UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

Documento de pruebas de concepto

Informe 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

1. Introducción

Debido a que el cliente de este proyecto pidió que se explorara la posibilidad de poder expandir las funcionalidades del graficador 3D con la tecnología existente, todas las pruebas de concepto realizadas fueron con el fin de validar los nuevos requisitos. Más aún, se analizaron nuevas características adicionales a los requisitos base, que podrían ser incluidas dependiendo de las estimaciones de tiempo.

2. Configuración del entorno de pruebas

Todas las pruebas fueron realizadas sobre un entorno de ejecución acotado a las tecnologías que usa el graficador 3D. Esto se hizo con el fin de enfocar las pruebas en las funcionalidades clave, sin tener en cuenta -en una primera instancia- la dificultad agregada por la integración de esas nuevas funcionalidades en todo el entorno de desarrollo MateFun.

La configuración del repositorio donde se fueron haciendo las pruebas de concepto, se puede encontrar en [1]. Para resumir la configuración de este, se contó con una aplicación NodeJS [2] basada en la versión de TypeScript 4.6.4 y la versión de ThreeJS 0.140.0. Para levantar un servidor local de ejecución, se utilizó live-server [3]. Todo el JavaScript generado por el proceso de transpilar TypeScript tenía como *target* la versión ES6 de JavaScript.

3. Pruebas de concepto

En las siguientes secciones se presenta un resumen de lo que se analizó en cada prueba de concepto para corroborar la factibilidad de las funcionalidades a agregar. Se van a obviar muchos ejemplos de código en cada apartado, ya que los mismos están presentes en el repositorio en donde se hicieron las pruebas de concepto y en la documentación de la API de ThreeJS que se nombra en cada sección.

3.1. Representar todos los poliedros regulares (tetraedro, octaedro, dodecaedro e icosaedro)

Se investigó la API de ThreeJS en busca de primitivas que permitieran representar el resto de los poliedros regulares que no estaban disponibles en MateFun. Se corroboró que ThreeJS tenía funciones para graficar los poliedros regulares faltantes:

- Tetraedro [4]
- Octaedro [5]
- Dodecaedro [6]
- Icosaedro [7]

Una vez examinadas las primitivas, mediante pruebas de concepto se validó que las mismas pudieran representar los poliedros regulares bajo distintas configuraciones de tamaños y colores.

3.2. Poder configurar la transparencia de las figuras

Se investigó la API de ThreeJS en busca de una primitiva que permitiera configurarles la transparencia a todas las figuras posibles de representar. Se corroboró que ThreeJS permitía configurar la opacidad[1] del tipo de mallas (MeshLambertMaterial [8]) que usa el graficador 3D.

Una vez examinada en detalle la API, se validó mediante pruebas de concepto que todas las figuras ya representables en el MateFun legado más los nuevos poliedros regulares, se pudiera configurar la transparencia y simultáneamente el color de estas.

[1] La opacidad es el opuesto de la transparencia. Si ambos valores se toman entre 0 y 1, la relación se explica por: opacidad = 1 - transparencia.

3.3. Poder mostrar aristas para todas las figuras

Se investigó la API de ThreeJS en busca de una primitiva que permitiera mostrar las aristas de todas las figuras. Se encontró una primitiva [9] de ThreeJS que permitía configurar la visualización de las aristas, dada su geometría. Es decir, esta primitiva permitía configurar las aristas de todas las figuras representables hasta el momento en adición a los poliedros regulares que se querían representar.

Una vez examinada la API en detalle, se realizaron pruebas de concepto para validar que todas las figuras ya representables en el MateFun legado más los nuevos poliedros regulares, podían mostrar sus aristas por medio de dicha primitiva. A pesar de esto, se encontró un problema con la figura Cilindro, debido a que sus aristas no quedaban del todo bien representadas con esta funcionalidad; aparecían líneas que no debían en la superficie curva del cilindro, debido a que su superficie era una descomposición de muchos polígonos. Es por esto que, se optó por representar las aristas del cilindro, graficando dos círculos para demarcar los bordes en cada tapa del mismo.

