los cuerpos presentes en la escena; mejorar el sistema de iluminación de las figuras y ajustar la definición de funciones primitivas existentes en el lenguaje.

En este documento se destacan cuatro secciones principales:

- **Estado del Arte**. Se exponen las características de MateFun para la representación de cuerpos 3D; se analizan las aplicaciones más populares para representar cuerpos 3D y se hace un análisis comparativo de las mismas contra MateFun, con el fin de analizar el estatus de MateFun en esta área y conocer mejor qué funcionalidades adicionales son candidatas para integrar en MateFun. Asimismo, se presenta un análisis sobre el estado de la extensibilidad de MateFun para así tomar decisiones sobre qué tecnologías usar para expandir el lenguaje.
- **Análisis y Diseño de la solución**. Se definen los requerimientos, casos de uso y los aspectos arquitectónicos y de diseño de la solución.
- **Implementación y pruebas**. Se presentan temas relacionados con el desarrollo de las nuevas funcionalidades de la aplicación, la integración de tecnologías para mejorar y facilitar el mantenimiento de la aplicación y finalmente el testing realizado.
- **Conclusiones y Trabajo a futuro**. Se presentan algunas conclusiones sobre el trabajo realizado y el alcance logrado, y se exponen distintas mejoras y funcionalidades para eventuales futuras extensiones del lenguaje MateFun.

2. Objetivos planteados y logrados

El objetivo inicial de este Proyecto de Grado era expandir el lenguaje MateFun con la posibilidad de representar poliedros regulares, también se quería realizar ajustes a la definición de funciones primitivas para mejorar su significado y se planteaba sacar completamente el sistema de iluminación de las figuras 3D. Asimismo, se buscaba realizar alguna de las dos alternativas siguientes:

- visibilizar las líneas exteriores e interiores de las figuras 3D y posibilitando configurar la transparencia de los cuerpos.
- representar poliedros irregulares.

Un primer análisis de estimación del tiempo arrojó como resultado que los requisitos planteados insumirían un tiempo menor al del total del proyecto, por lo cual se decidió realizar ambas alternativas y se propusieron funcionalidades adicionales -como resultado del análisis del estado del arte- para integrar adicionalmente en el lenguaje. Como resultado, se planteó un nuevo conjunto de requisitos, los cuales son:

- representar poliedros regulares.
- visibilizar las líneas exteriores e interiores de las figuras 3D y posibilitando configurar la transparencia de los cuerpos.
- representar poliedros irregulares por medio de una nueva primitiva que permite unir dos círculos, dos rectángulos o dos polígonos, en una nueva figura 3D.
- visibilizar los vértices de las figuras 3D.
- mejorar el sistema de iluminación, ya que se vio que al sacarlo completamente se perdía profundidad en la escena.
- renombrar la primitiva cubo a prisma y eliminar la primitiva cilindro del lenguaje MateFun.
- migrar el código JavaScript legado del graficador 3D a TypeScript.

Estos requisitos fueron los logrados al final del proyecto, en adición al manual de configuración (Ver anexo B) de cómo configurar una estación de trabajo desde cero, para que pueda soportar toda la infraestructura de un ambiente de desarrollo para MateFun; algo que no existía hasta el inicio del proyecto, lo cual hizo bastante complicado iniciar nuevos desarrollos de este estilo en los que se quiere extender MateFun.

3. Estado del arte

Para lograr los objetivos descritos en la sección 2, *Objetivos planteados y logrados*, se analizaron las características de la versión de MateFun legada al comenzar el proyecto, así como las opciones y tecnologías disponibles al comenzar el proyecto para representar los cuerpos 3D y para expandir sus características, como se describe en las siguientes secciones. Uno de los puntos más importantes del análisis consistió en evaluar si las tecnologías con las cuales está desarrollado MateFun permitían la integración de las nuevas funcionalidades. Como plan alternativo para el caso de que no fuera así, se planteó la búsqueda en la Web de otra tecnología de representación de figuras 3D para integrar las funcionalidades requeridas. Asimismo, en este análisis se incluyó la comparación de las funcionalidades aportadas por la versión recibida de MateFun con algunas de las brindadas por otras aplicaciones que existen para graficar figuras 3D.

3.1. Características de MateFun

MateFun es un lenguaje de programación funcional dirigido al aprendizaje del concepto matemático de función que está compuesto por la integración de dos componentes; un graficador de funciones 2D y 3D, y una aplicación de consola desarrollada en Haskell que permite compilar funciones escritas en un lenguaje de programación funcional muy simple.
La componente de graficador 3D de MateFun está desarrollada con tecnologías de Web JavaScript en conjunto con la librería Three.js de animación gráfica 3D para Web, la cual está basada WebGL.

A continuación, se describen características de la versión de MateFun legada al comenzar el proyecto, la cual permite representar las siguientes figuras 3D:

- Cilindro.
- Cubo.
- Esfera.
- Segmento de recta.
- Torus (toro).

Por otro lado, sobre dichas figuras 3D se permite hacer las siguientes transformaciones:

- Mover las figuras según la dirección de un vector 3D.
- Rotar las figuras según la dirección de un vector 3D.

- Escalar las figuras según una constante real.
- Juntar dos figuras 3D en otra nueva figura 3D.
- Pintar una figura 3D según un color.

En la versión recibida, las figuras 3D son dibujadas sin que sus aristas sean visibles al igual que las superficies de dichas figuras no contienen ningún porcentaje de transparencia, sino que son cuerpos opacos, como se puede ver en Figura 3.1.1. Adicionalmente, para diferenciar la curvatura de las figuras, MateFun las representa agregando un sombreado que varía la iluminación de las mismas según el ángulo con el cual se las observe.

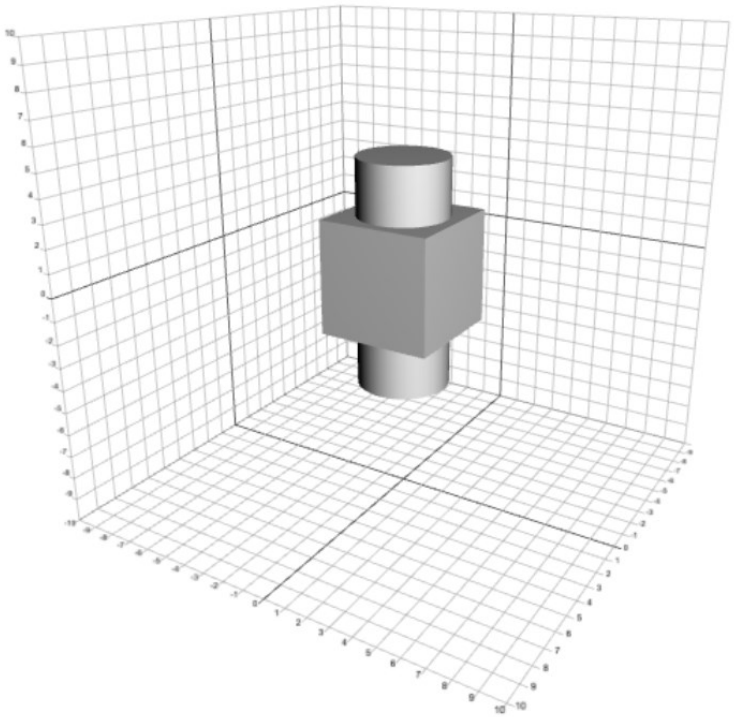

Figura 3.1.1: Sombreado de dos figuras 3D; un cilindro de base 2 y altura 10, compuesto con un cubo de laterales 5.

3.2. Estudio de extensibilidad del graficador 3D existente

En este proyecto, uno de los requisitos planteados por el cliente consistió en aprovechar al máximo la tecnología utilizada en la construcción del graficador 3D, con el objetivo de expandir su funcionalidad y agregar nuevas características. Por lo tanto, se investigó si la base sobre la cual está construido el graficador 3D permitía soportar las nuevas características que se le desea agregar. Por lo ya explicado en la sección 2, *Objetivos planteados y logrados*, esto se traduce en investigar si la librería Three.js de animación gráfica 3D para Web permitía:

- Representar los poliedros regulares más comúnmente usados en enseñanza secundaria según la información proporcionada por la profesora Sylvia Borbonet.
    - Tetraedro
    - Hexaedro (o cubo)[1]
    - Octaedro
    - Dodecaedro
    - Icosaedro
- Representar los poliedros irregulares más comúnmente usados en enseñanza secundaria.
- Poder agregar un porcentaje de transparencia configurable a la superficie de las figuras 3D.
- Poder ver las aristas y vértices de las figuras 3D, en conjunto con algunas otras líneas de interés (como lo es la altura del cilindro).

Para validar si las nuevas funcionalidades que requería la extensión de MateFun eran factibles con las tecnologías que usaba el graficador 3D, se hicieron análisis del código ya existente en el mismo, así como se abordaron pruebas de concepto para modelar las funcionalidades requeridas.

[1] Notar que el cubo ya tiene soporte en MateFun, pero como sucede con el resto de figuras, no cuenta con la posibilidad de configurar su transparencia, al igual que no se pueden ver sus aristas.

3.2.1. Análisis del código existente para encontrar limitaciones de extensibilidad

Un rápido análisis sobre el código legado reveló que el hecho de estar usando JavaScript "puro" -y no TypeScript- para la implementación de todo el comportamiento del graficador, llevaba a que existieran errores en producción que podían dar lugar a algunos inconvenientes en ciertos casos de uso. En la sección 3.2.3 se ahonda más sobre JavaScript y TypeScript, pero para dar un panorama general, se explica acá que al reemplazar el código JavaScript por código TypeScript en el graficador 3D, se detectaron errores de:

- Métodos no existentes para objetos de cierto tipo.
- Asignaciones no válidas entre objetos de distintos tipos.
- Falta de chequeos en métodos de objetos que posiblemente sean null.

Más allá de los errores, se encontraron otros aspectos faltantes en cuanto a la calidad del código:

- Variables declaradas cuyo valor nunca era leído.
- Métodos declarados que no usaban todos los parámetros que requerían.
- Llamados a funciones que se les pasaban más parámetros de los que se utilizaban. Por ejemplo, si una función utilizaba un parámetro, existían casos en el cual se le pasaban dos parámetros.
- No se hizo uso de un Linter y formateador de código en todo el Proyecto, lo cual daba lugar a muchas inconsistencias en cuanto al ordenamiento e indentación del código, dificultando así, su mantenimiento y legibilidad.
- Falta de documentación en las primitivas creadas. Es decir, no quedaba claro la utilidad de cada primitiva, de los parámetros de entrada usados y del resultado de la aplicación de estas.

Una vez terminado el análisis sobre el código legado, se realizaron chequeos de la arquitectura sobre la cual estaba diseñado el graficador, con el fin de validar la extensibilidad de las funcionalidades existentes. Se corroboró que el código había sido elaborado siguiendo los lineamientos que promueve Three.js en su documentación de uso, lo cual facilita la implementación de nuevas funcionalidades.

El análisis permitió por un lado detectar los errores que se señalan del código JavaScript y, por otro lado, concluir que la implementación del graficador 3D era

ampliamente mantenible y extensible, lo cual facilitaba la realización de pruebas de concepto para validar nuevas funcionalidades. Por ende, se decidió utilizar TypeScript para el código legado, y así poder corregir los errores. Con el tipado del código se pudo mejorar no solo la robustez del proyecto y la mantenibilidad, sino la legibilidad.

3.2.2. Pruebas de concepto para validar las funcionalidades requeridas

Se realizaron varias pruebas de concepto, las cuales se pueden encontrar en detalle en el documento de pruebas de concepto (ver anexo B). Dichas pruebas de concepto tenían el objetivo de validar si era posible:

- Representar los poliedros regulares requeridos, al igual que poliedros irregulares.
- Visualizar la transparencia de las superficies de las figuras 3D.
- Visualizar aristas y líneas de interés en las figuras 3D.

La conclusión fue que sí era posible satisfacer todos estos requerimientos con las tecnologías que estaba construido MateFun, con lo cual era posible satisfacer los objetivos planteados en su totalidad. En particular, para representar los poliedros regulares buscados, Three.js provee métodos muy simples que permiten representar dichos poliedros en cuestión de pocas líneas de código; además, sobre cualquier figura 3D se pudo comprobar que sus superficies son configurables con un porcentaje de opacidad y color; más aún, sobre todos los cuerpos 3D, Three.js provee un mecanismo para mostrar sus aristas, el cual, en la mayoría de los casos permitió mostrar con total facilidad las aristas de las figuras 3D representadas (hubieron unos pocos casos como el cilindro, en el cual se requirió agregar una circunferencia 3D para demarcar las líneas de las bases del mismo). En el caso de querer agregar líneas de interés a las figuras -como lo es la altura del cilindro-, se pudo corroborar que fácilmente se podía agregar un segmento de recta parametrizado con datos de la figura asociada, el cual se podía ajustar fácilmente según la posición y proporción inicial de la figura. Además, se comprobó que este segmento de recta mantenía su relación con respecto a la figura cuando ésta rotaba. Por último, Three.js tiene una primitiva que, determinando los vértices y caras, permite representar cualquier poliedro deseado. Por lo tanto, se puede representar cualquier poliedro irregular que se requiera. Esto se puede apreciar en las primitivas que cuenta Three.js para representar los poliedros regulares ya que las mismas están construidas sobre esta primitiva más general [2].

A modo de ejemplo, se muestra la Figura 3.2.2.1 la cual es una captura de una de las pruebas de concepto que contiene un cono, un cubo y un dodecaedro, en donde sus superficies tienen un grado de transparencia al 20% y sus aristas y líneas de interés son visibles.

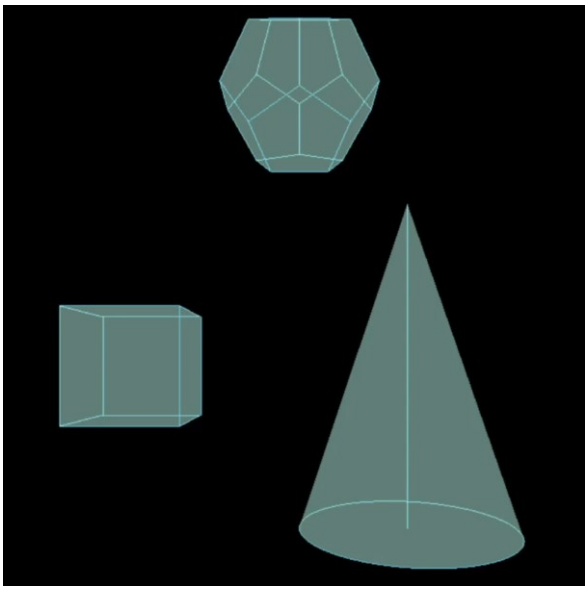

Figura 3.2.2.1: Cono, cubo y dodecaedro con transparencia al 20% junto a sus aristas y líneas de interés visibles.

Por último, con las pruebas de concepto realizadas se llegó a unas estimaciones de tiempo para implementar las funcionalidades iniciales que se estipulaban para el Proyecto de Grado. Estas estimaciones de tiempo concluyeron que el alcance del proyecto podía ser mayor, así que se decidió realizar más pruebas de concepto para medir el tiempo que podía llevar agregar nuevas funcionalidades y, así, obtener un nuevo conjunto de requerimientos que fueron validados con el cliente del proyecto.

3.2.3. JavaScript y TypeScript

Como ya se había mencionado al principio de esta sección, Matefun es un lenguaje de programación funcional cuyo Frontend está construido sobre dos tecnologías; JavaScript y la librería Three.js de animación gráfica 3D para Web. JavaScript es una de las tecnologías centrales del desarrollo Web hoy en día. Más del 97% de los sitios Web utilizan JavaScript para implementar el comportamiento del lado del cliente en sus páginas Web [3].

Es innegable que JavaScript está presente en todos los sitios Web hoy en día, pero esto no significa que no existan alternativas que mejoren y faciliten el desarrollo del comportamiento *client-side* de las páginas Web. Es así entonces que existe TypeScript, un lenguaje de programación el cual es un *superset* de

JavaScript, es decir, todo lo que se puede hacer en JavaScript es igual de válido en TypeScript, pero no a la inversa.

TypeScript nace como un lenguaje de programación para extender JavaScript y superar sus carencias. Por ejemplo, una de las características más importantes que agrega TypeScript sobre JavaScript es el tipado fuerte; el cual permite detectar fácilmente errores de tipado en tiempo de compilación, mientras que en JavaScript esto solo es posible en tiempo de ejecución, lo cual da lugar a una detección más tardía de los errores en el mejor de los casos. Para dar un ejemplo de este último argumento, Airbnb en un análisis *postmortem* concluyó que el 38% de bugs podrían haber sido evitados si hubieran usado TypeScript en su aplicación [4].
Una de las diferencias más grandes entre TypeScript y JavaScript, es que los navegadores no entienden TypeScript (solo entienden JavaScript), por lo tanto, existe un proceso intermedio para ejecutar el código TypeScript, el cual se llama transpilación. Este proceso convierte el código TypeScript -inentendible para los navegadores- en código JavaScript que sí puede ser interpretado por los navegadores.

