



UNIVERSIDAD DE JAÉN
Nombre del Centro

Trabajo Fin de Grado

HERRAMIENTA PARA LA DOCENCIA DE INFORMÁTICA GRÁFICA UTILIZANDO WebGL

Alumno: Luis René Domínguez Fuentes

Tutor: Prof. D. Francisco Daniel Pérez Cano
Cotutor: Prof. D. Juan José Jiménez Delgado
Dpto: Departamento de Informática

Septiembre, 2023

(Página intencionalmente en blanco)



Universidad de Jaén

Departamento de Informática

Don Francisco Daniel Pérez Cano y Don Juan José Jiménez Delgado, tutores del Trabajo Fin de Grado titulado: '**Herramienta para la docencia de Informática Gráfica utilizando WebGL**', que presenta Luis René Domínguez Fuentes, otorgan el visto bueno para su entrega y defensa en la Escuela Politécnica Superior de Jaén.

Jaén, Septiembre de 2023

El alumno:

Los tutores:

Luis René Domínguez Fuentes

Francisco Daniel Pérez Cano
y
Juan José Jiménez Delgado

Agradecimientos

En primer lugar me gustaría darle las gracias a mis padres por brindarme la oportunidad de poder estudiar la carrera, y por su apoyo durante todos estos años.

También a todos aquellos compañeros con los que he podido trabajar codo con codo. Sobre todo a Antonio Javier Armenteros y Martín Ángel Martín, por ser como son y porque sin ellos las prácticas no hubieran sido lo mismo.

Y por último a mi tutor, Francisco Daniel, por toda la ayuda y confianza depositada en mí, tanto para este trabajo como en las clases de IGV y PAG.

Contenido

1 Especificación del trabajo	13
1.1 Introducción	13
1.2 Descripción del punto de partida	14
1.3 Metodología de desarrollo de software	16
1.4 Alcance	17
2. Análisis	18
2.1 Alternativas y viabilidad	18
2.1.1 Bibliotecas	18
2.1.2 Frameworks	20
2.1.3 Entorno de desarrollo.	21
2.2 Solución propuesta	22
2.3 Requisitos	24
2.3.1 Funcionales	24
2.3.2 No Funcionales	27
2.4 Historias de Usuario	28
2.5 Planificación	32
2.5.1 Tareas	32
2.5.2 Diagrama de Gantt	35
2.5.3 Variaciones	36
2.6 Estimación de costes	36
3. Diseño	39
3.1 Diagrama de casos de uso	39
3.1.1 Transformaciones	40
3.1.2 Luz ambiental	41
3.1.3 Luz direccional	42
3.1.4 Luz puntual	43
3.1.5 Foco	44
3.1.6 Selección	45
3.1.7 Grafo de escena	46
3.1.8 Malla de triángulos	47
3.1.9 Aplicación de texturas	48
3.1.10 Bump Mapping	49
3.1.11 Modelos de reflexión	50
3.2 Interfaz	51
3.3 Story Board	52
3.3.1 Transformaciones geométricas	52
3.3.2 Luces	55
3.2.2.1 Luz ambiental	55
3.3.2.1 Luz direccional	57
3.3.2.1 Luz puntual	58
3.3.2.1 Foco de luz	60
3.3.3 Selección	61

3.3.4 Grafo	61
3.3.4 Mallas	63
3.3.5 Texturas	64
3.3.5.1 Aplicación de texturas	64
3.3.5.2 Bump Mapping	66
3.3.5.3 Reflexión	67
4. Desarrollo	68
4.1 Trabajo previo	68
4.2 Primera iteración	68
4.2.1 Escena Básica con Three.js	68
4.2.1 Escena de transformaciones geométricas	72
4.2.3 Luz ambiental	78
4.2.4 Luz direccional	81
4.2.5 Luz puntual	85
4.2.6 Foco	86
4.3 Segunda iteración	88
4.3.1 Selección de objetos	88
4.3.2 Grafo de escena	92
4.3.3 Malla de triángulos	96
4.4 Tercera iteración	99
4.4.1 Aplicación de texturas	99
4.4.2 Bump Mapping	103
4.4.3 Modelos de reflexión	107
4.4.4 Decoración	110
4.5 Cuarta iteración	114
4.5.1 Creación proyecto	114
4.5.2 Vue Router	116
5. Pruebas	118
6. Conclusiones	121
Bibliografía	122
Referencias	122
Ilustraciones	124
Apéndice I. Manual de instalación del sistema	125
Apéndice II. Manual de usuario	130
Apéndice III. Descripción contenido entregado	137

Índice de ilustraciones

■ Ilustración 1.3-1.....	12
■ Ilustración 1.3-2.....	12
■ Ilustración 1.4-1.....	14
■ Ilustración 2.1-1.....	16
■ Ilustración 2.1-2.....	16
■ Ilustración 2.1-3.....	17
■ Ilustración 2.1-4.....	18
■ Ilustración 2.1-5.....	18
■ Ilustración 2.1-6.....	19
■ Ilustración 2.5-1.....	32
■ Ilustración 2.5-2.....	32
■ Ilustración 3.1-1.....	36
■ Ilustración 3.1-2.....	37
■ Ilustración 3.2-2.....	38
■ Ilustración 3.2-2.....	39
■ Ilustración 3.2-4.....	40
■ Ilustración 3.2-5.....	41
■ Ilustración 3.2-5.....	42
■ Ilustración 3.2-5.....	43
■ Ilustración 3.2-5.....	44
■ Ilustración 3.2-5.....	45
■ Ilustración 3.2-5.....	46
■ Ilustración 3.2-5.....	47
■ Ilustración 3.2-1.....	48
■ Ilustración 3.3-1.....	49
■ Ilustración 3.3-2.....	50
■ Ilustración 3.3-3.....	50
■ Ilustración 3.3-4.....	51
■ Ilustración 3.3-5.....	52
■ Ilustración 3.3-6.....	52
■ Ilustración 3.3-7.....	53
■ Ilustración 3.3-8.....	54
■ Ilustración 3.3-9.....	54
■ Ilustración 3.3-10.....	55
■ Ilustración 3.3-11.....	55
■ Ilustración 3.3-12.....	56
■ Ilustración 3.3-13.....	57
■ Ilustración 3.3-14.....	57
■ Ilustración 3.3-15.....	58
■ Ilustración 3.3-16.....	59
■ Ilustración 3.3-17.....	59

■ Ilustración 3.3-18.....	60
■ Ilustración 3.3-19.....	61
■ Ilustración 3.3-20.....	61
■ Ilustración 3.3-21.....	62
■ Ilustración 3.3-22.....	62
■ Ilustración 3.3-23.....	63
■ Ilustración 3.3-24.....	64
■ Ilustración 4.2-1.....	69
■ Ilustración 4.2-2.....	70
■ Ilustración 4.2-3.....	71
■ Ilustración 4.2-4.....	74
■ Ilustración 4.2-5.....	77
■ Ilustración 4.2-6.....	80
■ Ilustración 4.2-7.....	81
■ Ilustración 4.2-8.....	83
■ Ilustración 4.2-9.....	84
■ Ilustración 4.3-1.....	89
■ Ilustración 4.3-2.....	90
■ Ilustración 4.3-3.....	93
■ Ilustración 4.3-4.....	95
■ Ilustración 4.4-1.....	100
■ Ilustración 4.4-2.....	100
■ Ilustración 4.4-3.....	104
■ Ilustración 4.4-4.....	104
■ Ilustración 4.4-5.....	107
■ Ilustración 4.4-6.....	109
■ Ilustración 4.4-6.....	110
■ ilustración 4.5-1.....	114
■ Ilustración AI-1.....	122
■ Ilustración AI-2.....	123
■ Ilustración AI-3.....	125
■ Ilustración AI-4.....	126
■ Ilustración AII-1.....	127
■ Ilustración AII-2.....	128
■ Ilustración AII-3.....	129
■ Ilustración AII-4.....	129
■ Ilustración AII-5.....	130
■ Ilustración AII-6.....	131
■ Ilustración AII-7.....	131
■ Ilustración AII-8.....	132
■ Ilustración AII-9.....	132
■ Ilustración AII-10.....	133

1 Especificación del trabajo

1.1 Introducción

El objetivo final de este Trabajo de Fin de Grado (TFG) es el desarrollo de una Aplicación de Página Única (SPA) haciendo uso de WebGL para ilustrar y explicar conceptos de la asignatura de Información Gráfica y Visualización (IGV), de tal manera que los usuarios puedan interactuar con modelos y escenas, poner en práctica sus conocimientos y facilitar la comprensión de dichos conceptos. Esto surge porque muchos alumnos al cursar dicha asignatura manifiestan dificultades a la hora de estudiar la materia, como por ejemplo:

- Visualización de un espacio tridimensional.
- Situar objetos y modelos en el espacio a través de transformaciones geométricas y comprender la relación entre ellas.
- Entender los sistemas de iluminación.
- Creación de mallas de triángulos.

Por ello para mejorar la comprensión en estos temas, la herramienta se centrará en ofrecer diferentes escenas interactivas donde el usuario pueda experimentar con:

- Aplicación de transformaciones geométricas a modelos en un espacio 3D.
- Utilización de distintos tipos de luces en diferentes escenas.
- Técnicas de selección interactiva de objetos.
- Construcción de un grafo de escena mediante jerarquía de objetos.
- Construcción de mallas de triángulos.
- Uso de materiales y texturas.

1.2 Descripción del punto de partida

WebGL [1] es un motor de rasterización, que sirve para dibujar puntos, líneas y triángulos según el código proporcionado. Este código es ejecutado en la GPU en forma de dos funciones: *vertex shader* y *fragment shader*. Estas funciones están escritas en un lenguaje basado en C/C++ conocido como GLSL [2].

El trabajo de un *vertex shader* es calcular las posiciones de los vértices. En función de los datos de salida, se puede rasterizar varios tipos de primitivas, incluidos puntos, líneas o triángulos. Al rasterizar estas primitivas, se llama a la segunda función proporcionada por el usuario, llamada fragment shader. Su trabajo es calcular un color para cada píxel de la primitiva que se esté dibujando en cada instante.

El inconveniente es que dar los primeros pasos en esta tecnología puede llegar hacerse muy complicado si no se tienen los conocimientos básicos necesarios. Conocer las fases del Pipeline Rendering [3] y saber el tipo de shader involucrado en cada una de ellas, para poder programarlos adecuadamente y conseguir los efectos visuales deseados. Aparte, como ya se ha mencionado, hay que saber GLSL, lo que requiere un cierto tiempo de aprendizaje para algo tan específico como lo es este lenguaje.

Actualmente existen bibliotecas que ayudan a iniciarse, facilitando la configuración inicial de una escena básica y ofreciendo muchas herramientas ya implementadas. Un ejemplo de estas bibliotecas es ThreeJS [4], en cuya página web se puede encontrar documentación introductoria para poder usar la librería, en la que se muestran ejemplos tanto teóricos como prácticos, y muchos de ellos interactivos. De esta manera los usuarios pueden manejar y aplicar conceptos fácilmente sin necesidad de descargarse programas o archivos adicionales.

En la ilustración 1.3-1, se puede observar el ejemplo [5] que explica la geometría de un cubo, y con los slides de abajo se pueden modificar los parámetros en tiempo real, para así poder ver y entender qué representa cada uno de ellos.

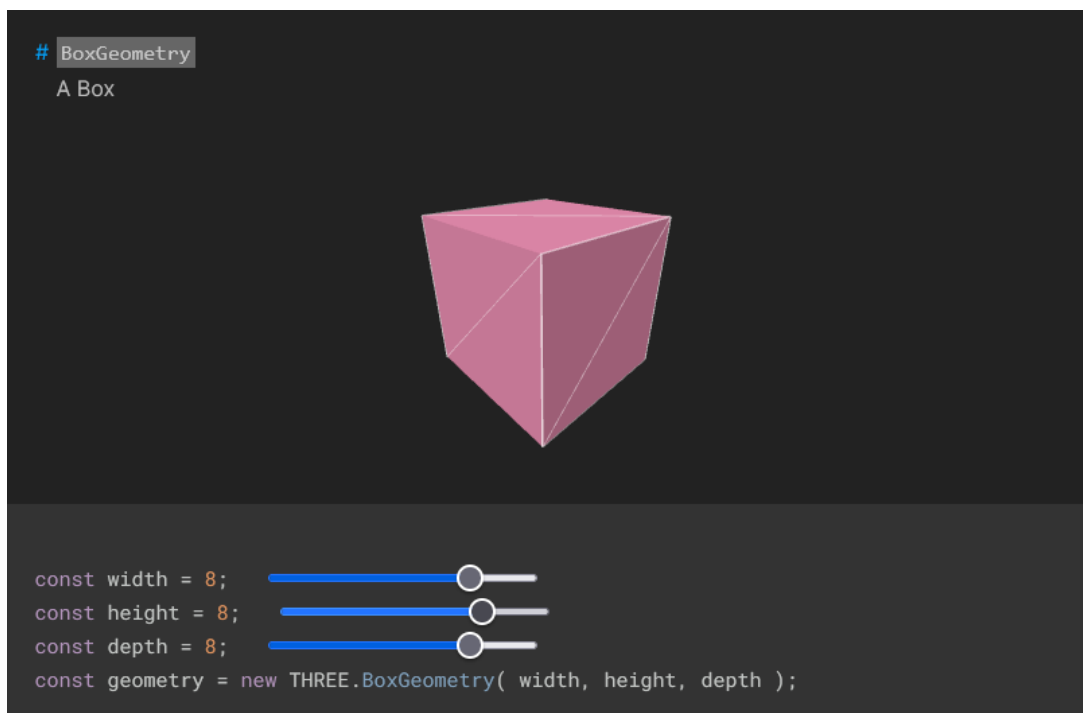


Ilustración 1.3-1: documentación de Three.js [3]

Otro apartado interesante de esta página es el de ejemplos [6], donde se pueden encontrar escenas, como la de la ilustración 1.3-2, más complejas que carecen de apartado teórico, pero se centran en ofrecer un nivel mayor de interacción. Lo que permite ver cómo se aplican diferentes conceptos y funciones al mismo tiempo.

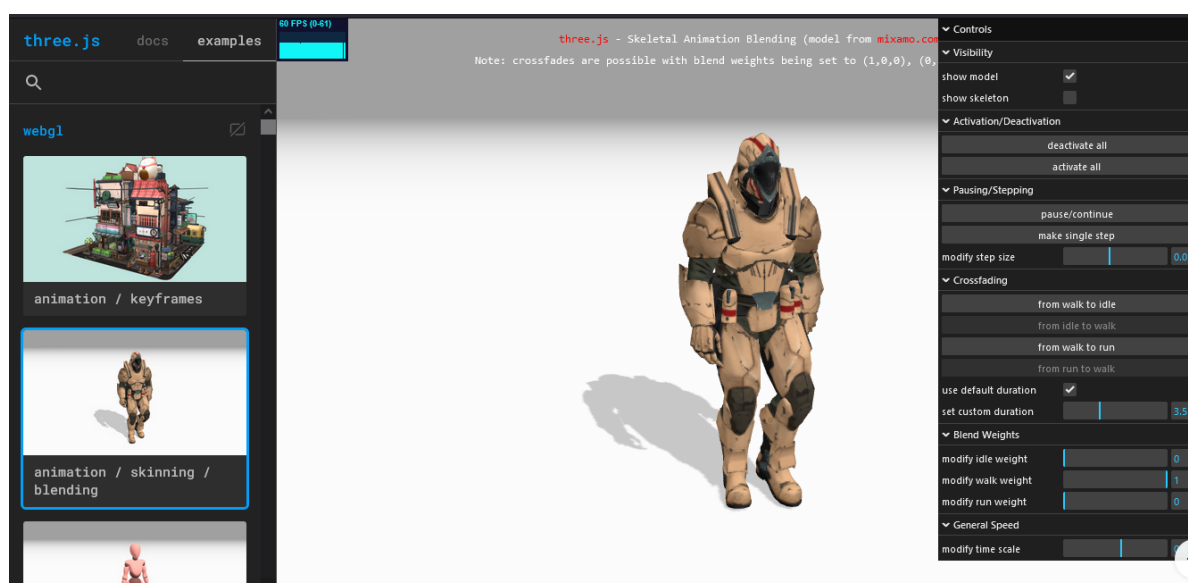


Ilustración 1.3-2: ejemplo interactivo de Three.js [3]

La documentación de este tipo y las escenas que ahí figuran han servido de inspiración para la aplicación web de este TFG.

1.3 Metodología de desarrollo de software

Las metodologías ágiles suelen estar pensadas para facilitar la comunicación y el seguimiento del proyecto en equipos de desarrollo compuestos por varias personas con diferentes roles, también establecen distintos tipos de reuniones tanto diarias como semanales, y entregas o incrementos cada cierto tiempo.

Por ejemplo el marco de trabajo Scrum [7] requiere de un Scrum Master y un entorno de trabajo configurado por Sprints. El Scrum Master ayuda al Product Owner brindándole herramientas y técnicas que respaldan la formación de objetivos de producto sólidos y alcanzables, así como la creación de Product Backlogs efectivos. Esto son listas emergentes y ordenadas de lo que se necesita para mejorar el producto.

Y un Sprint es una iteración de 1 a 4 semanas de duración, al final de las cuales se obtiene un incremento usable. En cada iteración las principales etapas son:

1. Un propietario del producto (Product Owner) ordena el trabajo de un problema complejo en un Product Backlog. Esto es un listado de todas aquellas tareas que se quieren realizar durante la fase de desarrollo con el objetivo de que estas sean visibles para todo el equipo.

El Product Owner es responsable de maximizar el valor del producto resultante del trabajo del Equipo Scrum. Se encarga de identificar, medir y maximizar el valor durante todo el ciclo de vida del producto.

2. El equipo de Scrum convierte una selección del trabajo en un Incremento de valor durante un Sprint.
3. El equipo de Scrum y sus partes interesadas (stakeholders) inspeccionan los resultados y realizan los ajustes necesarios para el próximo Sprint.
4. Repetir.

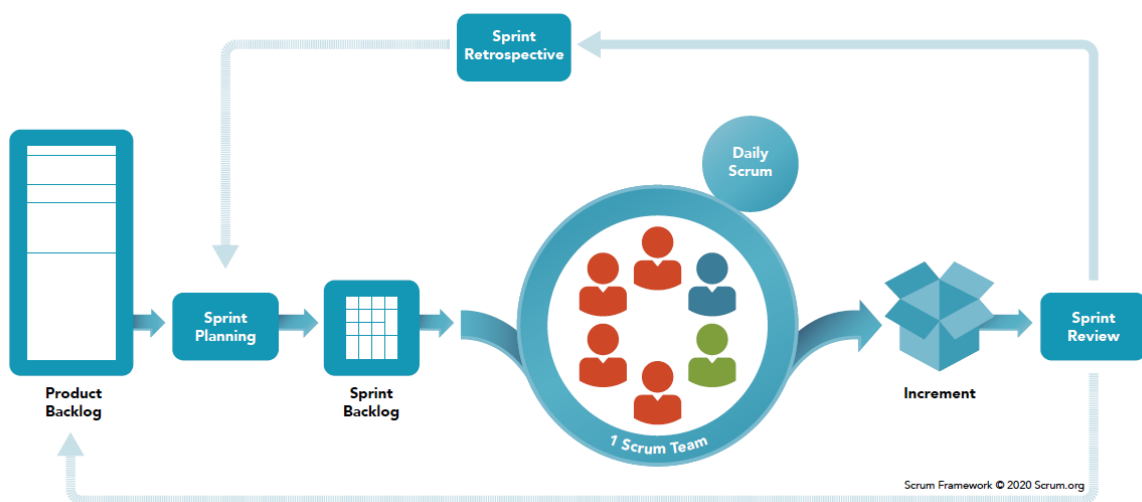


Ilustración 1.4-1: marco de Scrum [7]

Los roles, eventos, artefactos y reglas de Scrum son inmutables, y aunque es posible implementar solo partes de Scrum, el resultado no es Scrum.

Partiendo de esa idea, y teniendo en cuenta que el desarrollo va a estar llevado a cabo por una persona, y que esta no puede asumir todos los roles anteriormente mencionados, la metodología escogida es un método iterativo e incremental. En cada iteración se presentarán nuevas funcionalidades o se implementarán nuevas tecnologías así como las correcciones y mejoras detectadas en revisiones previas. También porque permite introducir nuevos requisitos que bien sustituyen a otros previos, o bien los complementan ampliando y concretando las especificaciones. Y también porque facilita establecer reuniones tanto para la entrega de cada iteración, como para confirmar las especificaciones. Por último, en caso de que surja algún problema durante el desarrollo, se podrá concertar reuniones improvisadas para abordarlo.

1.4 Alcance

Una vez definidos los objetivos del proyecto queda por determinar qué documentos se van a incluir dentro del mismo.

- Aplicación operativa. Se trata de una aplicación web que debe ser entregada en la fecha fijada y cumpliendo con todos los requisitos descritos en el punto

2.3. Pudiendo acceder a ella a través de internet, o siguiendo los pasos del manual de instalación para poder lanzarla de manera local.

- Código fuente. En la entrega final se presentará también el código fuente necesario no solo para arrancar la aplicación, sino para su consulta debido a las numerosas referencias a lo largo de esta memoria. Se explica detalladamente a lo largo del punto 4, la etapa de desarrollo.
- Manual de instalación. Como ya se ha mencionado, en este mismo documento se incluye una sección (apéndice I) donde se explica cómo instalar los programas y paquetes necesarios para ejecutar la aplicación de manera local.
- Manual de usuario. De igual manera, existe otra sección (apéndice II) donde se detallan los controles y funcionalidades de la interfaz de cada escena.

2. Análisis

En esta sección se estudian diferentes tecnologías, a partir de las cuales se ha obtenido una propuesta para el cliente, y los requisitos y tareas para cumplimentarlos, así como la planificación para la fase de desarrollo.

2.1 Alternativas y viabilidad

Como ya se ha comentado, el producto a desarrollar es una página web, por lo que se ha buscado una serie de bibliotecas que faciliten la implementación de escenas 3D, frameworks compatibles para manejar las vistas y un entorno de desarrollo adecuado.

2.1.1 Bibliotecas

- **Three.js** es una biblioteca JavaScript de código abierto para crear gráficos 3D en navegadores web. Una de las principales ventajas de Three.js es su simplicidad. Tiene una pequeña API que es fácil de aprender y usar, y se puede integrar con otras bibliotecas y frameworks. También tiene una comunidad grande y activa, con muchos recursos disponibles en línea para

aprender y solucionar problemas. Utiliza WebGL para renderizar gráficos en tiempo real.

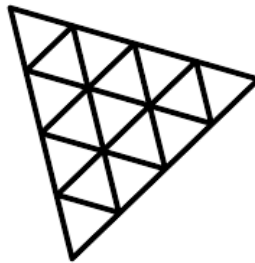


Ilustración 2.1-1: logo de Three.js

- **Babylon.js** [8] es otra biblioteca de código abierto para crear gráficos 3D en la web. Está diseñado específicamente para el desarrollo de juegos y ofrece características y funcionalidades más avanzadas. Su lenguaje original es Typescript. Una de las principales ventajas de Babylon.js es su motor de física y animación incorporado. Buena opción si se desea crear aplicaciones 3D avanzadas con física y animaciones complejas. Pero es más difícil de aprender y usar que Three.js



Ilustración 2.1-2: logo de Babylon.js

- **Similitudes.**
 - Buena documentación.
 - Variedad de ejemplos.
- **Diferencias**
 - La curva de aprendizaje en Three.js es menor.

- En Three.js la documentación está mejor organizada, es más fácil de consultar. Pero la de Babylon.js es más completa, aunque hay que acostumbrarse a su uso.
- Three.js cuenta con muchos más ejemplos.
- La API de Three.js puede ser algo más inestable que la de Babylon.js, pero su comunidad es más amplia y muchos problemas están resueltos en foros.
- Babylon.js cuenta con más funcionalidades ya implementadas.
- Babylon.js tiene herramientas de depuración.

2.1.2 Frameworks

- **Vue** [9] es un framework flexible y ligero basado en JavaScript que ofrece potentes herramientas web para desarrollar proyectos web de frontend. Permite agregar módulos personalizados y componentes visuales a la funcionalidad de la aplicación web.



Ilustración 2.1-3: logo de Vue.js

- **React** [10] es una biblioteca JavaScript de código abierto, creada por Facebook, que se puede utilizar para crear aplicaciones web con interfaces de usuario sofisticadas. React permite a los desarrolladores construir componentes personalizados reutilizables.

No es un framework de JavaScript, porque sólo es responsable de renderizar los componentes de la capa de vista de una aplicación.

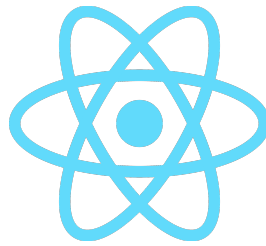


Ilustración 2.1-4: logo de React

- **Similitudes**

- Compatibilidad con aplicaciones web progresivas (PWA)
- Uso de Modelo de Objetos del Documento (DOM) virtual
- Visualizar componentes que son reactivos y personalizados
- Código JavaScript
- Ambas parten de una librería base a la que se puede añadir otras bibliotecas o frameworks complementarios.
- Capacidad de integración con cualquier aplicación web actual.

- **Diferencias**

- Comunidad. React está respaldado por FaceBook y posee una gran comunidad con variedad de recursos.
- Curva de aprendizaje. Vue es uno de los framework más sencillos de estudiar y comprender, posee una documentación clara y bien estructurada a diferencia de otros frameworks.

2.1.3 Entorno de desarrollo.

- **Visual Studio Code (VS)** [11]. Es un editor de código fuente ligero pero potente que se ejecuta en el escritorio y está disponible para Windows, macOS y Linux.

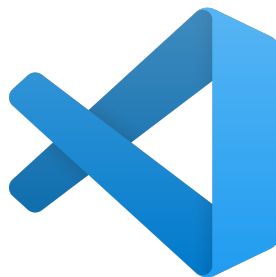


Ilustración 2.1-5: logo de Visual Studio Code

- **Web Storm** [12]. WebStorm es un entorno de desarrollo integrado para JavaScript y las tecnologías relacionadas. Al igual que otros IDE de JetBrains.

*Ilustración 2.1-6: logo de Web Storm*

- **Similitudes.**
 - Ambos son compatibles con lenguajes como JavaScript, TypeScript, HTML, CSS...
 - Integran extensiones para trabajar con React, Vue y Angular.
 - En ambos se puede integrar node Node.js [13].
 - Implementan su propia terminal para ejecutar comandos y gestionar paquetes.
- **Diferencias.**
 - La única diferencia notable detectada es que WebStorm cuenta con una vista previa de HTML integrada.

2.2 Solución propuesta

- **Biblioteca.** La biblioteca escogida ha sido Three.js porque dispone de una página web sencilla e intuitiva, la documentación es muy amplia, bien organizada y con numerosos ejemplos interactivos. También dispone de un curso de pago, aunque los primeros videos están disponibles gratuitamente y explican cómo instalar la biblioteca, cómo configurar una escena básica y arrancar un servidor para visualizar el resultado.

- **Frameworks.** Es necesario implementar algún framework que permita crear una SPA. Y el framework escogido es Vue.js, que proporciona un modelo de programación declarativo y basado en componentes que ayuda a desarrollar interfaces de usuario de manera eficiente. Permite al programador crear sus propios componentes y con Vue Router se inyectan fácilmente donde se requiera.
- **Entorno de desarrollo.** Como entorno de desarrollo se ha elegido Visual Studio Code principalmente porque ya se ha trabajado previamente con él en otros proyectos. Y como se ha comentado en la sección anterior, soporta diversos lenguajes de programación como Javascript y HTML/CSS, además de ofrecer extensiones como Vetur para desarrollo de aplicaciones con Vue.js. También integra una PowerShell con la que ejecutar comandos para instalación de sistemas de paquetes y dependencias, como Node y Vite, necesarias para arrancar fácilmente un servidor local con el que visualizar la página web en desarrollo.

2.3 Requisitos

Según el Instituto de Ingenieros Eléctricos y Electrónicos (IEEE) [14] un requisito se puede definir como:

1. Una condición o capacidad que necesita un usuario para resolver un problema o lograr un objetivo.
2. Una condición o capacidad que debe cumplir o poseer un sistema o componente del sistema para satisfacer un contrato, norma, especificación u otro documento impuesto formalmente.

En ingeniería de sistemas existen tres tipos de requisitos.

- Un requisito funcional puede ser una descripción de lo que un sistema debe hacer. Este tipo de requisito especifica algo que el sistema entregado debe ser capaz de realizar.
- Un requisito no funcional: especifica algo sobre el propio sistema, y cómo debe realizar sus funciones. Algunos ejemplos de aspectos solicitables son la disponibilidad, el testeo, el mantenimiento, la facilidad de uso, etc.
- Restricciones, que afectan indirectamente al desarrollo como adecuación a leyes o regulaciones del producto.

En este caso se van a dividir los requisitos en funcionales y no funcionales.

2.3.1 Funcionales

Requisitos relacionados con la funcionalidad e interacción de las escenas:

1. **Transformaciones geométricas.**
 - a. En la escena se presentará un modelo tridimensional de una primitiva, como un cubo o un cilindro, dispuesto en el centro de la escena.
 - b. A ese modelo se le podrá aplicar transformaciones geométricas (traslación, rotación y escalado).
 - c. Los valores de las transformaciones se introducirán a través de la interfaz dando la opción de realizarlas en los tres ejes (x, y, z).

- d. Se debe mostrar dos botones, uno para aplicar las transformaciones y otro para deshacerlas y volver al estado inicial.

2. Tipos de iluminación.

- a. Ambiental.
 - i. La escena debe incluir luz ambiental.
 - ii. Se podrá modificar los parámetros de intensidad y color de la luz.
- b. Direccional.
 - i. La escena debe incluir luz direccional.
 - ii. Se podrá modificar los parámetros de intensidad, color y posición de la luz.
- c. Puntual.
 - i. La escena debe incluir luz puntual.
 - ii. Se podrá modificar los parámetros de intensidad, color y atenuación.
- d. Foco.
 - i. La escena debe incluir luz focal.
 - ii. Y se podrá modificar los parámetros de intensidad, color, atenuación, ángulo de apertura y penumbra.

3. Tipos de selección con el ratón.

- a. La escena debe representar un gran número de objetos distribuidos por el espacio.
- b. Deben ser seleccionables por el ratón y cambiar de color cuando se clique sobre ellos.
- c. También debe aplicarse algún tipo de transformación siguiendo el movimiento del ratón.

4. Grafo de escena.

- a. La escena debe incluir un grafo de escena construido a través de jerarquía de objetos.
- b. La jerarquía debe tener por lo menos 3 niveles.

- c. Se deben aplicar transformaciones a los nodos de esos niveles, introduciendo los valores de la interfaz.
- d. Se añadirá un botón que active y desactive una animación predefinida del grafo.

5. Mallas de triángulos.

- a. La escena debe incluir un modelo tridimensional o una primitiva, y que se visualice tanto el material de la superficie como el *wireframe*.
- b. A través de la interfaz se podrá alterar la geometría y topología, modificando su tamaño y el número de divisiones longitudinales y transversales.

6. Uso de materiales y texturas.

- a. Aplicación de texturas.
 - i. La escena presentará un plano con dos texturas intercambiables.
 - ii. Las texturas podrán aplicar distintas técnicas de interpolación para la minificación y la magnificación.
- b. Bump Mapping.
 - i. La escena presentará dos modelos pudiendo alternar entre ellos.
 - ii. Y dos texturas distintas con su correspondiente mapa de alturas o normal map aplicados para visualizar el efecto de bump mapping.
 - iii. También se podrá alterar el nivel de detalle o de intensidad con el que se aplica el bump mapping.
- c. Reflexión.
 - i. La escena mostrará un modelo con dos tipos de materiales intercambiables.
 - ii. Unos de los materiales simulará metal pudiendo alterar las propiedades difusas y especular de este.
 - iii. Otro será una textura que simula madera con valores predefinidos.