Usando la geometría para círculos [10] fue posible representar las aristas para el cilindro. Con lo cual, fue factible representar las aristas para todas las figuras.

3.4. Poder configurar el grosor de las aristas

Se investigó la API de ThreeJS en busca de una primitiva o configuración que permitiera cambiar el grosor de las aristas de todas las figuras. Desafortunadamente, se halló que esta característica no era posible representarla, debido a que el motor de renderizado (WebGL) que usa el graficador 3D, no soporta configurar el grosor de las aristas. Tampoco era una opción cambiar el motor de renderizado que usaba el graficador, ya que esto es parte de la tecnología base de este.

Para más información sobre el problema, consultar [11].

3.5. Poder mostrar los vértices de todas las figuras

Se investigó la API de ThreeJS en busca de una primitiva que permitiera mostrar los vértices de todas las figuras. Se corroboró que ThreeJS no tenía primitiva alguna para graficar los vértices de las figuras.

Se analizó la factibilidad de renderizar los vértices de las figuras, representando cada uno de ellos mediante una esfera con un radio muy reducido, donde cada esfera se agrega a la escena. Para este propósito, se encontró la forma de ubicar los vértices de todos los poliedros regulares, ya que las implementaciones de estos (ver [12], [13], [14] y [15]) contenían toda la información de en qué coordenadas se debían ubicar, dada la información (proporciones) de los mismos. Con esto, se realizaron pruebas de concepto que validaron la factibilidad de representar los vértices de todas las figuras usando dicha forma.

3.6. Poder graficar líneas punteadas y para hacer que las mismas representen el radio de figuras como la esfera o cilindro

Se investigó la API de ThreeJS en busca de una primitiva que permitiera graficar líneas punteadas. Se corroboró que ThreeJS permitía graficar líneas punteadas usando [16], por lo cual, se hicieron pruebas de concepto para validar el funcionamiento de dicha primitiva. Una vez se pudo probar la primitiva para graficar líneas punteadas, se usó la misma para representar el radio de la esfera y de las tapas del cilindro, así como la altura del cilindro.

Al realizar pruebas de concepto, se pudo encontrar una relación entre las proporciones de la esfera y cilindro, con las coordenadas en donde debían ser ubicadas sus líneas punteadas, por lo cual, fue factible representar está otra información de las figuras mediante líneas punteadas.

3.7. Poder unir dos figuras 2D (dos polígonos, dos círculos o dos rectángulos) en 3D

Se investigó la API de ThreeJS en busca de alguna primitiva que permitiera unir dos figuras que están contenidas cada una en un plano 2D, en un nuevo cuerpo 3D. Desafortunadamente, dadas las características de la primitiva, no se encontró nada parecido para esta funcionalidad.

Por el anterior motivo, se tuvo que diseñar un algoritmo que, dados dos polígonos con la misma cantidad de vértices, se unieran los polígonos en 3D mediante caras laterales, como lo que muestra la figura 3.7.1.

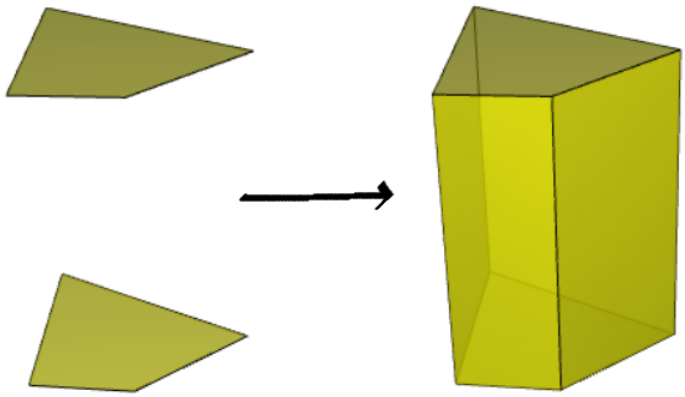

Figura 3.7.1: Dos polígonos unidos en 3D mediante caras laterales.