A continuación, se detalla una lista de otras características relevantes que puede aportar TypeScript sobre JavaScript para el lenguaje MateFun:

- TypeScript permite escribir una única base de código que luego se puede *transpilar* a cualquier versión existente de JavaScript, permitiendo así, escribir código actual que luego puede funcionar incluso en los navegadores más antiguos.

- Convertir los proyectos existentes de JavaScript a TypeScript no requiere esfuerzo alguno. Esto es algo que comprobamos nosotros mismos en nuestras pruebas de concepto; al ser TypeScript un *superset* de JavaScript, todo el código existente funciona directamente en TypeScript, por lo cual solo se requiere configurar algunos parámetros que se usan en el transpilador de TypeScript.

- Usar TypeScript evita tener que leer cantidades excesivas de documentación. Como TypeScript aporta tipos para las funciones y objetos, el *intellisense* de los IDEs permite aportar documentación sobre los métodos y funciones que disponen los objetos, por lo cual se evita tener que revisar la documentación para consultar estos.

Objetivamente, TypeScript parece ser un muy buen complemento de JavaScript, pero a la hora de la verdad, la simplicidad y comodidad de usar JavaScript puede hacer la diferencia entre usar o no TypeScript. Es así entonces, que todos los años se realiza una encuesta en StackOverflow -el foro Web más grande de preguntas y respuestas sobre programación-, la cual, entre otras cosas, revela las tecnologías más amadas por los desarrolladores. El pasado 2021 se hizo la encuesta [5], la cual reveló que con 18117 votos a favor (72,73%) y 6792 en contra, TypeScript se encontraba en el tercer puesto como tecnología más amada por lo votantes que participaron en la encuesta, mientras que, con 32964 votos a favor (61,51%) y 20623 en contra, JavaScript se encontraba en el puesto 15 como la tecnología más amada por los votantes. Este resultado demuestra que sobre la base de encuestados en StackOverflow, TypeScript es mucho más preferido que JavaScript.

En nuestra experiencia y en las pruebas de concepto que hicimos, usar TypeScript nos facilitó el entendimiento del código existente en el graficador 3D, nos permitió detectar posibles errores existente en la implementación actual del graficador 3D y nos indujo a una programación más ágil y documentada sobre el código; algo que con JavaScript no es tan directo.

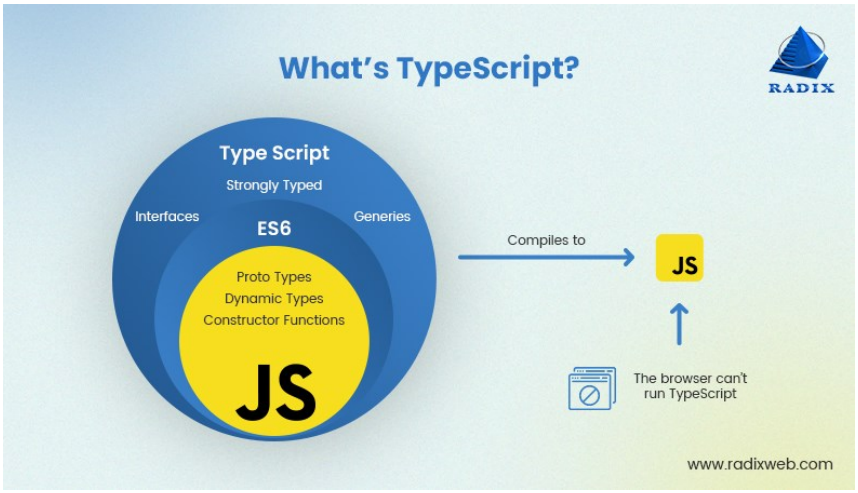

Figura 3.2.3.1: Resumen del funcionamiento de TypeScript.

3.2.4. Conclusiones sobre la extensibilidad del graficador 3D

Primero que nada, cabe remarcar la utilidad de utilizar TypeScript sobre JavaScript en el graficador 3D, no solo para seguir escalando y manteniendo con mayor facilidad la implementación del mismo, sino también para detectar tempranamente errores que pueden llegar a producción.

Resumiendo, los argumentos que justifican el cambio de JavaScript a TypeScript son:

- El cambio de JavaScript a TypeScript tiene un costo mínimo en esfuerzo por ser TypeScript un *superset* de JavaScript. Además, se pudo corroborar con una prueba de concepto este hecho.
- La comunidad prefiere usar TypeScript antes que JavaScript.
- Los beneficios de TypeScript sobre JavaScript permiten detectar muchos bugs antes de que lleguen a producción.

Se pudo comprobar que la tecnología usada en el graficador 3D de MateFun no es un impedimento para agregar las funcionalidades deseadas, lo cual mitiga riesgos y ahorra tiempo de desarrollo. Más aún, todas las opciones de extensibilidad que se quieren incluir en MateFun tienen una implementación relativamente sencilla y compatible con la actual implementación del graficador 3D, lo cual reduce de gran manera la cantidad de trabajo que se tiene que realizar para adaptar la arquitectura actual a los nuevos requerimientos.

3.3. Aplicaciones para graficar figuras geométricas

A continuación, se presentarán diferentes aplicaciones para graficar figuras 3D, con el fin de destacar sus principales características y debilidades, logrando así, tener un panorama general sobre el estado del arte del área, el cual permite desarrollar una lista de funcionalidades que estarían interesantes incluir en MateFun e, incluso, en las aplicaciones analizadas.

3.3.1. Desmos

Desmos es una herramienta gratuita para enseñar y graficar matemáticas. Está disponible en versión de navegador Web y en las aplicaciones móviles Android e iOS [6]. Esta aplicación está pensada para estudiantes y profesores.

Características claves.

- Es un graficador 2D, por lo que no permite representar figuras 3D de manera nativa, sino que dichas figuras tienen que ser representadas juntando varias funciones del plano 2D para asimilar un cuerpo 3D. Por ejemplo, para graficar un cubo se tiene que hacer uso de varias funciones afín para representar cada segmento del cubo.
- Cuenta con una amplia cobertura de los lineamientos de accesibilidad WCAG 2.1 [7], lo que permite que cada estudiante pueda utilizar la aplicación.
- La interfaz gráfica no es la más sencilla de utilizar (al menos en comparación con GeoGebra). Por ejemplo, no se puede mover una figura en ambos ejes de coordenadas a la vez.

3.3.2. Geogebra

Geogebra es una herramienta digital gratuita para graficar funciones, graficar figuras geométricas y realizar cálculos aritméticos. Cuenta con una pizarra interactiva, la cual permite interactuar con las figuras geométricas graficadas. Está disponible en versión de navegador Web y en las aplicaciones móviles Android y iOS. Esta aplicación brinda tutoriales sobre su uso y es popular entre estudiantes y profesores.

Características claves.

- Permite graficar las siguientes figuras 3D mediante una interfaz gráfica (no por comandos):
    - Cubo
    - Cilindro
    - Cono
    - Esfera
    - Pirámide
    - Prisma
    - Segmento de recta
    - Superficie de revolución
    - Tetraedro
- Permite rotar/ajustar la cámara con la cual se ven las figuras 3D según la dirección de las aristas de cada una.
- Permite trasladar y rotar los vértices de las figuras 3D (según una dirección de los ejes) desde la interfaz gráfica.
- Permite cambiar el color y grosor de las caras y vértices de las figuras 3D.
- Permite rotar de manera infinita (o a velocidad constante) las figuras sobre el eje z.
- Las superficies de las figuras 3D son representadas como cuerpos traslucidos y su porcentaje de transparencia es configurable.
- La aplicación *mobile* contiene un modo examen, GeoGebra Examen, diseñado para aprovechar las funcionalidades de GeoGebra limitando el acceso a Internet y a otras aplicaciones instaladas en el equipo durante el examen. Además, este modo examen emite una alerta visual muy evidente cuando se lo abandona sin autorización [8].

- En nuestras pruebas, el rendimiento del graficador 3D no se percibe tan alto, al menos comparado con la fluidez del graficador 3D de MateFun (Ver anexo A).

3.3.3. Análisis comparativo

A continuación, se presentará un análisis cualitativo de las diferentes funcionalidades que proveen las distintas herramientas digitales para representar figuras 3D. Esta comparativa busca dar un panorama de cómo se encuentra actualmente cada una de ellas, con el fin de encontrar sus puntos fuertes, carencias y cualquier otra cualidad de interés a destacar.

Características	Desmos	Geogebra	MateFun
Ejes de coordenadas visibles	Si	Si	No
Escalar figuras	No (*)	Si	Si
Figuras coloreables con todo el espectro RGB	Está limitado a unos pocos colores predefinidos	Si	Si
Grilla o cuadrícula visible	Si	Si	Si
Hacer Zoom In/Out en las figuras	Si	Si	Si
Muestra la intersección de dos figuras	No	Si	No
Muestra las aristas de las figuras	Si	Si	No
Muestra los vértices de las figuras	No	Si	No
Permite graficar cualquier función	Si	Si	Solo las que dispone el lenguaje
Permite graficar figuras 3D no representables por medio de una función	No	Solo las que dispone el lenguaje	Solo las que dispone el lenguaje
Permite representar varias figuras a la vez	Si	Si	Si
Permite ver las figuras desde cualquier ángulo de visión	Está limitado a ciertos ángulos de visión	Si	Si
Rotar figuras	No (*)	Si	Si
Tiene una aplicación móvil	Si	Si	Si
Trasladar figuras	No	Si	Si

Tabla 3.3.3.1: Análisis cualitativo entre Desmos, Geogebra y MateFun.

(*) Desmos sí permite escalar y rotar sus funciones, pero para hacerlo, hay que reescribir las mismas, ya sea multiplicandolas por un escalar o multiplicandolas por la función rotación del plano 2D. Lo que queremos decir con estos puntos, es que Desmos no posee un mecanismo más "amigable" o rápido para implementar estas características.

3.3.4. Conclusiones al comparar las distintas aplicaciones con MateFun

En primer lugar, es innegable que la aplicación Geogebra es una herramienta madura y completa para representar figuras tanto en 2D como en 3D. La amplia gama de funcionalidades que ofrece hace que sea la aplicación más comúnmente usada para la representación de funciones y cuerpos geométricos. Esto es algo bueno para MateFun, porque le permite tener una base de partida para incorporar futuras características, dado que se puede estudiar las cosas que hace bien GeoGebra para comparar e incorporar nuevas funcionalidades o similares en el esquema de diseño de MateFun.

Por otro lado, un punto en el que MateFun necesita mucho trabajo es en la accesibilidad. Esto se puede apreciar al comparar todos los lineamientos que cumple Desmos en el campo de la accesibilidad; campo el cual está en continua evolución y desarrollo.

En términos de representación de figuras 3D, MateFun es una herramienta con mucho potencial. Cuenta con un buen diseño base que permite extenderlo ampliamente y, su desempeño y fluidez para graficar figuras 3D, es superior al de GeoGebra (ver anexo A) (herramienta con muchos más años de desarrollo e investigación detrás).

Una fortaleza de MateFun es que es un lenguaje de programación funcional, por lo tanto, lo que se grafica o representa como figura es la solución a un problema computacional. Por tanto, el proceso de aprendizaje que fomenta es que, ante un problema, éste debe ser comprendido y su solución debe ser programada y representada mediante composición de funciones.

Por último, redondeamos las conclusiones con algunas características que estarían interesantes agregar en las distintas herramientas digitales para las figuras 3D:

- Ver líneas específicas e importantes de cada figura, como lo es la altura para el cono y para el cilindro o, el radio/diámetro para el cilindro y la esfera.
- Poder permitir múltiples colores en una misma cara de una figura 3D. Esto podría ser usado, por ejemplo, para indicar zonas de calor o zonas de densidad en las figuras.

4. Análisis y Diseño de la solución

Los análisis realizados que se describen en las secciones anteriores aportaron al análisis y diseño de la solución de la implementación para las funcionalidades estipuladas para el proyecto. Los principales aspectos se describen en esta sección.

4.1 Requerimientos del sistema

Dado que MateFun está principalmente orientado al área educativa, los usuarios son por lo general alumnos y docentes.

Se detalla a continuación las funcionalidades de las figuras geométricas en la aplicación para cada grupo de usuarios:

- **Alumno**: utiliza la herramienta Matefun para graficar funciones y figuras geométricas, crear directorios, mover archivos y la realización de las tareas.
- **Docente**: utiliza la herramienta Matefun para graficar funciones, crear directorios, mover archivos y compartir archivos.

La Figura 4.1.1 resume los casos de uso críticos de los usuarios[2], lo cual da una perspectiva gráfica de dichos casos críticos de MateFun.

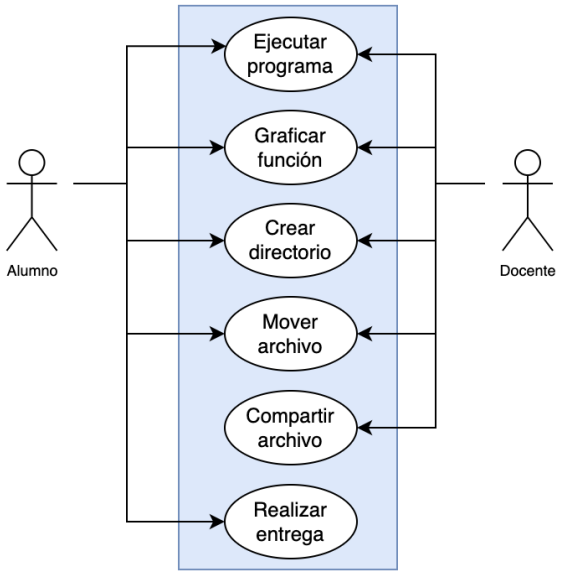

Figura 4.1.1: Diagrama de casos de uso críticos de los usuarios.

En este proyecto se busca potenciar y complementar las funcionalidades para graficar figuras 3D de MateFun, permitiendo así tener una amplia capacidad de modelado de figuras, así como un mayor enriquecimiento para las opciones de

[2] Casos de uso que constituyen funcionalidades clave del sistema y que determinan el diseño de la arquitectura

personalización de estas. En este contexto, se procura expandir las funcionalidades provistas por el caso de uso "Graficar función" de la Figura 4.1.1.

En las siguientes secciones abordaremos los requerimientos dados por el relevamiento de los requisitos.

4.1.1 Requerimientos funcionales

A continuación, se listan los requerimientos funcionales para los usuarios de esta aplicación. En particular, los dos tipos de usuarios tendrán las mismas funcionalidades.

1. Graficar poliedros regulares, tales como el tetraedro, octaedro, dodecaedro e icosaedro[3].
2. Visualizar las aristas de las figuras 3D.
3. Visualizar la otra información de las figuras 3D[4] mediante líneas punteadas.
4. Visualizar/resaltar los vértices de las figuras 3D.
5. Configurar la transparencia de las figuras 3D.
6. Dados dos polígonos, crear una figura 3D (un poliedro) que sea la unión de los dos polígonos mediante caras. Los dos polígonos serían la base de la nueva figura 3D. Entre otras figuras, esto permitirá graficar pirámides de base rectangular, circular y triangular.

Adicionalmente, se listan otras características a implementar/mejorar en el lenguaje con el fin de adaptar ciertos requerimientos:

- Mantener el diseño base de las interfaces de usuario, adaptando las mismas a las nuevas funcionalidades.
- Mejorar el sistema de iluminación de las figuras 3D, para así pasar a una escena en donde la luz de las figuras siempre esté dada por la dirección en que se las mira. En otras palabras, la fuente de luz de las figuras será la cámara.
- Mejorar el sistema de iluminación de las figuras 3D, para así pasar a una escena en donde la luz de las figuras siempre esté dada por la dirección en

[3] El hexaedro (cubo) ya se puede graficar en la versión legada de MateFun, pero sin las funcionalidades de 2 a 5.
[4] Líneas interiores tales como la altura del cilindro, radio de la esfera, entre otras.

que se las mira. En otras palabras, la fuente de luz de las figuras será la cámara.