2.3.2 No Funcionales

1. **Aspecto** de la aplicación. La página web ha de presentar como mínimo los siguientes elementos:
 - a. Un menú navegable con los títulos de todos los apartados, y de
 - b. un color distinto al fondo de la página.
 - c. Una vista distinta para cada apartado donde figuren el área de visualización de la escena
 - d. debajo de esta los textos explicativos correspondientes.
 - e. El área de visualización debe tener un menú desplegable con los controles necesarios para interactuar con la escena.
2. **Textos**. Se debe utilizar un lenguaje claro y preciso, las explicaciones no deben ser demasiado largas y deben contener imágenes que apoyen el contenido.
3. **Diseño adaptativo** (responsive). El diseño de la página se reorganiza y ajusta al tamaño de la pantalla detectada.

2.4 Historias de Usuario

Una Historia de Usuario (HU) [15] describe una funcionalidad que debe incorporar un sistema de software y aporta valor al cliente. Están descritas en lenguaje natural y su estructura es la siguiente.

- **ID.** Identificador de la HU.
- **Título.** Título descriptivo de la historia de usuario.
- **Descripción:** Descripción sintetizada
 - Como [rol de usuario].
 - Quiero [objetivo].
 - Para poder [beneficio].
- **Estimación:** Estimación del coste de implementación en Puntos de Historia (PH).
- **Valor de negocio:** valor (normalmente numérico) que aporta la historia de usuario al cliente o usuario.
- **Dependencias:** una historia de usuario no debería depender de otra, pero a veces es necesario mantener la relación.

En 2003 se desarrolló un método llamado INVEST [16] para asegurar la calidad en la escritura de historias de usuario. El método sirve para comprobar la calidad de una historia de usuario revisando que cumpla una serie de características:

- **Independiente.** Cada historia de usuario puede ser planificada e implementada en cualquier orden.
- **Negociable.** Sus detalles serán acordados con el cliente o el usuario durante la fase de conversación. Se debe evitar HUs con demasiados detalles porque limitaría la conversación acerca de las mismas.
- **Valiosa.** Cada historia tiene que tener valor para el usuario.

- **Estimable.** Debe de poder ser estimada con la precisión suficiente para ayudar al cliente, usuario o propietario del producto a priorizar y planificar su implementación.
- **Pequeña.** Debería englobar como mucho unas pocas semanas de trabajo.
- **Comprobable.** Disponer de pruebas de aceptación tangibles para saber que la HU se ha completado con éxito.

Para puntuar las HUs en relación al valor que aportan, se ha seguido el método MoSCoW [17] que divide los requerimientos en 4 categorías diferentes:

1. **Debe tener** (Must Have). Son las HUs que determinan el producto mínimo viable (MVP), sin estos requisitos el proyecto se considera que no ha tenido éxito, por lo que se consideran “no negociables”. En este caso se les ha dado puntuación de 10 sobre 10.
2. **Debería tener** (Should have). No son tan urgentes como los anteriores, pero sí son muy deseables. Pueden adquirir mucha relevancia conforme se avanza en el desarrollo. Se ha puntuado estas HUs con un 7 sobre 10.
3. **Podría tener** (Could have). Pueden ser funcionalidades interesantes de implementar, pero que no afectan a la viabilidad del proyecto. A estas HUs se las puntúa con un 5 sobre 10.
4. **No tendría** (Won't have). Funcionalidades que no merece la pena introducir durante el desarrollo de este proyecto, pero pueden incluirse en futuras versiones. Puntuadas con 3 sobre 10.

Las HUs redactadas siguiendo los criterios de INVEST y MoSCoW se muestran en la tabla 2.4-1. Todas entran en las categorías “Debe tener” y “Debería tener”.

ID	Título	Descripción	PH	Valor
1	Menú navegable	• Como usuario • Quiero que la página disponga de un menú navegable • Para acceder a todas las secciones	3	7

2	Transformaciones	• Como usuario • Quiero aplicar transformaciones geométricas a un modelo • Para ver el resultado final	7	10
3	Color Luces	• Como usuario • Quiero modificar el color de las luces • Para comprobar cómo afectan al color de las escenas	5	10
4	Intensidad Luces	• Como usuario • Quiero modificar la intensidad de la luz ambiental • Para comprobar cómo influye a la iluminación de las escenas	5	10
5	Luz Direccional Posición	• Como usuario • Quiero modificar la posición de la luz direccional • Para comprobar cómo cambia el sombreado de objetos	5	7
6	Luz Puntual Atenuación	• Como usuario • Quiero modificar la atenuación de la luz puntual • Para comprobar cómo cambia el sombreado de objetos	5	7
7	Luz Foco Penumbra	• Como usuario • Quiero modificar el efecto penumbra de la luz focal • Para comprobar cómo afecta al modelo de iluminación	5	7
8	Luz Foco Ángulo	• Como usuario • Quiero modificar el ángulo de apertura de la luz focal • Para comprobar cómo afecta al modelo de iluminación	5	7
9	Selección de objetos	• Como usuario • Quiero seleccionar objetos con el ratón • Para aplicarles alguna tipo de transformación geométrica	7	10
10	Grafo de escena	• Como usuario • Quiero aplicar rotaciones a los nodos articulados • Para ver cómo se comporta la jerarquía de objetos	5	10

11	Animación Grafo	• Como usuario • Quiero activar una animación predefinida • Para ver cómo se mueve el grafo de manera automática	7	7
12	Malla	• Como usuario • Quiero visualizar el wireframe de un modelo • Para ver cómo se compone una malla de triángulos	3	10
13	Malla Divisiones	• Como usuario • Quiero modificar las divisiones longitudinales y laterales de la malla • Para ver cómo afecta el número de triángulos a su resolución	5	7
14	Texturas	• Como usuario • Quiero aplicar diferentes texturas a varios modelos distintos • Para ver cómo cambian su aspecto	3	7
15	Texturas Interpolación	• Como usuario • Quiero aplicar diferentes técnicas de interpolación • Para ver cómo afectan a la minificación y magnificación	5	10
16	Bump Mapping	• Como usuario • Quiero aplicar diferentes mapas de normales o altura • Para ver cómo afectan a la apariencia de los modelos	5	10
17	Materiales	• Como usuario • Quiero modificar las propiedades difusa y especular • Para ver cómo afectan a la apariencia de los modelos	5	10

Tabla 2.4-1: lista de HUs

2.5 Planificación

2.5.1 Tareas

Una vez obtenidas las HUs con el esfuerzo y el valor estimado, se procede con la redacción de tareas para luego planificar los Sprints.

ID	Tarea	Subtareas	Precedentes	Duración (días)
A	Configuración Proyecto	i. Crear proyecto en VS ii. Descargar Three.JS iii. Repositorio GitHub iv. Crear el diseño inicial de la página web v. Creación primera escena Three.JS	-	1
B	Escena#1 Transformaciones	i. Escena base ii. Añadir primitiva iii. Añadir GUI iv. Aplicar interacciones v. Test	A	3
C	Escena#2 Luz Ambiental	i. Escena base ii. Añadir primitivas iii. Configurar luz ambiental iv. Añadir GUI v. Aplicar interacciones vi. Test	A	3
D	Escena#3 Luz Direccional	i. Escena base ii. Añadir primitivas iii. Configurar luz direccional iv. Añadir GUI v. Aplicar interacciones vi. Test	C	2
E	Escena#4 Luz Puntual	i. Escena base ii. Añadir primitivas iii. Configurar luz puntual iv. Añadir GUI v. Aplicar interacciones vi. Test	D	2
F	Escena#5 Luz Focal	i. Escena base ii. Añadir primitivas iii. Configurar luz focal iv. Añadir GUI	E	2

		v. Aplicar interacciones vi. Test		
G	Escena#6 Selección de objetos	i. Escena base ii. Añadir primitivas aleatorias iii. Añadir GUI iv. Aplicar interacciones v. Test	A	4
H	Escena#7 Grafo de escena	i. Escena base ii. Añadir primitivas iii. Configurar jerarquía iv. Configurar animación v. Añadir GUI vi. Aplicar interacciones vii. Test	A	4
I	Escena#8 Malla de triángulos	i. Escena base ii. Añadir primitiva iii. Configurar malla iv. Añadir GUI v. Aplicar interacciones vi. Test	A	3
J	Escena#9	i. Escena base ii. Añadir primitivas iii. Añadir texturas iv. Añadir GUI v. Aplicar interacciones vi. Test	A	3
K	Escena#10	i. Escena base ii. Añadir primitivas iii. Añadir texturas iv. Añadir GUI v. Aplicar interacciones vi. Test	J	2
L	Escena#11	i. Escena base ii. Añadir primitivas iii. Añadir materiales iv. Añadir GUI v. Aplicar interacciones vi. Test	K	2
M	Decorar Escenas	i. Integrar Node y Vite ii. Buscar modelos 3d en formato obj o fbx. iii. Implementar modelos y configurarlos. iv. Añadir fondos. v. Añadir helpers.	B, F, G, H, I, L	3

N	Implementar VUE	i. Descargar VUE.js ii. Crear vista inicial iii. Integrar Vue-Router iv. Crear componentes y vistas provisionales. v. Test	A	2
O	Creación de vistas	i. Crear componentes de las escenas Three.js. ii. Crear vistas. iii. Añadir textos e imágenes. iv. Crear barra navegable con Vue-Router. v. Aplicar estilos CSS y Bootstrap	M, O	5

Tabla 2.5-1: definición de tareas y subtareas

Finalmente, la planificación de las tareas divididas por sprints se muestra a continuación.

Sprint	Inicio	Fin	Tareas
1	5/4/2023	21/4/2023	A, B, C,D, E, F,
2	7/6/2023	21/6/2023	G, H, I
3	28/6/2023	11/7/2023	J, K, L, M
4	18/7/2023	26/7/2023	N, O

Tabla 2.5-2: definición de sprints

2.5.2 Diagrama de Gantt

En este apartado se muestra el diagrama de Gantt resultante incluyendo tareas previas de análisis, planificación y diseño, representadas en verde.

Luego en tonalidades de amarillo y rojo, se muestra la duración de los sprints. Y entre medias de estos, en azul, la entrega de los incrementos y las correcciones de diferentes aspectos que pudieran surgir tras cada entrega.

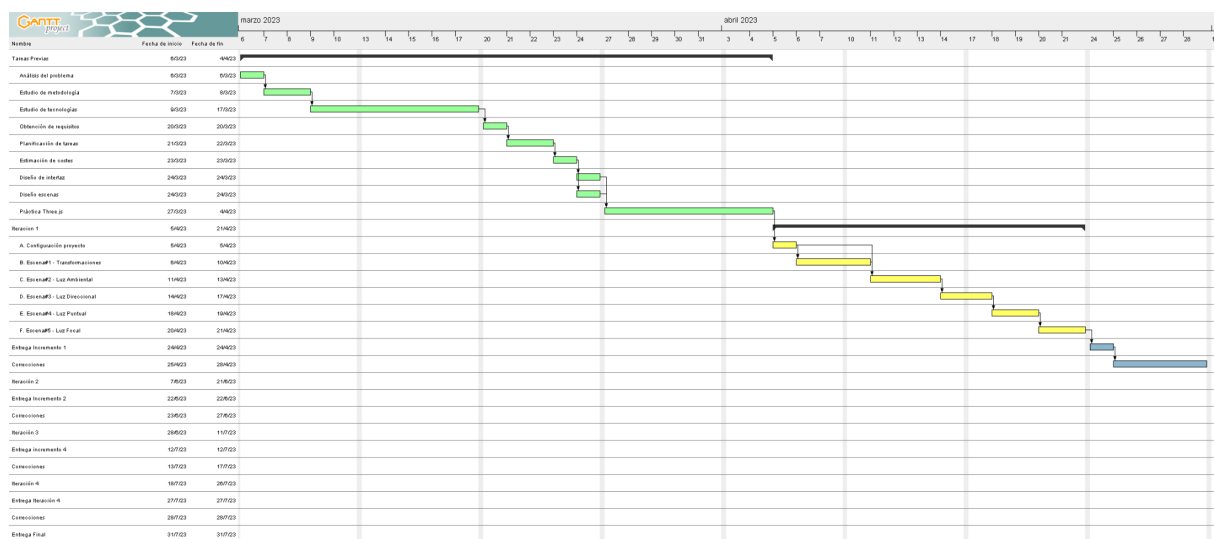


Ilustración 2.5-1: diagrama de Gantt (1ª parte)

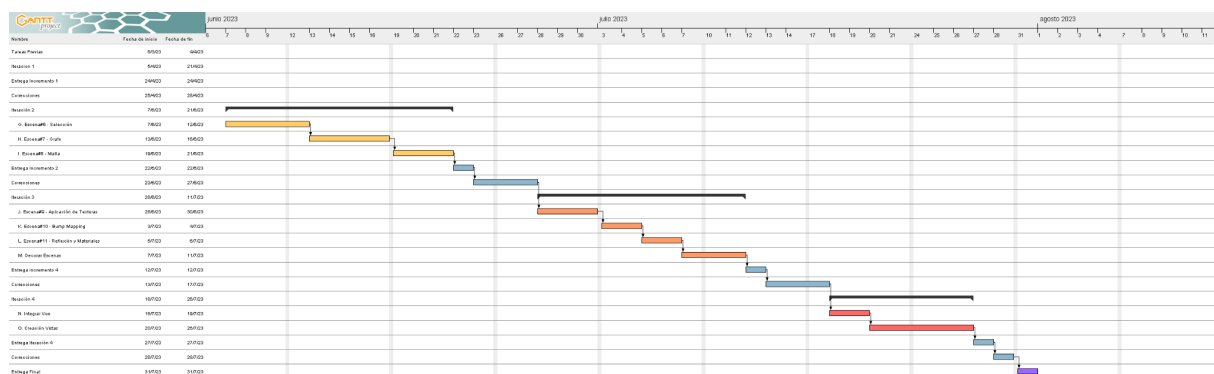


Ilustración 2.5-2: diagrama de Gantt (2ª parte)

Los diagramas también están disponibles en formato PNG en el siguiente enlace:

<https://drive.google.com/drive/folders/1ujintf2EyQ70Te2o0uLIMcMxK-sGcu2V?usp=sharing>

A pesar de que en la planificación de tareas, tabla 2.5-1, no todas las tareas son paralelizables, en este caso se muestran de manera secuencial porque han sido llevadas a cabo por una sola persona.

2.5.3 Variaciones

- Originalmente la escena de selección se había implementado correctamente con la técnica de selección por color. Pero en la última iteración, al integrar las escenas con Vue.js, en esta escena en concreto, se detectó que no funcionaba de la misma manera porque daba numerosos fallos. Como consecuencia, se optó por sustituirla e implementar la técnica de intersección de rayos.
- Al concluir la etapa de desarrollo, al tratarse de una aplicación web, se decidió subirla a un servidor para que sea fácilmente accesible. El enlace se puede encontrar en el primer apéndice de esta memoria: manual de instalación.

2.6 Estimación de costes

Como en todo proyecto software, el gestor de proyecto debe preparar una propuesta para el cliente en función de las necesidades del proyecto. En dicha propuesta se estima el coste de los trabajadores, amortización de los equipos informáticos, costes indirectos... etc. En este caso, el coste está basado principalmente en el trabajo realizado por una persona que cubre diferentes roles: analista y programador.

Basándose en la planificación anterior, el analista ha trabajado 5h diarias durante 15 días, lo que supone un total de 75h. Con un precio aproximado de 16€/h según Indeed [18]. Entonces el coste total es de 1440€.

Por otro lado, el programador ha trabajado 5h diarias durante 64 días. Si el precio de la hora es de 10€ según Indeed, el coste total es de 3.200€.

Puesto Trabajador	Coste mensual	Coste (€/h)	Tiempo trabajado	Coste Total
-------------------	---------------	-------------	------------------	-------------

Analista	1.600€	16€	75h	1.200 €
Programador	1.000€	10€	320h	3.200 €
Total				4.400 €

Tabla 2.6-1: costes de personal

En cuanto al coste de material software, se han usado programas gratuitos y librerías de código abierto, como se ha descrito en el apartado 2.2. Por lo que no repercuten en el presupuesto.

Por otro lado, sí hay que tener en cuenta los costes relacionados con material hardware y energía. La vida útil de un equipo informático se estima que puede llegar a 5 años y su valor es de unos 800€. Por lo tanto, los costes derivados son los siguientes.

Valor hardware	Vida útil (años)	Amortización mensual	Amortización en 3 meses
800€	5	13.33€	40 €
Total			40 €

Tabla 2.6-2: costes hardware

Finalmente el presupuesto total es la suma de los costes presentados anteriormente.

Coste de personal	4.400 €
Coste hardware	40 €
Subtotal	4.440 €
Costes Indirectos (5%)	222 €
Margen (10%)	444 €
Beneficio (10%)	444 €
Antes de impuestos	5328 €
IVA (21%)	1.118,90 €
Presupuesto total	6.446,90 €

Tabla 2.6-4: coste total

Al coste del desarrollo de la aplicación se le han sumado costes indirectos relacionados con el pago de la luz e internet. También un margen en caso de que surjan problemas o imprevistos durante el desarrollo y haya que dedicarle más tiempo y esfuerzo que el estimado para cumplir con las tareas. Por otro lado, un margen de beneficio. Y por último el IVA.

3. Diseño

En esta sección se explica cómo se estructura la página web, presentando primero los diagramas de caso de uso, que describen las posibles acciones del usuario en cada escena. Luego un diseño general para todas las vistas, y más adelante, en distintos apartados se detalla la interfaz para cada una. También se incluyen bocetos de las escenas.

3.1 Diagramas de casos de uso

Como ya se ha mencionado, el objetivo es crear una SPA, en la que el usuario tiene como dos principales acciones: realizar alguna interacción con la escena que se muestra en el momento, o bien, cambiar de escena. Por lo que el diagrama de casos de uso genérico es el siguiente.

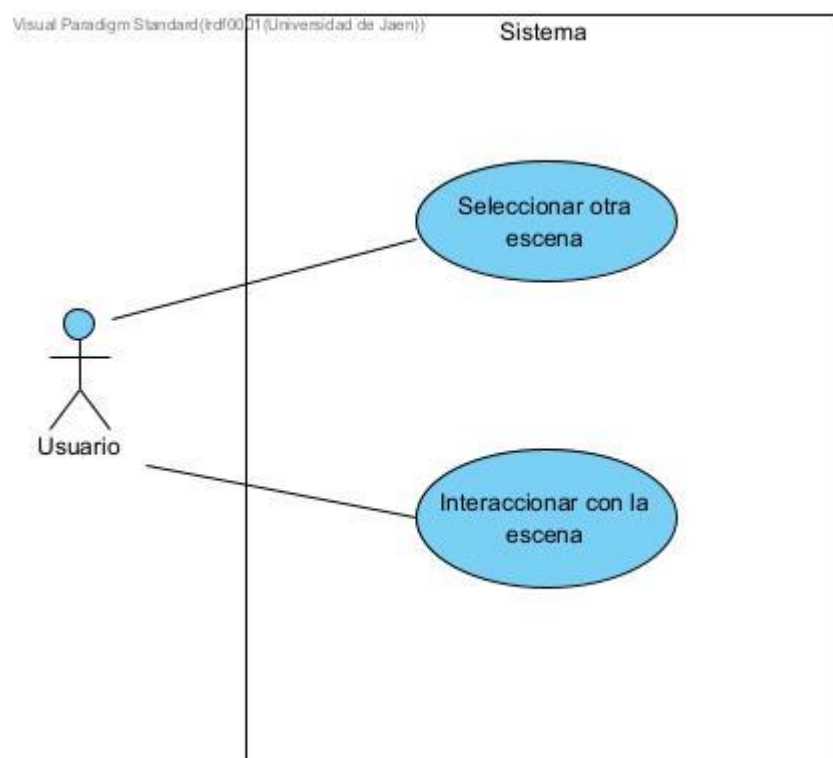


Ilustración 3.1-1: diagrama de caso de uso general

En los siguientes sub-apartados se muestran los diagramas de caso de uso para cada una de las escenas.

3.1.1 Transformaciones

En esta escena las acciones que puede realizar el usuario son algo más complejas que en el resto de escenas, porque unas dependen de otras:

- Introducir los valores correspondientes a cada transformación geométrica.
- Aplicar las transformaciones, en caso de que se hayan introducido los valores correspondientes.
- Deshacer las transformaciones previamente aplicadas y devolver al modelo su estado original.
- Cambiar la visualización del modelo para mostrar solo el wireframe de este.

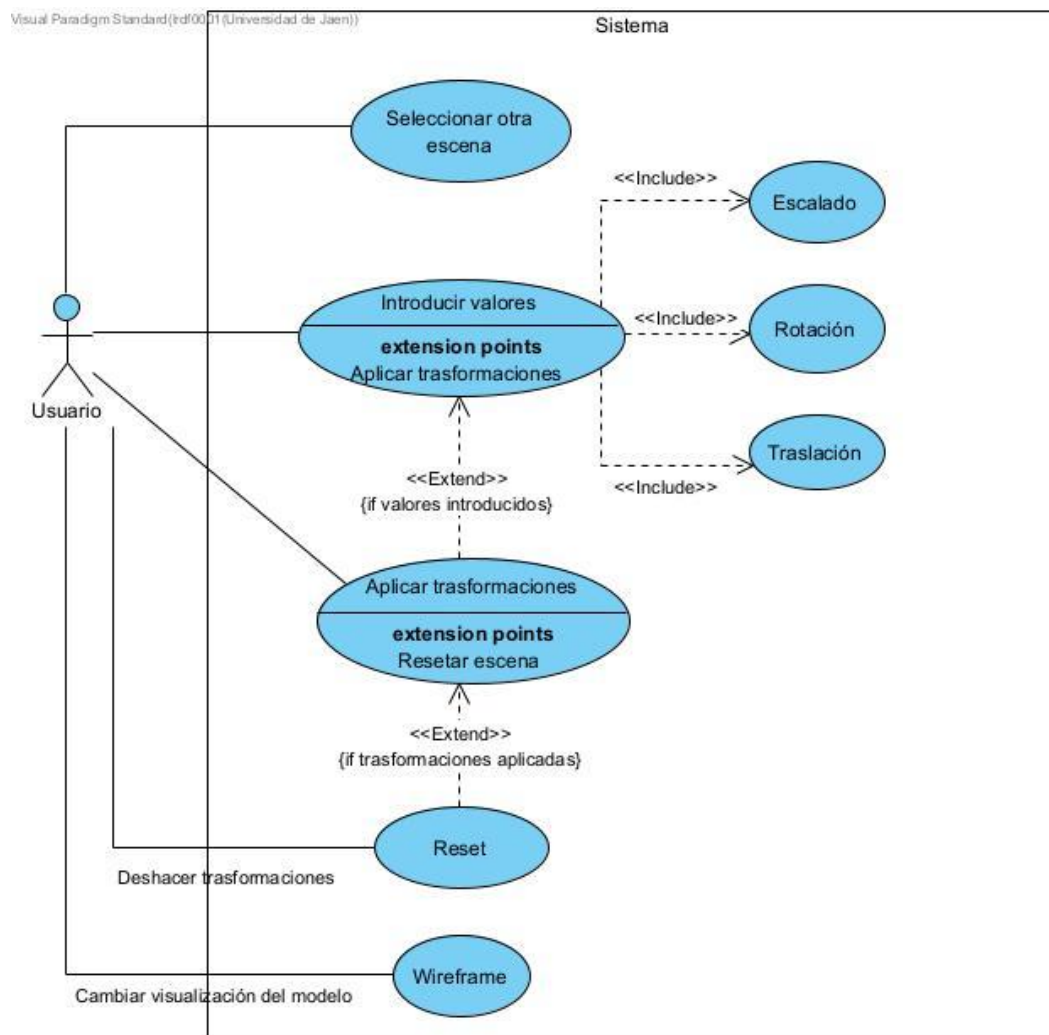


Ilustración 3.1-2: diagrama de casos de uso para la escena 1

3.1.2 Luz ambiental

Los casos de uso para esta escena son bastante sencillos, ya que la luz ambiental depende principalmente de dos parámetros:

- Cambiar el color.
- Cambiar la intensidad.

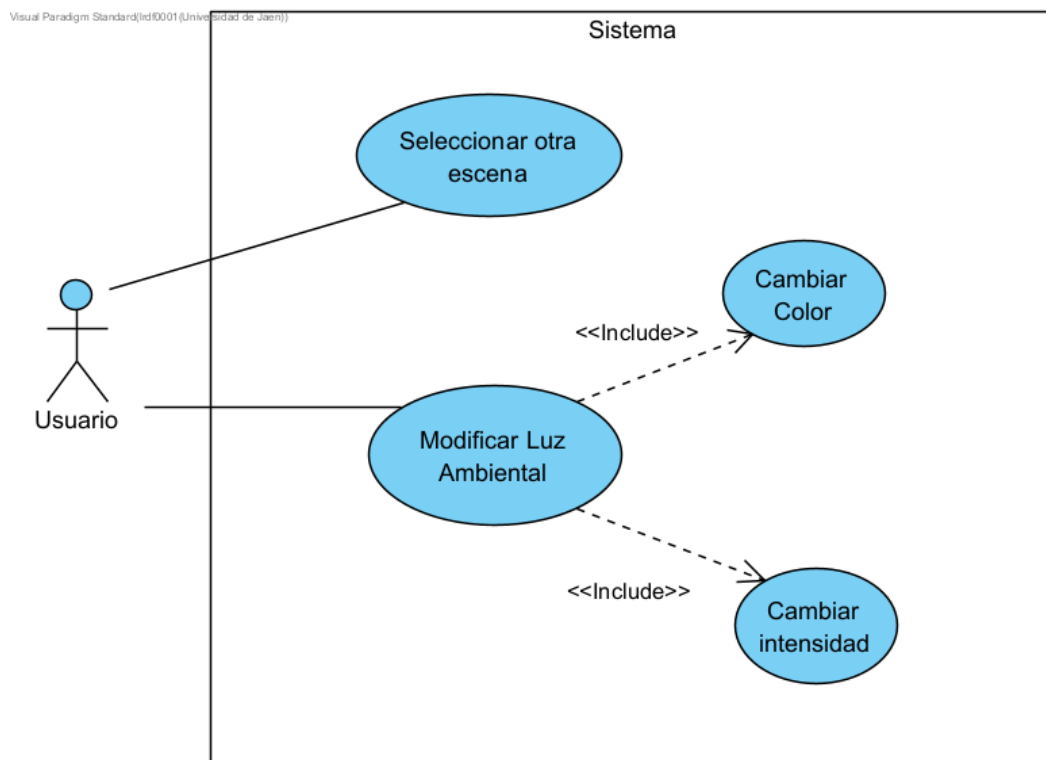


Ilustración 3.2-2: diagrama de casos de uso para la escena 2

3.1.3 Luz direccional

Las acciones en esta escena son parecidas a la anterior, se le suma el poder cambiar la posición de la luz. Aunque este tipo de luz también depende de la dirección, se ha decidido que no sea un parámetro ajustable por el usuario y que la luz siempre apunte al origen de coordenadas, donde está ubicado el modelo.

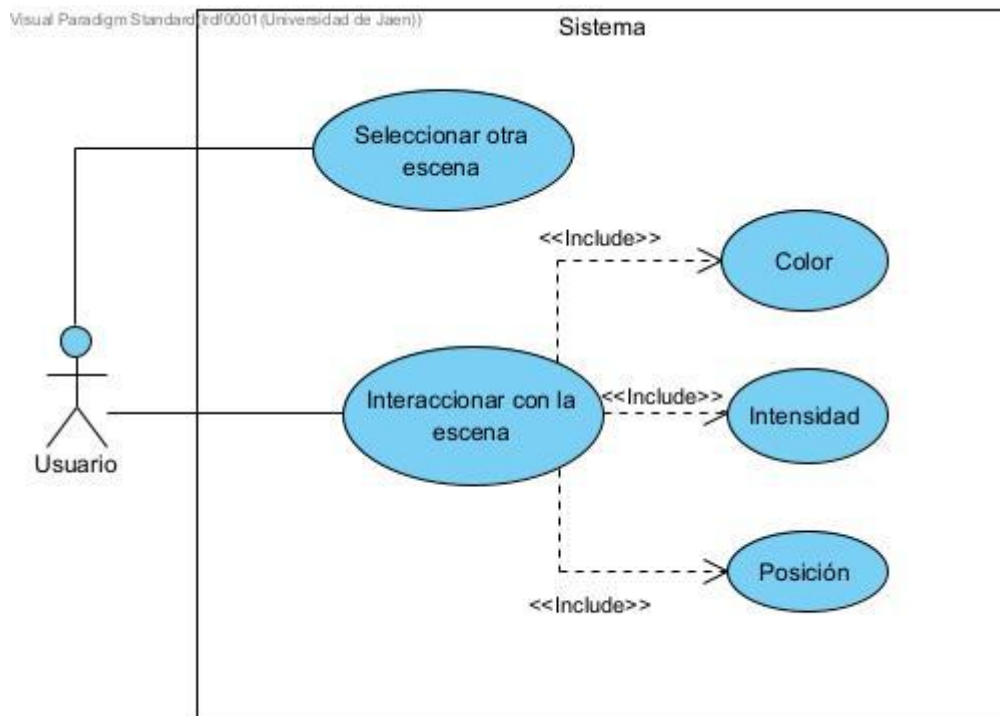


Ilustración 3.2-2: diagrama de casos de uso para la escena 3

3.1.4 Luz puntual

Siguiendo el esquema de las anteriores escenas, las acciones para la luz puntual son casi idénticas, solo que también se puede ajustar la atenuación.

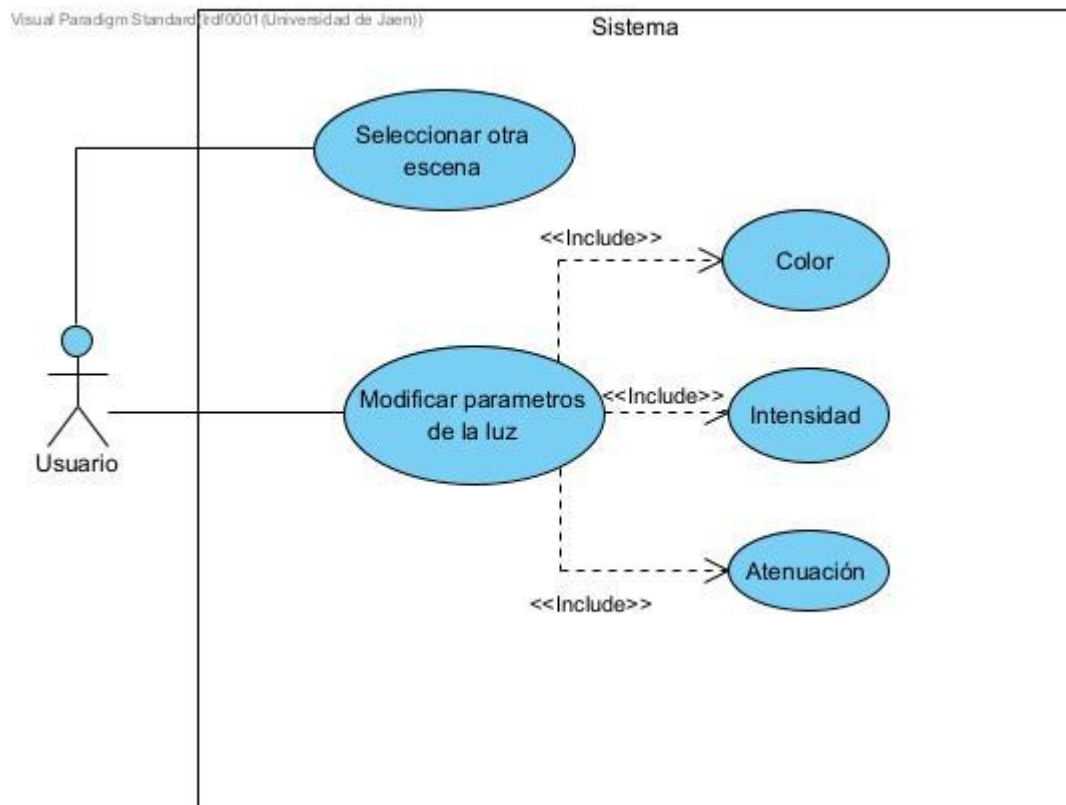


Ilustración 3.2-4: diagrama de casos de uso para la escena 4

3.1.5 Foco

Para acabar con las fuentes de iluminación, la luz focal, sus parámetros son parecidos a los de la luz puntual pero incluyendo el ángulo de apertura y la penumbra.