El algoritmo en cuestión se basa en graficar y agregar varios polígonos a la escena, en donde los polígonos se identifican en dos grupos:

1. Polígonos superior e inferior que forman parte de las "tapas" de la nueva figura.
2. Cada lateral de la nueva figura.

Para graficar cada polígono, se hicieron pruebas de concepto para implementar una función [17] que, dado los triángulos que se descompone el polígono, gráfica un polígono en un espacio 3D. Por este motivo, graficar el segundo grupo de polígonos fue muy directo, dado que cada lateral se descompone en dos triángulos.

El primer grupo de polígonos (las tapas de la nueva figura) requirió de investigación, porque se debía triangularizar cada polígono[2]. Dado que no se encontró en ThreeJS alguna primitiva que permitiera triangularizar un polígono, se investigaron librerías de JavaScript que permitieran hacer esto. Se llegó a que la librería más apropiada en términos de rendimiento, escalabilidad y popularidad era

[2] Descomponer en triángulos cada polígono, donde la unión de todos los triángulos vuelve a reconstruir el polígono.

Delaunator [18], con lo cual, se usó esta librería para triangularizar los polígonos de las tapas de la figura.

Una vez completado el algoritmo anterior, se hicieron pruebas de concepto para validar que también se podía representar las aristas y vértices de estas nuevas figuras, al igual que se validó que se podía configurar la transparencia y color.

Con el algoritmo y todas las características de una figura ya probadas, se concluyó que era factible esta funcionalidad.

3.8. Poder rotar y mover las nuevas representaciones

Se comprobó y validó que todas las nuevas figuras representables, así como todas las nuevas configuraciones de aristas, vértices y líneas punteadas, podían ser rotadas y trasladadas a lo largo de la escena, manteniendo la composición de los subcuerpos. En otras palabras, se comprobó que las nuevas construcciones funcionaban correctamente cuando se las trasladaban y rotaban en la escena.

3.9. Trabajar con TypeScript para realizar las pruebas de concepto

Todas las pruebas de concepto realizadas se programaron usando TypeScript en lugar de JavaScript. Esto se realizó con motivos de comprobar que no había ninguna limitante o dificultad a la hora de implementar parcialmente las nuevas funcionalidades, ya que en el repositorio del graficador 3D, se quería evaluar la posibilidad de migrar todo el código JavaScript existente a TypeScript. Dado que no hubo dificultades a la hora de programar con TypeScript y, más aún, las diferencias de programación entre JavaScript y TypeScript, eran prácticamente nulas, se vio factible el hecho de migrar el código JavaScript del repositorio del graficador 3D a TypeScript.

Referencias

[1] DSilvera-Oct y ncamera (2022). *MatefunPruebas*.
https://github.com/ncamera/MatefunPruebas

[2] Herrera, D. (2023). *Qué es Node.js: Casos de uso comunes y cómo instalarlo*
https://www.hostinger.com.ar/tutoriales/que-es-node-js#:~:text=Conclusión-,Node.,JavaScript%20V8%20de%20Google%20Chrome.

[3] Vierros, T. (2022). *Live Server: A little development server with live reload capability*.
https://www.npmjs.com/package/live-server

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

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

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

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

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

[9] 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

[10] Three.js. (2016). *CircleGeometry*
https://threejs.org/docs/#api/en/geometries/CircleGeometry

[11] Three.js. (2016). *LineBasicMaterial: A material for drawing wireframe-style geometries.*
https://threejs.org/docs/#api/en/materials/LineBasicMaterial.linewidth

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

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

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

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

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

[17] DSilvera-Oct y ncamera (2022). *MatefunPruebas: graphPolygon*
https://github.com/ncamera/MatefunPruebas/blob/main/src/figures.ts#L275

[18] mapbox (2022). *Delaunator: An incredibly fast and robust JavaScript library for Delaunay triangulation of 2D points.*
https://github.com/mapbox/delaunator