- Uniformizar la definición de las funciones para que así haya menos redundancia en la especificación de las mismas. Para ello, se realizan los siguientes cambios a dos de las funciones ya existentes en MateFun:
    - La función `cubo :: (R X R X R) -> Fig3D` fue renombrada como la función `prisma :: (R X R X R) -> Fig3D`
    - La función `cilindro :: (R X R X R) -> Fig3D` fue eliminada del conjunto básico de funciones primitivas, ya que puede ser construida con la nueva funcionalidad de unir dos figuras 2D en una nueva figura 3D (ver sección 4.2.3.2).

4.1.2 Requerimientos no funcionales

A continuación, se listan los requerimientos no funcionales para los usuarios de esta aplicación.

Requerimientos de hardware.

Para utilizar la aplicación Web MateFun, se necesita contar con

- Conexión a Internet.
- Procesador Intel Pentium 4 / AMD Athlon 64 o superior compatible con SSE2.
- Al menos 2GB de RAM, en caso de usar una distribución de Windows a 64 bits. En caso de usar Windows a 32 bits u otro sistema operativo, al menos 512MB de RAM.
- Disponer de una GPU, ya sea integrada al procesador o dedicada[5].

Los requisitos de hardware principalmente se deducen de los requisitos mínimos para utilizar un navegador Web [10] [11] [12].

Requerimientos de software.

Como requisitos mínimos de software, los usuarios de la aplicación deberán utilizar un navegador Web con la función de JavaScript activada, renderizado de gráficos SVG [13], renderizado de gráficos WebGL [14], soporte para API

[5] El motor de renderizado que usa MateFun para graficar en 3D requiere disponer de una GPU [9].

WebSocket [15] y soporte para ECMAScript 2015 [16]. Estos requisitos y otros más se cumplen al usar algunos de los siguientes navegadores:

- Chrome 51 en adelante
- Edge 15 en adelante
- Safari 10 en adelante
- Firefox 54 en adelante
- Opera 38 en adelante

En esta lista no aparece ninguna versión de Internet Explorer, porque desde el 15 de junio del presente año (2022) Microsoft ha dejado de dar soporte para el mismo [17].

Usabilidad.

La usabilidad es uno de los aspectos más importante de toda aplicación. En el contexto de MateFun y de las nuevas funcionalidades integradas, la usabilidad a la que se apuntó fue seguir manteniendo una coherencia y uniformidad con la forma de escribir y usar los comandos del intérprete.

Es por lo anterior que, las nuevas funcionalidades siguieron los lineamientos de modelado que se han construido en el intérprete. Esto es:

- Simplicidad y facilidad al momento de usar los comandos.
- Similitud de los comandos con el concepto de función matemática.
- Conjunto de comandos minimal, pero con mucha potencia de modelado. Es decir, un conjunto de comandos que permita construir una biblioteca de funciones más específicas.

Escalabilidad.

La arquitectura del sistema siguió los lineamientos de diseño ya planteados en el graficador 3D, los cuales aportan una alta escalabilidad y mantenibilidad al módulo de dibujado 3D. Estos lineamientos se pueden observar en la Figura 4.1.2.1. Para ejemplificar, el módulo de luz fue modificado para sustituir el actual sistema de iluminación y el módulo de figuras fue expandido para dar lugar a las nuevas construcciones que se modelaron.

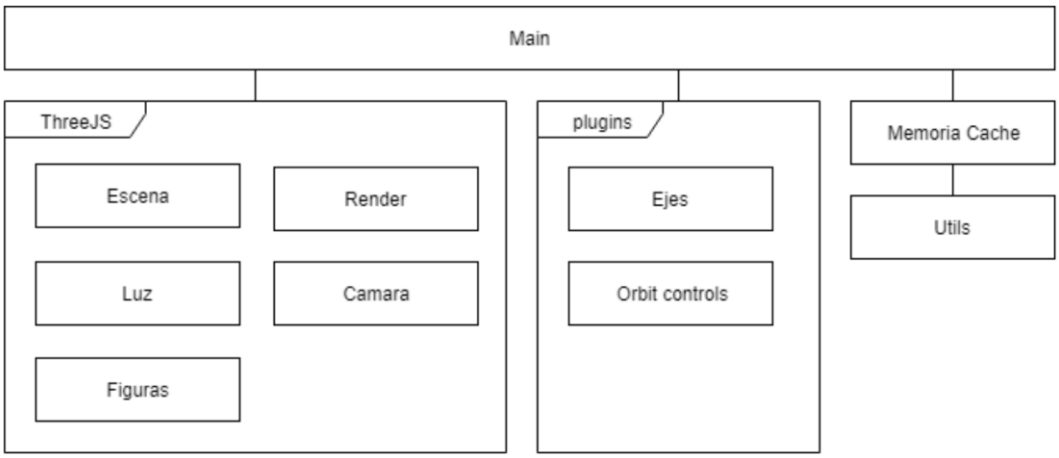

Figura 4.1.2.1: Arquitectura del graficador 3D [9].

Lo mismo sucedió con la arquitectura del intérprete, la misma fue ampliada siguiendo los lineamientos de diseño que fueron establecidos como base para la mantenibilidad y modularización. Para ejemplificar, el módulo *Figures* del intérprete fue expandido con las nuevas construcciones de figuras que se querían modelar.

Por último, el código del graficador 3D fue portado a TypeScript para hacer uso de todos los beneficios que este trae en la detección temprana de errores y en la alta escalabilidad que este aporta.

4.2. Modelo de casos de uso

A continuación, se muestran las descripciones de los casos de uso a implementar para las distintas funcionalidades. Todos los casos de uso aplican para ambos tipos de usuarios: Docente y Alumno.

Más detalles sobre los casos de uso se pueden encontrar en el documento de Requerimientos y Casos de Uso (ver anexo B). Asimismo, se explica el uso de cada una de las funciones en el documento Resumen de cambios MateFun 0.16 (ver anexo B), que se elaboró como guía para el usuario.

4.2.1. Caso de uso básicos

Para los siguientes casos de uso, el usuario solo usará una primitiva con sus correspondientes argumentos para poder llevarlo a cabo.

4.2.1.1. Graficar poliedro regular

El caso de uso comienza cuando un usuario quiere graficar un poliedro regular. Para ello, el usuario ingresa alguno de los comandos (o el comando) para graficar

un poliedro regular en el intérprete, y el sistema dibuja en el graficador 3D el resultado de la ejecución de la función, o se devuelve un error en el intérprete en caso de que se haya instanciado mal el comando.

4.2.2. Caso de uso en el despliegue del menú de configuración

Para los siguientes casos de uso, el usuario solo usará el menú de configuración de las figuras 3D para poder llevarlo a cabo.

4.2.2.1. Visualizar aristas de figuras 3D

El caso de uso comienza cuando un usuario quiere visualizar las aristas de las figuras 3D del panel de gráficos 3D. Para ello, el usuario despliega el menú de configuración, clickea el CheckBox cuya leyenda es "Aristas", y el sistema dibuja en el graficador 3D el resultado de la nueva configuración.

4.2.2.2. Visualizar otra información de una figura 3D

El caso de uso comienza cuando un usuario quiere visualizar la otra información de las figuras 3D del panel de gráficos 3D. En los distintos casos, puede ser la altura y/o el radio de la figura. Para ello, el usuario despliega el menú de configuración, clickea el CheckBox cuya leyenda es "Otra información", y el sistema dibuja en el graficador 3D el resultado de la nueva configuración.

4.2.2.3. Visualizar vértices de una figura 3D

El caso de uso comienza cuando un usuario quiere visualizar los vértices de las figuras 3D del panel de gráficos 3D. Para ello, el usuario despliega el menú de configuración, clickea el CheckBox cuya leyenda es "Vértices", y el sistema dibuja en el graficador 3D el resultado de la nueva configuración.

4.2.3. Casos de uso resultado de la programación

Para los siguientes casos de uso, el usuario tiene que componer funciones para poder llevarlos a cabo.

4.2.3.1. Configurar transparencia de una figura 3D

El caso de uso comienza cuando un usuario quiere configurar la transparencia de una figura 3D. Para ello, el usuario ingresa alguna función para graficar una figura 3D, la compone en otra función que habilita la transparencia de la figura, y el sistema dibuja en el graficador 3D el resultado de la ejecución de la función, o

se devuelve un error en el intérprete en caso de que se haya instanciado mal la función.

4.2.3.2. Crear una figura 3D mediante dos polígonos

El caso de uso comienza cuando un usuario quiere graficar una figura 3D (un poliedro) a partir de dos polígonos, cuya cantidad de vértices coincide. Para ello, el usuario ingresa la función en el intérprete y le pasa como parámetro dos polígonos y la distancia que hay entre ellos, y el sistema dibuja en el graficador 3D el resultado de la ejecución de la función, o se devuelve un error en el intérprete en caso de que se haya instanciado mal la función.

4.3. Arquitectura del sistema

La arquitectura base de la aplicación MateFun sigue siendo la misma que de la versión legada de MateFun, es decir, no se agregó, ni se eliminó ninguna capa. La integración de los nuevos casos de uso se realizó modificando los distintos módulos de las capas existentes.

Como se puede apreciar en la Figura 4.3.1, la arquitectura sigue siendo la misma presentada inicialmente en [18].

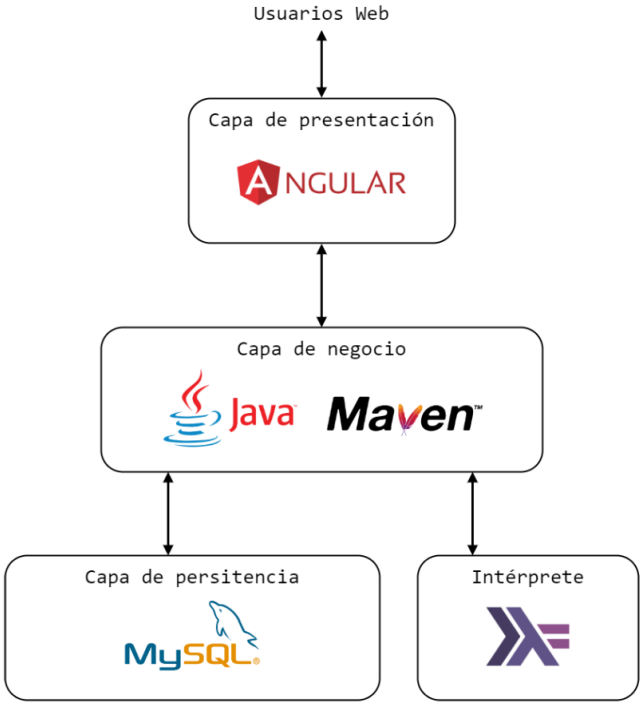

Figura 4.3.1: Diagrama de capas de MateFun.

Presentación: La capa de presentación se refiere a la interfaz gráfica. Es la encargada de la interacción con el usuario y del despliegue de los elementos visuales del sistema.

Negocio: Refiere a la lógica de negocio del sistema. Aquí se encuentra la implementación de las funciones que resuelven las funcionalidades que el usuario ejecuta desde la interfaz. Además, esta capa se encarga de comunicar la capa de presentación con el compilador del lenguaje y gestiona las conexiones.

Persistencia: La capa encargada de la gestión de la base de datos del sistema. Provee una API que permite a la capa de negocios realizar operaciones sobre la base de datos.

Intérprete: Consiste en un ejecutable binario capaz de compilar y ejecutar en modo interactivo código Matefun. Almacena archivos temporales en el sistema de archivos del servidor.

Más detalles sobre la arquitectura y diseño del sistema (ver anexo B) se pueden encontrar en el documento de arquitectura y diseño.

5. Implementación y pruebas

En esta sección se presentará como se realizó el proceso de implementación y testing de los casos de uso. Cabe destacar que en algunas secciones se mostrará parte del código implementado para ejemplificar las decisiones tomadas a lo largo de la implementación. En otras ocasiones este código estará como una referencia del documento.

5.1. Metodología de trabajo

La metodología aplicada para el desarrollo de este proyecto fue principalmente *pair programming*[6], con algunas influencias de metodologías ágiles. Las tareas más grandes se trabajaban de a pares, mientras que las tareas más autocontenidas se hacían de forma individual, con un posterior *review* del otro par. Ambos integrantes éramos desarrolladores y testers, donde las tareas que completaba uno eran *testeadas* por ambos.

Si bien la planificación de tareas fue un elemento que siempre se mantuvo en desarrollo y evolución a lo largo del proyecto, la división de estas pasó a un segundo plano, dado que gran parte del tiempo se realizó *pair programming*. Para realizar el seguimiento de tareas, se llevaba una hoja de Google Docs con el fin de mantener las líneas principales de trabajo, mientras que en Discord se iba haciendo un seguimiento más exhaustivo de la tarea que se iba realizando en cada momento.
No se optó por una herramienta más sofisticada como Trello, ClickUp, entre otras, para realizar el seguimiento de las tareas, porque se detectó de manera temprana que al estar haciendo *pair programming* estas herramientas agregan más inconvenientes que beneficios.

La comunicación efectiva dentro del equipo fue un elemento clave en el desarrollo del proyecto. Se optó utilizar WhatsApp para la comunicación diaria, y Discord para las reuniones que teníamos. Discord resultó ser una herramienta versátil y beneficiosa para el equipo.

Como la infraestructura estaba preestablecida para este proyecto, se utilizó GitLab y GitHub para el código fuente, permitiendo hacer un seguimiento detallado del progreso del proyecto. Asimismo, para la gestión de la documentación, se recurrió a la versatilidad de Google Drive, que facilitó el acceso y la colaboración

[6] Técnica de desarrollo de software en la que dos programadores trabajan juntos en una estación de trabajo. Uno, el desarrollador, escribe el código mientras que el otro, el tester, revisa el código. Los dos programadores intercambian roles con frecuencia.

en tiempo real sobre distintos documentos. En este sentido, Google Docs se mostró como la herramienta perfecta para la documentación.

5.2. Entorno de desarrollo

Si bien en este proyecto los entornos de desarrollo ya venían dados por la naturalidad de este, se tuvo que decidir en qué IDE se iba a programar la implementación de cada área. Basado en la experiencia de ambos y en lo que dice la comunidad de programadores [23], se optó por usar el IDE Visual Studio Code [24] para programar en todas las áreas de desarrollo de este proyecto:

- Interprete MateFun (Back-end)
- Aplicación Web MateFun (Front-end)
- Módulo graficador 3D (Front-end)

Otro de los motivos por el cual se optó por utilizar el IDE Visual Studio Code, es porque cuenta gran variedad de *plugins* que lo hacen muy personalizable. En particular, al principio del desarrollo de este proyecto se descubrió el plugin Live Share [25], el cual permitió programar de manera compartida y en vivo sobre una misma base de código, facilitando así el *pair programming* de manera remota.

Para finalizar, como parte de la evolución y mantenimiento de los repositorios trabajados, se vio una oportunidad de mejora en cuanto configuración y buenas prácticas en el repositorio usado para llevar el módulo graficador 3D. Es por esto que, este repositorio ahora cuenta con las tecnologías mencionadas en las siguientes secciones.

5.2.1. ESLint en el graficador 3D

ESLint [26] es un analizador estático de código, el cual permite rápidamente encontrar y reportar problemas en el mismo, antes de que el código sea ejecutado. Sin este analizador de código, muchos errores existentes pueden no ser detectados y así llegar a producción.

Este analizador de código fue configurado en el repositorio del graficador 3D como una dependencia de desarrollo[7], sobre el cual se le configuraron varias reglas [27] para analizar y reportar patrones de programación que inducen a errores y/o malas prácticas. Esto permite que todos los futuros colaboradores que trabajen

[7] Es decir, una dependencia que solo se utiliza para desarrollar y que no está presente en el código usado para producción [28].

sobre el repositorio puedan tener una buena configuración base que les va a facilitar la mantenibilidad del código y ayudará en la detección temprana de errores.

Con este analizador de código también se detectaron varios errores de programación y malas prácticas existentes con el código legado del repositorio, por lo cual, ESLint sirvió para mejorar la calidad de la implementación legada.

5.2.2. Prettier en el graficador 3D

Prettier [29] es un formateador de código que permite uniformizar el estilo del mismo, lo que posibilita que el código escrito por varios desarrolladores siempre termine teniendo el mismo estilo. Este formateador logra eliminar la necesidad de tener que pensar en el estilo del código, ya que el estilo se rige por las reglas de configuración del Prettier, a su vez, sus reglas dan como resultado un código más legible y fácil de mantener.

Prettier soporta lenguajes tales como JavaScript, TypeScript, HTML, CSS, entre otros.

Este formateador de código fue configurado en el repositorio del graficador 3D como una dependencia de desarrollo, sobre la cual se le configuraron varias reglas [30] para definir el estilo del código. Esto permite que todos los futuros colaboradores que trabajen sobre el repositorio puedan realizar aportes que seguirán manteniendo el estilo y uniformidad de todo el trabajo realizado.