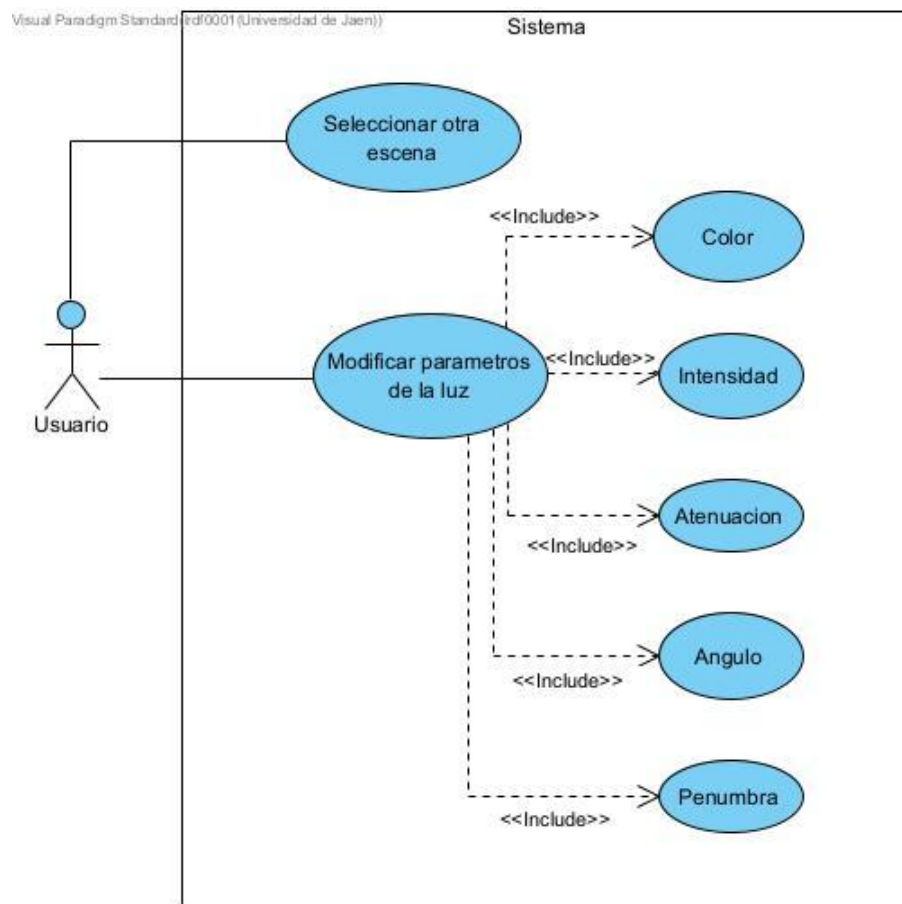


Ilustración 3.2-5: diagrama de casos de uso para la escena 5

3.1.6 Selección

En el caso de esta escena la principal acción es poder seleccionar un objeto al clicar, y al mover el ratón cuando esté marcado, este rota automáticamente.

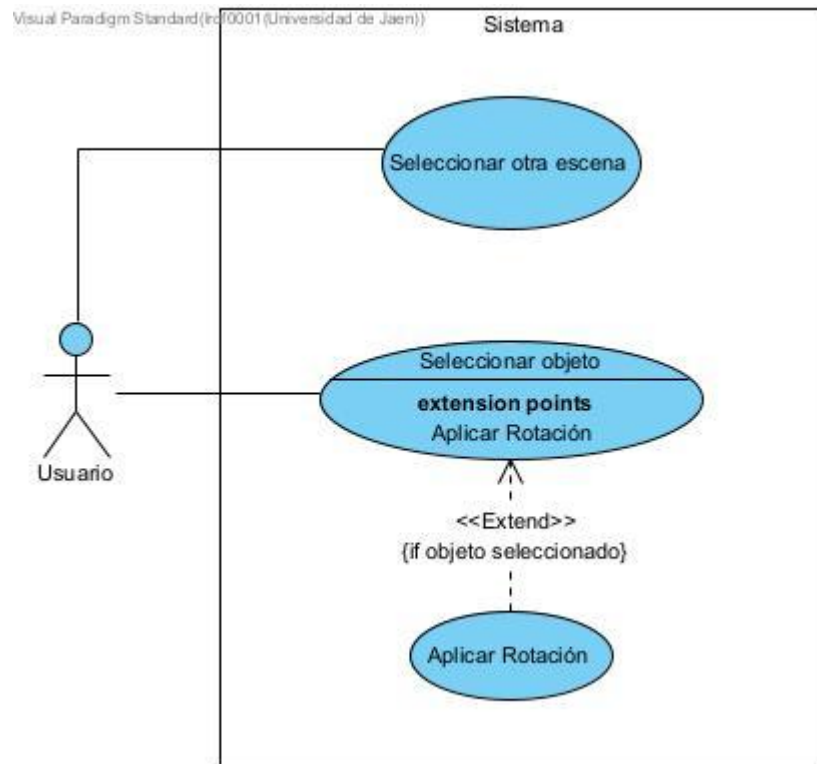


Ilustración 3.2-5: diagrama de casos de uso para la escena 6

3.1.7 Grafo de escena

En esta escena se representa un brazo robótico con 3 puntos articulables, simulando un brazo humano, y las principales acciones son aplicar rotaciones a estos puntos así como activar una animación, para que roten de manera autónoma.

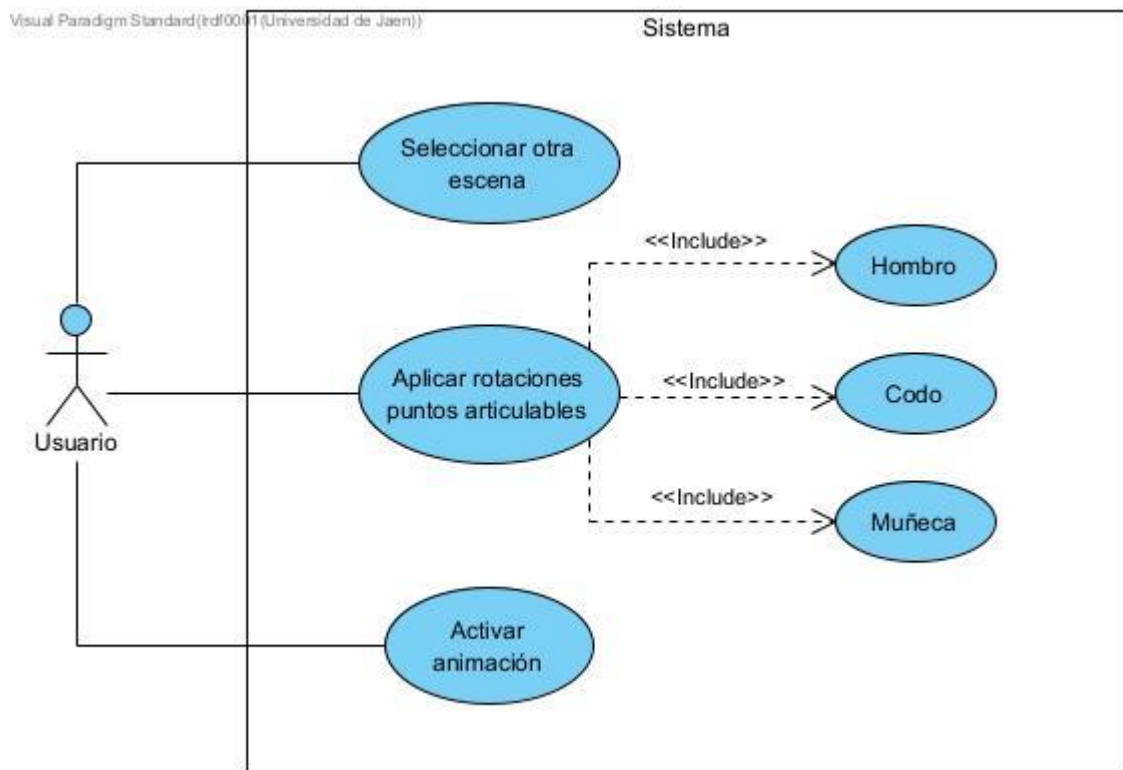


Ilustración 3.2-5: diagrama de casos de uso para la escena 7

3.1.8 Malla de triángulos

Esta escena muestra una malla de triángulos que representa una esfera, por lo que las acciones que puede hacer el usuario son:

- Modificar su radio.
- Aumentar y disminuir sus divisiones laterales y longitudinales.

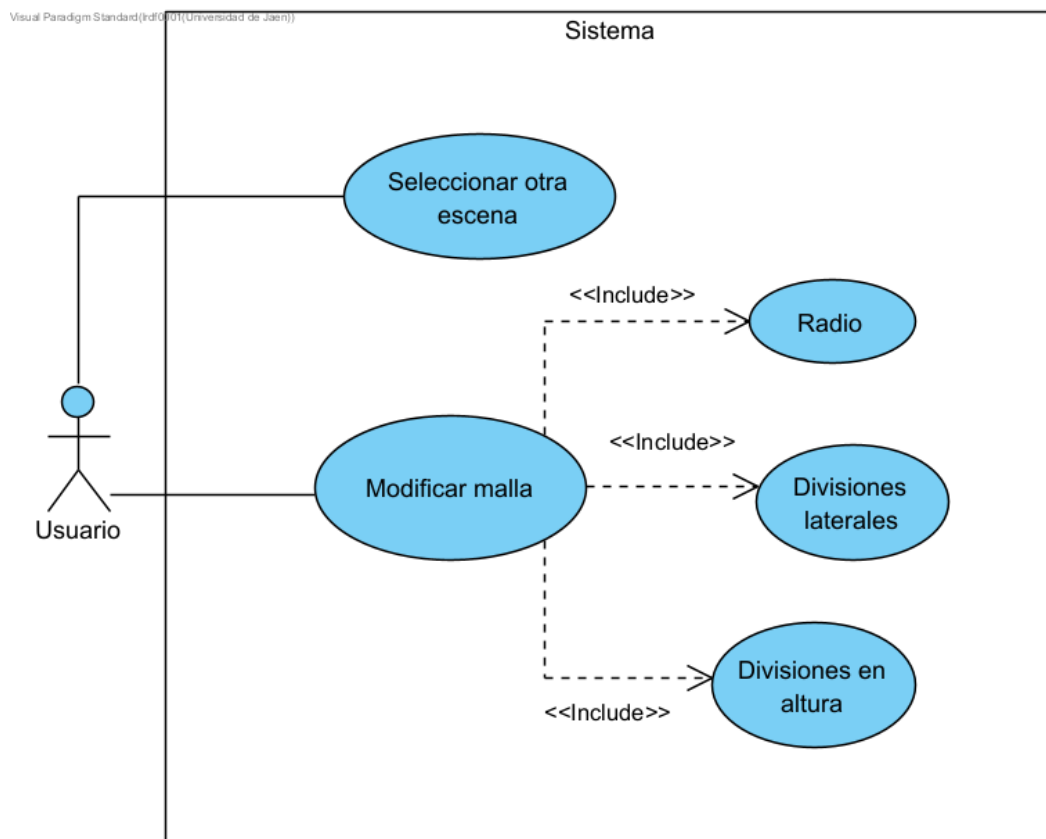


Ilustración 3.2-5: diagrama de casos de uso para la escena 8

3.1.9 Aplicación de texturas

En esta sección se muestra un plano con una textura, al que se le podrá configurar los tipos de interpolación para los efectos de minificación y magnificación. Y alternar entre dos texturas distintas.

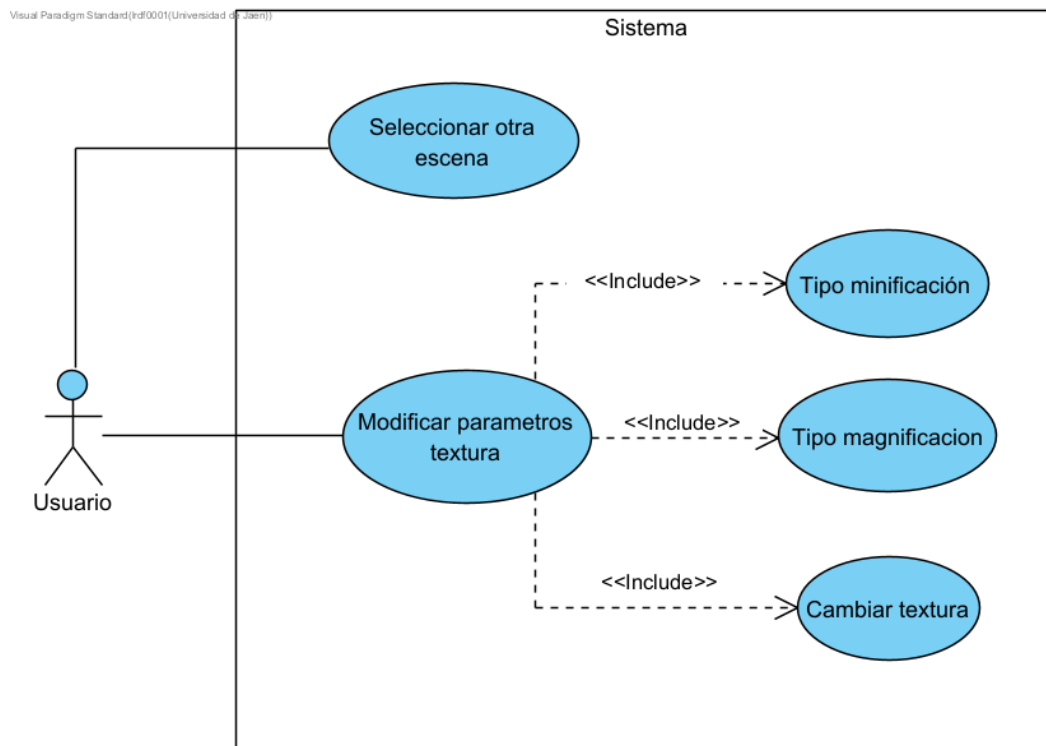


Ilustración 3.2-5: diagrama de casos de uso para la escena 9

3.1.10 Bump Mapping

En esta escena se muestra una primitiva como un cubo o una esfera, con una textura aplicada a la que se le puede modificar el nivel intensidad con el que se aplica el bump mapping, también se puede cambiar la primitiva y alternar entre dos texturas diferentes.

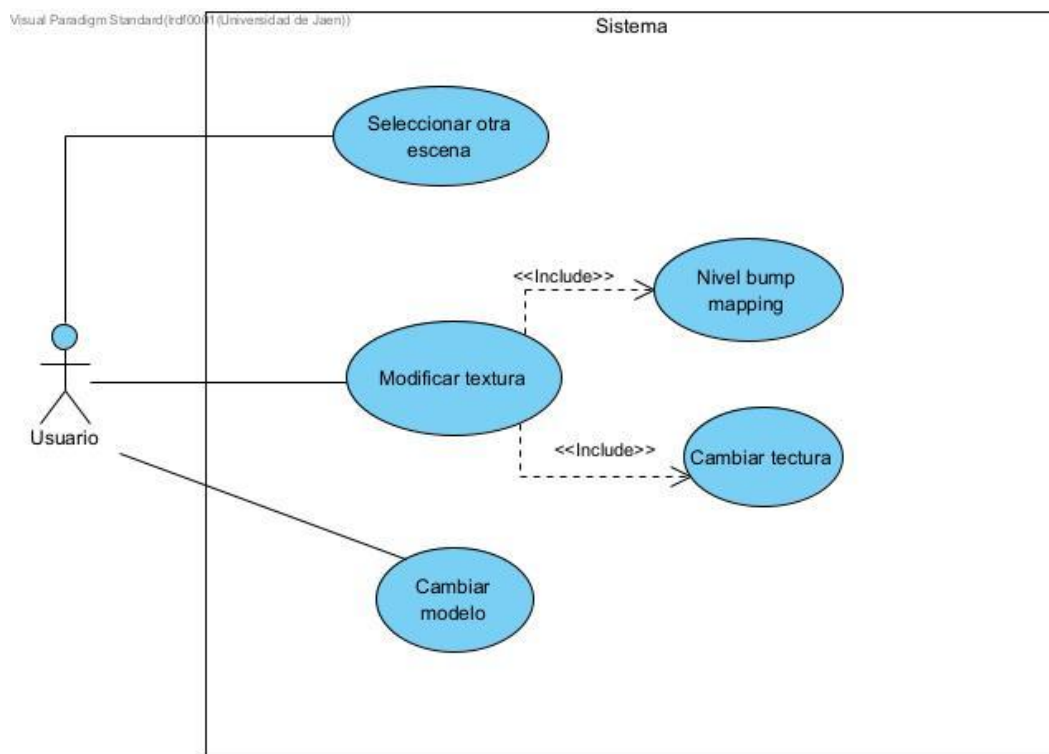


Ilustración 3.2-5: diagrama de casos de uso para la escena 10

3.1.11 Modelos de reflexión

Para esta escena el usuario puede alterar las propiedades difusa y especular del material del modelo mostrado, o bien aplicar una textura con esas propiedades previamente establecidas. También puede modificar la geometría del modelo.

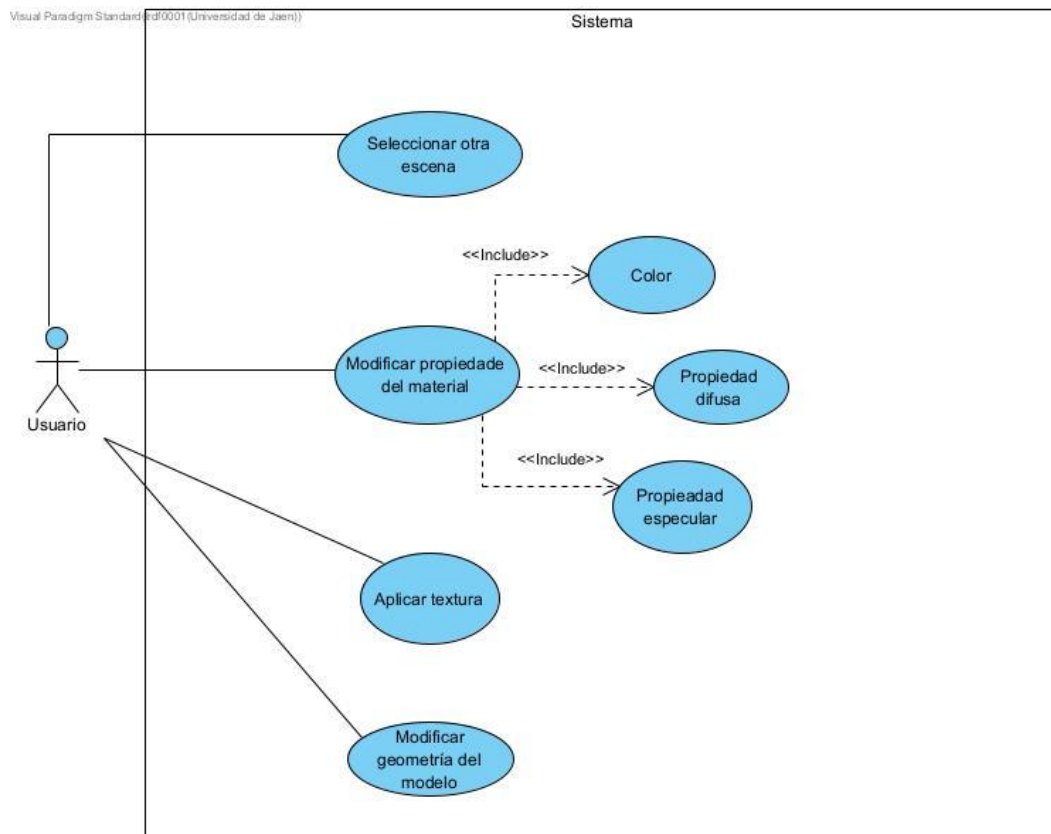


Ilustración 3.2-5: diagrama de casos de uso para la escena 11

3.2 Interfaz

Cumpliendo con los requisitos relacionados con el aspecto de la aplicación, para el diseño de la página web solo ha sido necesario crear un único wireframe, ya que al ser una SPA, el diseño parte de un único DOM donde figura un menú navegable en la parte de arriba, y el resto de componentes se irán reemplazando unos por otros para generar las vistas, pero todas ellas con la misma estructura:

1. Canvas, para renderizar la escena.
2. Título de la sección.
3. Textos e imágenes.

En cuanto a la escena, para crear un sistema con la que interactuar con los modelos se estableció decidió poner diferentes botones creados con HTML y Javascript. Pero más adelante, en la etapa de desarrollo se detectó que Three.js se podía combinar fácilmente con dat.GUI [19], que permite modificar variables de Javascript y establecer rangos de valores y el valor de incremento. El diseño final es el que se muestra en la ilustración 3.1-1.

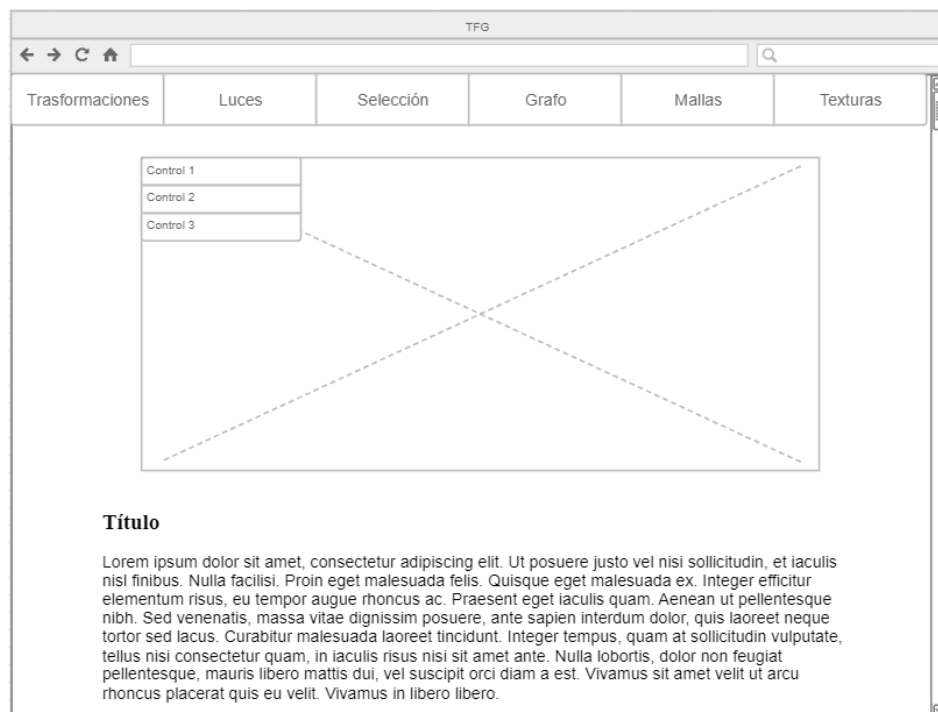


Ilustración 3.2-1: wireframe

3.3 Story Board

A continuación se muestra el storyboard de la aplicación por secciones.

3.3.1 Transformaciones geométricas

Primero, al entrar en la página web, la primera vista mostrada es la sección de transformaciones geométricas (1). En el canvas se podrá ver un menú desplegable (2) con los siguientes submenús desplegables: escalado, rotación y traslación.

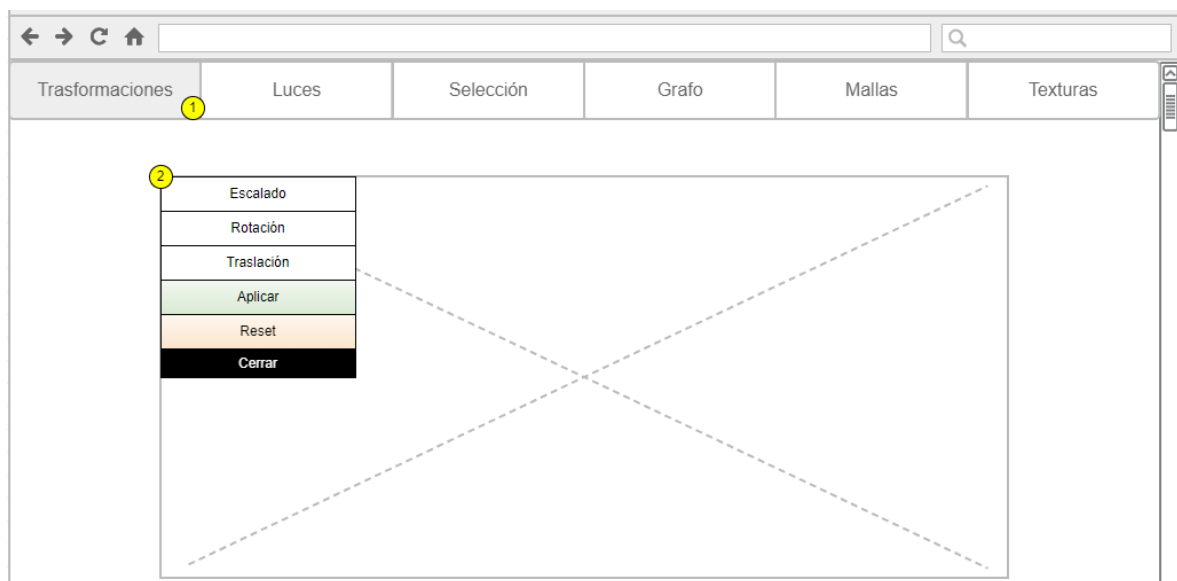


Ilustración 3.3-1: storyboard del apartado de transformaciones

Luego, al clicar en uno de ellos, por ejemplo el de escalado (3), se despliegan las variables (4) que representan las 3 dimensiones (x,y,z) en la que se introducen los valores deseados para escalar el modelo. Los submenús de rotación (5) y traslación (6) son análogos.

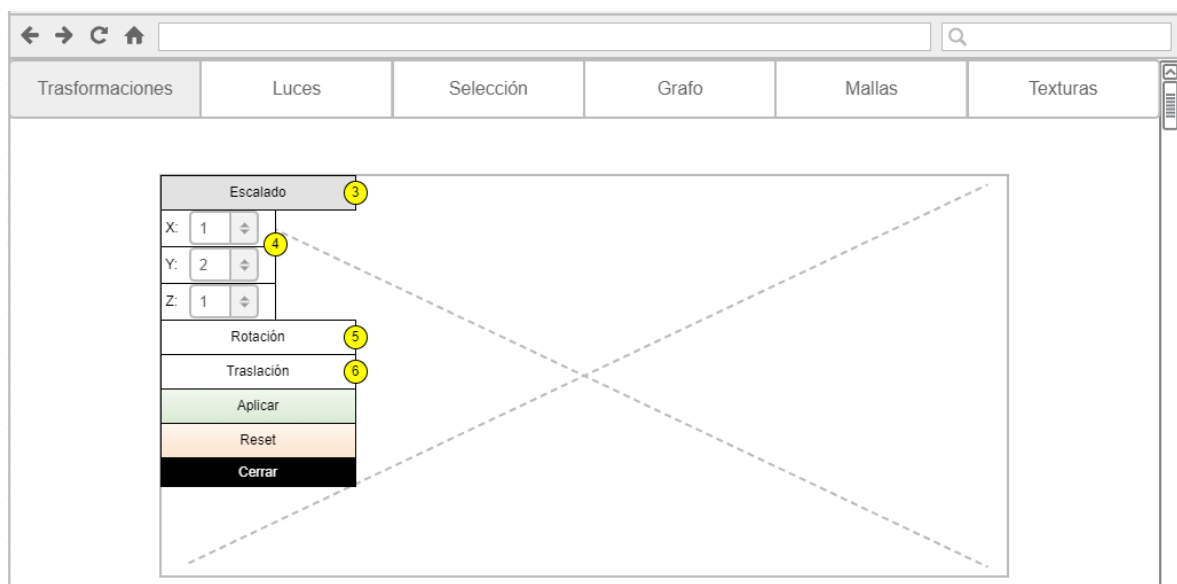


Ilustración 3.3-2: storyboard del apartado de transformaciones

Finalmente, los botones del final se emplean para aplicar (7) las transformaciones con los valores introducidos, el de reset (8) devuelve al objeto a su estado inicial, y el de cerrar (9) sirve para ocultar el menú desplegable.

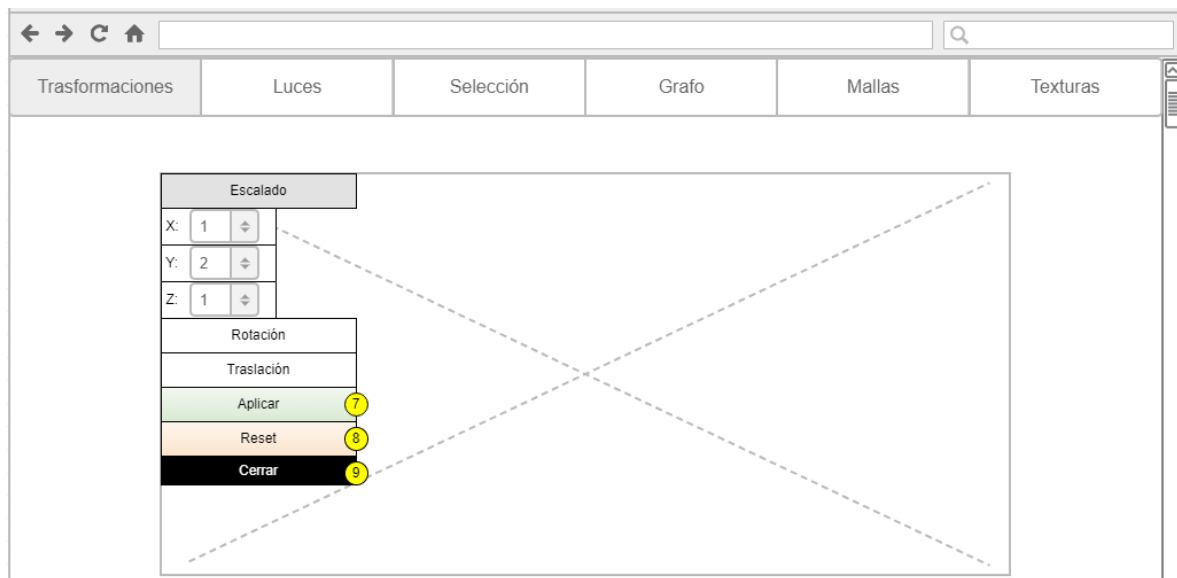


Ilustración 3.3-3: storyboard del apartado de transformaciones

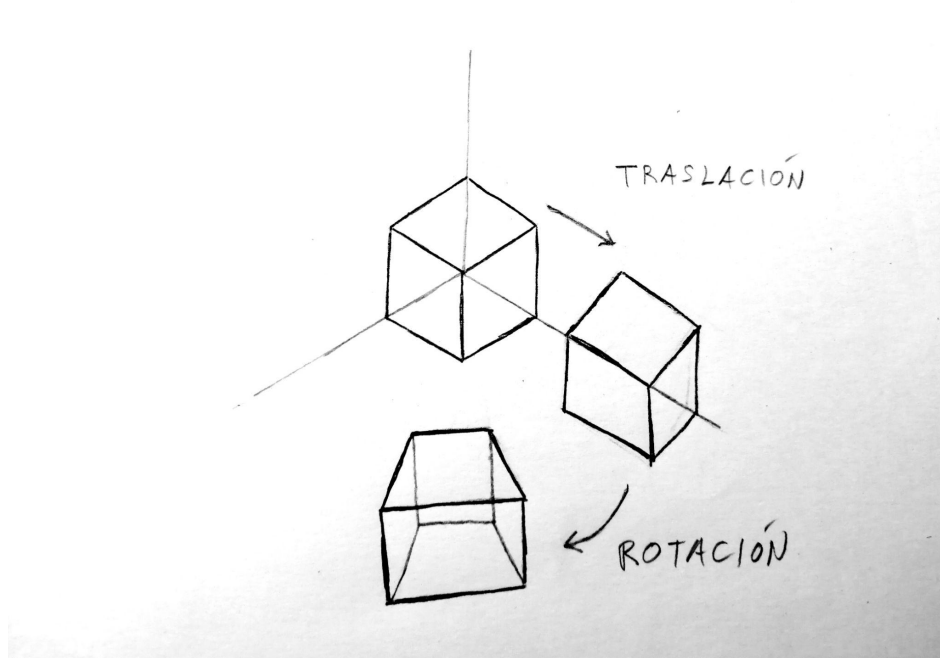


Ilustración 3.3-4: boceto de escena

3.3.2 Luces

La segunda sección es la de luces que contiene 4 apartados donde se ve en detalle cada uno de los tipos de luces. Clicando en el segundo enlace de la barra navegable (1) aparece un desplegable (2) con los apartados.