5.2.3. Plugins recomendados para el desarrollo del graficador 3D

En el graficador 3D se configuraron plugins recomendadas para el desarrollo con el IDE Visual Studio Code, así, todos los futuros colaboradores que participen de este repositorio podrán hacerse con un conjunto recomendado de extensiones que los ayudarán a automatizar tareas cotidianas y a mejorar la calidad del código. Cada vez que se abre el proyecto del graficador 3D con Visual Studio Code, se muestra un mensaje para instalar las extensiones recomendadas, en caso de que todas no estén instaladas. Dichas extensiones se pueden encontrar acá [31]. A modo de resumen, estas:

- facilitan las detecciones de errores reportados por ESLint;
- permiten detectar las faltas de ortografías en los comentarios del código
- y permiten desplegar el histórico de cambios de Git en cada línea de código, lo que mejora la auto documentación del código, ya que ahora cada línea

es enriquecida con la documentación que dejó el colaborador al introducir el cambio.

Junto a las extensiones recomendadas, se dejó una configuración preestablecida [32] en el repositorio para aprovechar el potencial de las extensiones. Esta configuración permite hacer dos cosas al guardar un archivo:

- formatear el código usando Prettier, por lo cual, los usuarios no tienen que preocuparse de ejecutar un comando para aplicar Prettier.
- resolver automáticamente todos los errores "resolubles" reportados por ESLint. Como ESLint es un analizador estático de código, también sabe cómo se pueden resolver ciertos errores reportados por las reglas, por lo cual, al guardar un archivo se intentan resolver todos los errores posibles del código.

El fin de todas estas extensiones y configuración es dar a los futuros colaboradores una ayuda extra para trabajar con el repositorio, manteniendo así los criterios de calidad y reduciendo los tiempos ociosos que se pueden inducir por distintos factores.

5.2.4. TypeScript para el código fuente del graficador 3D

Como se concluyó en las pruebas de concepto (ver anexo B), en este proyecto se integró TypeScript para realizar la implementación del graficador 3D en sustitución del código JavaScript.

En la primera fase de migración del código TypeScript a JavaScript, se pudieron detectar más de 30 errores al momento de *transpilar* el TypeScript a JavaScript. Varios de esos errores eran en realidad reportes de que no se realizaban controles para chequear ciertas precondiciones en las funciones, lo cual podría inducir a errores en el *runtime*; otros reportes venían de malas prácticas de programación que podían provocar bugs según cómo se diera el orden de ejecución en el *runtime*; y el resto de los errores eran patrones de implementación que no eran válidos y, por ende, no tenían efecto alguno en el *runtime*.

Pasada la primera fase de resolución de errores y migración del código a TypeScript, se creó un archivo definiciones de tipos [34] para mejorar el tipado estático de las expresiones. Esto permitió definir la estructura de los JSONs que devuelve el módulo del intérprete MateFun para expresar las figuras 3D, tener

intellisense y detección de errores en el IDE en todas las funciones y expresiones que usan figuras 3D; algo que no era posible con el código JavaScript, lo cual dificultaba la detección temprana de errores y la programación, ya que se tenían que acceder a propiedades que podían existir o no en la estructura de objetos que se usaban en la implementación.

El proceso de migración del código fuente a TypeScript no solo permitió detectar errores y mejorar la calidad del código escrito, sino que permitió detectar más fácilmente la presencia de código muerto[8] que llegaba al JavaScript de la aplicación en producción. Con esto, se lograron hacer optimizaciones en el peso total del paquete final, lo que logró amortiguar casi completamente el aumento de peso de los JavaScripts del repositorio dado por la implementación de las nuevas funcionalidades. En la tabla 5.8.3.1 se puede ver una comparación del peso de los paquetes de la descarga inicial de la aplicación MateFun legada, contra el actual MateFun el cual contiene todas las nuevas funcionalidades más la eliminación del código muerto.

5.3. Entorno de ejecución

En este proyecto no fue necesario actualizar el entorno de ejecución Wildfly en donde se aloja la aplicación Back-end y Front-end, dado que la actualización de tecnologías del Front-end no impacta en este entorno y, además, uno de los requisitos del cliente era apegarnos a las tecnologías existentes siempre que se pudiera.

5.4. Implementación de casos de uso

En esta sección detallaremos en alto nivel cómo fueron implementados cada caso de uso, para así comentar sobre las tecnologías y las decisiones tomadas en el proceso.

Se hace énfasis en que, como se describe en el documento de pruebas de concepto, gran parte de la implementación de las funcionalidades ya venían dadas por las pruebas de concepto, lo que permitió tener tiempo para reforzar la arquitectura del código del graficador 3D legado, manteniendo así la modularización y calidad del código. Bajo este punto, la primera sección que se presenta en este apartado habla de mejoras y *refactorings* en la implementación que se hicieron

[8] Código no usado

sobre el repositorio del graficador 3D, para así evitar la duplicación de código y permitir integrar más fácilmente la implementación de los casos de uso.

5.4.1. Mejoras en la arquitectura del código del graficador 3D

Una de las primeras tareas que se realizó antes de empezar a implementar los casos de uso, fue entender lo que hacían las diferentes variables y funciones existentes en el código, para así comprender mejor los diferentes flujos del mismo. En este punto, se vio una oportunidad de mejora en términos de código autodocumentado [37], ya que varias funciones y variables estaban presentadas con muchos acrónimos. Es por esto que se decidió mejorar la nomenclatura de nombres a lo largo del repositorio, además, para los casos en que las variables y/o funciones eran muy complejas, se usó JSDoc [38] para documentar el funcionamiento de las mismas, lo que permitió que mejorar el *intellisense* del IDE.

Se realizaron varios *refactorings* para mejorar la reutilización de código existente. Entre otras funcionalidades, la característica principal que se mejoró para evitar su duplicación en el código era el graficador de figuras. El mismo estaba dividido en dos implementaciones por separado; una para graficar figuras y otra para graficar figuras que están dentro de una secuencia. La diferencia entre ambas implementaciones radicaba en que el segundo caso *cacheaba* los resultados de la creación de figuras, para que así pudieran ser reutilizados próximamente cuando se volviera a tener que crear dichas figuras. Debido a que las figuras iban a poder tener aristas, vértices y/u otra información al momento de graficarlas, fue necesario factorizar la implementación del graficador de figuras para reutilizar todo el código posible, evitando así tener que implementar en ambos algoritmos la creación de las aristas, vértices y otra información. Con esto, se llegó a una implementación compartida para estos dos graficadores de figuras, donde todo el código era reutilizado y el graficador de secuencias además podía implementar caché para las figuras.

También se trabajó en modernizar el JavaScript existente en el graficador 3D legado, usando así sintaxis más comprimida (*sintaxis sugar*) para implementar las funcionalidades, sin perder así la claridad y legibilidad del código. Además, se mejoraron patrones de diseño para bajar los órdenes de complejidad en algunos algoritmos. Un ejemplo de los patrones aplicados se encuentra acá [39].

5.4.2. Modificaciones de los casos de uso del MateFun legado.

En esta sección se describen los ajustes que se hicieron para los casos de uso ya existentes en el MateFun legado.

5.4.2.1. Mejoras en el sistema de iluminación de las figuras 3D

Se ajustó la implementación legada del sistema de iluminación del graficador 3D, para así pasar a una escena en donde la luz de las figuras siempre esté dada por la dirección en que se las mira. En la Figura 5.4.2.1.1 se puede apreciar cómo funcionaba el antiguo sistema de iluminación, mientras que en la 5.4.2.1.2 se puede apreciar cómo funcionaba el nuevo sistema.

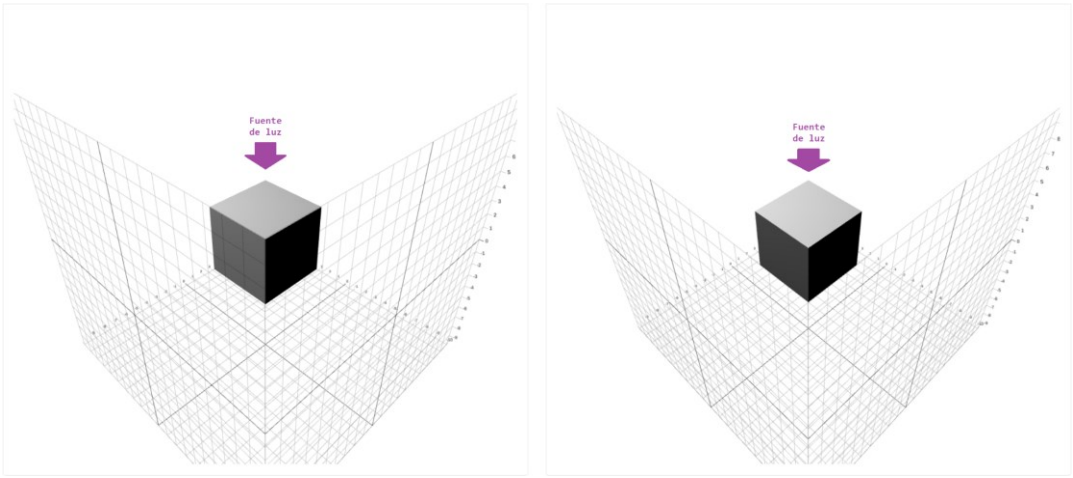

Figura 5.4.2.1.1: Escena con un prisma visto desde distintos ángulos en el sistema de iluminación del MateFun legado.

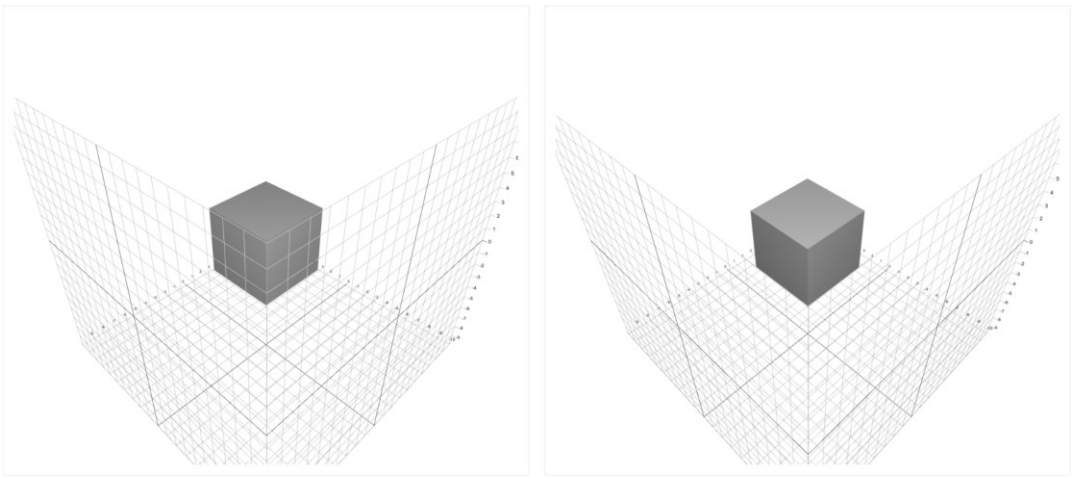

Figura 5.4.2.1.2: Escena con un prisma visto desde distintos ángulos en el sistema de iluminación del MateFun actual.

5.4.2.2. Renombrar la primitiva cubo a prisma

La primitiva `cubo` del intérprete MateFun fue renombrada a `prisma`, cambiando las traducciones existentes de la misma en el intérprete MateFun. Este enfoque permitió no tener que modificar la implementación existente en el intérprete y en el graficador de figuras 3D para que se adapte al cambio de nombres, así este cambio fue totalmente transparente y sin impacto alguno en la implementación de las distintas áreas.

5.4.2.3. Eliminar la primitiva cilindro

Para eliminar la primitiva cilindro del conjunto básico de funciones MateFun, se eliminó dicha primitiva del intérprete -se borraron constructores, chequeos de tipos, evaluaciones y demás- y se eliminaron las traducciones de esta en el intérprete. La implementación de dicha primitiva no fue borrada del graficador de figuras 3D, ya que como se verá en la sección 5.4.4.2, fue usada para implementar `juntarFigEn3D`.

5.4.3. Casos de uso básicos

En los casos de uso de esta sección el usuario solo usa una primitiva con sus correspondientes argumentos para realizar las acciones.

5.4.3.1. Graficar poliedro regular

Para este caso de uso se agregaron cuatro nuevas primitivas, `tetraedro`, `octaedro`, `dodecaedro` e `icosaedro`, al intérprete MateFun y se agregó soporte para dibujar sus respectivas figuras en el graficador 3D.

Las cuatro nuevas primitivas del intérprete MateFun siguen las mismas ideas que la primitiva esfera, por lo que ambas implementaciones son muy parecidas en el intérprete.

Siguiendo con el graficador 3D, para cada nueva primitiva del intérprete MateFun se agregó su correspondiente representación en el módulo Figures (ver Figura 4.1.2.1). Las representaciones de las figuras están basadas en las primitivas que expone ThreeJS para crear las mismas:

- Tetraedro [40]
- Octaedro [41]
- Dodecaedro [42]
- Icosaedro [43]

5.4.4. Casos de uso resultado de la programación

En los casos de uso de esta sección el usuario tiene que componer funciones para poder llevarlos a cabo.

5.4.4.1. Configurar transparencia de una figura 3D

Para este caso de uso se agregó una nueva primitiva, `transparencia3D`, al intérprete MateFun y se adaptó la construcción de figuras en el graficador 3D para que soporten transparencia.

La nueva primitiva del intérprete MateFun sigue las mismas ideas que la primitiva `color3D`, por lo que ambas implementaciones son muy parecidas en el intérprete.

Siguiendo con el graficador 3D, como en la primera toma de contacto se mejoró la arquitectura del código de este para así aumentar la reutilización, la creación de las mallas MeshLambertMaterial [44] para las figuras quedó abstraída a una única función, así, este caso de uso se pudo implementar agregando un parámetro más a la implementación de dicha primitiva, como se puede ver en el Código 5.5.4.1.1.

```
/**
 * Dada una figura, retorna un objeto que corresponde a la
 * configuración de su malla.
 * @param figure Una figura de MateFun
 */
export const getFigureConfiguration = (
  figure: Figure3DStyle
): MeshLambertMaterial => {
  const opacity = 1 - figure.transparency;

   const configuration: MeshLambertMaterialParameters = {
     color: new Color(figure.color),
     opacity,
     transparent: true
   };

   return new MeshLambertMaterial(configuration);
};
```

Código 5.4.4.1.1: Implementación de obtener la configuración de una figura.

5.4.4.2. Crear una figura 3D mediante dos polígonos

Para este caso de uso se agregó una nueva primitiva, `juntarFigEn3D`, al intérprete MateFun y se agregó soporte para dibujar las posibles figuras de esta primitiva en el graficador 3D.

La nueva primitiva del intérprete MateFun sigue las mismas ideas que la primitiva `juntar3D`, por lo que ambas implementaciones son muy parecidas en el intérprete. La diferencia más notoria que tiene `juntarFigEn3D` sobre `juntar3D` -en

términos de implementación-, es que `juntarFigEn3D` hace dos chequeos de errores sobre sus parámetros:

- Comprobar que las dos figuras 2D pasadas son círculos, rectángulos o polígonos.
- Si se cumple lo anterior y son polígonos, comprobar que la cantidad de vértices de los polígonos coinciden.

Si no se cumple alguna de las condiciones, el intérprete devuelve un error. El chequeo de errores fue implementado en el evaluador de expresiones del intérprete y no en el chequeo de tipos, ya que estos controles son semánticos y no sintácticos[9]. Para implementar dichos controles de errores, se siguió el ejemplo de la primitiva `primero`[10] del intérprete MateFun.

Siguiendo con el graficador 3D, la implementación de la primitiva `juntarFigEn3D` del intérprete deriva en tres subcasos que se detallan a continuación.

Dos círculos. Acá se distinguen dos subcasos más:

- El centro de los círculos coincide. En este caso se tiene que dibujar un cilindro, donde los radios de las tapas son los radios de los círculos. Para hacer esto, se usa la primitiva cilindro que estaba implementada en el graficador 3D legado.
- El centro de los círculos no coincide[11]. En este caso se tiene que dibujar un cilindro oblicuo. Para hacerlo, se discretizan[12] ambos círculos en polígonos, derivando así la implementación al caso de usar dos polígonos. El motivo por el cual se hizo esto, es porque no se encontró en ThreeJS alguna primitiva que permitiera representar cilindros oblicuos. La cantidad de aristas (la calidad de la aproximación) de los polígonos, depende del parámetro calidad configurado en el menú desplegable de la Web de MateFun (ver Figura 5.4.5.1). En la Figura 5.4.3.2.2 se puede ver una comparativa de la calidad de aproximación a medida que mejora la cantidad de aristas.