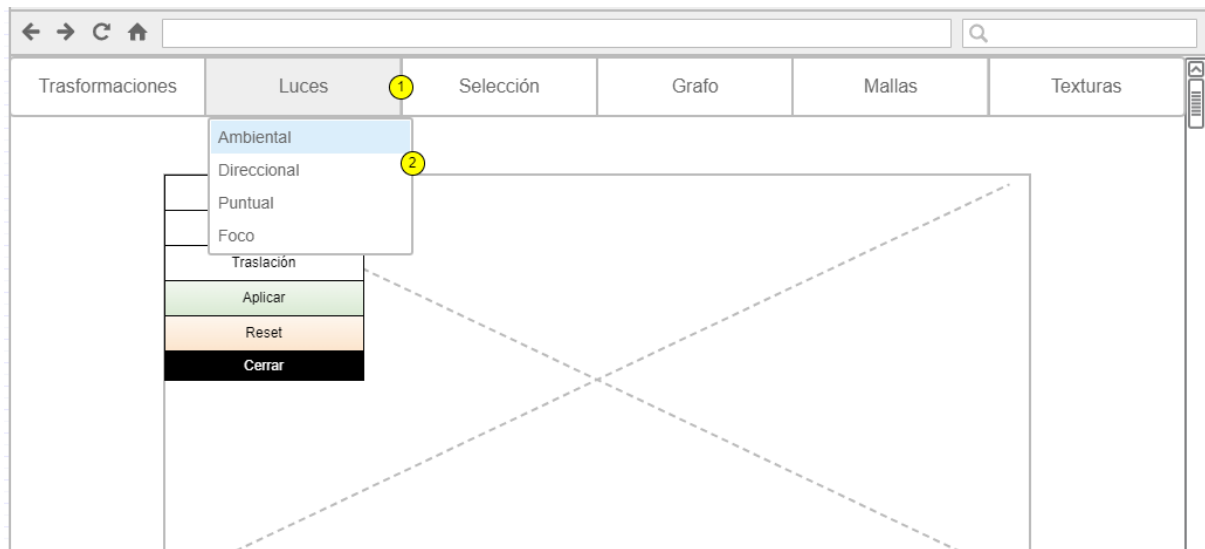


Ilustración 3.3-5: storyboard del apartado de luces

3.2.2.1 Luz ambiental

Si se accede al primer apartado, ilustración 3.2-4, se muestra una escena distinta con otras variables en el menú, en este caso son: intensidad y color.

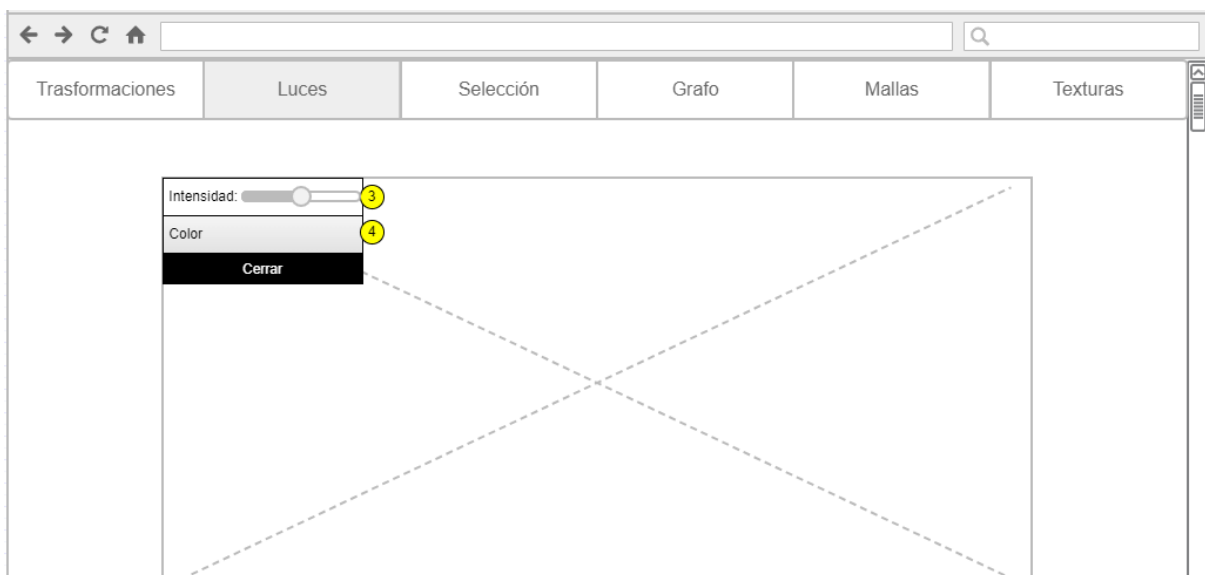


Ilustración 3.3-6: storyboard de luz ambiental

La intensidad (3) será modificable a través de un slide con valores decimales comprendidos en el intervalo $[0, 2]$.

Por otro lado, el color será un botón (4) que permita mostrar y ocultar un desplegable (5) para modificar los valores RGB con valores enteros comprendidos en el intervalo $[0, 255]$.

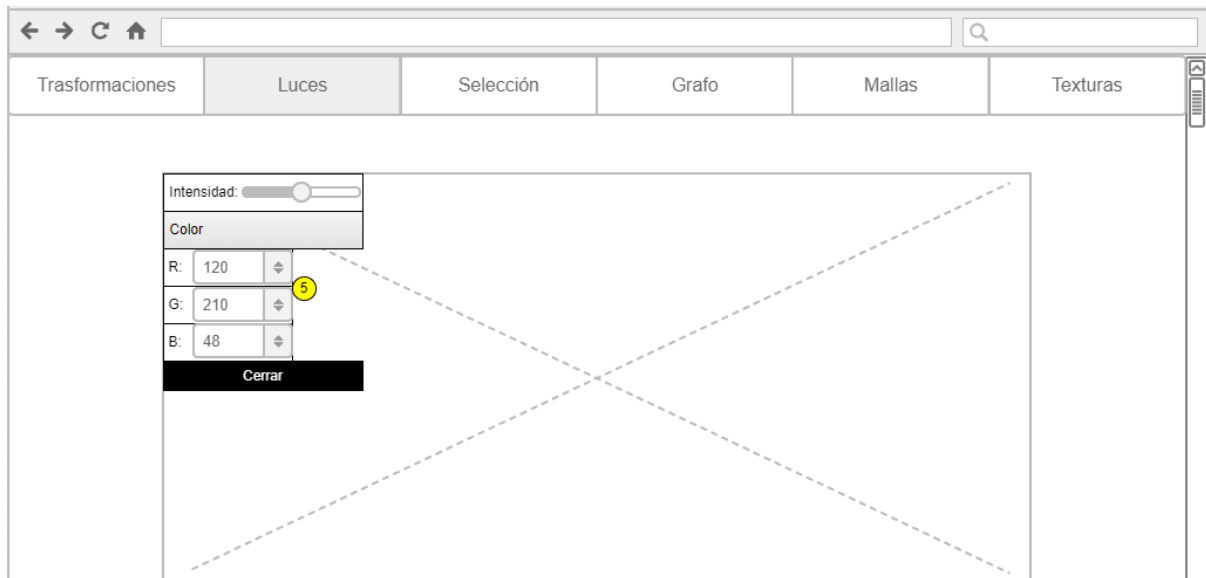


Ilustración 3.3-7: storyboard de luz ambiental

A diferencia de los controles de la escena de transformaciones geométricas, no es necesario un botón de “aplicar” porque la escena se actualizará conforme se modifiquen los parámetros.

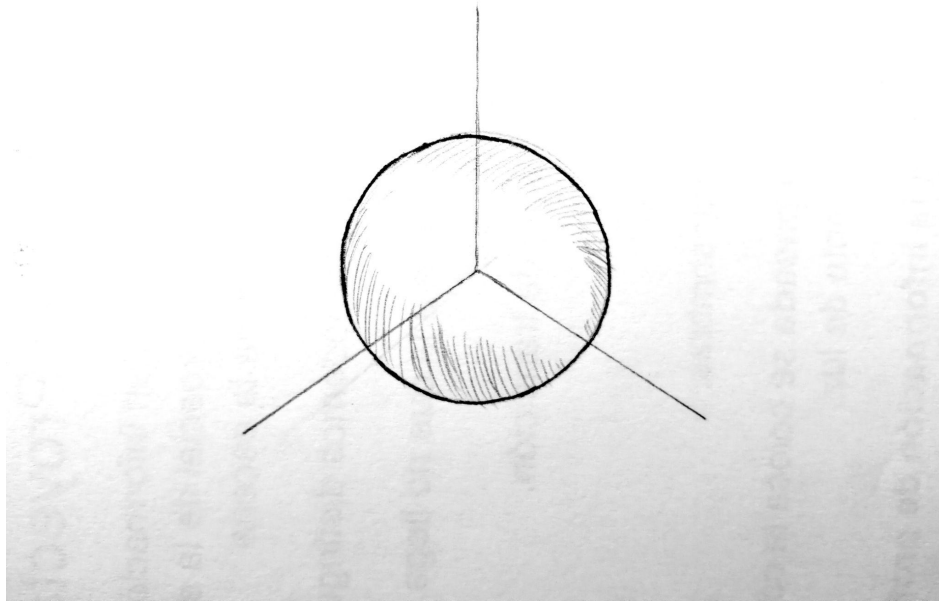


Ilustración 3.3-8: boceto de luz ambiental

3.3.2.1 Luz direccional

Para la luz direccional los controles son los mismos pero se le añaden dos nuevos que permiten rotar la luz respecto los ejes X y Z para cambiar su posición, para ello se emplearán dos slides (6) con valores entre 0 y 180 para que represente los grados.

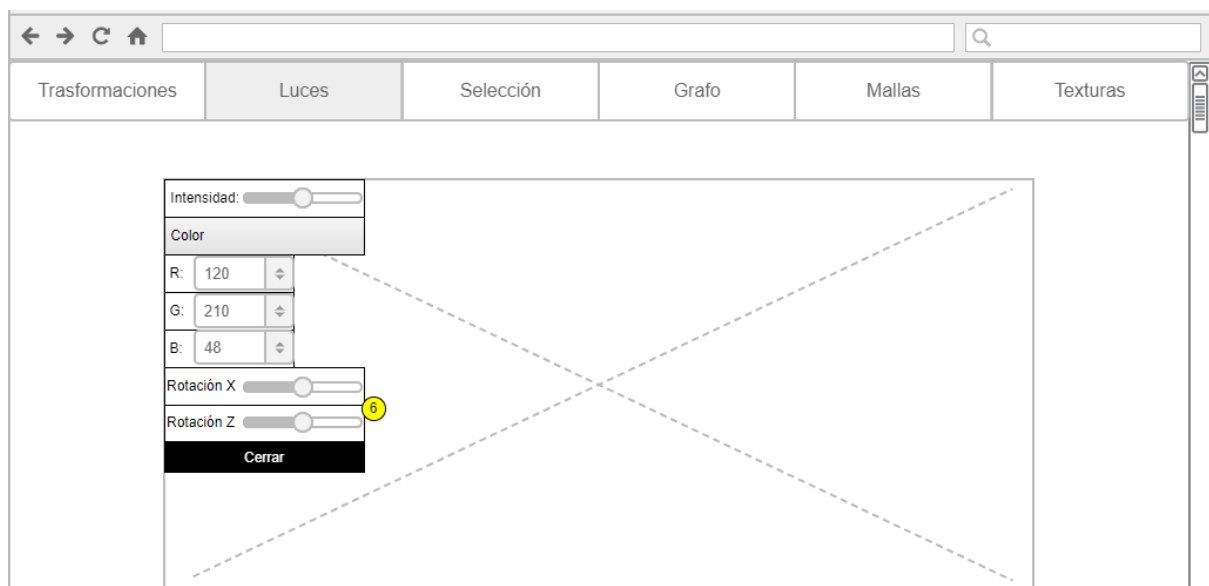


Ilustración 3.3-9: storyboard de luz direccional

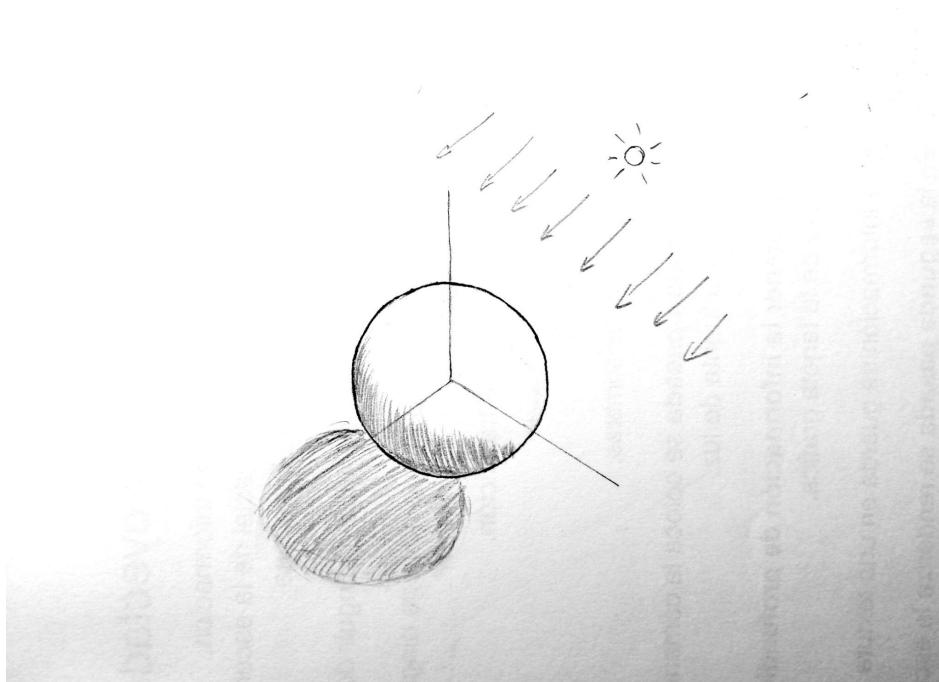


Ilustración 3.3-10: boceto de luz direccional

3.3.2.1 Luz puntual

La luz puntual tendrá los mismos controles que la ambiental pero se le añade uno adicional para controlar la atenuación, que será mediante un slide (7) con valores enteros entre 0 y 10.

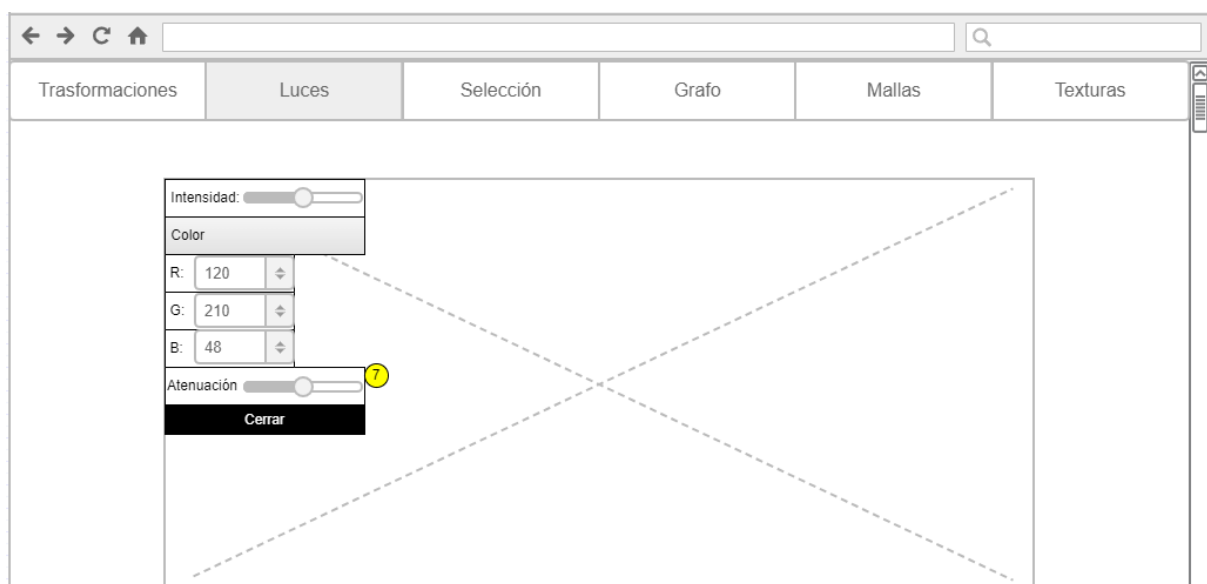


Ilustración 3.3-11: storyboard de luz puntual

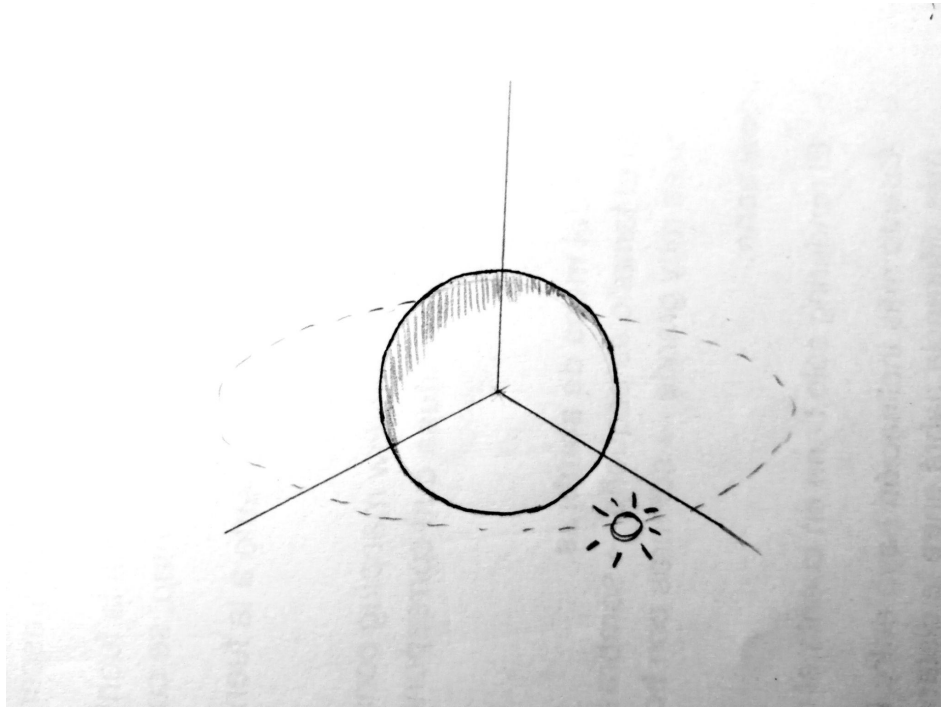


Ilustración 3.3-12: boceto de luz puntual

3.3.2.1 Foco de luz

Finalmente el foco de luz, sus parámetros son parecidos a los de la luz puntual pero incluyendo un modificador para el ángulo de apertura (8), con valores enteros entre 0 y 90, y otro para definir la penumbra (9), con valores entre 0 y 1.

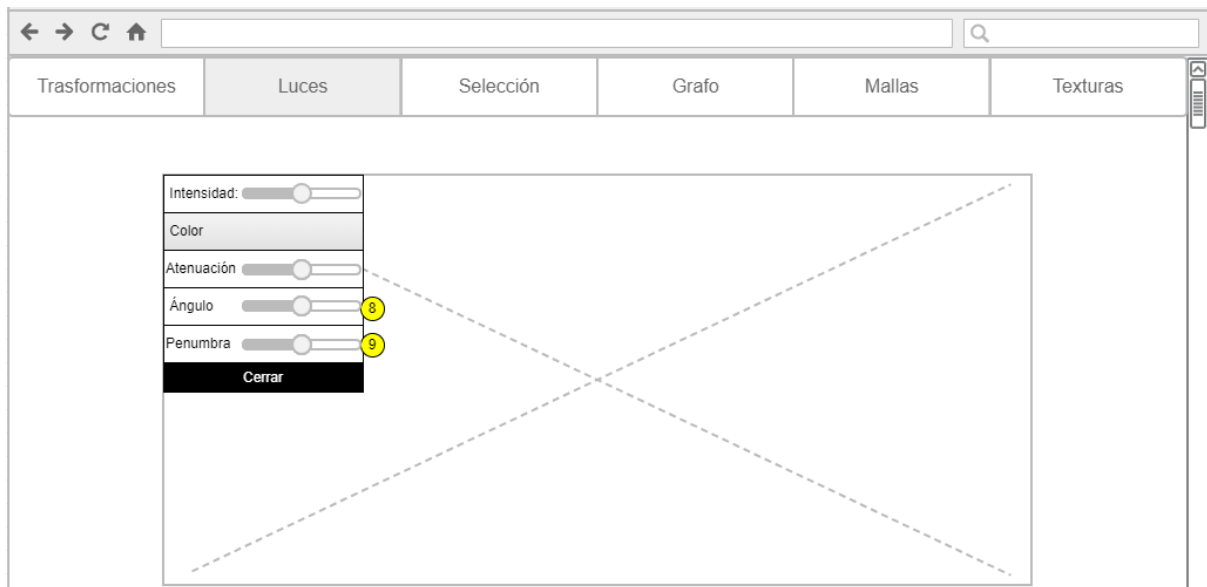


Ilustración 3.3-13: storyboard de luz focal

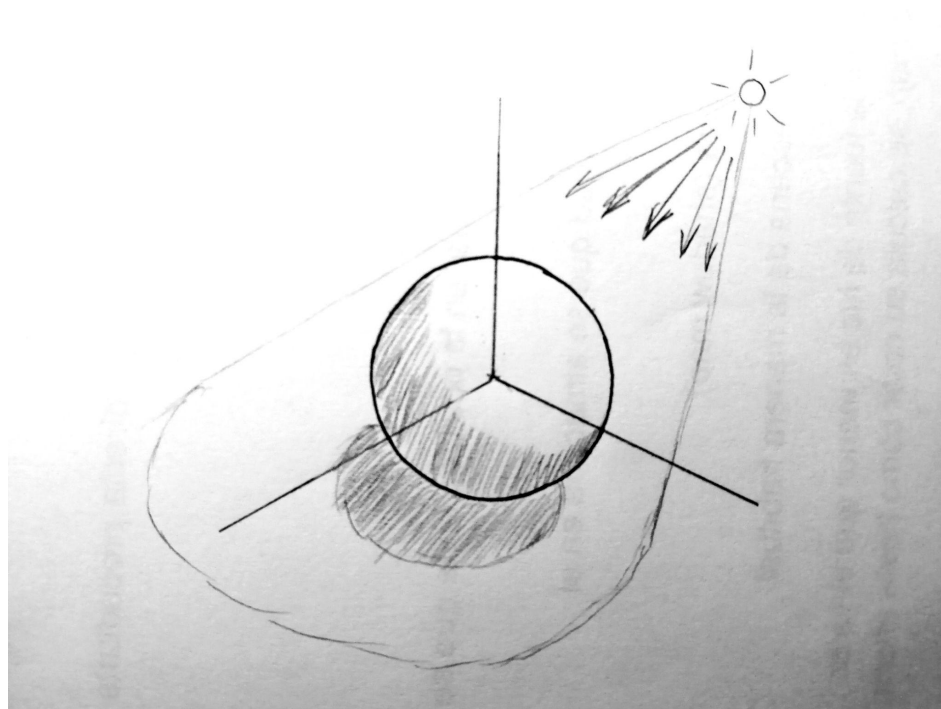


Ilustración 3.3-14: boceto de luz focal

3.3.3 Selección

En este caso no será necesario incluir un menú con elementos interactivos ya que la interacción recae en acciones que se podrán hacer con el ratón, como clicar en un elemento y acto seguido mover el ratón para rotarlo.

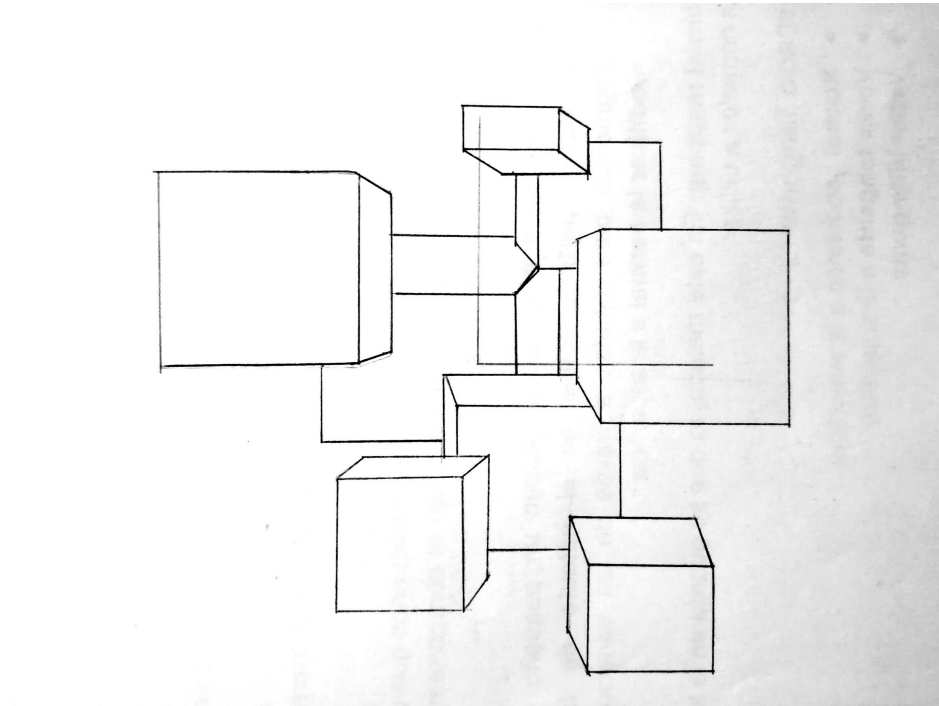


Ilustración 3.3-15: boceto de selección

3.3.4 Grafo

En esta escena se representará un brazo robótico cuyos puntos articulables están denominados como: hombro, codo y muñeca. Para que sea intuitivo qué controles y valores se aplican a cada parte. Por lo que el menú contará con 3 apartados en los que se incluyen las rotaciones aplicables a cada articulación: rotar hombro (1), rotar codo (2) y rotar muñeca (3). Cada apartado contará con los modificadores (4) donde se introducirán valores enteros en el intervalo $[0, 180]$. Aparte, se incluye un botón (5) con el que activar y desactivar una animación predefinida de la jerarquía de objetos.



Ilustración 3.3-16: storyboard de grafo de escena

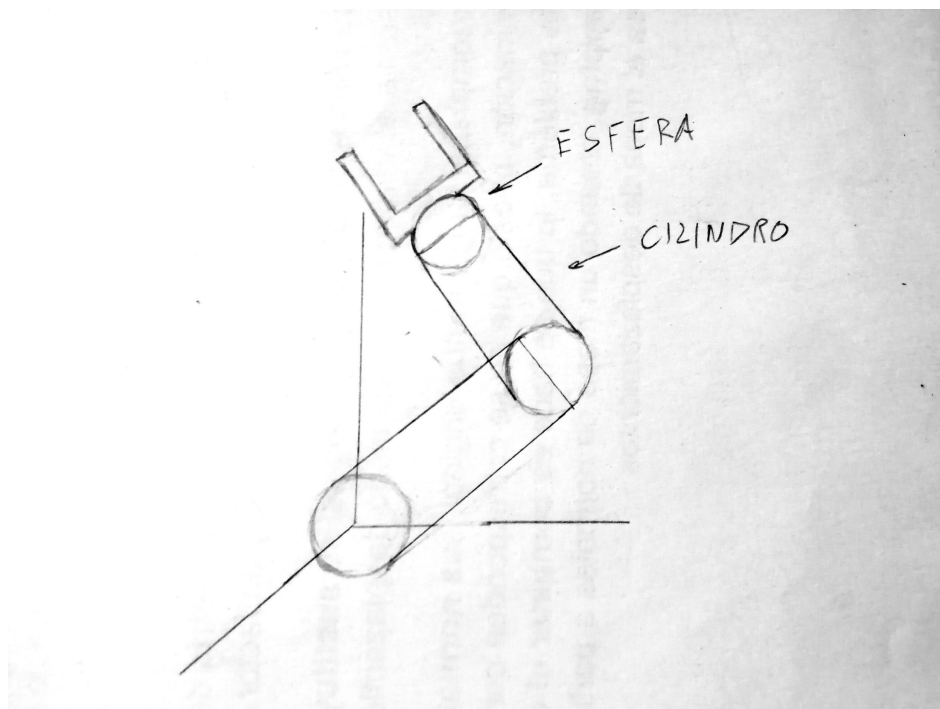


Ilustración 3.3-17: boceto de grafo de escena

3.3.4 Mallas

En esta escena se mostrará una esfera con su wireframe visible, por lo que se verán los triángulos por los que está compuesta. En el menú se incluirá interacciones para modificar su radio (1), así cambiará su tamaño, también para aumentar y disminuir sus divisiones laterales (2), denominadas “meridianos”, y sus divisiones longitudinales (3), llamadas “paralelos”.

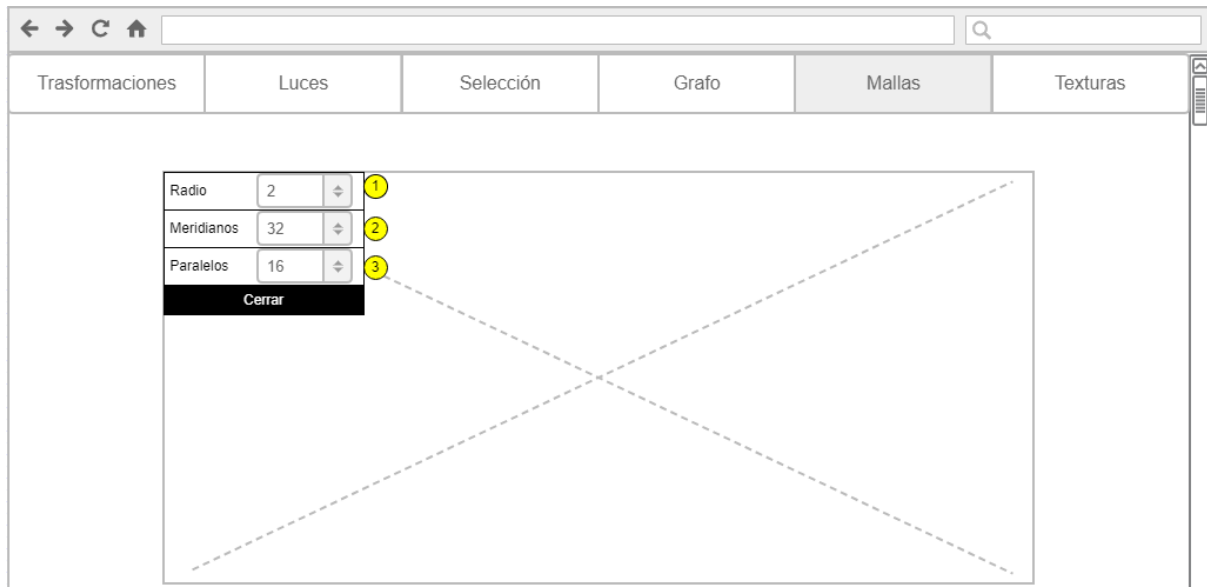


Ilustración 3.3-18: storyboard de mallas

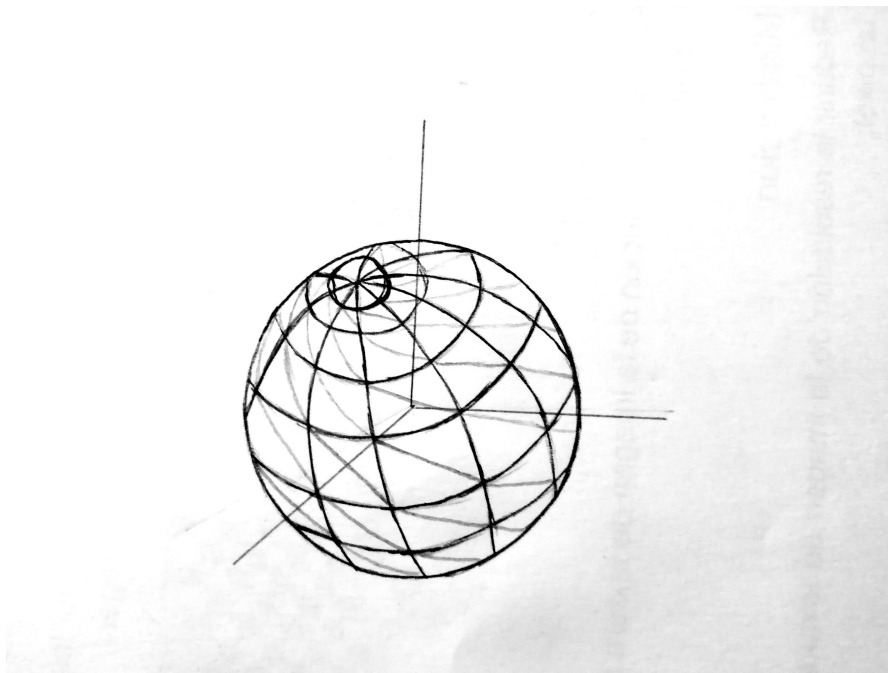


Ilustración 3.3-19: boceto de malla

3.3.5 Texturas

3.3.5.1 Aplicación de texturas

En esta sección se muestra un plano con una textura, al que se le podrá configurar los tipos de interpolación tanto para la minificación (1) como para la magnificación (2). También se podrá alternar entre dos texturas distintas (3).

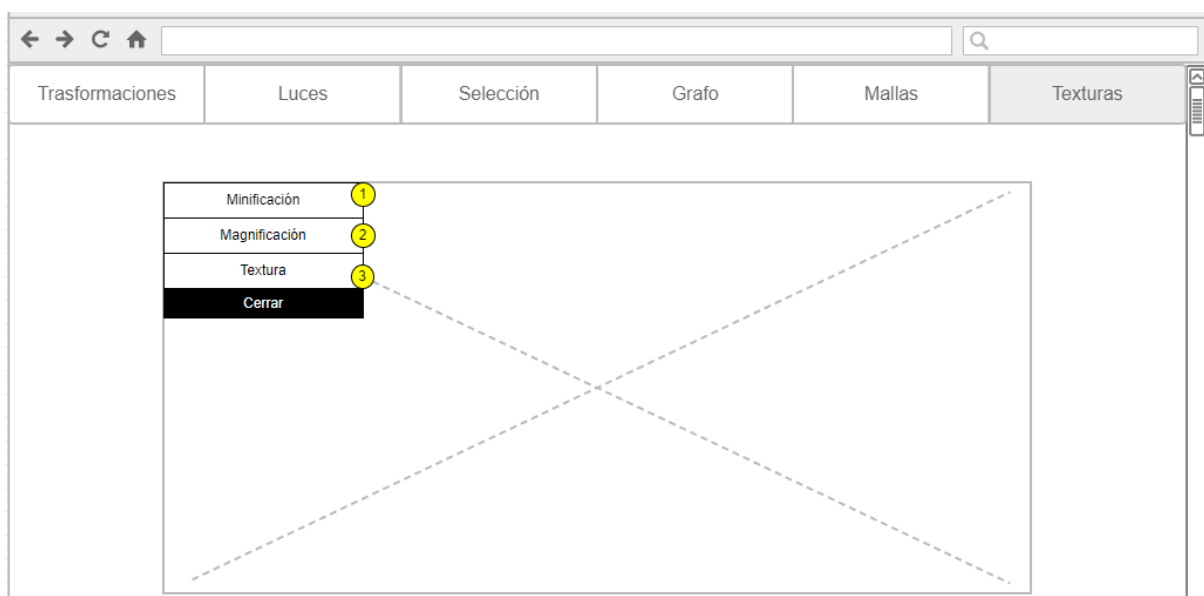
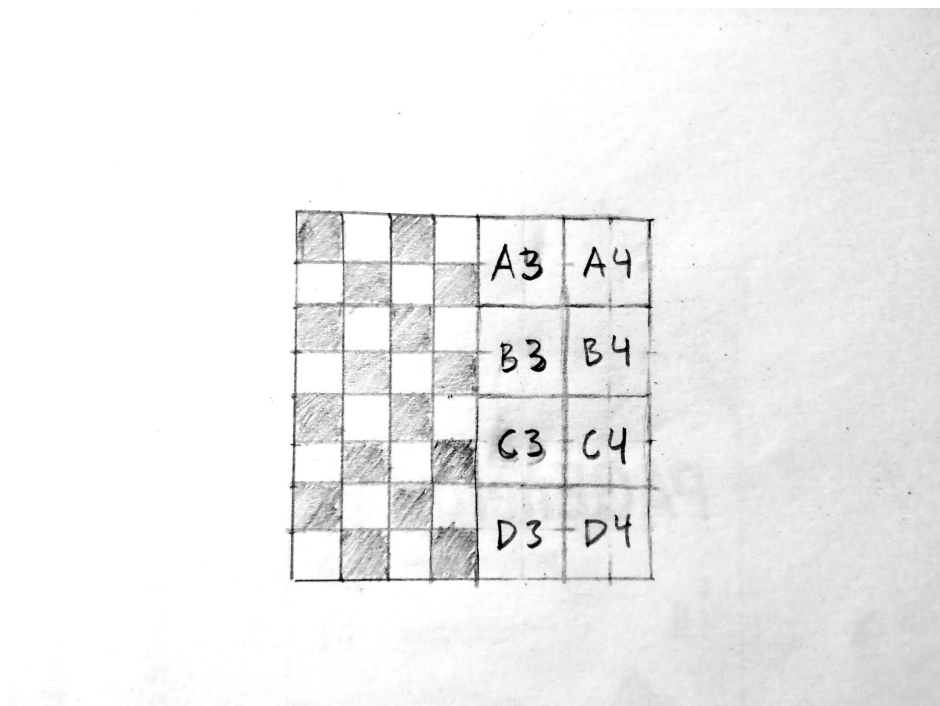


Ilustración 3.3-20: storyboard de aplicación de texturas

Cada elemento del menú será un desplegable como se muestra a continuación.

*Ilustración 3.3-21: storyboard de aplicación de texturas**Ilustración 3.3-22: boceto de escena de texturas*

3.3.5.2 Bump Mapping

En esta sección se muestra una primitiva o un modelo, con una textura aplicada sobre él y a la que se le podrá modificar el nivel de detalle del bump mapping con un slide (1) con valores entre 0 y 1, a parte se podrá visualizar otro modelo distinto (2) y alternar entre dos texturas diferentes (3).



Ilustración 3.3-23: storyboard de aplicación de texturas

3.3.5.3 Reflexión

En el último apartado se tratarán las propiedades de los materiales como la reflexión difusa y especular. Al igual que se ha mostrado anteriormente se podrá escalar el modelo y modificar su color. Adicionalmente se podrá alterar los factores de reflexión difusa (1) y especular (2). También se podrá aplicar una textura (3) que tenga precargadas mapas de reflexión difusas y especular.

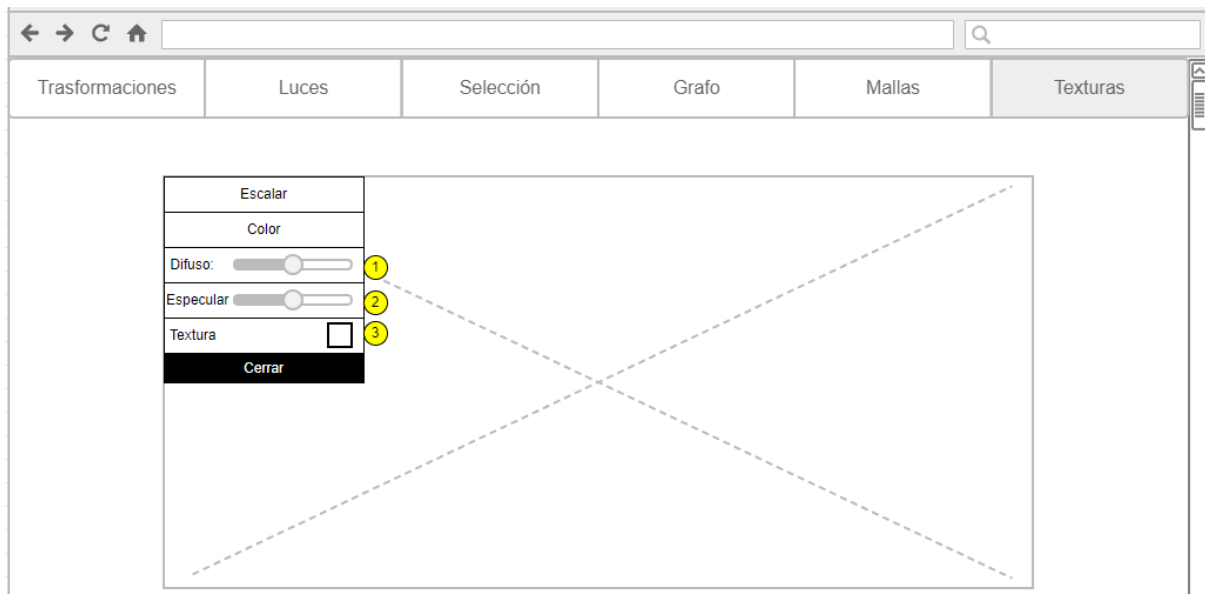


Ilustración 3.3-24: storyboard de aplicación de materiales

4. Desarrollo

La fase de desarrollo del proyecto se ha realizado siguiendo la metodología y planificación definidas en la fase de análisis, descritas en el punto 2. Por lo que, al tratarse de un marco de trabajo iterativo e incremental, esta sección estará dividida en tantos apartados como iteraciones existen.

4.1 Trabajo previo

En la planificación se muestra la fase de análisis y diseño en esta etapa, en la que también se incluyen el estudio de tecnologías y algunas pruebas realizadas para comprobar su funcionamiento y utilidad. Así como la búsqueda de un entorno de desarrollo adecuado y las bibliotecas correspondientes. Y un periodo de práctica con estas tecnologías antes de iniciar el desarrollo. Por lo que la primera tarea, *A - Configuración Proyecto* es llevada a cabo, a pesar de que estaba planificada para la primera iteración.

En las siguientes iteraciones se da por hecho que se dispone de las herramientas necesarias (ver apartado 2.2) por lo que es recomendable visitar los apéndices donde figuran el manual de instalación y el manual de usuario, antes de proceder con la lectura de las iteraciones.

4.2 Primera iteración

Para esta iteración se planea llevar a cabo las tareas desde la B a la C, asociadas a las HUs desde la 2 a la 9, que consiste en crear una escena que sirva de punto de partida para el resto, y crear las escenas donde se trabaja con transformaciones geométricas y fuentes de iluminación.

4.2.1 Escena Básica con Three.js

El primer paso fue configurar el documento Html para poder renderizar la escena. Para ello hubo que crear un canvas dentro del cuerpo y asociar el script que se iba a usar. Es importante establecer el tipo a “module”.

```
<main class="container">
```

```
<div class="p-4 p-md-5 mb-4">
  <div id="lienzo" class="px-0 col-md-6">
    <div id="gui"></div>
    <canvas id="mi_canvas" class="rounded"></canvas>
  </div>
</div>

<script type="module" src="../Ruta_del_script/script.js"></script>
</main>
```

Y aunque no era estrictamente necesario añadir las clases de estilo css, sí era muy recomendable para definir un área de visualización lo suficientemente grande como para poder renderizar la escena sin problemas.

A continuación, se creó el script y se incluyó la dependencia a la biblioteca de Three.js, y se adjuntó el controlador de eventos para que cuando se cargue la página, llame al método main.

```
import * as THREE from 'three';
function main(){
  //Código de La escena
}
window.addEventListener('load', main);
```

Dentro de la función main hubo que indicar cual es el canvas que va a emplear el código para la visualización de la escena. Acto seguido se creó y se inicializó una escena con un color de fondo que facilitase diferenciar el espacio delimitado por el canvas en el DOM.

```
function main(){
  var micanvas = document.querySelector('#mi_canvas');
  const scene = new THREE.Scene();
  scene.background = new THREE.Color(0x1b2631);
}
```

Cámaras

Por otro lado, se creó y se configuró una cámara. Existen dos tipos de cámaras:

- Perspectiva o *frustum*. Como su nombre indica es en forma de pirámide sólida con la punta cortada. Ofrece una vista en 3D donde las cosas en la

distancia parecen más pequeñas que aquellos elementos a menor distancia.

Y está definida por los siguientes parámetros:

- *Fov*, es el ángulo de apertura.
 - *Aspect*, define qué tan anchos son el frente y la parte posterior del tronco.
 - *Near*, define dónde comienza el frente del tronco.
 - *Far*, define dónde termina.
- Ortográfica. Esta cámara define un cubo y hace que los objetos aparezcan a la misma escala: las líneas paralelas permanecen paralelas y una unidad de longitud parecerá tener la misma longitud en todas partes del dibujo.

La cámara empleada en todas las escenas es la de tipo *frustum*. Y en el código se define de la siguiente manera.

```
const fov = 75;
const aspect = window.innerWidth / window.innerHeight;
const near = 0.1;
const far = 100;
const camera = new THREE.PerspectiveCamera(fov, aspect, near, far);
scene.add(camera);
```

También se puede situar en un punto concreto del espacio modificando directamente su posición. Es importante dirigir la cámara al origen para más adelante visualizar los objetos que se añadan.

```
camera.position.x = 3;
camera.position.y = 3;
camera.position.z = 3;
camera.lookAt(0,0,0);
```

Render

Finalmente para poder ejecutar el script y renderizar la escena se instanció un objeto *render*, que permite visualizar la escena en el canvas, junto con una función, *animate()*, que sirve como loop de renderizado.

```
const renderer = new THREE.WebGLRenderer({canvas: micanvas});
```

```
renderer.render(scene, camera);

function animate() {
    requestAnimationFrame( animate );
    renderer.render( scene, camera );
};

animate();
```

Resumiendo, el código resultante es el siguiente.

```
import * as THREE from 'three';
function main(){
    var micanvas = document.querySelector('#mi_canvas');
    const scene = new THREE.Scene();
    scene.background = new THREE.Color(0x1b2631);

    const fov = 75;
    const aspect = window.innerWidth / window.innerHeight;
    const near = 0.1;
    const far = 100;
    const camera = new THREE.PerspectiveCamera(fov, aspect, near, far);
    scene.add(camera);

    camera.position.x = 3;
    camera.position.y = 3;
    camera.position.z = 3;

    const renderer = new THREE.WebGLRenderer({canvas: micanvas});
    renderer.render(scene, camera);

    function animate() {
        requestAnimationFrame( animate );
        renderer.render( scene, camera );
    };

    animate();
}
window.addEventListener('load', main);
```

Una vez realizados estos pasos, abriendo el archivo HTML desde el navegador la vista es como la siguiente:

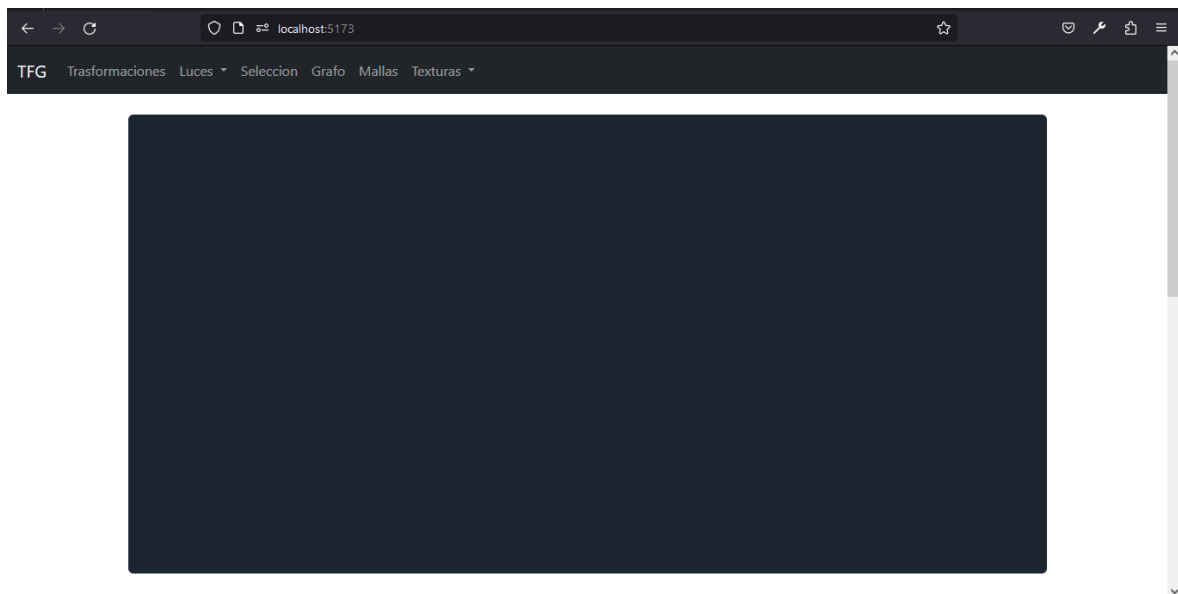


Ilustración 4.2-1: visualización inicial

4.2.1 Escena de transformaciones geométricas

Hasta este punto solo se podía ver un lienzo con un color plano. Por lo que concluyó que era conveniente introducir algún elemento fijo en la escena que indicase dónde se halla el origen de coordenadas, para que tanto el programador y el tester, como el usuario pudieran entender lo que se está representado y si los objetos se comportan como deberían. Para ello se dibujaron tres segmentos de colores distintos que representan las tres dimensiones del espacio.

Líneas

A continuación se muestra como dibujar una línea a lo largo del eje X usando Three.js.

```
const ejeXvertices = []; // Buffer
ejeXvertices.push( new THREE.Vector3( 0, 0, 0 ) ); // Coordenada inicial (0,0,0)
ejeXvertices.push( new THREE.Vector3( 3, 0, 0 ) ); // Coordenada final (3,0,0)

// Color rojizo
const colorEjeX = new THREE.LineBasicMaterial( { color: 0x943126 } );

const geometriaX = new THREE.BufferGeometry().setFromPoints( ejeXvertices );
const ejeX = new THREE.Line( geometriaX, colorEjeX );
scene.add(ejeX);
```

Primero se introducen las coordenadas inicial y final en un buffer. Luego se define

un material, y finalmente se crea la línea a partir de la geometría y el material especificados. Siguiendo estos pasos y cambiando únicamente la coordenada final y el color, se crean los ejes Y y Z. Como resultado, al actualizar el navegador, en la escena puede verse algo como la siguiente imagen.

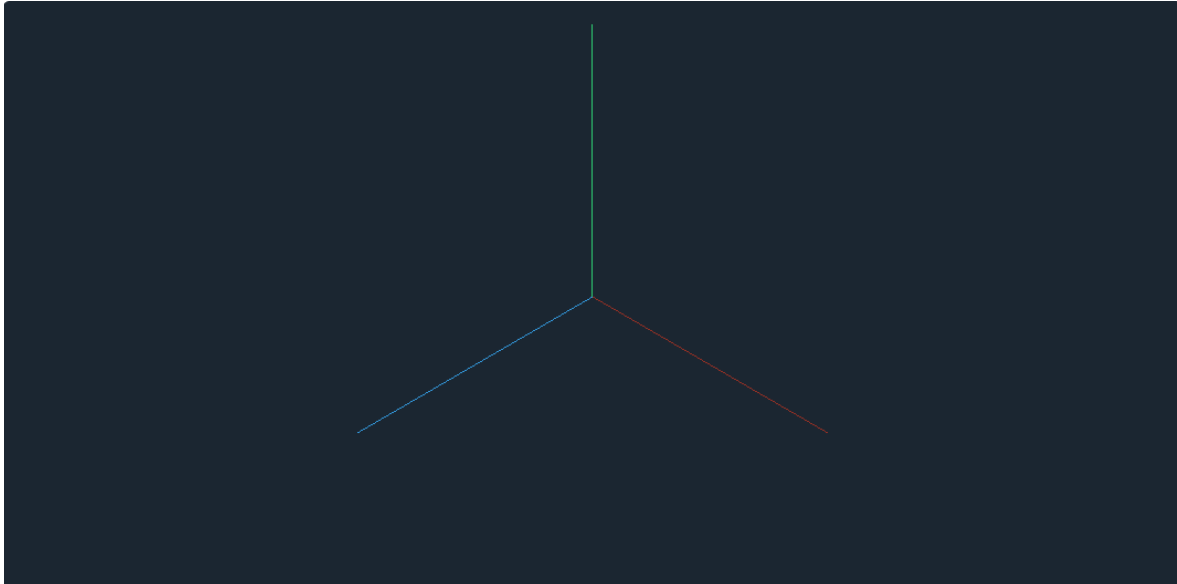


Ilustración 4.2-2: visualización ejes de coordenadas

Primitivas

El siguiente paso fue añadir la primitiva o el modelo sobre el que se aplican las transformaciones, para este caso se emplea un cubo. En Three.js todas las primitivas se definen de la misma manera: primero se define una geometría, luego un material, y por último se crea una malla con esas propiedades.

```
const geometriaCaja = new THREE.BoxGeometry( 1, 1, 1 );           // Geometría
var mat = new THREE.MeshPhongMaterial( { color: 0x673ab7 } );     // Material

const cubo = new THREE.Mesh( geometriaCaja, mat );               // Malla
scene.add( cubo );
```

Al actualizar el navegador, el cubo se veía de color negro, esto era porque aún no existía ninguna fuente de luz en la escena. Pues era la siguiente tarea a realizar, crear una luz ambiental y una puntual. Por ahora no se entra en detalle sobre estos elementos, ya que se estudia cada uno en sus respectivas secciones más adelante.

```
const color = 0xFFFFFF;  
const intensity = 1;  
  
const ambientLight = new THREE.AmbientLight(color, intensity);  
scene.add(ambientLight);  
  
const pointLight = new THREE.PointLight(color, intensity);  
pointLight.position.set(0, 7, 0);  
scene.add(pointLight);
```

Ahora al aplicar las luces sí se aprecia el color del cubo.

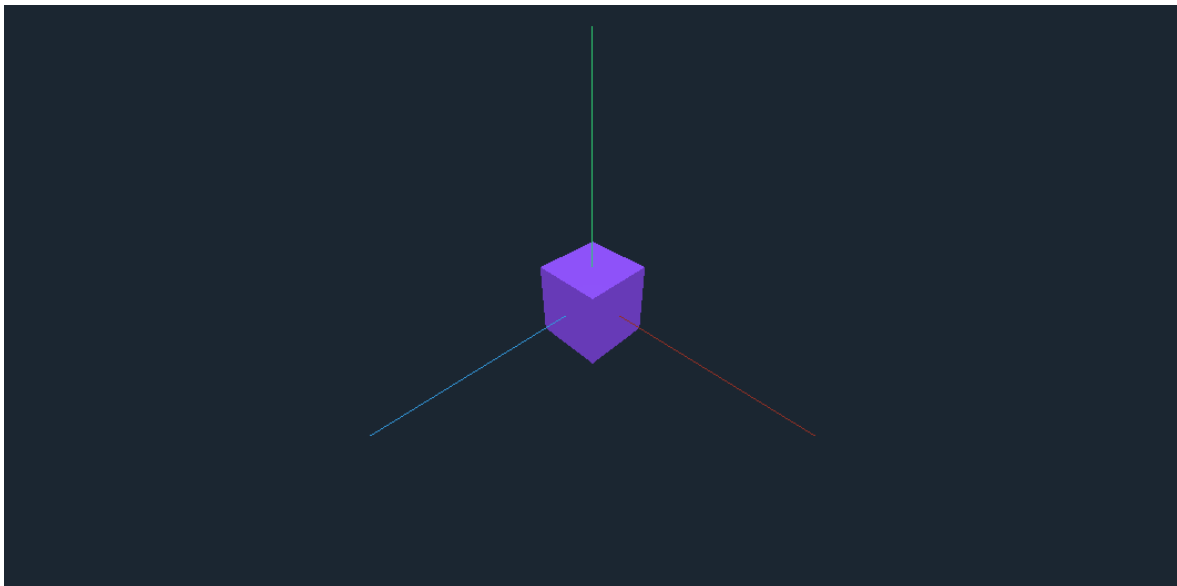


Ilustración 4.2-3: visualización del cubo

Transformaciones

El código que se explica a continuación no es el código implementado durante el primer sprint, sino que es el fruto de realizar varias correcciones y revisiones tras varios sprints. Tal como se indica en el apartado 2.5.2, en el diagrama de Gantt, se muestra en azul periodos de tiempo dedicados a correcciones. A pesar de ello, el código final se explica en este apartado para seguir un orden claro y facilitar su comprensión.

Antes de implementar las transformaciones cabe mencionar que la mayoría de las clases de Three.js, como las primitivas y las mallas, heredan de la clase `Object3D` [22] que implementa los siguientes métodos:

- `rotateX(rad)`. Gira el objeto alrededor del eje x en el espacio local. Igual manera para `rotateY` y `rotateZ`.
- `translateX(distancia)`. Traslada el objeto a lo largo del eje x en el espacio del objeto.
- `scale : Vector3`. Modificando esta propiedad se puede escalar el objeto directamente.

Al usar estos métodos no importa en qué orden se empleen las operaciones, porque la matriz resultante siempre va a ser $M = T \cdot R \cdot S$. Como se tiene que dar la posibilidad al usuario de introducir las transformaciones en cualquier orden, se ha implementado esas mismas operaciones realizando los cálculos de matrices aprovechando las operaciones de clase `Matrix4` [23] y aplicando la matriz resultante al modelo.

Por lo tanto, las variables a manejar son: una matriz identidad 4x4 a la que se le aplican los cálculos de traslación, rotación y escalado; y también 3 vectores del tipo (x,y,z) para almacenar temporalmente los datos introducidos por el usuario.

```
const transformM = new THREE.Matrix4();
transformM.identity();

const traslacion = new THREE.Vector3();
const rotacion = new THREE.Vector3();
const escalado = new THREE.Vector3();
```

Luego para aplicar las transformaciones, se creó una variable `options` que encapsula las funciones `aplicar()` y `reset()`, que son las opciones que se muestran en la interfaz.

```
var options = {
  aplicar: function(){
    // 1. Calcular traslación
    // 2. Calcular rotación
    // 3. Calcular Escalado
    // 4. Aplicar matriz resultado
    // 5. Reiniciar variables
    // 6. Actualizar display
  },
```

```
reset: function(){ ... }  
}
```

Como en un principio se desconoce las operaciones que introducirá el usuario, se implementó el cálculo de la matriz de transformación siguiendo el esquema $M = T \cdot R \cdot S$

La idea es que si se da el caso en que el usuario no introduce valores para alguna transformación, por ejemplo el escalado, al estar inicializado a (1,1,1), cuando se multiplique dicho vector por la matriz, el resultado seguirá siendo la misma matriz, es decir, como si no se hubiera realizado ningún cálculo.

Finalmente para reiniciar la escena, y devolver al modelo, en este caso al cubo, sus valores predeterminados se emplea la siguiente operación.

```
reset: function(){  
    cubo.matrix.identity().decompose(  
        cubo.position, cubo.quaternion, cubo.scale);  
}
```

GUI

Al igual que hay que especificarle a Three.js donde ubicar el canvas, con el GUI hay que hacer algo parecido, hay que especificar dónde se va a insertar.

```
const gui1 = new GUI( { autoPlace: false } );  
var customContainer = document.querySelector('#gui').append(gui1.domElement);
```

A continuación se añadieron los inputs y se asocian a las variables, para que se puedan modificar desde la vista. Estos controles se establecen dentro de unos desplegados llamados “folder”, de esta manera el usuario podrá ver y ocultar los controles que necesite.

Los parámetros que hay que introducir son: la variable a modificar, el valor mínimo, el valor máximo y el incremento. Luego con `setValue()` se indica el valor inicial que toma cuando se muestra por primera vez, y con `name()` se pone el nombre.

```
const escaladoGUI = gui1.addFolder('1. Escalado');
```

```

    escaladoGUI.add(escalado, 'x', 0.1, 3, 0.1).setValue(1).name('Ancho');
    escaladoGUI.add(escalado, 'y', 0.1, 3, 0.1).setValue(1).name('Alto');
    escaladoGUI.add(escalado, 'z', 0.1, 3, 0.1).setValue(1).name('Largo');

    const rotacionGUI = gui1.addFolder('2. Rotación');
    rotacionGUI.add(rotacion, 'x', -180, 180, 1).setValue(0).name('x');
    rotacionGUI.add(rotacion, 'y', -180, 180, 1).setValue(0).name('y');
    rotacionGUI.add(rotacion, 'z', -180, 180, 1).setValue(0).name('z');

    const traslacionGUI = gui1.addFolder('3. Traslación');
    traslacionGUI.add(traslacion, 'x', -3, 3, 0.1).setValue(0);
    traslacionGUI.add(traslacion, 'y', -3, 3, 0.1).setValue(0);
    traslacionGUI.add(traslacion, 'z', -3, 3, 0.1).setValue(0);

```

Por último se crearon unos botones para aplicar las transformaciones y reiniciarlas, tal como se ha explicado anteriormente. Y añadir una checkbox que permite visualizar el modelo en wireframe.

```

gui1.add(options, 'aplicar').name("Aplicar");
gui1.add(options, 'reset').name("Reset");

gui1.add(cubo.material, 'wireframe').name("Wireframe").listen();

```

Con todos estos añadidos si se actualiza la vista se debe de ver algo parecido a la siguiente imagen.

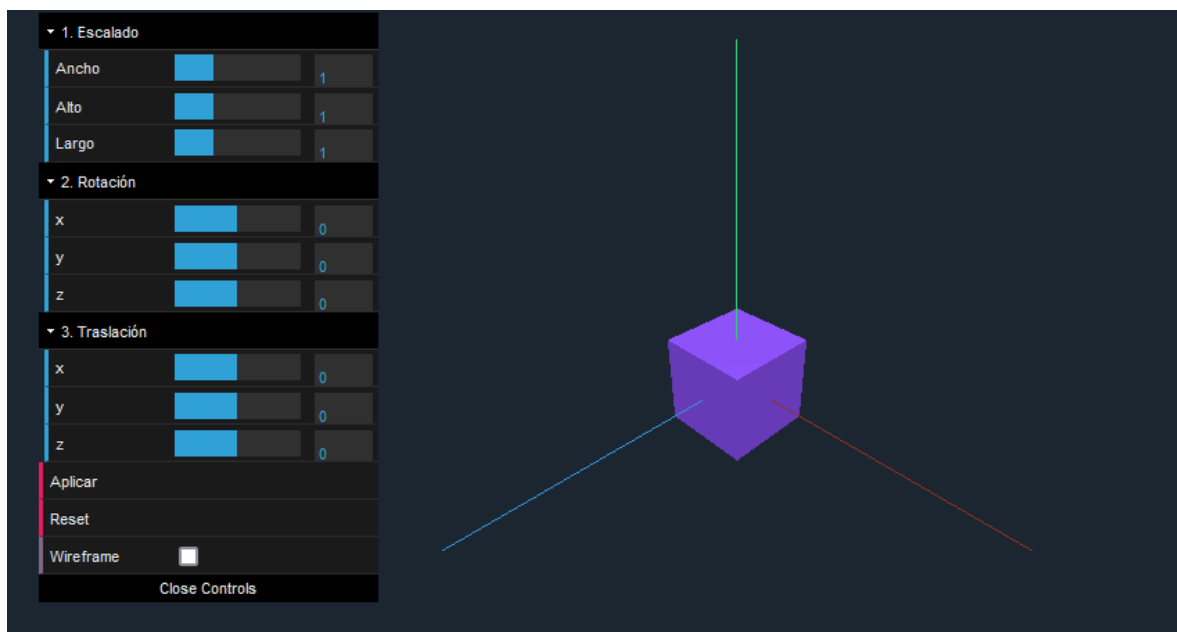


Ilustración 4.2-4: escena transformaciones

4.2.3 Luz ambiental

Tomando como punto de partida la escena básica lo primero es representar los ejes de coordenadas al igual que en la escena anterior, y también definir el elemento principal de esta escena: una luz ambiental. Para este tipo de luz únicamente son necesarios dos parámetros: el color y la intensidad.

```
const color = 0xFFFFFF;  
const intensity = 1;  
var ambientLight = new THREE.AmbientLight(color, intensity);  
scene.add(ambientLight);
```

Por otro lado, Three.js también incluye otro tipo de luz ambiental llamada luz hemisferio, esta pretende emular dos fuentes de luz a la vez, una proveniente desde arriba que sería el reflejo del cielo, y otra desde abajo que es la luz que rebota en el suelo.

```
const skyColor = 0xB1E1FF; // azul claro  
const groundColor = 0xB97A20; // naranja-marrón  
const hemisphereLight = new THREE.HemisphereLight(skyColor, groundColor,  
intensity);  
  
scene.add(hemisphereLight);
```

Incluir este tipo de luz no formaba parte de los requisitos, pero al ser un recurso más que ofrece la biblioteca y que aporta cierto realismo a la escena, se consideró que podía ser interesante incluirla, para que el usuario experimente y vea sus efectos.