[9] Estos errores aparecen cuando los tipos de los parámetros es correcto -la sintaxis es correcta-, pero las funciones 2D usadas no son las adecuadas -la semántica no es correcta-
[10] La función `primero :: A* -> A`, devuelve el primer elemento de una lista. Si la lista está vacía, tiene que devolver un error.
[11] Este caso puede suceder cuando en unos de los círculos se aplicó la primitiva mover.
[12] En matemáticas aplicadas, la discretización es el proceso de transferir funciones continuas, modelos, variables y ecuaciones a contrapartes discretas [46].

Dos rectángulos. En este caso se tiene que dibujar un poliedro. Para hacer esto, se convierten las representaciones de ambos rectángulos a polígonos y así se deriva la implementación al caso de usar dos polígonos.

Dos polígonos. Al igual que antes, en este caso se tiene que dibujar un poliedro. Para dibujar el poliedro, se ideó un algoritmo que descompone el problema en dos subpartes:

1. Graficar los polígonos superior e inferior que forman parte de las "tapas" de la nueva figura.
2. Graficar cada lateral de la nueva figura.

Una ilustración de lo que hace el algoritmo se puede apreciar en la Figura 5.4.4.2.1.

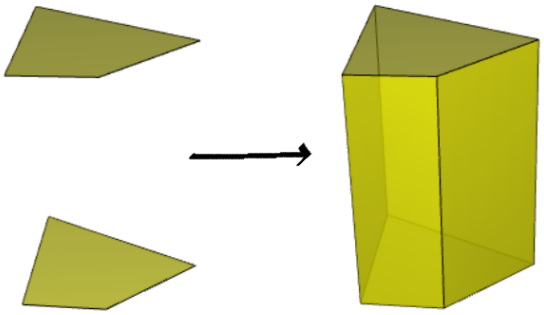

Figura 5.4.4.2.1: Dos polígonos unidos en 3D mediante caras laterales.

Para graficar cada polígono se implementó una función que, dada la triangulación[13] del polígono, gráfica un polígono en un espacio 3D. Por este motivo, la segunda parte del algoritmo es muy directa, dado que cada lateral se descompone en dos triángulos.

Para la primera parte del algoritmo (graficar las tapas de la nueva figura) se usa una librería de JavaScript llamada Earcut [36], la cual permite descomponer en triángulos los polígonos que se usan en `juntarFigEn3D`. El motivo por el cual se hizo esto, es porque en ThreeJS no se encontró alguna primitiva que permitiera triangularizar un polígono, entonces en una posterior fase de investigación y pruebas, se llegó a que la librería más indicada para implementar esta funcionalidad, en términos de rendimiento, escalabilidad, robustez y popularidad era Earcut. No se optó por implementar esta funcionalidad desde cero, ya que requería de investigación de varios *papers* y muchas horas de implementación, las

[13] La triangulación de un polígono o área poligonal es una partición de dicha área en un conjunto de triángulos por un conjunto maximal de diagonales que no se cruzan [47].

cuales tampoco aseguraban el mismo nivel de calidad que una librería que lleva años haciendo dicha funcionalidad.

Vale la pena señalar que se realizaron varias optimizaciones para la implementación de algunos casos bordes `juntarFigEn3D` en el graficador 3D:

- Cuando se usa `juntarFigEn3D` con círculos, si alguno de los círculos tiene radio 0, el círculo no se dibuja en la escena, ya que no es necesario, dado que se quiere representar un cono en la escena.
- Cuando se usa `juntarFigEn3D` con dos polígonos (idem para dos rectángulos), si alguno de los polígonos tiene que todos sus vértices son la misma coordenada (por ejemplo, un polígono es `[(1,1), (1,1), (1,1)]`), no dibuja el polígono en la escena y tampoco se haya su descomposición en triángulos, ya que no es necesario, dado que se quiere representar una pirámide en la escena.

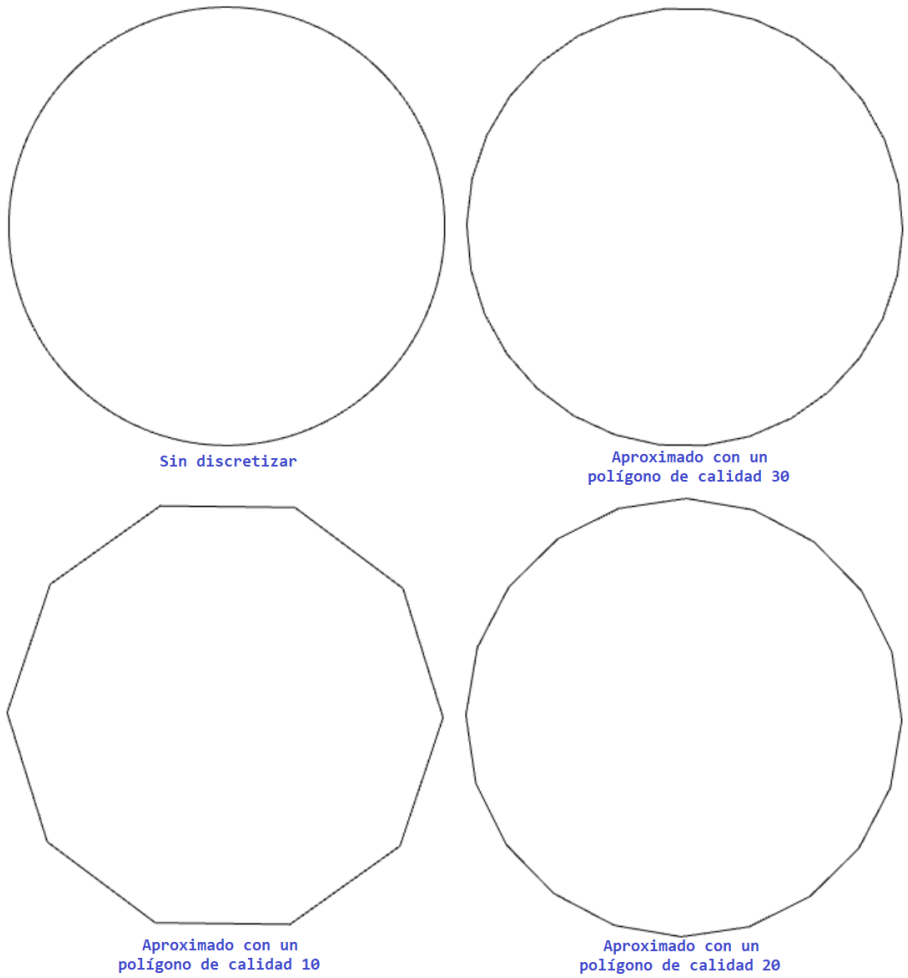

Figura 5.4.3.2.2: Comparación de aproximaciones de un círculo de radio 2 por polígonos con 10, 20 y 30 aristas.

5.4.5. Caso de uso en el despliegue del menú de configuración

En los casos de uso de esta sección el usuario solo usa el menú desplegable de configuración de las figuras 3D para poder realizar las acciones.

Para los casos de uso referentes a esta sección, en la Web de MateFun se agregaron nuevas opciones en el menú desplegable de configuración de las figuras 3D. Estas nuevas opciones se pueden apreciar en la Figura 5.4.5.1.

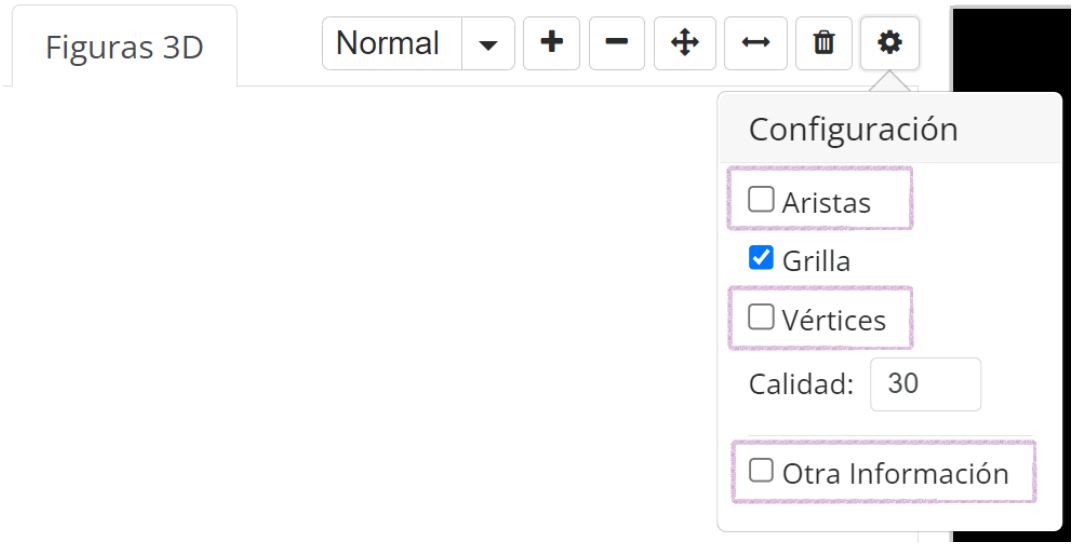

Figura 5.4.5.1: Nuevas opciones en la Web de MateFun para el menú desplegable de configuración de las figuras 3D.

5.4.5.1. Visualizar aristas de figuras 3D

Para este caso de uso, en la Web de MateFun se agregó una nueva opción en el menú desplegable de configuración de las figuras 3D, "Aristas", y se adaptó la construcción de figuras en el graficador 3D para que soporten la configuración de aristas.

La implementación de esta nueva opción en la Web de MateFun siguió las ideas usadas para la implementación de la opción "Grilla" del mismo menú.
Siguiendo con el graficador 3D, sin ser por las figuras creadas con `juntarFigEn3D`, el representado de las aristas de las figuras se hizo utilizando la geometría de las figuras en conjunto con la primitiva EdgesGeometry [45] de ThreeJS, la cual permite obtener la malla de aristas de una geometría dada. Luego, para el caso de `juntarFigEn3D`, la implementación se hizo en dos casos:

- Dos círculos con el mismo centro. En este caso, con `juntarFigEn3D` se dibuja un cilindro, por lo cual, se agregan dos círculos para las tapas del cilindro.

Combinando la primitiva CircleGeometry [54] con la ya nombrada EdgesGeometry, se logra obtener estas aristas. Esta funcionalidad quedó abstraída en una función para su futura reutilización.

- El resto de los casos. Como se vio en la sección 5.4.4.2, estos casos terminaban siendo adaptados a dos polígonos, así, se procede a representar las aristas de los polígonos usando la primitiva EdgesGeometry y luego para las aristas de los laterales se dibujan segmentos de recta usando la información de los polígonos y la primitiva Line [48] de ThreeJS, como se ve en la Figura 5.4.4.2.1.

Al igual que se mencionó al final de la sección 5.4.4.2, esta primitiva tiene algunas optimizaciones para `juntarFigEn3D`. En particular, no se dibujan las aristas si alguno de los círculos tiene radio cero o si alguno de los polígonos tiene que todos sus vértices son la misma coordenada.

5.4.5.2. Visualizar otra información de figuras 3D

Para este caso de uso, en la Web de MateFun se agregó una nueva opción en el menú desplegable de configuración de las figuras 3D, "Otra información", y se adaptó la construcción de figuras en el graficador 3D para que soporten la configuración de "Otra información".

La implementación de esta nueva opción en la Web de MateFun siguió las ideas usadas para la implementación de la opción "Grilla" del mismo menú.
Siguiendo con el graficador 3D, las figuras que tienen "Otra información" son la esfera, cilindro y cilindro oblicuo; estos últimos dos representados por medio de `juntarFigEn3D`. Para representar las líneas punteadas que corresponden a la información complementaria de las figuras, se usó la primitiva LineDashedMaterial [49] de ThreeJS.

Al igual que se mencionó al final de la sección 5.4.4.2, esta primitiva tiene una optimización para `juntarFigEn3D`, y es que no se crean las líneas punteadas cuando el radio de algún círculo es 0, ya que no es necesario, debido a que se quiere representar un cono.

5.4.5.3. Visualizar vértices de figuras 3D

Para este caso de uso, en la Web de MateFun se agregó una nueva opción en el menú desplegable de configuración de las figuras 3D, "Vértices", y se adaptó la construcción de figuras en el graficador 3D para que soporten la configuración de vértices.

La implementación de esta nueva opción en la Web de MateFun siguió las ideas usadas para la implementación de la opción "Grilla" del mismo menú.
Siguiendo con el graficador 3D, las figuras que tienen vértices son los poliedros regulares y las figuras hechas con `juntarFigEn3D` con dos rectángulos o dos polígonos. Debido a que ThreeJS no cuenta con alguna primitiva para representar los vértices de las figuras, los mismos son implementados mediante esferas de tamaño reducidos, las cuales se ubican en los correspondientes vértices de las figuras 3D. Para ubicar los vértices del prisma y de las figuras hechas con `juntarFigEn3D`, se utiliza toda la información disponible de las figuras y la ubicación es bastante directa. Para ubicar los vértices del resto de poliedros regulares, se tuvo que además conocer parte de la implementación del bajo nivel de los mismos [50] [51] [52] [53], para así poder ubicar sus vértices.

Al igual que se mencionó al final de la sección 5.4.4.2, esta primitiva tiene algunas optimizaciones para `juntarFigEn3D`. En particular, si alguno de los polígonos tiene que todos sus vértices son la misma coordenada, solo se agrega una esfera para representar el vértice de dicho polígono, ya que se está queriendo dibujar una pirámide en dicho caso.

Por último, se destaca que debido a los *refactorings* mencionados en la sección 5.4.1, la implementación de estos casos de uso está abstraída a funciones apartes, con lo cual no hubo que hacer casi ningún ajuste más para el graficador de figuras pudiera representar bien las aristas, otra información y vértices, cuando se usan figuras normales o figuras dentro de una secuencia (las cuales se cachean). Así, si en algún momento se cambia la implementación de estos casos de uso, el graficador de figuras es transparente ante dichos cambios.

5.5. Problemas encontrados

Cuando se inició el proyecto el primer problema que se encontró fue que no existía documentación para poder configurar las estaciones de trabajo, para poder tener un entorno de desarrollo para poder extender la aplicación MateFun. Esto era debido a que la documentación existente (Wiki de Gitlab y Readme de los repositorios) era muy reducida, carecía de detalles, y la configuración e instalación de las distintas tecnologías estaba explicada por medio de enlaces antiguos los cuales a día de hoy son inaccesibles porque están caídos. Esto provocó un gran retraso -en términos de semanas de trabajo- al inicio del proyecto, porque no era posible configurar las estaciones de trabajo con todas las tecnologías necesarias para desarrollar.

Otro de los problemas encontrados fue al trabajar con el código legado del graficador 3D; el mismo carecía de comentarios, documentación en las distintas funcionalidades, y los nombres de las variables y funciones no eran nemotécnicos. En general, le quedaba bastante trabajo para que se considerara "código autodocumentado".

Uno de los problemas que surgió al momento del desarrollo y el cual no fue relevado antes en las pruebas de concepto, fue la librería que se optó en primera instancia para triangularizar los polígonos de la primitiva `juntarFigEn3D`. Como resultado de las pruebas de concepto se optó por usar la librería Delaunator [35] para triangularizar los polígonos, ya que cumplía con todos los requisitos de calidad que se buscaban y además satisfació todos los casos de uso probados en su momento. El problema que tuvo esta librería es que no servía para triangulizar polígonos cóncavos [57] -casos de usos que no fueron probados en las pruebas de concepto-, con lo cual, se tuvo que buscar otra librería que permitiera triangulizar cualquier polígono. Como resultado se halló la librería Earcut que nuevamente volvió a cumplir con todos los requisitos de calidad y demostró su eficacia a la hora de triangularizar cualquier tipo de polígono, ya sea convexo o cóncavo.

Otro de los problemas al trabajar con figuras 3D, fue entender las diferencias entre el espacio continuo -con precisión infinita- de las figuras 3D y el espacio discreto -con precisión finita- de las computadoras. Esto se vio especialmente reflejado a la hora de implementar la figura cilindro oblicuo (que puede ser presentada uniendo dos círculos de centros distintos con `juntarFigEn3D`). Como dicha figura no es una primitiva de ThreeJS, había que implementar el *render* de

esta a mano, lo cual consumió gran parte de tiempo puesto que no encontrábamos una solución para implementar el *render* de la figura. Una vez tuvimos implementado `juntarFigEn3D` para unir dos polígonos cualesquiera en 3D, nos dimos cuenta de que podíamos aproximar los dos círculos del cilindro oblicuo por polígonos y así usar lo que teníamos de `juntarFigEn3D` para dibujar un cilindro oblicuo.

5.6. Dedicación relativa

En la Figura 5.6.1 se encuentra la dedicación realizada en este proyecto relativa a cada mes y dividida por área; donde "Pruebas de la aplicación" abarca todas las pruebas que fueron realizadas con el fin de testear algún ámbito de la aplicación.

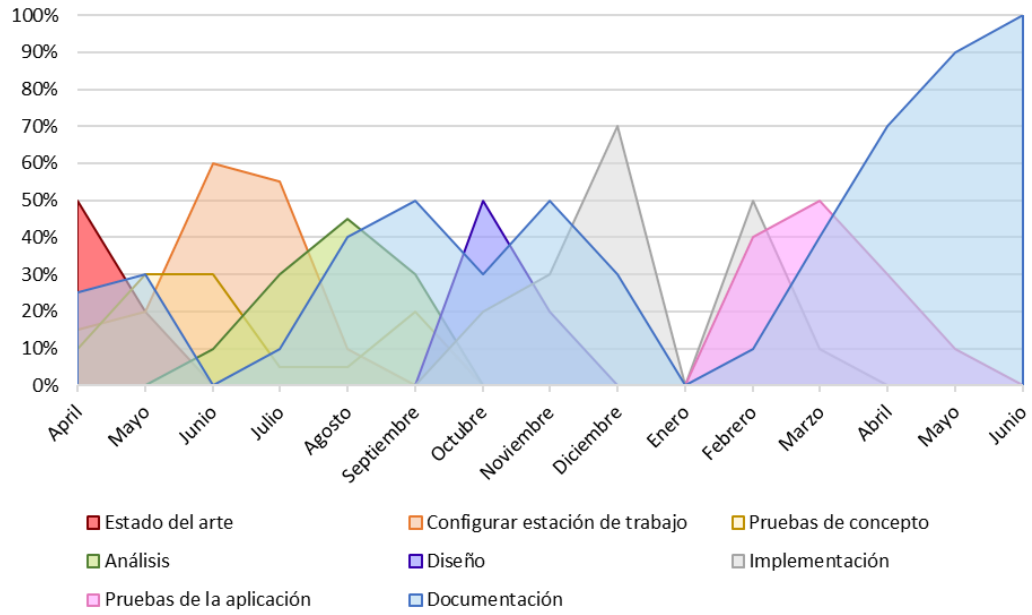

Figura 5.6.1: Dedicación relativa por cada mes.

5.7. Pruebas realizadas

Debido a que gran parte del desarrollo del proyecto estuvo enfocado en el Frontend del módulo graficador 3D, se hicieron muchas pruebas sobre la aplicación cliente MateFun. A su vez, también se hicieron pruebas de las funcionalidades del módulo intérprete.

Para realizar las pruebas de la aplicación MateFun se desplegó la misma en un ambiente muy cercano al de producción (*staging*), para emular lo mejor posible el comportamiento que iba a tener la aplicación legada luego de los cambios. Este ambiente era un servidor local Wildfly, que contaba con la configuración que tiene la aplicación desplegada en producción, en particular, la aplicación MateFun usaba

el modo producción de Angular [55]. Adicionalmente, se desplegó otra instancia de la aplicación MateFun, pero en su versión legada, con motivos de poder corroborar que los casos de uso legados -y otros factores como el rendimiento-, seguían teniendo el mismo resultado esperado en el MateFun actual.

El equipo de pruebas contó con un procesador AMD Ryzen 9 7900X, 2*16GB de RAM a 6000MHz CL32 y disco duro de estado sólido. El entorno de pruebas se obtuvo virtualizando en el programa VMWare el sistema operativo Fedora KDE Plasma 35, donde se le configuró para que use todos los núcleos del procesador, 12GB de RAM y 60 GB del disco duro de estado sólido. El navegador usado para probar la aplicación MateFun fue Opera, donde se le deshabilitó el caché de recursos (desde las *developer tools*) para obtener resultados más cercanos a los que experimentaría un usuario en el peor caso.

5.7.1. Pruebas unitarias y de regresión del módulo intérprete

Se realizaron pruebas unitarias y de regresión sobre el módulo del intérprete con el fin de corroborar el correcto funcionamiento primitivas. Las pruebas de regresión consistieron en probar todos las primitivas existentes con relación a las figuras 3D del MateFun legado, en el nuevo MateFun. En este caso, el resultado esperado se corroboraba cuando los resultados de las primitivas del MateFun legado y el actual coincidían, exceptuando la información de la transparencia que se agregaba en el resultado del MateFun actual.

Para probar las funciones, se hizo un plan de test que consistió en probar cada primitiva de manera unitaria, y luego ir componiendo esa primitiva con variantes posibles para formar nuevas representaciones. Por ejemplo, se tomó la primitiva `prisma`, se probó esta primitiva con algunos valores posibles y luego se probó la misma componiéndola con las transformaciones:

- `color3D`
- `mover3D`
- `rotar3D`
- `escalar3D`
- `juntar3D`

Sobre cada una de estas nuevas representaciones, se repitió el proceso componiendo con otras transformaciones, y así sucesivamente hasta completar la cadena de transformaciones posibles para la primitiva `prisma`.

Una vez completadas las pruebas de regresión, se hizo un plan de test muy parecido al anterior para probar todas las nuevas primitivas y posibilidades que brinda el MateFun actual. Por ejemplo, se tomó la primitiva `tetraedro`, se probó esta primitiva con algunos valores posibles y luego se probó la misma componiéndola con las transformaciones

- `color3D`
- `mover3D`
- `rotar3D`
- `escalar3D`
- `juntar3D`
- `transparencia3D`
- `juntarFigEn3D`

Para las nuevas primitivas se hicieron pruebas exhaustivas para el control de errores que implementaban, con el fin de corroborar que cumplían con la especificación dada por el análisis y diseño planteado. Por ejemplo, para primitiva la `transparencia3D` se probaron valores de transparencia enteros y reales con valores positivos y negativos, e incluso con valores que excedían el rango típico de uso. Otro caso fue el de la primitiva `juntarFigEn3D`, que se probó con distintas configuraciones de polígonos corroborando que sólo admitiera dos círculos, dos rectángulos o dos polígonos y, en el caso de especificar dos polígonos, se probaron polígonos con distintas cantidades de vértices para chequear que se controlaba que los polígonos tuvieran la misma cantidad de vértices.

5.7.2. Pruebas de sistema y de regresión de la aplicación MateFun

Se realizaron pruebas de sistema y de regresión sobre la aplicación MateFun con el fin de corroborar el correcto funcionamiento de los casos de uso en el MateFun actual. Las pruebas de regresión consistieron en probar todos los casos de uso existentes con relación a las figuras 3D del MateFun legado, en el nuevo MateFun. En este caso, el resultado esperado se corroboraba cuando los resultados de los casos de uso del MateFun legado y el actual coincidían.

Para probar los casos de uso legados, se hizo un plan de test que consistió en probar cada primitiva de manera unitaria, y luego ir componiendo esa primitiva con variantes posibles para formar nuevas representaciones. Por ejemplo, se tomó la primitiva `prisma`, se probó esta primitiva con algunos valores posibles y luego se probó la misma componiendola con las transformaciones:

- `color3D`
- `mover3D`
- `rotar3D`
- `escalar3D`
- `juntar3D`

Sobre cada una de estas nuevas representaciones, se repitió el proceso componiendo con otras transformaciones, y así sucesivamente hasta completar la cadena de transformaciones posibles para la primitiva `prisma`.

Para el caso de uso de graficar secuencia de figuras 3D también se hicieron pruebas de regresión muy parecidas a las anteriores, en donde se tenía una lista de figuras cuya complejidad iba aumentando según se iba recorriendo la misma.

En simultáneo con estas pruebas, se realizaron pruebas de los casos de uso tales como rotar las figuras, el zoom in/out, cambiar la calidad de las representaciones, mostrar/ocultar la grilla de la escena, con el fin de verificar el correcto funcionamiento de los casos de uso en el MateFun actual.

Una vez completadas las pruebas de regresión, se hizo un plan de test muy parecido al anterior para probar todas las nuevas primitivas y posibilidades que brinda el MateFun actual. Por ejemplo, se tomó la primitiva `tetraedro`, se probó esta primitiva con algunos valores posibles y luego se probó la misma componiendola con las transformaciones

- `color3D`
- `mover3D`
- `rotar3D`
- `escalar3D`
- `juntar3D`
- `transparencia3D`
- `juntarFigEn3D`

A su vez, se probaron estas combinaciones alternando las nuevas opciones de configuración de figuras: aristas, vértices y otra información.
Adicionalmente, se corroboró que las traducciones siguieran funcionando en el nuevo MateFun y, más aún, que las traducciones estuvieran presentes en las nuevas primitivas y mensajes de errores al usar las mismas.

Por último, se probó el editor de texto que tiene integrado MateFun para crear archivos, con el fin de corroborar que funcionaba el *intellisense* del mismo para las nuevas primitivas, cambios y controles de errores que tiene el MateFun actual. Como ejemplo de esto, se hizo el manual que resume cómo usar todas las nuevas funcionalidades y todos los cambios del MateFun actual (ver anexo B).

5.7.3. Pruebas de rendimiento de la aplicación MateFun

Se realizaron varias pruebas de rendimiento en la aplicación MateFun con el fin de corroborar que los nuevos cambios no habían impactado gravemente en el rendimiento del MateFun legado.

La primera prueba que se realizó fue un test de Lighthouse [56] en el MateFun legado y en el MateFun actual, con el objetivo de medir el impacto en los tiempos de carga inicial que habían tenido los nuevos cambios. Este test es muy importante en las aplicaciones Web, porque realiza un muy buen análisis con gran cobertura sobre el estado de la aplicación y todos los problemas que puede tener. Este test fue realizado 5 veces y promediado para cada versión de MateFun (legado y actual). Los resultados se pueden ver en la tabla 5.8.3.1 en dónde además se agrega el tamaño del paquete (en comprimido) que se tiene que descargar al inicio de la aplicación para poder usarla.

Métrica		**MateFun legado**	**MateFun actual**
Performance	First Contentful Paint	1.8s	1.9s
	Largest Contentful Paint	2.6s	2.6s
	Total Blocking Time	0.05s	0.07s
	Cumulative Layout Shift	0.001	0.001
	Speed Index	1.8s	1.9s
	Puntaje	77	76
Accessibility		65	65
Best Practices		100	100
SEO		78	78
Tamaño de paquete de la carga inicial (comprimido)		2.21MB	2.3MB

Tabla 5.7.3.1: Comparación de resultados de Lighthouse y peso de paquete inicial del MateFun legado y actual.

Como se puede observar en la Tabla 5.8.3.1, los resultados de Lighthouse y tamaño del paquete inicial para ambas versiones de MateFun son muy parecidos,

siendo que el nuevo MateFun tiene un punto menos en la métrica Performance, lo cual es un resultado muy bueno teniendo en cuenta la gran cantidad de funcionalidades que fueron incluidas en la versión actual de MateFun. Este resultado fue posible gracias a todas las optimizaciones realizadas que fueron mencionadas en la sección 5.2.4. El motivo del porque se tiene un punto menos en la métrica Performance está directamente relacionado con el tamaño del paquete de la carga inicial, dado que ahora el MateFun actual tiene que descargar más JavaScript para implementar todas las nuevas funcionalidades (menos de 100 KBs), la carga inicial toma unos pocos milisegundos más en ocurrir, lo que penaliza mínimamente la Performance.

Las siguientes pruebas que se hicieron estuvieron más orientadas al rendimiento en *runtime* de la aplicación. Para ello, se hizo uso de la sección Performance que incluyen las *developer tools* de los navegadores. Esta sección permite realizar mediciones de muchas características de las aplicaciones, destacándose poder medir:

- Consumo de CPU
- Consumo de heap
- Frames animados
- Cantidad de nodos HTML
- Cantidad de *listeners* de eventos
- Tiempos de *loading*, *scripting*, *rendering* y *painting*

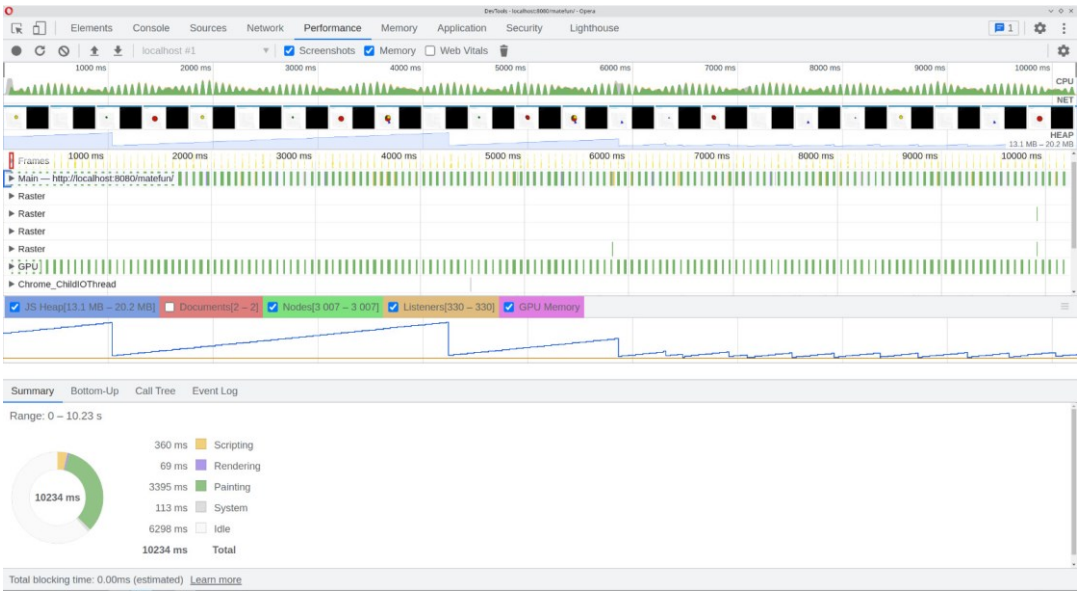

Figura 5.7.3.2: Sección Performance de las *developer tools* del navegador Opera.

Se utilizó esta herramienta para medir el rendimiento de varios escenarios típicos de uso -desde los más simples hasta los más complejos-, en ambas versiones de MateFun (legado y actual), con el fin de analizar si existían grandes diferencias en el rendimiento del *runtime*. Se realizaron los siguientes conjuntos de pruebas para ambas versiones:

1. Probar cada comando de manera unitaria; luego mostrar muchas figuras y rotarlas con el mouse y por último ejecutar una secuencia figuras.
2. Ejecutar una secuencia durante 2 minutos con el intervalo de tiempo por defecto.
3. Ejecutar una secuencia durante 10 segundos con el intervalo de tiempo más corto.

En la tabla 5.8.3.3 se puede ver el resumen del resultado de estas pruebas.

	Prueba 1 (duración: 46s)		**Prueba 2** (duración: 120s)		**Prueba 3** (duración: 10s)	
Métrica	Legado	Actual	Legado	Actual	Legado	Actual
JS Heap	13.0 - 18.6 MB	10.9 - 14.4 MB	11.6 - 15.6 MB	10.1 - 11.8 MB	13.1 - 20.2 MB	11.8 - 13.1 MB
Nodes	3007 - 3561	3168 - 3596	3022 - 3062	2142 - 2180	3007 - 3007	3205 - 3206
Listeners	328 - 339	327 - 337	328 - 330	244 - 246	330 - 330	329 - 329
Loading	5 ms	5 ms	0 ms	0 ms	0 ms	0 ms
Scripting	492 ms	448 ms	372 ms	328 ms	360 ms	272 ms
Rendering	115 ms	113 ms	79 ms	78 ms	69 ms	77 ms
Painting	2461 ms	2359 ms	2572 ms	2754 ms	3395 ms	3756 ms
System	204 ms	204 ms	150 ms	151 ms	113 ms	53 ms

Tabla 5.7.3.3: Pruebas de rendimiento entre MateFun legado y actual para las tres pruebas.

Como se puede observar en la Tabla 5.7.3.3, ambas versiones de MateFun obtienen resultados muy parecidos en cada prueba, siendo que ninguna versión de MateFun parece mejor que la otra. La única tendencia que parece existir es que la versión de MateFun actual consumió un poco menos de JS Heap en las ejecuciones, al igual que los tiempos de Scripting fueron levemente reducidos. Estas mejoras son directamente justificadas por las optimizaciones realizadas al código (comentadas en la sección 5.4.1), con el fin de mejorar la complejidad de los algoritmos, reutilizar más el código y modernizar las implementaciones.