GUI

A diferencia de los parámetros de la anterior escena, hubo que crear un selector de color. Para ello, en la documentación de Three.js se puede encontrar el siguiente fragmento de código.

```
class ColorGUIHelper {  
  constructor(object, prop) {  
    this.object = object;  
    this.prop = prop;  
  }  
  get value() { return `#${this.object[this.prop].getHexString()}`; }
```

```
set value(hexString) { this.object[this.prop].set(hexString); }  
}
```

Esto sirve para que el valor seleccionado desde el GUI se convierta en un tipo de valor válido para Three.js. Para implementar la interfaz se hace de la siguiente manera: se definen dos carpetas, una para cada luz, se crean los selectores de color y el modificador para la intensidad. En este caso no hace falta crear variables adicionales, ya que se pueden modificar las propiedades de los objetos directamente.

```
const ambientalGUI = gui1.addFolder('Ambiental');  
  
ambientalGUI.addColor(new ColorGUIHelper(ambientLight, 'color'),  
  'value').name('Color');  
  
ambientalGUI.add(ambientLight, 'intensity', 0, 2, 0.01).name("Intensidad");  
  
const hemisphereGUI = gui1.addFolder('Hemisferio');  
  
hemisphereGUI.addColor(new ColorGUIHelper(hemisphereLight, 'color'),  
  'value').name('Superior');  
  
hemisphereGUI.addColor(new ColorGUIHelper(hemisphereLight, 'groundColor'),  
  'value').name('Inferior');  
  
hemisphereGUI.add(hemisphereLight, 'intensity', 0, 2, 0.01).name("Intensidad");
```

Esfera

Como modelo escogido, para comprobar el efecto de las luces, ha sido una esfera. Para la cual se necesitan 3 parámetros: radio, segmentos laterales y longitudinales. También se define un material para poder apreciar los brillos de ambas luces, por lo que el modelo de Phong puede ser el más adecuado.

```
const radius = 1;  
const widthSegments = 32;  
const heightSegments = 16;  
  
var esferaGeo = new THREE.SphereGeometry(radius, widthSegments, heightSegments)  
var esferaMat = new THREE.MeshPhongMaterial({ color: 0xbcbcbc });
```

```
const esfera = new THREE.Mesh( esferaGeo, esferaMat );  
scene.add(esfera);
```

Como resultado final se obtiene la siguiente escena, en la que se puede apreciar como el contorno alto es de un color, y el contorno bajo de otro. Y disminuyendo la intensidad de la luz ambiental se observa cómo se oscurece.

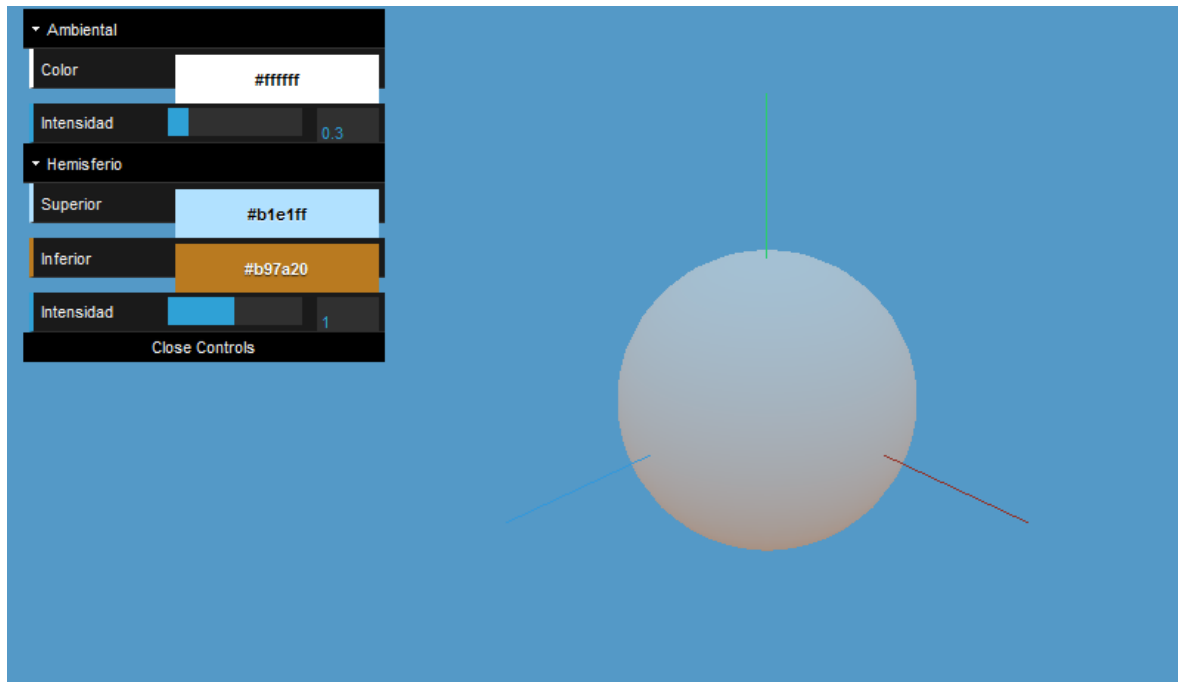


Ilustración 4.2-5: escena transformaciones

4.2.4 Luz direccional

Esta escena es parecida a la anterior, pero realizando algunos cambios. El primero de todos consiste en cambiar las luces y poner una luz direccional, estableciendo el color y la intensidad, al igual que la ambiental. Además hay que especificar otros dos parámetros: un punto al cual va a dirigir los rayos de luz, y una posición donde colocarla.

```
const color = 0xFFFFFF;  
const intensity = 1;  
const light = new THREE.DirectionalLight(color, intensity);  
light.position.set(0, 5, 0);  
light.target.position.set(0, 0, 0);
```

Para luego poder cambiar su posición y rotarla sin problemas desde la interfaz, se ha creado un objeto vacío al cual van enganchados tanto la luz como el objeto objetivo. Este objeto es al que se le aplican las transformaciones (explicado más adelante).

```
const pivot = new THREE.Object3D();  
scene.add(pivot);  
  
pivot.add(light);  
pivot.add(light.target);
```

Plano

En cuanto a los modelos se añade una esfera, con los mismos parámetros que la anterior, y un plano que está situado justo debajo.

```
const width = 10;  
const height = 10;  
const planeGeometry = new THREE.PlaneGeometry( width, height );  
  
const planeMat = new THREE.MeshPhongMaterial({ color: 0x7b7d7d });  
  
const plano = new THREE.Mesh( planeGeometry, planeMat );  
scene.add(plano);  
  
plano.translateY(-1.5);  
plano.rotateX(degToRad(-90));
```

GUI

Antes de implementar el GUI, hubo definir una variable para almacenar el valor de los ángulos introducidos por el usuario, `rotacionGUI`, y otra para almacenar el ángulo de rotación que tenga la luz en cada instante, `rotacionPivot`. Así como una función auxiliar para rotar la luz correctamente.

```
const rotacionGUI = {x: 90, z: 90};
const rotacionPivot = {x: 90, z: 90};

function rotateLigth(){
    var angX = -(rotacionPivot.x - rotacionGUI.x);
    rotacionPivot.x += angX;
    pivot.rotateX(degToRad(angX));

    var angZ = -(rotacionPivot.z - rotacionGUI.z);
    rotacionPivot.z += angZ;
    pivot.rotateZ(degToRad(angZ));
}
```

A continuación se definió la interfaz de manera parecida al de la luz ambiental, poniendo los controladores para la intensidad, el selector de colores, y para la variable `rotacionGUI`.

```
const gui1 = new GUI( { autoPlace: false } );
var customContainer = document.querySelector('#gui').append(gui1.domElement);

const direccionalGUI = gui1.addFolder('Direccional');

direccionalGUI.addColor(new ColorGUIHelper(light, 'color'),
'value').name('Color');

direccionalGUI.add(light, 'intensity', 0, 2, 0.01).name("Intensidad");

direccionalGUI.add(rotacionGUI, 'x', 0, 180).setValue(90).onChange(rotateLigth);
direccionalGUI.add(rotacionGUI, 'z', 0, 180).setValue(90).onChange(rotateLigth);
```

Si se coge el código explicado hasta ahora, se ejecuta y se actualiza el navegador se puede ver un resultado como el de la ilustración 4.2-4, en el que se puede mover la luz, modificar el color y la intensidad. A pesar de ello no queda muy realista ya que no se proyecta ninguna sombra. Aparte, la ubicación de la luz en cada instante no se aprecia de manera precisa. Por todo esto se han incluido algunas mejoras.

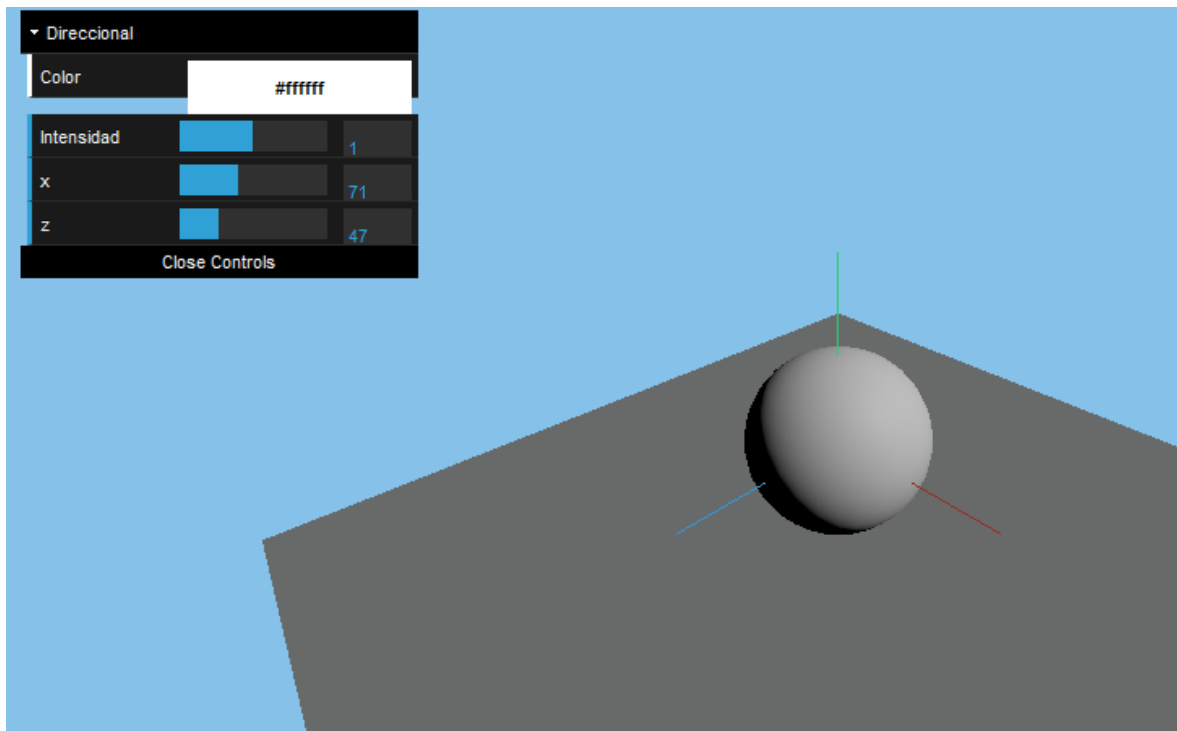


Ilustración 4.2-6: luz direccional versión 1

Sombras

Para poder visualizar las sombras, en la definición de la luz se debe activar la propiedad `castShadow`, y en los modelos también. Por otro lado hay que definir el tamaño del mapa de sombras.

```
light.castShadow = true;

light.shadow.mapSize.width = 800;
light.shadow.mapSize.height = 800;
light.shadow.camera.near = 0.5;
light.shadow.camera.far = 500;

...
esfera.castShadow = true;
...
plano.receiveShadow = true;
```

También se han dibujado, de manera representativa con unas líneas, la dirección de los rayos de luz. De esta manera se podrá ubicar fácilmente la luz en el espacio y

será más intuitivo para el usuario.

```
const helper = new THREE.CameraHelper( light.shadow.camera );  
scene.add( helper );
```

Finalmente en el renderer también hay que activar ciertas propiedades para que se visualicen las sombras.

```
const renderer = new THREE.WebGLRenderer({canvas: canvas});  
renderer.shadowMap.enabled = true;  
renderer.shadowMapSoft = true;
```

Como resultado, tras haber realizado estas mejoras, se puede observar la dirección con la que incide la luz y una sombra proyectada en el plano que se corresponde con esa trayectoria.

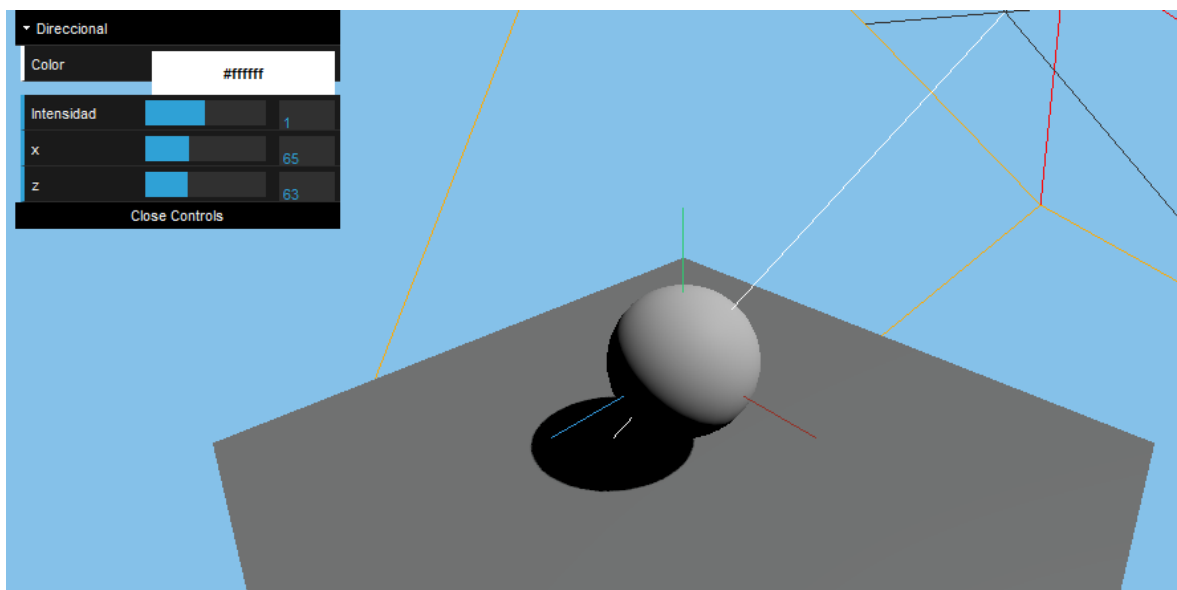


Ilustración 4.2-7: luz direccional versión 2

4.2.5 Luz puntual

Esta escena es bastante más sencilla que la anterior, lo único que hubo que hacer es implementar una luz puntual, estableciendo la intensidad, color y posición. De igual manera se dispuso dentro de un objeto vacío al que se le aplican las transformaciones. Y también se le asoció una esfera amarilla para indicar en cada momento dónde se encuentra.

```
const pivot = new THREE.Object3D();
pivot.position.set(0, 0, 0);
scene.add(pivot);

const color = 0xFFFFFF;
const intensity = 1;
const light = new THREE.PointLight(color, intensity);
light.position.set(0, 0, 0);
pivot.add(light);

const bombillaGeometria = new THREE.SphereGeometry(0.05, 8, 8);
const bombillaMaterial = new THREE.MeshToonMaterial( { color: 0xFFFF00} );
var bombilla = new THREE.Mesh(bombillaGeometria, bombillaMaterial);
light.add(bombilla);
```

Dentro del loop de renderizado se rota el objeto padre, pivot, tanto en X como en Z para hacer una animación y que la luz rote alrededor de la esfera, e ilumine desde diversos puntos.

```
function animate() {
    requestAnimationFrame( animate );

    pivot.rotation.y += 0.01;
    pivot.rotation.x += 0.01;

    renderer.render( scene, camera );
};
```

Finalmente la interfaz se configura como ya se ha visto en otras escenas.

```
const gui1 = new GUI( { autoPlace: false } );
var customContainer = document.querySelector( '#gui' ).append(gui1.domElement);

const puntualGUI = gui1.addFolder( 'Puntal' );
```

```
puntualGUI.addColor(new ColorGUIHelper(light, 'color'), 'value').name('Color');  
puntualGUI.add(light, 'intensity', 0, 2, 0.01).name("Intensidad");  
puntualGUI.add(light, 'distance', 0, 10).set_value(5).name("Atenuación");
```

El resultado final es el siguiente.

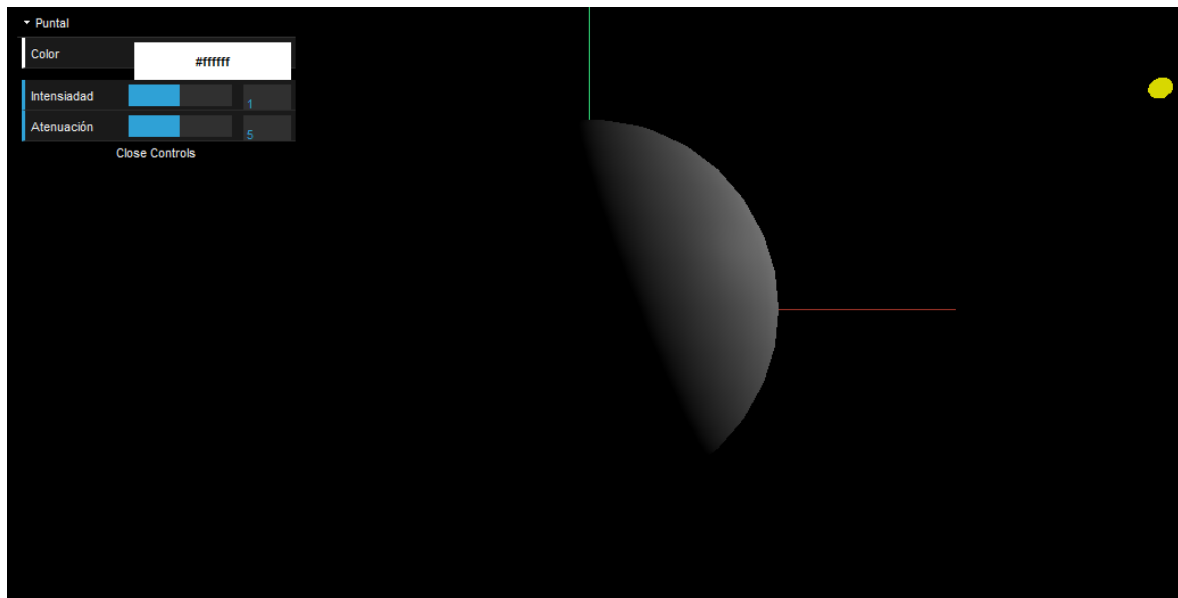


Ilustración 4.2-8: escena luz puntual

4.2.6 Foco

Para acabar con las escenas relacionadas con iluminación queda por implementar un foco. Tomando el mismo código empleado en la escena de la luz direccional, pero cambiando el tipo de luz, cuyos parámetros son parecidos a la direccional: color, intensidad, posición y objetivo.

```
const color = 0xFFFFFF;  
const intensity = 1;  
const light = new THREE.SpotLight(color, intensity);  
light.position.set(0, 4.3, -2.5);  
scene.add(light);  
scene.add(light.target);
```

Y también se modificó la interfaz para que se puedan controlar no solo la intensidad y el color, sino también la atenuación, la penumbra de los bordes y el ángulo de apertura.

```
const gui1 = new GUI( { autoPlace: false } );
var customContainer = document.querySelector('#gui').append(gui1.domElement);

const focalGUI = gui1.addFolder('Focal');
focalGUI.addColor(new ColorGUIHelper(light, 'color'), 'value').name('Color');
focalGUI.add(light, 'intensity', 0, 2, 0.01).name("Intensidad");
focalGUI.add(light, 'distance', 0, 40).onChange(
  helper.update()).name("Atenuacion");

focalGUI.add(new DegRadHelper(light, 'angle'), 'value', 0,
  90).name('angle').setValue(60).onChange(updateLight).name("Angulo");

focalGUI.add(light, 'penumbra', 0, 1, 0.01).setValue(0.1).name("Penumbra");
```

También, para que se represente el ángulo de apertura y la dirección mediante líneas, se ha creado un helper asociado a la luz.

```
const helper = new THREE.SpotLightHelper(light);
scene.add(helper);

function updateLight() {
  light.target.updateMatrixWorld();
  helper.update();
}
```

Por lo que el resultado final es el siguiente.

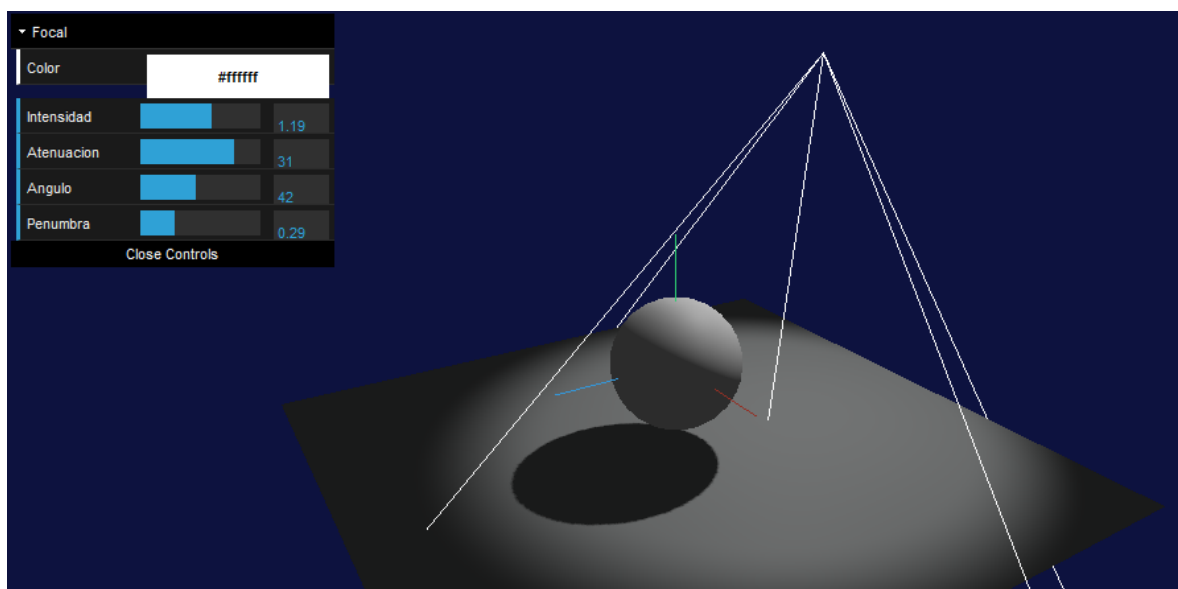


Ilustración 4.2-9: escena foco

4.3 Segunda iteración

En esta iteración las tareas a realizadas son G, H e I, que corresponden con las HUs: 9, 10, 11, 12 y 13. Que consisten principalmente en desarrollar las escenas de selección de objetos, grafo de escena y representación de una malla de triángulos.

4.3.1 Selección de objetos

La selección de objetos se puede realizar mediante dos técnicas: intersección de rayos o selección por colores. La intersección de rayos consiste en lanzar un rayo y calcular la intersección con el resto de objetos. Para la selección de colores hay que implementar dos escenas con los mismos objetos en las mismas posiciones, pero una de ellas no incluye luces y los materiales son emisivos, y hay que trabajar con ambas escenas a la vez.

Cámara

Para esta escena la cámara se colocó sobre el eje Z y a una distancia prudencial, a 30 unidades de distancia respecto del origen.

```
const camera = new THREE.PerspectiveCamera( 75, window.innerWidth /  
window.innerHeight, 0.1, 200);  
  
camera.position.z = 30;  
camera.lookAt(0,0,0);  
scene.add(camera);
```

Luces

Se añadieron también dos luces, una direccional y otra ambiental, para poder distinguir las caras de los cubos repartidos por la escena, ya que algunas tienen otra tonalidad al estar sombreadas.

Cubos

Para poder repartir los cubos por la escena se creó una función auxiliar que dados dos números, devuelve un valor aleatorio en ese intervalo. Y otra función que devuelve un color aleatorio.

```
function randomColor() {  
    var r = (Math.random() * 6) / 10.0 ;  
    var g = (Math.random() * 6) / 10.0 ;  
    var b = (Math.random() * 6) / 10.0 ;  
    return new THREE.Color(r,g,b);  
}
```

Finalmente, haciendo uso de estas dos funciones, se crean 100 cubos de manera aleatoria, tanto su color, como la posición y escalado.

```
const boxWidth = 1;  
const boxHeight = 1;  
const boxDepth = 1;  
  
const geometry = new THREE.BoxGeometry(boxWidth, boxHeight, boxDepth);  
  
const numObjects = 100;  
  
for (let i = 0; i < numObjects; i++) {  
    const material = new THREE.MeshPhongMaterial({ color: randomColor()});  
    const cube = new THREE.Mesh(geometry, material);  
    scene.add(cube);  
  
    cube.position.set(rand(-20, 20), rand(-20, 20), rand(-20, 20));  
    cube.scale.set(rand(3, 6), rand(3, 6), rand(3, 6));  
}
```

Raycast

Antes de definir el comportamiento del ratón al clicar, hubo que definir una serie de variables. Primero el raycaster de Three.js que lanza un rayo dada una posición, una variable para almacenar la posición del ratón al clicar, `x_pos` y `y_pos` para almacenar la posición del ratón en cada instante cuando es movido. Otra variable (`seleccionado`) para almacenar el primer objeto intersectado por el rayo y su color original (`colorSeleccionado`). Y un contador de tiempo (`seconds`).

```
const raycaster = new THREE.Raycaster();
const pointer = new THREE.Vector2();// Guardar la posición del puntero al clicar
var x_pos, y_pos;                // Guardar la posición del puntero al moverlo
var seleccionado = null;
var colorSeleccionado;
var seconds = 0;
```

Posteriormente, una función que permite devolver el color original al objeto previamente seleccionado en caso de que se clique en otro objeto.

```
function quitarSeleccionado(){
    if(seleccionado){
        seleccionado.material.emissive.setHex(colorSeleccionado);
        seleccionado = undefined;
        colorSeleccionado = undefined;
    }
}
```

También una función que devuelve la posición relativa del puntero cuando se clicca en el canvas.

```
function getCanvasRelativePosition(event) {
    const rect = canvas.getBoundingClientRect();
    return {
        x: (event.clientX - rect.left) * canvas.width / rect.width,
        y: (event.clientY - rect.top) * canvas.height / rect.height,
    };
}
```

Usando estas dos funciones, se implementó la función que se llama cada vez que se clicca en el canvas. La cual primero calcula la posición del puntero, lanza un rayo en esa dirección, en caso de haya un objeto previamente seleccionado, le devuelve su color original, finalmente comprueba si hay objetos intersectados: si el primer objeto es una malla, lo establece como seleccionado y lo resalta con un color intermitente.

```
function onPointerClick( event ) {
    const pos = getCanvasRelativePosition(event);
    pointer.x = (pos.x / canvas.width) * 2 - 1;
    pointer.y = (pos.y / canvas.height) * -2 + 1;

    // Se lanza un rayo en la posición del puntero
    raycaster.setFromCamera( pointer, camera );
```

```

// Obtenemos Los objetos impactados
const intersects = raycaster.intersectObjects( scene.children );

quitarSeleccionado();

if(0 < intersects.length){
    seleccionado = intersects[0].object;
    colorSeleccionado = seleccionado.material.emissive.getHex();

    seleccionado.material.emissive.setHex((seconds * 5) % 2 > 1 ?
    0x00FFFF : 0xFF0000);
}
}

```

Por otro lado se definió otra función que actualiza la posición del ratón cada vez que este se mueve por encima del canvas.

```

function onMouseMove(event){
    const pos = getCanvasRelativePosition(event);
    x_pos = (pos.x / canvas.width ) * 2 - 1;
    y_pos = (pos.y / canvas.height) * -2 + 1;
}

```

Y otra que rota el objeto seleccionado en dirección al puntero.

```

function rotarSeleccionado(){
    if(seleccionado){
        seleccionado.lookAt(x_pos*10, y_pos*10, camera.position.z);
        seleccionado.material.emissive.setHex((seconds * 5) % 2 > 1 ?
        0x00FFFF : 0xFF0000);
    }
}

```

Finalmente se establecieron los eventos asociados a esas funciones, para que se ejecuten cada vez que se haga una acción con el ratón.

```

canvas.addEventListener("mousedown", onPointerClick);
canvas.addEventListener("mousemove", onMouseMove);
canvas.addEventListener("mouseleave", quitarSeleccionado);

```

Render

Para terminar, queda la función `render` o `animate`, que se ejecuta en cada instante: actualiza el contador de tiempo y rota el objeto seleccionado.

```
function render(time) {  
    seconds = time * 0.001; // convert to seconds;  
  
    rotarSeleccionado();  
  
    renderer.render(scene, camera);  
  
    requestAnimationFrame(render);  
}  
  
requestAnimationFrame(render);
```

Como resultado se obtiene la siguiente escena



Ilustración 4.3-1: escena selección

4.3.2 Grafo de escena

El grafo de escena está inspirado en un brazo robótico con 3 puntos articulables, el primero se llama hombro, el segundo codo y el tercero muñeca para

que sea análogo a las partes del cuerpo humano, y sea más sencillo referenciar cada punto. La estructura está compuesta mediante primitivas: esferas para las articulaciones y cilindros para los fragmentos de unión.

Materiales

Los materiales empleados son del tipo MeshToonMaterial [24], este tipo de material está incluido en Three.js, sirve para simular un sombreado estilizado parecido al de dibujos animados, empleando varias tonalidades de un mismo color para resaltar brillos y sombras.



Ilustración 4.3-2: mapa de gradientes

Para generar esas tonalidades se emplea un mapa de gradientes como el de la ilustración 4.3-2. Este mapa hay que cargarlo como si fuera una textura y asignarlo a la propiedad de gradientMap del material.

```
const threeTone = new
THREE.TextureLoader().load('./src/assets/img/gradientMaps/threeTone.jpg');

threeTone.minFilter = THREE.NearestFilter;
threeTone.magFilter = THREE.NearestFilter;

const yellowToon = new THREE.MeshToonMaterial( { color: 0xf7dc6f });
yellowToon.gradientMap = threeTone;

const blackToon = new THREE.MeshToonMaterial( { color: 0x212f3d });
blackToon.gradientMap = threeTone;
```

Modelos

Como ya se ha mencionado, para la creación del brazo se emplean primitivas conectadas jerárquicamente, de esta manera al aplicar una rotación a un punto articulable, afecta también a las primitivas de los niveles inferiores. Los puntos articulables están representados con esferas:

1. Se sitúa la esfera de hombro en la escena

```
const hombro = new THREE.Mesh( esferaGeo, blackToon );
scene.add(hombro);
```

2. Se crea la esfera del codo, se traslada, y luego se enlaza con el hombro.

```
const codo = new THREE.Mesh( codoGeo, blackToon );
codo.translateY(1.5);
hombro.add(codo);
```

3. Y la esfera de la muñeca se añade al codo. Hay que tener en cuenta que las transformaciones se realizan en el espacio del codo, por lo que se separa una distancia de 1.5 unidades.

```
const muneca = new THREE.Mesh( munecaGeo, blackToon );
muneca.translateY(1.5);
codo.add(muneca);
```

Y para las partes que conectan visualmente cada punto articulable, se sigue el mismo proceso pero empleando cilindros.

GUI

Para crear las interacciones se definió una serie de variables numéricas asociadas a la interfaz, que representan los grados que se ha rotado cada articulación, y un booleano para indicar si la animación está activa.

```
const hombroGUI = {x:90, y:90, z:90};
const codoGUI = {x:270, y:270, z:270};
const munecaGUI = {y: 90};
const animacion = { play: false};
```

También se definieron unas funciones para rotar los 3 puntos articulables. Las rotaciones se realizan calculando la diferencia entre los grados indicados por la interfaz y los grados aplicados hasta ese instante. Por ejemplo, la función de rotación del hombro es tal que así:

```
function rotarHombro(){
    var angX = -(dataHombro.x - hombroGUI.x);
```

```
dataHombro.x += angX;
hombro.rotateX(degToRad(angX));

var angY = -(dataHombro.y - hombroGUI.y);
dataHombro.y += angY;
hombro.rotateY(degToRad(angY));

var angZ = -(dataHombro.z - hombroGUI.z);
dataHombro.z += angZ;
hombro.rotateZ(degToRad(angZ));
}
```

De esta manera se puede interactuar en tiempo real, el usuario puede ver como el modelo cambia a par que modifica los parámetros en la interfaz, sin necesidad de tener que depender de un botón adicional para aplicar los cambios.

Animación

Para la animación se creó la función `animar()`, a la cual se llama en cada vuelta del loop de renderizado siempre que se active dicha opción. Esta función actualiza los valores de los ángulos de rotación, y llama a las funciones encargadas de rotar cada parte. Los ángulos se incrementan cuando sus valores se encuentran en un intervalo válido, para así evitar hacer movimientos incorrectos.

```
function animar(){
    if(hombroGUI.x < 145 && hombroGUI.z < 145){
        hombroGUI.x += 0.1;
        hombroGUI.z += 0.1;

        rotarHombro();
    }

    if(codoGUI.x > 220 && codoGUI.z > 220){
        codoGUI.x -= 0.1;
        codoGUI.z -= 0.1;

        rotarCodo();
    }

    if(munecaGUI.y < 180){
        munecaGUI.y += 0.15;
        rotarMuneca();
    }
}
```

```
    gui.updateDisplay();  
}
```

Por último, para activar y desactivar la animación en la interfaz se añadió una checkbox. Con un click permite alternar entre “true” y “false”.

```
gui.add(animacion, 'play').name("Animar");
```

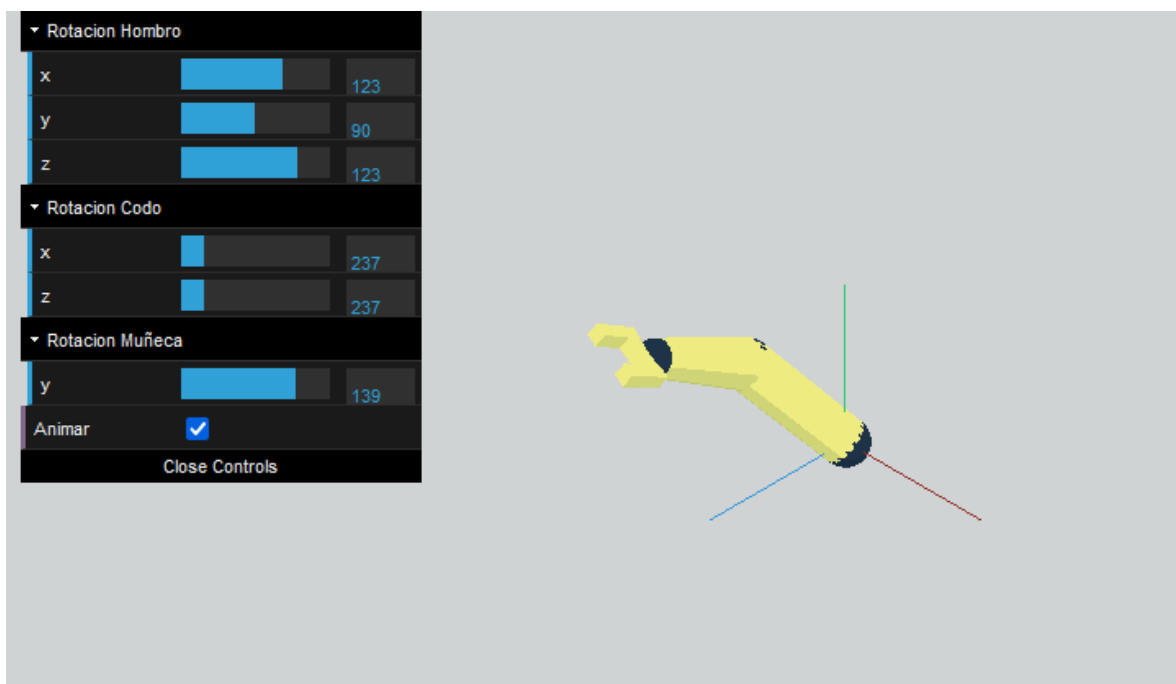


Ilustración 4.3-3: escena grafo

4.3.3 Malla de triángulos

Como ya se ha podido observar, casi todos los objetos de Three.js son mallas, `THREE.Mesh()`, construidas a partir de un tipo concreto de geometría como por ejemplo: `THREE.CylinderGeometry()` o `THREE.SphereGeometry()`; entre otros tantos [25]. Lo malo es que no todos los constructores comparten el mismo número de parámetros y tampoco en el mismo orden. Por lo que para esta escena solo se va a escoger un tipo de malla.

Esfera y wireframe

El tipo de malla escogido fue la esfera, porque de base su configuración es de las más sencillas y se puede apreciar mejor las divisiones en altura y anchura, de manera similar a como se ve en las prácticas de IGV.

Antes de empezar con la instanciación de la malla, se creó un objeto vacío, sin geometría ni material, que sirve como nodo raíz para aplicar rotaciones, como se vió anteriormente en el grafo de escena.

```
const pivot = new THREE.Object3D();
scene.add(pivot);
```

También se creó una función que permite instanciar una nueva esfera y añadirla a la escena. Al constructor de la geometría se le pasan los valores especificados a través. El material empleado es `THREE.MeshNormalMaterial()` [26] que para cada punto toma como color el valor de la normal. Una vez definida la malla se crea su wireframe, para el cual se usa un material básico para líneas, y se visualiza con `THREE.LineSegments()`.

```
function generateSphere() {
    pivot.remove(esfera);

    var esferaGeo = new THREE.SphereGeometry(data.radius, data.widthSegments,
        data.heightSegments, data.phiStart, data.phiLength, data.thetaStart,
        data.thetaLength);

    var esferaMat = new THREE.MeshNormalMaterial();
    esferaMat.side = THREE.DoubleSide;

    esfera = new THREE.Mesh( esferaGeo, esferaMat );

    var geo = new THREE.WireframeGeometry( esfera.geometry );
    var mat = new THREE.LineBasicMaterial( { color: 0xffffff } );
    var wireframe = new THREE.LineSegments( geo, mat );
    esfera.add( wireframe );

    pivot.add(esfera);
}
generateSphere();
```

GUI

Para visualizar una sección de la malla se emplean los ángulos Phi y Theta, que son propiedades de la geometría que ofrece Three.js, y se crean las funciones para poder modificarlas.

```
const ángulos = {phi: 0, theta: 0}

function updatePhi(){
  data.phiLength = degToRad(angulos.phi);
  generateSphere();
}
function updateTheta(){
  data.thetaLength = degToRad(angulos.theta/2.0);
  generateSphere();
}
```

Y finalmente se crearon los controles de la interfaz, al igual que en las otras escenas, de tal manera que cada vez que se modifique un parámetro, se genere una nueva esfera.

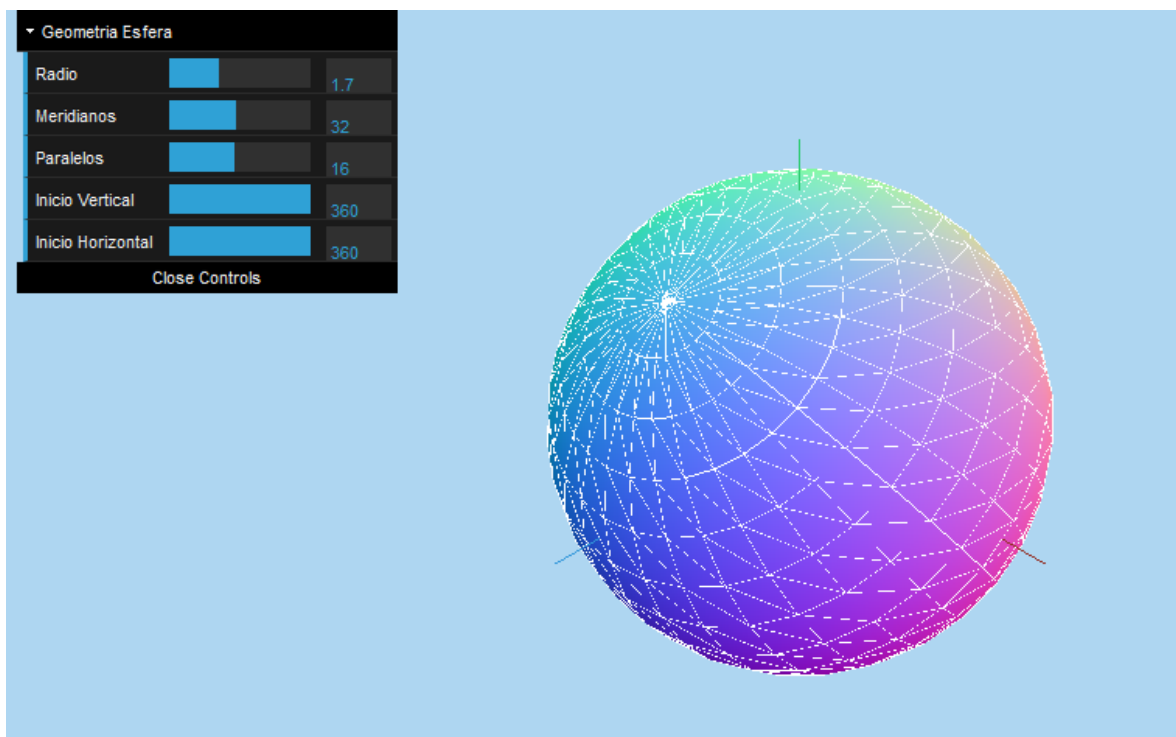


Ilustración 4.3-4: escena grafo

4.4 Tercera iteración

En esta iteración las tareas realizadas fueron: J, K, L y M, que corresponden con las HUs: 9, 10, 11 y 12. Estas consisten principalmente en desarrollar las escenas donde se apliquen texturas y se modifiquen sus propiedades, también el uso de materiales y los tipos de reflexión de la luz. Por otro lado se implementa Vite para poder lanzar un servidor local, de esta manera se puede cargar y visualizar modelos 3D más complejos.

4.4.1 Aplicación de texturas

En esta escena se estudian los principales problemas que surgen a la hora de aplicar texturas, como la minificación y la magnificación, y las técnicas más empleadas para solventarlos, como mipmapping e interpolaciones. Para ello se crearon dos planos con las mismas dimensiones y a cada uno se le aplicó unas texturas pensadas para que surjan estos efectos.

```
const planoGeo = new THREE.PlaneGeometry(10,10);

var textura1 = new THREE.TextureLoader().load('./src/assets/img/lineas.png');

var mat = new THREE.MeshBasicMaterial( {
  map: textura1
} );

var plano = new THREE.Mesh(planoGeo, mat);
scene.add(plano);
```

Para poder visualizarlo correctamente se añadió una luz ambiental (no son necesarias luces de otro tipo) y se puso la cámara sobre el eje Z a una distancia a la que se pudiera ver el plano al completo, por ejemplo en (0,0,8).

```
//===== Cámara =====
const camera = new THREE.PerspectiveCamera( 75, window.innerWidth /
window.innerHeight, 0.1, 100);
camera.position.z = 8;
camera.lookAt(0,0,0);
scene.add(camera);
```

Controles

Por otro lado, se creó una instancia de OrbitControls [27] para poder hacer zoom con la rueda del ratón y rotarla alrededor del punto objetivo. De esta manera, disminuyendo el zoom se fuerza la aparición de minificación, y aumentándolo, se aprecia el efecto de magnificación.

```
//===== Controls =====  
const controls = new OrbitControls( camera, renderer.domElement );  
controls.target.set(0, 0, 0);  
controls.update();
```

Propiedades de texturas

Por último, se configuró el menú de opciones: alternar entre las dos texturas, aplicar interpolación lineal o no cuando surge el problema de magnificación, y el tipo de mipmapping cuando aparece la minificación.

- **minFilter.** Define la función de minimización de la textura que se utiliza cada vez que el píxel, que se está texturizando, se le asigna a un área mayor que un elemento de textura (texel).
 - **NearestMipMapNearestFilter.** Elige el mipmap que más se aproxima al tamaño del píxel que se está texturizando y utiliza el criterio NearestFilter (el elemento de textura más cercano al centro del píxel) para producir la textura.
 - **NearestMipmapLinearFilter.** Elige los dos mipmaps que más se acercan al tamaño del píxel que se está texturizando y utiliza el criterio NearestFilter para producir la textura de cada mipmap. El valor de textura final es un promedio ponderado de esos dos valores.
 - **LinearMipmapNearestFilter.** Elige el mipmap que más se aproxima al tamaño del píxel que se está texturizando y utiliza el criterio LinearFilter (un promedio ponderado de los cuatro elementos de textura más cercanos al centro del píxel) para producir un valor de

textura.

- LinearMipmapLinearFilter. Es el predeterminado y elige los dos mipmaps que más se acercan al tamaño del píxel que se está texturizando y utiliza el criterio LinearFilter para producir un valor de textura de cada mipmap. El valor de textura final es un promedio ponderado de esos dos valores.
- magFilter. Define la función de magnificación de textura que se utilizará cuando el píxel que se está texturizando se asigna a un área menor o igual a un texel.
 - NearestFilter. Devuelve el valor del elemento de textura más cercano (en la distancia de Manhattan) a las coordenadas de textura especificadas.
 - LinearFilter. Devuelve el promedio ponderado de los cuatro texels que están más cerca de las coordenadas de textura especificadas.

```
const data = {seleccion: 1}

const options = {
  minFilters: {
    NearestFilter: THREE.NearestFilter,
    NearestMipMapLinearFilter: THREE.NearestMipMapLinearFilter,
    NearestMipMapNearestFilter: THREE.NearestMipMapNearestFilter,
    LinearFilter: THREE.LinearFilter,
    LinearMipMapLinearFilter: THREE.LinearMipMapLinearFilter,
    LinearMipmapNearestFilter: THREE.LinearMipmapNearestFilter,
  },

  magFilters: {
    NearestFilter: THREE.NearestFilter,
    LinearFilter: THREE.LinearFilter,
  },

  imagen:{
    opcion1: 1,
    opcion2: 2,
  },
};
```

GUI

También fue necesario crear funciones para alternar entre texturas y actualizar sus propiedades: `updateMinFilter()`, para cambiar el tipo de función de minimización; `updateMagFilter()`, para cambiar la función de magnificación y `changeTexture()` para cambiar la textura.

```
function updateMinFilter() {
    if(data.seleccion == 1){
        textura1.minFilter = Number(laTextura.minFilter) ;
        textura1.needsUpdate = true
    }else{
        textura2.minFilter = Number(laTextura.minFilter) ;
        textura2.needsUpdate = true
    }
}

function updateMagFilter() {
    if(data.seleccion == 1){
        textura1.magFilter = Number(laTextura.magFilter);
        textura1.needsUpdate = true
    }else{
        textura2.magFilter = Number(laTextura.magFilter) ;
        textura2.needsUpdate = true
    }
}

function changeTexture(){
    //console.log(data.seleccion);
    if(data.seleccion == 2){
        scene.remove(plano);
        scene.add(plano2);
    }else{
        scene.remove(plano2);
        scene.add(plano);
    }
}
```

Si carga el archivo desde el navegador, deberá aparecer una escena parecida a la siguiente.

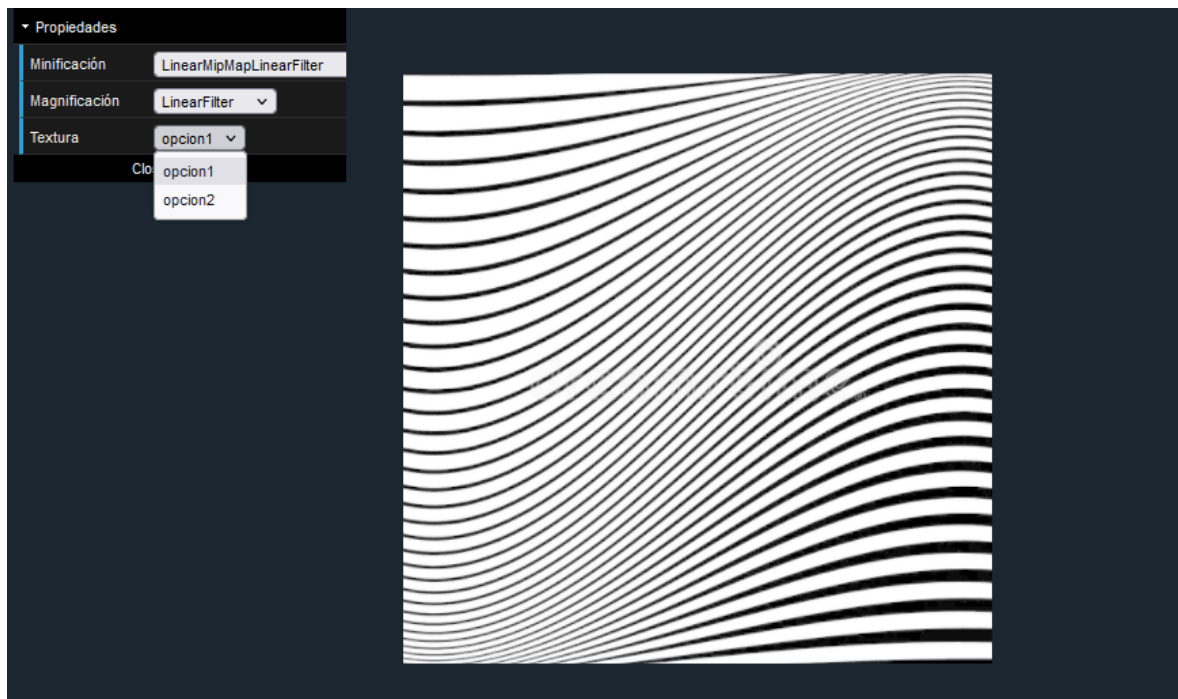


Ilustración 4.4-1: escena textura

4.4.2 Bump Mapping

El objetivo de esta escena es mostrar el efecto de aplicar un mapa de alturas o un mapa de normales a una textura, para simular algún tipo de rugosidad sobre una superficie y que parezca más realista. En este caso se emplea un mapa de alturas, que es una versión en blanco y negro de un mapa de color, donde los pixeles negros toman el valor 0 y los blancos el 1. Y esto es porque Three.js usa por defecto este tipo de mapas para el bump mapping, se pueden incorporar usando StandardMaterial [28].

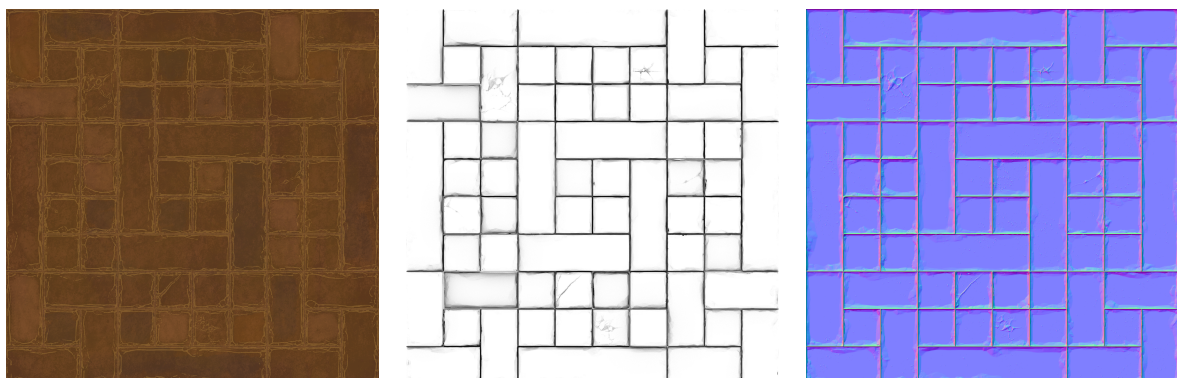


Ilustración 4.4-2: Mapa de color, mapa de altura y mapa de normales (de izquierda a derecha)

Estos mapas no dejan de ser archivos de imagen en formato png que se pueden obtener de diferentes formas, una puede ser accediendo a páginas web especializadas en ofrecer assets para modelado en 3D, como puede ser Unity Asset Store [29], o a través de herramientas como NormalMap-Online [30].

Iluminación y cámara

Una vez obtenidas las texturas, se procedió con la implementación de la escena, donde uno de los elementos principales es una fuente de luz direccional, ya que el uso de estos mapas hace que se alteren los cálculos de iluminación en la superficie del modelo. A parte se puede añadir luz ambiental, con poca intensidad, para atenuar las sombras.

```
//===== Luces =====  
const color = 0xFFFFFF;  
const intensity = 1;  
  
const ambientLight = new THREE.AmbientLight(color, intensity/2);  
scene.add(ambientLight);  
  
const light = new THREE.DirectionalLight(color, intensity);  
light.position.set(5, 5, 5);  
light.target.position.set(0, 0, 0);  
light.castShadow = true;  
  
scene.add(light.target);  
scene.add(light);
```

En cuanto a la cámara se dispuso de manera que ofreciese una perspectiva isométrica, no muy alejada del modelo. Y también se incluyó los controles orbitales como se hizo anteriormente.

```
//===== Cámara =====  
const camera = new THREE.PerspectiveCamera( 75, window.innerWidth /  
window.innerHeight, 0.1, 100);  
  
camera.position.x = 3;  
camera.position.y = 3;  
camera.position.z = 3;
```



```
camera.lookAt(0,0,0);
scene.add(camera);
```

Materiales

Se crearon los materiales con los mapas de color y de alturas, uno simulando una pared de ladrillos y otro una pared rocosa.

```
//----- Materiales -----
const loader = new THREE.TextureLoader();

const textura1 = loader.load('./src/assets/img/bricks/P4_BaseColor.png');
const bump1 = loader.load('./src/assets/img/bricks/P4_Ao.png');
var material1 = new THREE.MeshStandardMaterial({
    map: textura1,
    bumpMap: bump1
});

const textura2 = loader.load('./src/assets/img/bricks/rocks.png');
const bump2 = loader.load('./src/assets/img/bricks/rock_bump2.png');
var material2 = new THREE.MeshStandardMaterial({
    map: textura2,
    bumpMap: bump2
});

var elMaterial = material1;
```

Modelos

Posteriormente se definieron dos modelos, un cubo y una esfera, aplicándoles uno de los dos materiales, el de los ladrillos. Inicialmente solo se añade uno de los dos modelos a la escena, para cambiarlo desde la interfaz.

```
//----- Cubo -----
const cubeGeo = new THREE.BoxGeometry(2.5, 2.5, 2.5);
var cubo = new THREE.Mesh(cubeGeo, material1);
cubo.castShadow = true;
cubo.receiveShadow = true;

scene.add(cubo);

//----- Esfera -----
const sphereGeo = new THREE.SphereGeometry(1.5, 30, 30);
var esfera = new THREE.Mesh(sphereGeo, material1);
```

```
esfera.castShadow = true;
esfera.receiveShadow = true;
```

GUI

Al igual que en las otras escenas, se crearon unas variables que almacenan las posibles opciones elegibles desde la interfaz. Y las funciones que permiten cambiar de modelos, cambiar las texturas y actualizar el factor del bump map aplicado en cada momento.

```
function actualizarBump(){
    if(select.text == 1){
        material1.bumpScale = elMaterial.bumpScale;
        material1.needsUpdate = true
    }else{
        material2.bumpScale = elMaterial.bumpScale;
        material2.needsUpdate = true
    }
}

function cambiarModelo(){
    if(select.model == 1){
        scene.remove(cubo);
        scene.add(esfera);
    }else if(select.model == 0){
        scene.remove(esfera);
        scene.add(cubo);
    }
}

function cambiarTextura(){
    if(select.text == 1){
        esfera.material = material1;
        cubo.material = material1;
    }else{
        esfera.material = material2;
        cubo.material = material2;
    }
    actualizarBump();
}
```

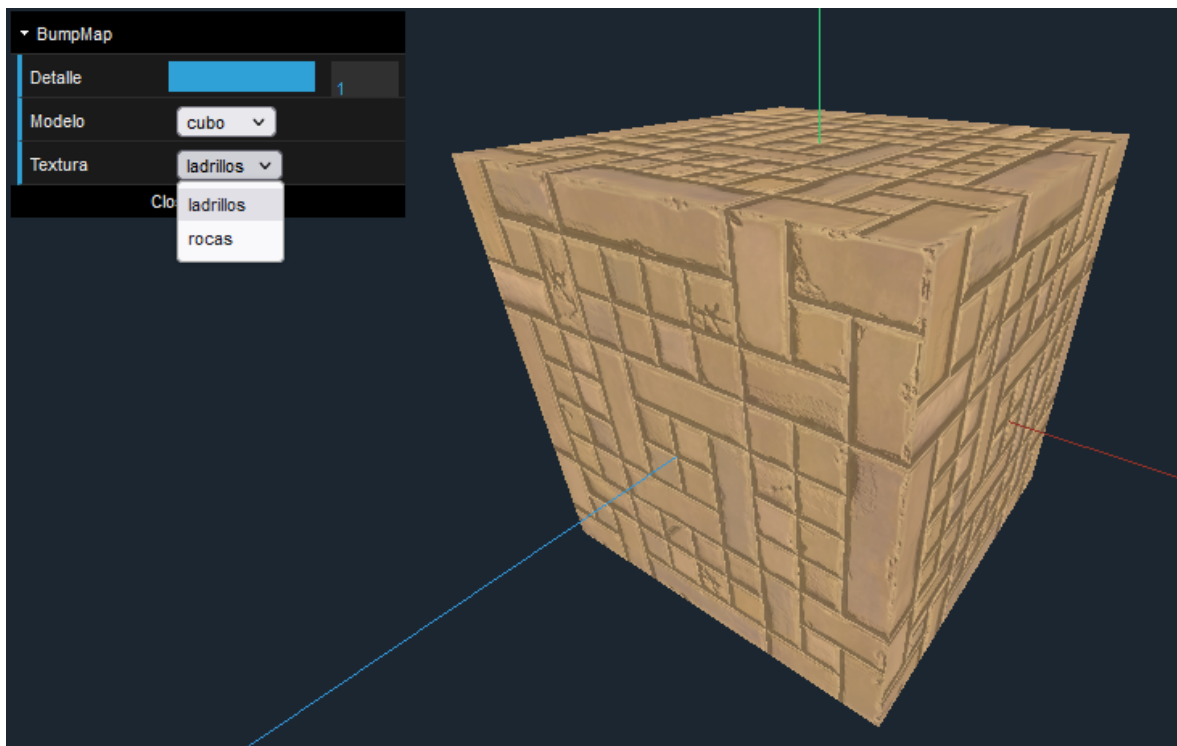


Ilustración 4.4-3: escena bump mapping

4.4.3 Modelos de reflexión

En esta última escena se estudian propiedades de los materiales como la especularidad y la rugosidad, para ver cómo se producen los reflejos, si los hay, y cuál es el color de ese reflejo, que es el color que percibimos. Al igual que en la escena anterior, se hace uso de `StandardMaterial` [28] pero de dos maneras distintas: una es modificando las propiedades del material directamente, y otra aplicando *maps* que aportan la información necesaria para generar los brillos.

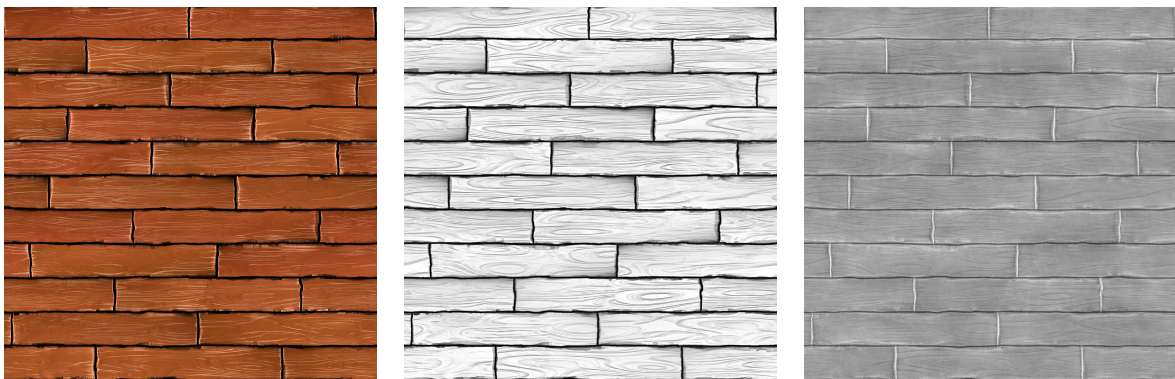


Ilustración 4.4-4: Mapa de color, mapa especular y mapa difuso (de izquierda a derecha)

Iluminación

El primer paso, como hasta ahora, fue configurar la iluminación de la escena. Como se trata de generar brillos sobre la superficie del modelo lo mejor es emplear una luz direccional y opcionalmente una luz ambiental para atenuar las sombras.