Adicionalmente para la prueba 3 se midieron los tiempos entre frames de animación para ambas versiones de MateFun. Estos tiempos en su mayoría fueron de 16,7ms, con esperas mínimas de 33,4ms y 50,1ms. Este es un resultado muy bueno, porque demostró que ambas versiones mantenían su *frametime* en el nivel de lo esperado[14].

Como resultado de las pruebas, el rendimiento y fluidez del MateFun actual sigue siendo igual de bueno -o incluso mejor- que sobre el MateFun legado, lo cual es un resultado muy bueno teniendo en cuenta la cantidad de agregados que aporta esta versión.

Por último, se realizó una cuarta prueba para medir el rendimiento en *runtime* del MateFun actual, con respecto a los nuevos casos de uso implementados. Para ello, se probó mediante una secuencia varias figuras que se formaban como composiciones de funciones de todos los casos de uso[15]. Algunos ejemplos de las figuras logradas en esta secuencia se pueden observar en las Figuras 5.7.3.4 y 5.7.3.5. Adicionalmente, en el ciclo de ejecución de la secuencia se probó:

- Mostrar y ocultar las aristas
- Mostrar y ocultar los vértices
- Mostrar y ocultar la otra información de las figuras
- Rotas las figuras
- Hacer Zoom In y Out
- Mostrar y ocultar la grilla

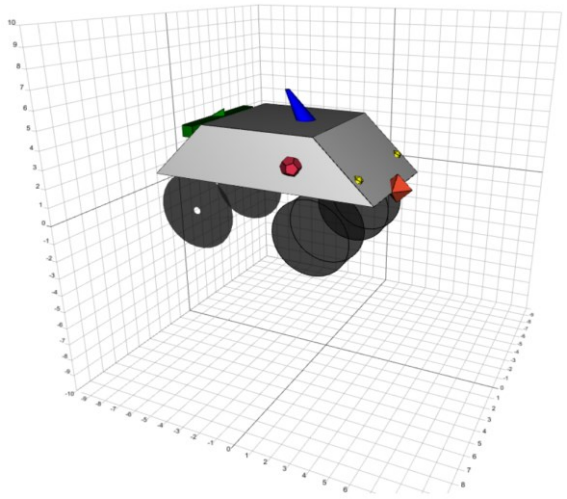

Figura 5.7.3.4: Auto hecho con las primitivas del MateFun actual.

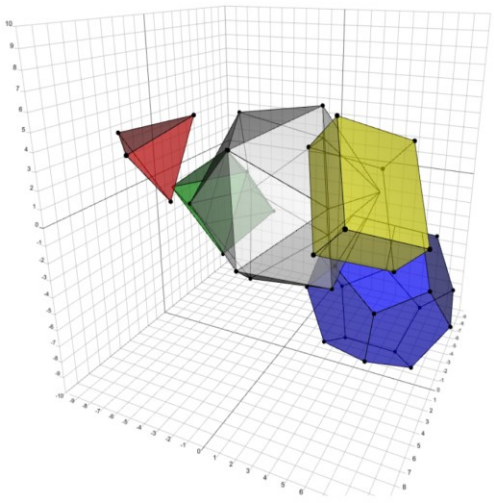

Figura 5.7.3.5: Poliedro hecho con las primitivas del MateFun actual.

[14] El valor esperado para el *frametime* es de 16,7ms, dado que la velocidad de refresco del monitor empleado para las pruebas era de 60 frames por segundos. Entonces 1 seg / 60 frames = 0.01666… seg ≈ 16,7ms

[15] Esta secuencia de pruebas se puede encontrar en el manual que resume cómo usar todas las nuevas funcionalidades y todos los cambios del MateFun actual.

La prueba realizada duró un minuto y sus resultados se pueden apreciar en la figura Figura 5.7.3.6.

Figura 5.7.3.6: Resultados de la cuarta prueba para el MateFun actual.

Como se puede apreciar en la figura 5.7.3.6 en la prueba el consumo de JS Heap se mantuvo acotado entre los 11.0 MB y 17.4MB, valores muy buenos teniendo en cuenta que en la prueba hubo altos momentos de intensidad (rotas las figuras, hacer zoom in/out, cambiar la visibilidad de las aristas, etc), los cuales se reflejan por los picos verdes en la gráfica de CPU. Además, cuando los momentos de intensidad cesaban, el consumo de JS Heap disminuía indicando que la memoria era liberada después de que no era más usada.

En general, el resultado refleja el comportamiento esperado que se venía viendo en las anteriores pruebas; en los momentos de intensidad el consumo de CPU y JS Heap incrementan, pero luego de que bajan los niveles de intensidad, los recursos son liberados. Lo cual demuestra que los nuevos casos de uso no tienen *memory leaks* ni reflejan mayores costes en su implementación, al compararlos con los resultados de las pruebas anteriores para las funcionalidades ya existentes.

5.7.4. Pruebas de sistema en la aplicación MateFun desplegada en producción

Para la aplicación MateFun actual desplegada en producción, se realizó una cobertura muy parecida a las pruebas nombradas de la sección 5.7.2 con el fin de verificar que los casos de uso seguían funcionando correctamente en el ambiente de producción. Adicionalmente, se realizaron algunas pruebas de rendimiento a las nombradas en la sección 5.7.3, donde se percibió la misma tendencia de rendimiento -entre el MateFun legado de producción y el actual de producción- sobre la comparativa realizada en dicha sección.

6. Conclusiones y trabajo a futuro

Una vez finalizado el proyecto, en esta sección se presentan las conclusiones obtenidas sobre el mismo y, a su vez, se presentan diversas líneas de investigación futuras.

6.1. Conclusiones

Las implementaciones de las nuevas funcionalidades y mejoras a MateFun de este proyecto de grado fueron desarrolladas a lo largo de, aproximadamente 10 meses de la asignatura Proyecto de Grado de Ingeniería en Computación de la Facultad de Ingeniería UdelaR.

Lo primero que se hizo fue investigar el estado del arte de las tecnologías para representar cuerpos 3D en los navegadores Web, con el fin de corroborar que la tecnología legada de MateFun cumplía todos los requerimientos funcionales necesarios para este proyecto. A su vez, se comparó la aplicación MateFun contra otras aplicaciones para representar cuerpos 3D, con el fin de analizar fortalezas y debilidades de estas. Esto permitió comprender cómo está situado MateFun en este ámbito y además dio lugar a varias líneas de desarrollo futuras.

Luego de varías iteraciones de pruebas de concepto y análisis de requerimientos, se logró llegar a un conjunto de requerimientos mayor que el planteado inicialmente en el proyecto, lo cual derivó en una cantidad mayor de casos de uso. Esto permitió dar lugar a un proyecto más grande, el cual satisfacía los requisitos de varias funcionalidades que las docentes deseaban tener.

Fue una experiencia desafiante, que nos aportó en varias áreas y nos permitió conocer más sobre cómo funcionan las computadoras en el campo del renderizado 3D.

Relativo al aporte que legamos para futuros proyectos de MateFun, el código del graficador 3D fue actualizado y mejorado para seguir las buenas prácticas de la industria en cuanto a la mantenibilidad y extensibilidad, dando como resultado un código robusto y prácticamente autodocumentado.

Finalmente, el proyecto excede las expectativas, se cumplieron todos los requerimientos funcionales, se documentó un manual de cómo configurar la estación de trabajo para desarrollar sobre MateFun, y se documentó un manual de

apoyo para poder utilizar la aplicación con las nuevas funcionalidades. Esto enriquecerá la enseñanza y el conocimiento de aquellos quienes lo usen.

6.2. Trabajo a Futuro

En las líneas de investigación futura se listan a continuación problemas que quedaron sin resolver, o funcionalidades que podrían expandirse.

Configurar el grosor de las líneas de las aristas

Esta característica no es posible implementarla [19], debido a que el motor de renderizado (WebGL) que usa el graficador 3D, no soporta configurar el grosor de las aristas. Tampoco en este proyecto fue una opción cambiar el motor de renderizado que usaba el graficador, ya que esto es parte de la tecnología base del mismo, la cual se debía mantener siempre que fuera posible por requisitos del cliente.

Visualizar área y volumen de las figuras puestas en la escena

Esta visualización podría darse como un recuadro de texto donde se encuentra la grilla, pasando el mouse por cada figura y obtenerlo de forma individual, o que lo devuelva la consola.

Líneas punteadas para las figuras que se encuentran detrás de otras

En el caso de que haya dos figuras superpuestas con transparencia no nula, con una de ellas ubicada detrás de la otra, se desearía que las aristas de la figura que está detrás se represente mediante una línea punteada. De esta manera, se puede evitar la confusión visual entre las aristas de ambas figuras y facilitar la comprensión de la geometría subyacente. Un ejemplo de esto es la Figura 6.2.2.

Visualización de la "red" de caras que componen la figura.

Un ejemplo de esto se encuentra en la Figura 6.2.2.

Visualización de las figuras con un sistema de iluminación real

Dar la opción de usar un sistema de iluminación en el graficador, para que las figuras tengan una visualización con parámetros físicos con respecto a la luz (sombra proyectada, luz difusa); más cercano a la realidad.

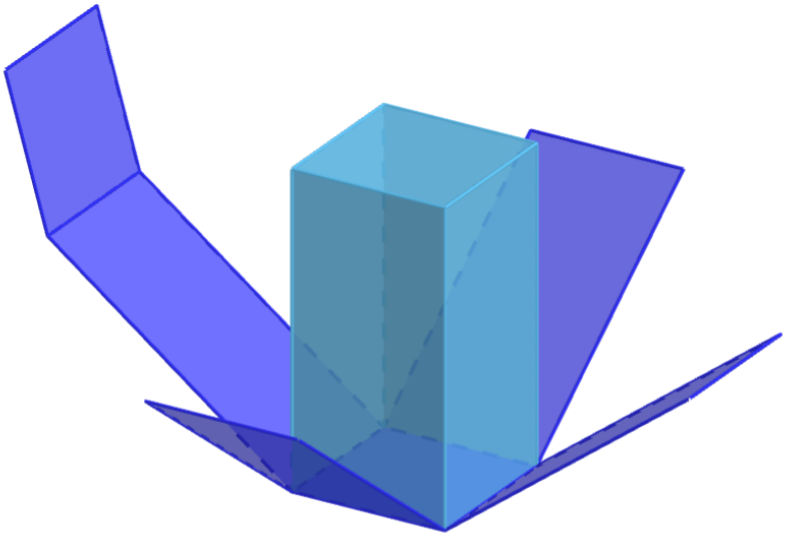

Figura 6.2.2: Prisma con su "red" y líneas punteadas detrás de la superficie.

Cambiar la forma en que se rotan figuras que están agrupadas con juntar3D

Cuando se agrupan figuras 3D, intuitivamente el usuario creería que sí aplica la transformación `rotar3D`, se rotaría todo el centro del conjunto de las figuras, y no cada figura por separado. Actualmente, las figuras rotan sobre su propio eje a pesar de pertenecer a un conjunto que forma una nueva figura. Esto hace que se den comportamientos contraintuitivos al juntar varias figuras para formar un elemento de la realidad y luego rotarlo. En la Figura 6.2.3 se muestra un ejemplo de esto.

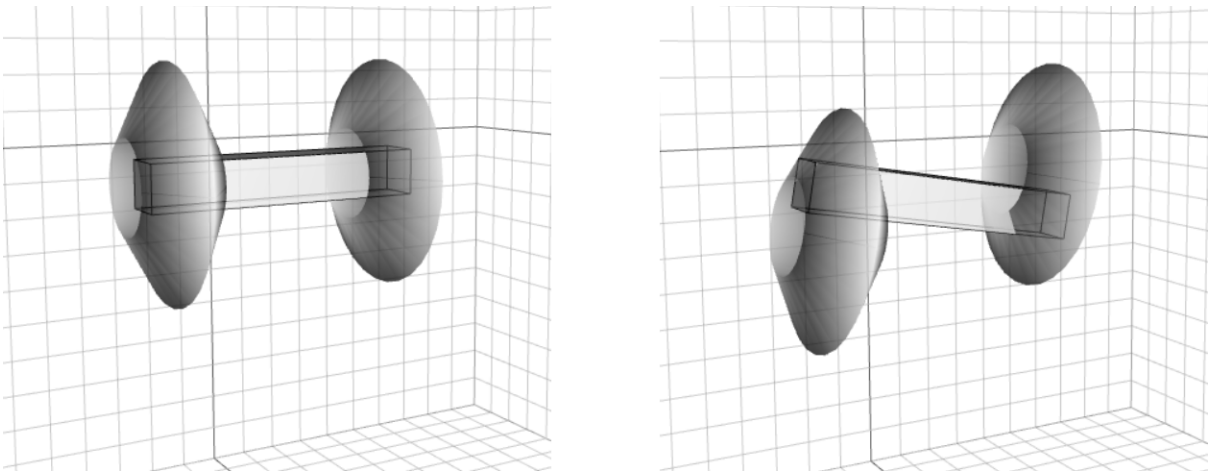

Figura 6.2.3: Comparación del antes y después al aplicar la rotación (0, -10, 0) a la figura `transparencia3D(juntar3D(juntar3D(anillo(1, 3, 1.5), mover3D(anillo(1, 3, 1.5), (5,0,0))), mover3D(prisma(1,1,6), (2.5,0,0))), 0.25)`

Determinar si la figura resultado tiene sentido cuando se usa juntarFigEn3D

Cuando se usa la primitiva `juntarFigEn3D` con dos polígonos con la misma cantidad de vértices, no hay garantía alguna que asegure que la figura resultado tenga algún sentido, según los polígonos que usó el usuario. Por ejemplo, el usuario podría intentar unir dos polígonos que son muy incompatibles o incluso podría tratar de rotar el conjunto de vértices de un polígono. Para estos casos se podría intentar cualificar mediante algún algoritmo cuando una figura resultado "se va a ver bien", según algún criterio definido, así, se puede enviar una advertencia, error o directamente se puede intentar corregir el resultado.

Poder representar polígonos con agujeros.

Actualmente, los polígonos que admite la primitiva `juntarFigEn3D` son polígonos que no contienen agujeros [22], ya que no hay una forma explícita para representar dichas figuras. Si se soportara esta funcionalidad, se podrían representar poliedros como los de la Figura 6.2.4

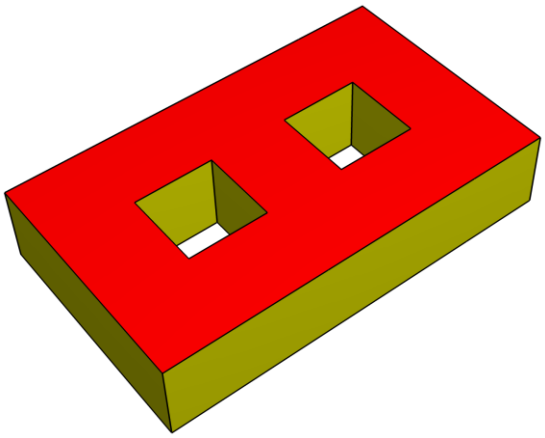

Figura 6.2.4: Poliedro con dos agujeros.

Escribir tests unitarios con captura de pantalla para probar el correcto funcionamiento de las figuras

En la versión existente de Matefun, el graficador 3D no cuenta con ningún tests unitario que ayude a la mantenibilidad del repositorio y promueva la detección de errores ante la regresión de funcionalidades. La forma de hacer tests unitarios para este módulo, sería haciendo tests de captura de pantalla, en donde, ante una secuencia de acciones, se toma una captura de pantalla de lo obtenido y se compara contra el resultado esperado. Jest [20] y Puppeteer [21] son dos tecnologías que sirven para desarrollar este tipo de tests con JavaScript.

Optimizar el resultado de los JSON de respuesta del intérprete para no incluir los valores por *default*

En la versión existente de MateFun, el intérprete ante el uso de una primitiva (o composición de primitivas) de manera exitosa, devuelve un JSON con la configuración de la figura a graficar. Esta configuración contiene valores por defecto que son rellenados por el intérprete, los cuales no son necesarios si estos valores por defecto se asumen en los módulos del graficador 2D y 3D. Por ejemplo, al pedir el JSON resultado de la función **`rotar3D(transparencia3D(tetraedro(3), 0.75), (90,45,75))`** se obtiene la Figura 6.2.5. En este resultado se muestra la información por defecto del centro y color de la figura a graficar. Dicha información podría no ser incluída, tal como se ve en la Figura 6.2.6.


```
FIG3D: [
  {
    "kind": "tetrahedron",
    "r": 3.0,
    "x": 0.0,
    "y": 0.0,
    "z": 0.0,
    "color": "white",
    "rot": (90.0, 45.0, 75.0),
    "transparency": 0.75
  }
]
```


Figura 6.2.5: JSON de la figura **`rotar3D(transparencia3D(tetraedro(3), 0.75), (90,45,75))`** sin optimizar


```
FIG3D: [
  {
    "kind": "tetrahedron",
    "r": 3.0,
    "rot": (90.0, 45.0, 75.0),
    "transparency": 0.75
  }
]
```


Figura 6.2.6: JSON de la figura **`rotar3D(transparencia3D(tetraedro(3), 0.75), (90,45,75))`** optimizado

7. Referencias

[1] MateFun. (2020). *Qué es MateFun*.
http://d11.com.uy/web/matefun/dev03/que-es-matefun/

[2] Three.js. *PolyhedronGeometry*.
https://threejs.org/docs/#api/en/geometries/PolyhedronGeometry

[3] Wikipedia. (2022). *JavaScript*.
https://en.wikipedia.org/wiki/JavaScript

[4] Couriol, B. (2020). *Airbnb Releases Tool to Convert Large Codebases to Typescript*.
https://www.infoq.com/news/2020/08/airbnb-typescript-migration/

[5] Stack Overflow. (2021). *Stack Overflow, most loved and dreaded technology*.
https://insights.stackoverflow.com/survey/2021#section-most-loved-dreaded-and-wanted-programming-scripting-and-markup-languages

[6] Desmos. *Desmos*, Elegant graphing calculator and math simulations promote collaboration.
https://www.commonsense.org/education/website/desmos

[7] Desmos. (2022). *Desmos*, *Accessibility*.
https://www.desmos.com/accessibility?lang=es

[8] GeoGebra. Tutorial: *GeoGebra Examen*.
https://www.geogebra.org/m/hybx6mzs

[9] Rey, D., Fagian, I. y Rosano, L. (2019). *Representación gráfica interactiva de funciones matemáticas, figuras geométricas y animaciones en la Web*.
https://gitlab.fing.edu.uy/matefun/Frontend/-/wikis/uploads/4a9f5d4a3ee140511353e54f4f27e1c4/Matefun__1_.pdf

[10] Google Support. Requisitos del sistema para el navegador Chrome.
https://support.google.com/chrome/a/answer/7100626?hl=es-419

[11] Mozilla. (2022). *Firefox System Requirements*.

https://www.mozilla.org/en-US/firefox/104.0.1/system-requirements/

[12] Opera. (2022). Opera Browser requirements.
https://www.opera.com/es-419/download/requirements

[13] caniuse. (2022). *Can I use: SVG*.
https://caniuse.com/?search=SVG

[14] caniuse. (2022). *Can I use: WebGL*.
https://caniuse.com/?search=webgl

[15] caniuse. (2022). *Can I use: WebSocket*.
https://caniuse.com/?search=web%20socket

[16] caniuse. (2022). *Can I use: ES6*.
https://caniuse.com/?search=es6

[17] Microsoft. (2022). *Internet Explorer Downloads*.
https://support.microsoft.com/en-us/windows/internet-explorer-downloads-d49e1f0d-571c-9a7b-d97e-be248806ca70

[18] Cameto, G. y Méndez, M. (2017). *Proyecto de Grado MateFun. Documento de Arquitectura y diseño*.
https://gitlab.fing.edu.uy/matefun/Frontend/uploads/41c0624b45ad9d14d855c08262cc31cd/arquitectura-y-diseno__1_.pdf

[19] ThreeJS (2016). *LineBasicMaterial: A material for drawing wireframe-style geometries.*
https://threejs.org/docs/#api/en/materials/Linea Basic Material.linewidth

[20] Facebook. (2023). A comprehensive JavaScript testing solution. Works out of the box for most JavaScript projects.
https://github.com/facebook/jest

[21] Puppeteer. (2023). *Puppeteer: A Node.js library which provides a high-level API to control Chrome/Chromium over the DevTools Protocol*.
https://github.com/puppeteer/puppeteer

[22] Wikipedia. (2023). *Polygon with holes*. https://en.wikipedia.org/wiki/Polygon_with_holes#:~:text=In%20geometry%2C%20a%20polygon%20with,they%20are%20not%20frequently%20needed.

[23] StackOverflow. (2022). *Developer Survey 2022*. https://survey.stackoverflow.co/2022/

[24] Visual Studio Code https://visualstudio.microsoft.com/es/#vscode-section

[25] Visual Studio Live Share. (2022). *Desarrollo en colaboración en tiempo real* https://visualstudio.microsoft.com/es/services/live-share/

[26] ESLint. (2023). *Find and fix problems in your JavaScript code* https://eslint.org/

[27] MateFun. (2023). *Reglas de configuración para ESLint en el graficador 3D*. https://gitlab.fing.edu.uy/matefun/graph3d/-/blob/master/.eslintrc.json

[28] npm. (2023). *Specifying dependencies and devDependencies in a package.json file*. https://docs.npmjs.com/specifying-dependencies-and-devdependencies-in-a-package-json-file

[29] Prettier. (2022). *Prettier: an opinionated code formatter*. https://prettier.io/

[30] MateFun. (2023). *Reglas de configuración para Prettier en el graficador 3D*. https://gitlab.fing.edu.uy/matefun/graph3d/-/blob/master/.prettierrc.json

[31] MateFun. (2023). *Extensiones recomendadas cuando se usa Visual Studio Code con el repositorio del graficador 3D*. https://gitlab.fing.edu.uy/matefun/graph3d/-/blob/master/.vscode/extensions.json

[32] MateFun. (2022). *Configuración del TypeScript*. https://gitlab.fing.edu.uy/matefun/graph3d/-/blob/master/tsconfig.json

[33] Blue, L. (2022). *Discover three.js.*
https://discoverthreejs.com/

[34] MateFun. (2023). *Definiciones de tipos*.
https://gitlab.fing.edu.uy/matefun/graph3d/-/blob/master/src/interfaces.ts

[35] mapbox. (2021). *Delaunator: An incredibly fast and robust JavaScript library for Delaunay triangulation of 2D points.*
https://github.com/mapbox/delaunator

[36] mapbox. (2022). *Earcut: The fastest and smallest JavaScript polygon triangulation library.*
https://github.com/mapbox/earcut

[37] hmn.wiki. *Código autodocumentado*.
https://hmn.wiki/es/Self-documenting

[38] JSDoc. (2017). *JSDoc: An API documentation generator for JavaScript*.
https://github.com/jsdoc/jsdoc

[39] Durán, E. (2018). *Why I prefer objects over switch statements*.
https://enmascript.com/articles/2018/10/22/why-I-prefer-objects-over-switch-statements

[40] Three.js. (2016). *TetrahedronGeometry: A class for generating a tetrahedron geometries.*
https://threejs.org/docs/#api/en/geometries/TetrahedronGeometry

[41] Three.js. (2016). *OctahedronGeometry: A class for generating an octahedron geometry.*
https://threejs.org/docs/#api/en/geometries/OctahedronGeometry

[42] Three.js. (2016). *DodecahedronGeometry: A class for generating an dodecahedron geometry.*
https://threejs.org/docs/#api/en/geometries/DodecahedronGeometry

[43] Three.js. (2016). *IcosahedronGeometry: A class for generating an icosahedron geometry.*

https://threejs.org/docs/#api/en/geometries/IcosahedronGeometry

[44] Three.js. (2016). *MeshLambertMaterial: A material for non-shiny surfaces, without specular highlights.*
https://threejs.org/docs/#api/en/materials/MeshLambertMaterial

[45] Three.js. (2016). *EdgesGeometry: This can be used as a helper object to view the edges of a geometry.*
https://threejs.org/docs/#api/en/geometries/EdgesGeometry

[46] Wikipedia. (2022). *Discretización*.
https://es.wikipedia.org/wiki/Discretización

[47] Wikipedia. (2022). *Triangulación de un polígono*.
https://es.wikipedia.org/wiki/Triangulación_de_un_polígono

[48] Three.js. (2016). *Line: A continuous line*.
https://threejs.org/docs/#api/en/objects/Line

[49] Three.js. (2016). *LineDashedMaterial: A material for drawing wireframe-style geometries with dashed lines*.
https://threejs.org/docs/#api/en/materials/LineDashedMaterial

[50] Three.js. (2022). *TetrahedronGeometry.js: Implementation*
https://github.com/mrdoob/three.js/blob/master/src/geometries/TetrahedronGeometry.js#L7

[51] Three.js. (2022). *OctahedronGeometry.js: Implementation*
https://github.com/mrdoob/three.js/blob/master/src/geometries/OctahedronGeometry.js#L7

[52] Three.js. (2022). *DodecahedronGeometry.js: Implementation*
https://github.com/mrdoob/three.js/blob/master/src/geometries/DodecahedronGeometry.js#L10

[53] Three.js. (2022). *IcosahedronGeometry.js: Implementation*
https://github.com/mrdoob/three.js/blob/master/src/geometries/IcosahedronGeometry.js#L9

[54] Three.js. (2022). *CircleGeometry: a simple shape of Euclidean geometry*. https://threejs.org/docs/?q=CircleGeometry#api/en/geometries/CircleGeometry

[55] Google. (2023). *Angular - Deployment: Production optimizations* https://angular.io/guide/deployment#production-optimizations

[56] Google. (2023). *Lighthouse overview*. https://developer.chrome.com/docs/lighthouse/overview/

[57] Aguirre, I.l (2023). *Clasificación de polígonos: Cóncavos y convexo*. https://www.geogebra.org/m/G8dWmRJh

8. Anexos

8.1. Anexo A

Para comparar el rendimiento de GeoGebra y MateFun se realizó un test simple que consistía en graficar una esfera de radio 6 y rotar la misma durante 5 a 6 segundos. Este experimento se realizó monitoreando la Performance con las opciones de desarrollador del navegador Opera. En la Figura 8.1.1 se puede ver el resultado del experimento para GeoGebra y en la Figura 8.1.2 para MateFun.

Adicionalmente, se utilizó una configuración de CPU AMD Ryzen 5 5600X (6 núcleos y 12 hilos) acompañada de 4x8 GB de RAM a 3600 Mhz CL18 con un monitor de resolución 2560x1440p a 144Hz. Como el monitor tiene 144Hz, se espera que el tiempo entre frames sea de 6,944ms[16] para asegurarse de que se renderizan la tasa correcta de frames (144 frames) para el refresco del monitor y la fluidez al rotar la esfera es la óptima.

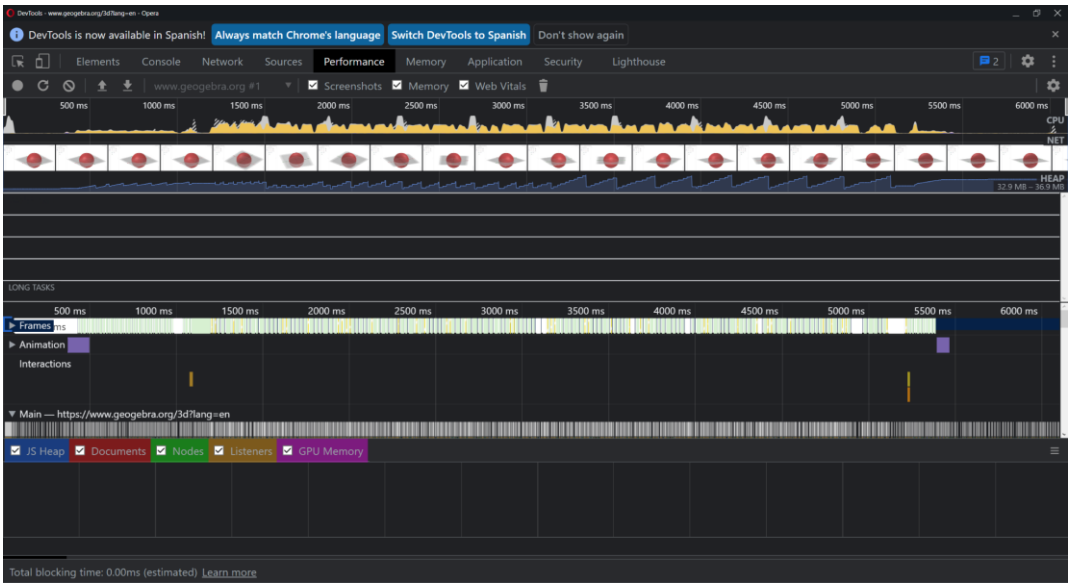

Figura 8.1.1: *Profiling* del experimento para GeoGebra.

[16] 1 segundo / 144 = 0,006944444 segundos. Multiplicado por 1000 (1 seg = 1000 ms), dá como resultado 6,944 ms.

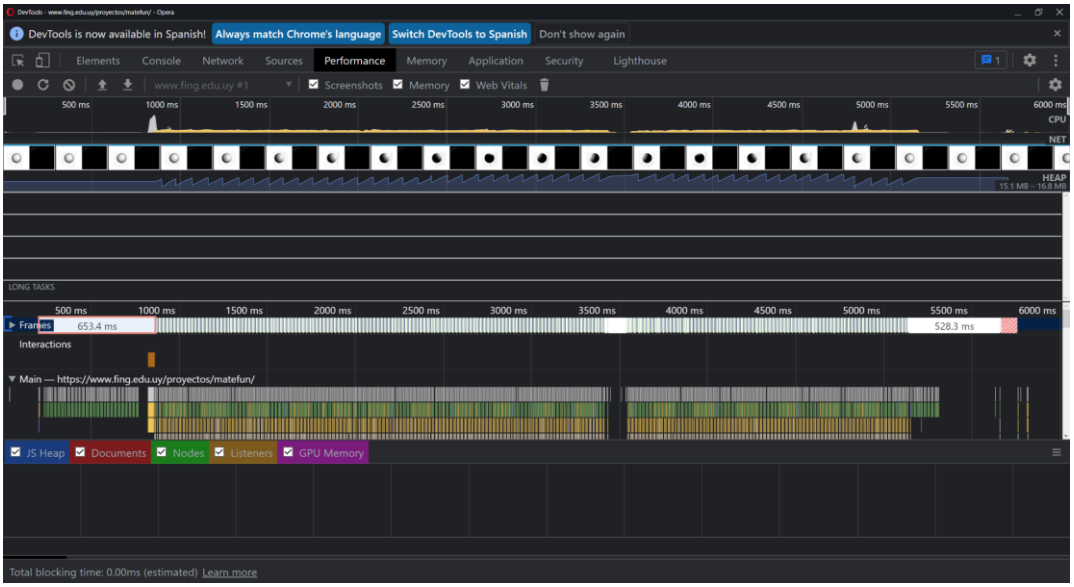

Figura 8.1.2: *Profiling* del experimento para MateFun.

Como se observa en las comparativas, el consumo del Heap es menor para MateFun: 15,1MB - 16,8MB contra 32,9MB - 36,9MB. Además, algo que no se puede apreciar en la captura, pero salía al pasar el mouse por encima de la sección "Frames", es que el tiempo entre los frames para MateFun oscilaba entre 6,9 ms a 7,0 ms, lo esperado dada la configuración que se usó para el experimento. Mientras que, para GeoGebra, el tiempo entre frames varió mucho, desde 13,9 ms a 27,8 ms, lo cual resultó en una experiencia menos fluida y que se sentía "a tirones" en varios momentos, lo cual en MateFun no sucedió.

Por último, GeoGebra presenta mayores picos de consumo de CPU en comparación a MateFun. Esto se puede observar en la sección superior "CPU" de cada figura.

La configuración elegida para este experimento está muy por encima del rendimiento medio que cualquier usuario puede tener en su estación de trabajo, pese a eso, se pudo apreciar que GeoGebra no rendía bien con esta configuración avanzada. En contraparte, como se observó, MateFun si tuvo un buen desempeño.

8.2. Anexo B

Resumen de cambios MateFun 0.16: Manual que resume cómo usar todas las nuevas funcionalidades y todos los cambios del MateFun actual.

Manual de configuración de la estación de trabajo: Documento que describe cómo configurar una estación de trabajo para poder desarrollar en MateFun.

Documento de pruebas de concepto: Este documento tiene todas las pruebas de concepto que fueron realizadas a lo largo del desarrollo del proyecto.

Documento de arquitectura y diseño del sistema: Este documento describe la arquitectura y diseño de los requerimientos de este proyecto.

Documento de requerimientos y casos de uso: Este documento describe el alcance del proyecto, identificando los requerimientos funcionales y no funcionales. A su vez lista los casos de uso.