Modelo

Como modelo se utiliza TorusKnotGeometry [31] una especie de toro retorcido, parecido a un nudo hecho con una goma elástica, ya que por su forma se generan brillos en varias zonas de su superficie. Se iniciaron sus parámetros de la siguiente manera.

```
const model = {  
  radius: 3,  
  tubeRadius: 1,  
  tubularSegments: 75,  
  radialSegments: 50,  
  p: 2,  
  q: 3  
};
```

Luego, se creó el material que es configurable a través de la interfaz, por defecto simula metal, ya que el color es gris con el factor de rugosidad a la mitad y el brillo al máximo.

```
var mat = new THREE.MeshStandardMaterial({  
  color: 0xbcbcbc,  
  emissive: 0x000000,  
  roughness: 0.5,  
  metalness: 1.0  
});
```

Por otro lado, se instanció el material creado a partir de las texturas mostradas al principio, simulando un suelo de madera. Para cada map hay que especificar la ruta del archivo y configurar las siguientes propiedades:

- wrapS y wrapT. Esto define cómo la textura “envuelve” al modelo horizontal y

verticalmente. Es posible que la textura no cubra una cara al completo, por lo que existen técnicas para rellenar el espacio sobrante:

- ClampToEdgeWrapping, donde el borde se “estira”.
 - RepeatWrapping, la imagen de la textura se repite.
 - MirroredRepeatWrapping la textura se repite reflejándose en cada repetición.
- offset. Cuánto se desplaza una sola repetición de la textura desde el principio, en cada dirección U y V.
 - repeat. Cuántas veces se repite la textura a lo largo de la superficie, en cada dirección U y V, según los valores de wrapS y wrapT

```
const textureLoader = new THREE.TextureLoader();
const baseColor = textureLoader.load('./src/assets/img/Wood/wood_basecolor.png',
  function( texture ) {
    texture.wrapS = texture.wrapT = THREE.RepeatWrapping;
    texture.offset.set( 0, 0 );
    texture.repeat.set( 25, 3 );
  } );

// Resto de maps

const woodTexture = new THREE.MeshStandardMaterial({
  map: baseColor,
  aoMap: especular,
  bumpMap: altura,
  bumpScale: 0.15,
  roughnessMap: rugoso,
});
```

Y para terminar con la creación del modelo, se creó una función que genera el modelo con los parámetros configurados como hasta ahora, de tal manera que al modificar uno de ellos, desde la interfaz, se llama a la función que genera el toro para así actualizar la vista.

```
var slipknot;
var geometry;
const maderita = { bool:false };
function generateTorus(){
  pivot.remove(slipknot);
```

```
geometry = new THREE.TorusKnotGeometry(model.radius, model.tubeRadius,  
    model.tubularSegments, model.radialSegments, model.p, model.q);  
  
if(madera.bool)  
    slipknot = new THREE.Mesh(geometry, woodTexture);  
else  
    slipknot = new THREE.Mesh(geometry, mat);  
  
slipknot.castShadow = true;  
slipknot.receiveShadow = true;  
  
pivot.add(slipknot);  
}  
generateTorus();
```

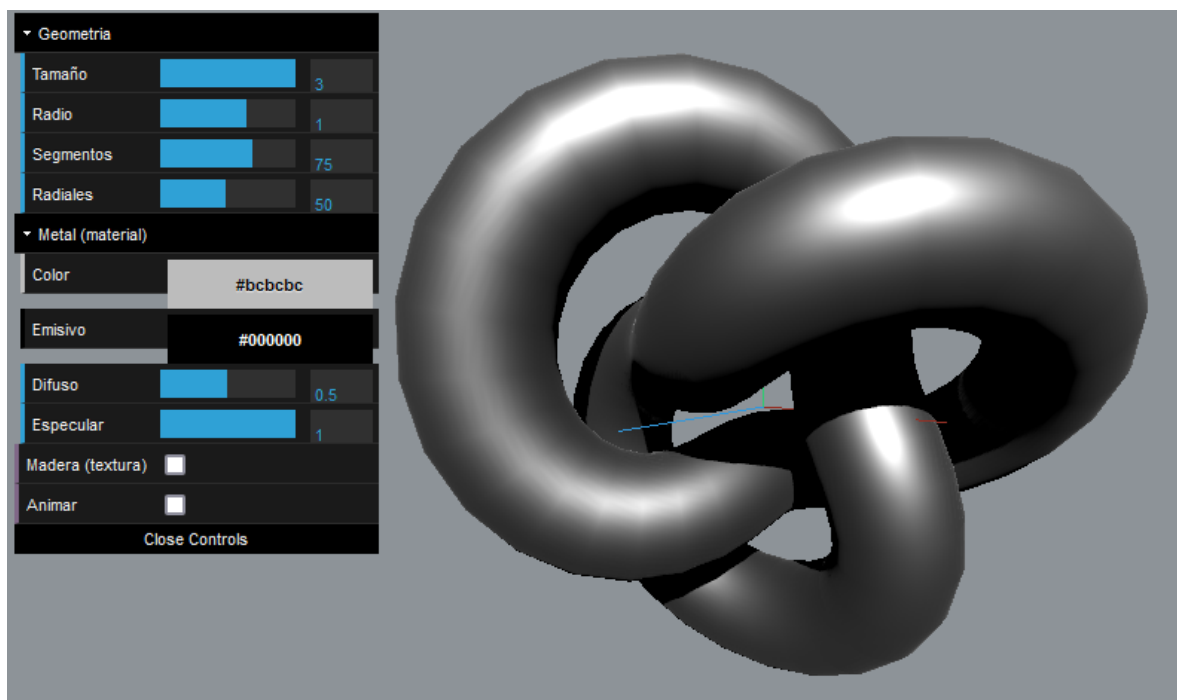


Ilustración 4.4-5: escena modelos reflexión

4.4.4 Decoración

Hasta ahora las escenas solo incluyen modelos básicos como esferas y cubos, por lo que es de interés cargar modelos 3D en formato FBX u OBJ y aprovechar la potencia de Three.js para hacer una página web más atractiva. Para poder cargar modelos de este tipo es necesario arrancar la aplicación desde un

servidor local, ver manual de instalación.

Actualmente existen diversas páginas web que ofrecen modelos completamente gratuitos, los sitios que se han consultado han sido Poly Pizza [32] y Sketchfab [33]. Y una vez descargados los archivos necesarios, hay que guardarlos dentro del proyecto para que sean fácilmente accesibles.

FBX

En la escena de la luz ambiental se ha incluido un modelo FBX de una ballena que trae una animación por defecto. Para cargar modelos de este tipo se hace uso de la clase `FBXLoader`, instanciando un objeto de este tipo se puede acceder al método `load()` al cual hay que especificarle la ruta donde se encuentra el modelo y una función donde se configure las animaciones (si las tiene) y aplicar transformaciones.

Para las animaciones es tan sencillo como crear un objeto del tipo `AnimationMixer` donde se le pasa el modelo, y con el método `clipAction()` se puede acceder a las animaciones, como en este caso solo dispone de una, el índice es el 0.

```
import { OrbitControls } from 'three/addons/controls/OrbitControls.js';
import { FBXLoader } from 'three/addons/loaders/FBXLoader.js';

let mixer;
const loader = new FBXLoader();
loader.load( './src/assets/Models/Whale/Whale.fbx', function ( object ) {

    mixer = new THREE.AnimationMixer( object );

    const action = mixer.clipAction( object.animations[ 0 ] );
    action.play();

    scene.add( object );
    object.scale.x = 0.005;
    object.scale.y = 0.005;
    object.scale.z = 0.005;
} )
```

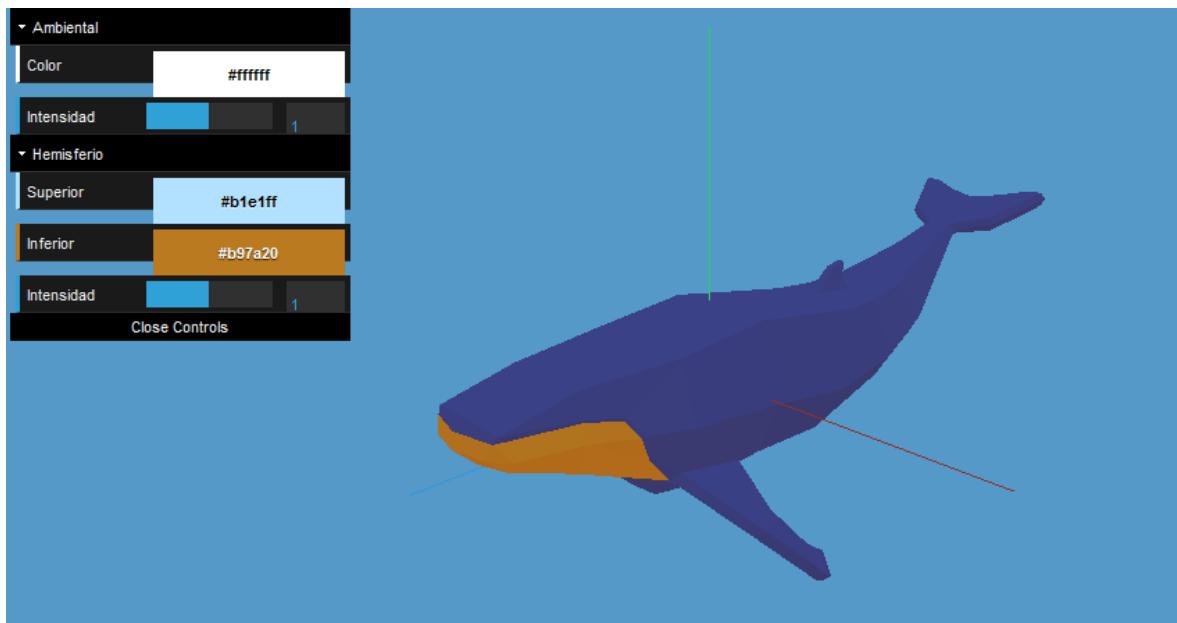


Ilustración 4.4-6: modelo fbx animado

OBJ

Para cargar objetos de este tipo se hace de un modo parecido, pero es necesario precargar también el archivo MTL para aplicar las texturas. Por lo que se necesitan objetos de `MTLLoader` y `OBJLoader`. Luego, una vez cargado el modelo, hay que detectar qué componentes son mallas para “activar” las sombras.

```
import { OBJLoader } from 'three/addons/loaders/OBJLoader.js';
import { MTLLoader } from 'three/addons/loaders/MTLLoader.js';
```

```
const mtlLoader = new MTLLoader();
const objLoader = new OBJLoader();

mtlLoader.load('./src/assets/Models/Tree Assets/materials.mtl', (mtl) => {
  mtl.preload();
  objLoader.setMaterials(mtl);

  objLoader.load('./src/assets/Models/Tree Assets/model.obj', (root) => {
    scene.add(root);

    root.traverse(function(node){
      if(node.isMesh){
        node.castShadow = true;
      }
    });
  });
});
```



```
node.receiveShadow = true;
    }
  });

  root.scale.x = 5;
  root.scale.y = 5;
  root.scale.z = 5;
});
});
```

En la imagen 4.4-6 se muestra como queda la escena de la luz direccional tras aplicar estos cambios. Y de manera análoga, en otras escenas: luz puntual, luz focal y grafo de escena; se ha aplicado este mismo método, pero con otros modelos.



Ilustración 4.4-6: modelo obj

4.5 Cuarta iteración

En esta última iteración se realizan las tareas N y O, que consisten en crear la aplicación utilizando Vue.js. Más adelante se explica cómo crear la base para un proyecto de este tipo, y cómo se estructuran los componentes y las vistas empleando Vue Router.

4.5.1 Creación proyecto

Componentes

Una vez se tenga la configuración básica del proyecto se puede eliminar tanto las vistas y los componentes que ya están creados. Porque es conveniente crear unos nuevos desde cero. El primero de ellos fue el canvas para visualizar las escenas. Ya que los componentes son reutilizables y se pueden insertar en todas las vistas que se necesite, y el canvas al ser un elemento común a todas las vistas, al tenerlo como componente no es necesario escribir el mismo código en cada una de ellas, basta con inyectarlo cuando se crean. Para ello se crea un archivo, desde el inspector, llamado “Canvas.vue” con el siguiente contenido.

```
<template>
  <div class="p-4 p-md-5 mb-4">
    <div id="lienzo" class="px-0 col-md-6">
      <div id="gui"></div>
      <canvas id="mi_canvas" class="rounded"></canvas>
    </div>
  </div>
</template>

<style>
  @import '../assets/canvas.css';
</style>
```

Dentro de la etiqueta <template> se definen los elementos html que tendrá el componente, y dentro de <style> se puede escribir en formato css los estilos de dichos elementos, o como en este caso, importar una hoja de estilos.

Para las escenas se crearon componentes siguiendo este formato:

```
<script>
import * as THREE from 'three';
import { GUI } from 'dat.gui'

export default {

  name: '[Nombre de la escena]',

  methods: {
    main:function(){
      // Código de La escena
    }
  },

  mounted() { this.main(); }
}
</script>

<template>
// Textos e imágenes
</template>
```

Esto es porque los componentes de Vue incluyen métodos establecidos por defecto, denominados *hooks* [34], que se ejecutan según su ciclo de vida. Por otro lado, se tienen los métodos que se van a llamar en cada una de esas fases. El *hook* empleado es `mounted()`, que llama a la función `main()` cuando se crea la vista. La función `main()` encapsula el código de la escena.

Vistas

Las vistas no dejan de ser un componente más grande que se inyecta en la página principal. Están formadas a partir de los componentes que se han creado previamente, es decir, a partir del canvas y del script de la escena. La ventaja de esto es que no todas las vistas tienen por qué tener los mismos componentes, y tampoco tienen que estar dispuestos de la misma manera. En este caso al tener solo dos componentes no es algo que se tenga en cuenta.

Para usar los componentes es tan sencillo como importarlos en la vista, darles un nombre, que va a ser el que se use para declarar las etiquetas de html, e incluirlos dentro del *template*. El siguiente código corresponde a la vista de transformaciones geométricas, y todas siguen el mismo patrón.

```
<script setup>
import Canvas from '../components/Canvas.vue'
import Escena1 from '../components/transformaciones2_escena.vue'
</script>

<template>
  <Canvas />
  <Escena1 />
</template>
```

4.5.2 Vue Router

Al integrar Vue Router durante la creación del proyecto, se crea automáticamente un script en “index.js” en la carpeta router. Al igual que en los scripts de las vistas se importaron los componentes, ahora se importan las vistas al completo y se añaden dentro de las rutas del Router. Este tipo de rutas tienen 3 atributos: un nombre, el path a usar en el navegador y el componente a inyectar, en este caso serán las vistas. La ventaja que ofrece es que se pueden personalizar los paths y no tienen porqué coincidir con el nombre del componente. A continuación se muestra, de manera resumida, cómo queda la configuración del Router.

```
import { createRouter, createWebHistory } from 'vue-router'

import EsceneView1 from '../views/transformaciones_vista.vue'
// Resto de imports

const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),
  routes: [
    {
      path: '/',
      name: 'transformaciones',
      component: EsceneView1
    },
    // Resto de direcciones
  ]
})

export default router
```

Menú navegable

Para implementar el menú navegable se escogió un ejemplo de Bootstrap [35], y se modificó para que incluyera los títulos de todas las secciones. En cuanto a los estilos, se importaron a través de un link dentro de la cabecera del archivo “index.html”.

Esta barra navegable se incluyó en el componente principal “App.vue”. Para emplear los enlaces de Vue Router no hay que usar la clásica etiqueta <a> sino que se debe usar <RouterLink>, por ejemplo:

```
<RouterLink class="dropdown-item" to="/luzAmbiental">Ambiental</RouterLink>
```

Y para que pueda inyectar las vistas, es con la etiqueta <RouterView /> dentro del contenido principal.

```
<main class="container">
  <RouterView />
</main>
```

El resultado final se puede comprobar en la siguiente imagen.

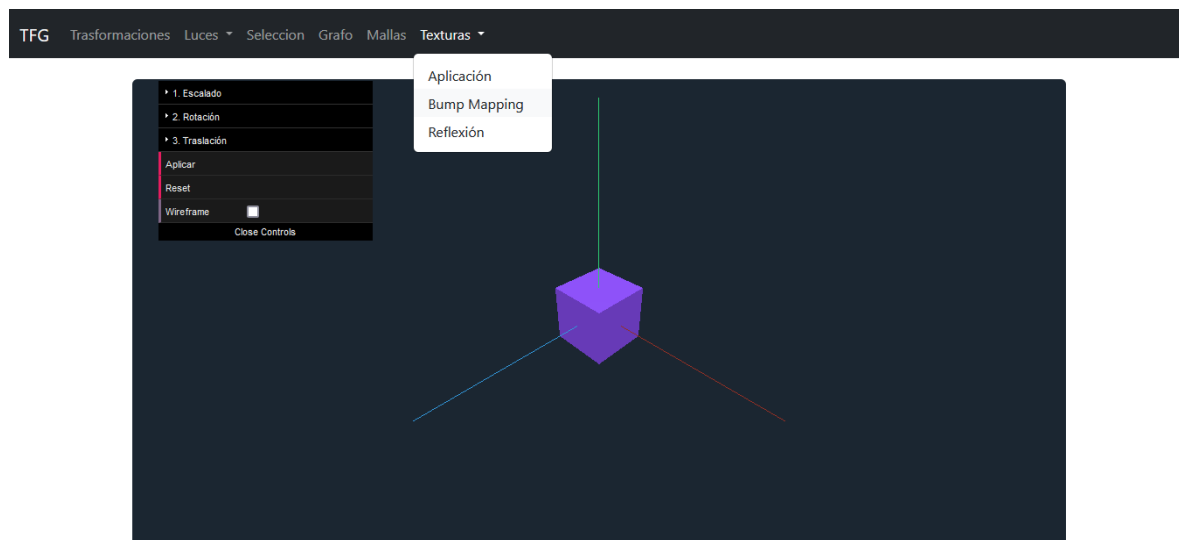


ilustración 4.5-1: vista de la aplicación en pruebas

5. Pruebas

La realización de pruebas es una fase fundamental, estas buscan cumplimentar dos criterios importantes que son la validación y la verificación. La validación responde a la pregunta de si el software en desarrollo es correcto, o dicho de otra manera, si el software se ajusta a la especificación. Y por otro lado, la verificación resuelve la cuestión de si el software desarrollado es el producto que el cliente espera. También cabe mencionar que los principios básicos de las pruebas son los siguientes.

1. Las pruebas muestran la presencia de defectos, no su ausencia. El software que se está probando siempre tendrá un defecto así haya pasado por todas las pruebas de software. No hay ningún software completamente libre de defectos.
2. Las pruebas exhaustivas son imposibles. No es posible probar todas las funcionalidades con todas las combinaciones válidas en calidad de datos de entrada durante la prueba real.
3. Las pruebas tempranas ahorran tiempo y dinero. Es mucho más económico reparar un defecto en las primeras etapas de prueba.
4. Los defectos tienden a agruparse. Puede suceder que la mayoría de los defectos encontrados estén relacionados con una pequeña cantidad de módulos.
5. Las pruebas dependen del contexto. Las pruebas no serán las mismas para un software con finalidades médicas que otro con finalidad recreativa.
6. La ausencia de errores es una falacia. La ausencia de nuevos errores detectados no implica necesariamente que el software sea útil, sino que la utilidad del mismo es indicada por la satisfacción de las expectativas del cliente que el producto pueda brindar.

En el caso de este proyecto, como ha sido llevado a cabo por una persona, el mismo programador ha hecho de tester, por lo que las pruebas se fueron llevando a cabo durante los sprints y al final de los mismos. Las pruebas han sido principalmente llevadas a cabo sobre los requisitos funcionales descritos al principio, en el apartado 2.3 .

El uso de la interfaz dat.GUI [19] no solo ha sido útil para el desarrollo, sino también para poder realizar pruebas de valores límite fácil y rápidamente. Ya que, de la manera en la que se ha implementado, introduciendo o cambiando un parámetro de la interfaz se puede comprobar instantáneamente su efecto sobre la escena, como por ejemplo la intensidad de las luces, los ángulos de rotación del grafo... etc.

A continuación se muestran los errores detectados en algunas de las escenas, y su correspondiente solución.

- Transformaciones. Inicialmente las interacciones se implementaron de tal manera que si se cambiaba el valor de algún parámetro, la escena cambiaba instantáneamente. Tras aplicar numerosas transformaciones sobre el modelo, se observó que el resultado no era el correcto. Por ejemplo, si se quería introducir las siguientes operaciones:

$$M = R_2 \cdot T \cdot R_1$$

Es decir, primero una rotación, luego una traslación y una segunda rotación.

El resultado producido era el siguiente:

$$M = T \cdot R_2 \cdot R_1$$

Three.js internamente reordena las operaciones. Por esto mismo se optó por el modelo final, en el que el usuario introduce una transformación, o varias en orden T·R·S, y pulsando un botón se aplica sobre la matriz actual. Esto es gracias al cálculo de matrices comentado anteriormente.

- Luz direccional. Al cambiar la posición de la luz, cambiaba también su dirección, es decir, no apuntaba al origen de coordenadas donde está situado el modelo, por lo que se dejaban de ver los efectos de la iluminación y las sombras proyectadas. Por lo que se optó por rotar la luz respecto a los ejes X y Z globales, en lugar de aplicar traslaciones. La interfaz no sufrió cambios,

tan solo las operaciones llamadas al realizar cada acción.

Y para establecer los valores que puede introducir el usuario son dentro del conjunto $[0, 180]$, ya que fuera de este intervalo son ángulos que hacen que la luz se coloque por debajo del modelo, lo cual no es realista ya que se pretende representar los rayos del sol sobre el modelo de un bosque.

- Foco de luz. En esta escena se detectó que la luz no debía tomar el valor 0 ya que se producía un efecto contrario al deseado, hacía que la escena se iluminase al completo. Y con valores por encima de 40 dejaba de tener efecto. Por lo que el intervalo de valores válidos establecido es $[1, 40]$.

Por otro lado, el ángulo de apertura, siguiendo la teoría se estableció entre 0 y 90, ya que los valores negativos no son posibles, y por encima de 90 no se obtienen efectos realistas.

- Selección de objetos. La primera versión se hizo con la técnica de selección por color. Para emplear este método con Three.js se crean dos escenas, en ambas se disponen los mismos objetos en las mismas posiciones. En una de ellas no se ponen luces y los materiales se asignan de manera única y de tipo emisoro, para que no les afecte la iluminación. Esta escena no se renderiza, solo sirve de apoyo para detectar el objeto seleccionado.

Cuando se implementó por primera vez, el resultado fue exitoso, pero al integrar Vue.js, daba muchos fallos. Y al no detectar la causa y encontrar una solución, se optó por aplicar la técnica de intersección de rayos, que se incluye en la versión final.

- Grafo de escena. Inicialmente el rango de ángulos para cada punto articulable era $[0, 180]$. Haciendo numerosas combinaciones en el GUI se pudo observar varios movimientos que se consideraban como ilegales, ya que algunas partes atravesaban a otras. Por lo cual se estableció el rango para el primer punto $[45, 145]$ en el eje Z. Y para el segundo punto se cambió a $[220, 330]$ para que no diera la impresión de que el antebrazo atravesase al resto del modelo.

6. Conclusiones

Llegado hasta este punto se puede decir que el desarrollo de la aplicación ha sido un éxito, se ha conseguido implementar una SPA empleando Vue.js con el que se gestiona el cambio de vistas en las que se renderizan escenas de WebGL implementadas con Three.js. Con el añadido de su mayor atractivo: ofrecer al usuario la posibilidad de interactuar con modelos en tiempo real. Cumpliendo con la finalidad didáctica que es el motivo principal de este TFG.

Todo ello no solo ha permitido poner en práctica conocimientos relacionados con asignaturas del grado como: Ingeniería de Requisitos, Diseño Software, Desarrollo de Aplicaciones Web, Programación de Aplicaciones Gráficas (PAG)... etc. Sino que también ha servido para adquirir conocimiento en tecnologías con las que no se había experimentado previamente, y sobre todo para aprender a manejar su documentación y aprender por uno mismo.

Por otro lado, hay que reconocer que el contenido de la herramienta se puede ampliar más aún, para ofrecer una aplicación más completa. Por ejemplo, en la escena donde se explican las mallas, se podría incluir una función para seleccionar otro tipo de primitiva como un cilindro o una malla de otro tipo, no solo una esfera. O en la del grafo de escena, explicar las aplicaciones que tiene dentro del campo de la animación e incluir un modelo de un humanoide, de esta manera sería más atractivo para el usuario al ver una aplicación de este tipo.

Por último, cabe destacar que la idea de este TFG puede ser aplicable a más asignaturas, dando como resultado más proyectos como este que sirvan de apoyo o de introducción para presentar las materias. Para orientar a los alumnos y que se animen a cursar ciertas asignaturas.

Bibliografía

- [1] Fundamentos de WebGL, Available: <https://webglfundamentals.org/webgl/lessons/webgl-how-it-works.html> (Último acceso: 24/07/2023)
- [2] Wiki GLSL, Available: [https://www.khronos.org/opengl/wiki/Core_Language_\(GLSL\)](https://www.khronos.org/opengl/wiki/Core_Language_(GLSL)) (Último acceso: 24/07/2023)
- [3] OpenGL Rendering Pipeline, Available: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview (Último acceso: 24/07/2023)
- [4] Three.js, Available: <https://threejs.org/> (Último acceso: 24/07/2023)
- [5] Primitivas de Three.js, Available: <https://threejs.org/manual/#en/primitives> (Último acceso: 24/07/2023)
- [6] Three.js ejemplos, Available: https://threejs.org/examples/#webgl_animation_skinning_blending (Último acceso: 24/07/2023)
- [7] Guia Scrum, Available: <https://scrumguides.org/download.html> (Último acceso: 24/07/2023)
- [8] Babylon.js, Available: <https://www.babylonjs.com/> (Último acceso: 24/07/2023)
- [9] Vue.js, Available: <https://vuejs.org/guide/introduction.html> (Último acceso: 24/07/2023)
- [10] React, Available: <https://es.react.dev/learn> (Último acceso: 24/07/2023)
- [11] Visual Studio Code, Available: <https://code.visualstudio.com/> (Último acceso: 24/07/2023)
- [12] Funcionalidades de WebStorm, Available: <https://www.jetbrains.com/es-es/webstorm/features/#features> (Último acceso: 24/07/2023)
- [13] Node.js, Available: <https://nodejs.org/es> (Último acceso: 24/07/2023)
- [14] IEEE standard glossary of software engineering terminology, Available: http://www.informatik.htw-dresden.de/~hauptman/SEI/IEEE_Standard_Glossary_of_Software_Engineering_Terminology%20.pdf (Último acceso: 24/07/2023)

- [15] Historias de Usuario, Available: https://www.scrummanager.com/files/scrum_manager_historias_usuario.pdf (Último acceso: 25/07/2023)
- [16] INVEST - Scrum Manager BoK, Available: <https://www.scrummanager.com/bok/index.php/INVEST> (Último acceso: 25/07/2023)
- [17] MoSCoW, Available: <https://soka.gitlab.io/blog/post/2019-06-05-tecnicas-metodologias-priorizacion/> (Último acceso: 25/07/2023)
- [18] Indeed, Available: <https://es.indeed.com/career/analista-programador/salaries> (Último acceso: 25/07/2023)
- [19] dat.GUI, Available: <https://github.com/dataarts/dat.gui> (Último acceso: 25/07/2023)
- [20] Patrón MVVM, Available: <https://learn.microsoft.com/es-es/dotnet/architecture/maui/mvvm> (Último acceso: 25/07/2023)
- [21] Patrón MVC, Available: <https://revistatelematica.cujae.edu.cu/index.php/tele/article/view/15/10> (Último acceso: 25/07/2023)
- [22] Three.js - Object3D, Available: <https://threejs.org/docs/index.html?q=obje#api/en/core/Object3D> (Último acceso: 25/07/2023)
- [23] Three.js - Matrix4, Available: <https://threejs.org/docs/index.html?q=matri#api/en/math/Matrix4> (Último acceso: 25/07/2023)
- [24] Three.js - MeshToonMaterial, Available: <https://threejs.org/docs/?q=toon#api/en/materials/MeshToonMaterial> (Último acceso: 25/07/2023)
- [25] Three.js - Primitivas, Available: <https://threejs.org/manual/#en/primitives> (Último acceso: 25/07/2023)
- [26] Three.js - MeshNormalMaterial, Available: <https://threejs.org/docs/#api/en/materials/MeshNormalMaterial> (Último acceso: 25/07/2023)

[27] Three.js - OrbitControls, Available: <https://threejs.org/docs/#examples/en/controls/OrbitControls> (Último acceso: 25/07/2023)

[28] Three.js - MeshStandardMaterial.bumpMap, Available: <https://threejs.org/docs/index.html#api/en/materials/MeshStandardMaterial.bumpMap> (Último acceso: 26/07/2023)

[29] Unity Asset Store, Available: <https://assetstore.unity.com/> (Último acceso: 26/07/2023)

[30] NormalMap-Online, Available: <https://cpetry.github.io/NormalMap-Online/> (Último acceso: 26/07/2023)

[31] Three.js - TorusKnotGeometry, Available: <https://threejs.org/docs/#api/en/geometries/TorusKnotGeometry> (Último acceso: 26/07/2023)

[32] Poly Pizza, Available: <https://poly.pizza/> (Último acceso: 26/07/2023)

[33] Sketchfab, Available: <https://sketchfab.com/> (Último acceso: 26/07/2023)

[34] Vue.js - Composition API: Lifecycle Hooks, Available: <https://vuejs.org/api/composition-api-lifecycle.html> (Último acceso: 26/07/2023)

[35] Bootstrap - Navbar, Available: <https://getbootstrap.com/docs/4.0/components/navbar/> (Último acceso: 26/07/2023)

Ilustraciones

Ilustración 1.3-1: Primitivas de Three.js, Available: <https://threejs.org/manual/#en/primitives>

Ilustración 1.3-2: Ejemplos de Three.js, Available: https://threejs.org/examples/#webgl_animation_skinning_blending

Ilustración 1.4-1: The Scrum Framework Poster, Available: <https://www.scrum.org/resources/scrum-framework-poster>

Apéndice I. Manual de instalación del sistema

La aplicación web está disponible en el siguiente enlace:

<https://tfg-lrdf0001-ujaenes.vercel.app/>

En caso de que se quiera emplear la aplicación de manera local, en los siguientes apartados se muestra lo necesario para su ejecución.

Visual Studio Code

1. Acceder a la página oficial de descargas:

<https://code.visualstudio.com/download>

2. Descargar el instalador compatible con el sistema operativo del equipo. Para este proyecto se ha empleado la versión de Windows 1.81.1.

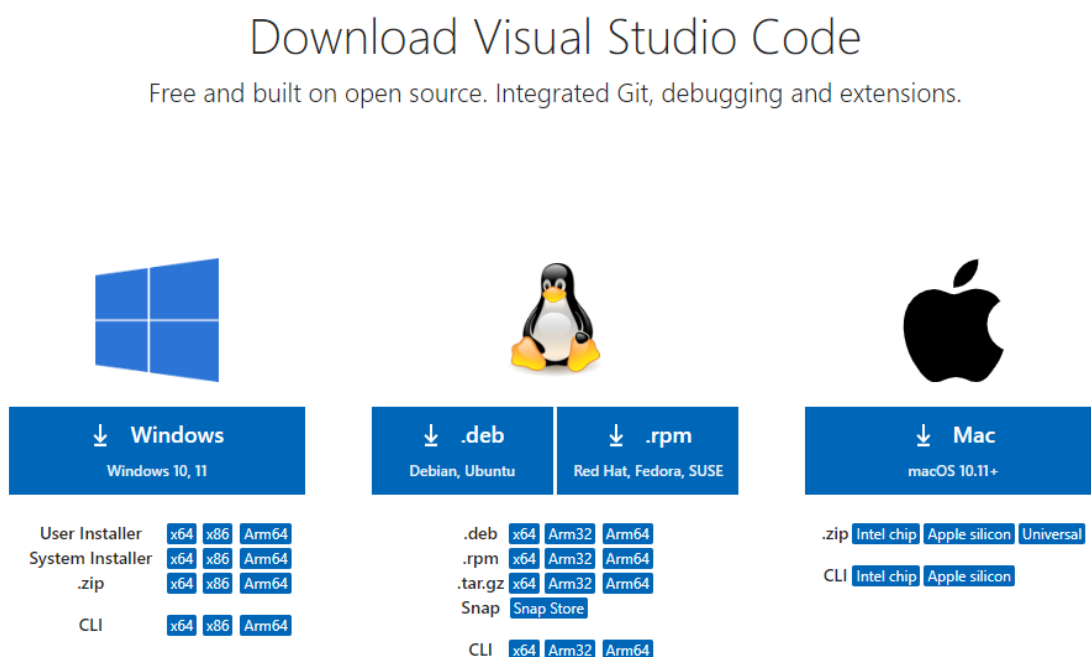


Ilustración AI-1: página de descarga de VS

3. Abrir el instalador desde el explorador de archivos y seguir los pasos que marca el asistente.

Node.js

1. Acceder a la página oficial de descargas: <https://nodejs.org/es>
2. Descargar el instalador, preferiblemente el de la versión de Soporte a Largo Plazo (LTS).



Ilustración AI-2: página de descarga de Node

3. Abrir el instalador desde el explorador de archivos y seguir los pasos que marca el asistente.

Three.js y Vite.js

Una vez se tenga node descargado e instalado, se puede instalar rápidamente las dependencias necesarias escribiendo los siguientes comandos desde la terminal.

1. Desde el explorador de archivos, crear una carpeta donde posteriormente se alojarán los archivos del proyecto.
2. Abrir VS y seleccionar la carpeta creada: File → Open Folder.
3. Una vez abierta la carpeta desde VS, hay que abrir la terminal usando `ctrl+J` o desde el inspector: Terminal → New Terminal.

4. Comprobar que el path del pront corresponde con la ruta de la carpeta del proyecto, ej: PS C:\Users\pc\Desktop\Tutorial\Proyecto>
5. Ejecutar la línea de comando: `npm install --save three`
6. Ejecutar la línea de comando: `npm install --save-dev vite`
7. Y para dat.GUI: `npm install --save dat.gui`
8. Ejecutar el proyecto: `npm run dev`

Hasta este punto, junto al código, ya se dispone de todo lo necesario para lanzar la aplicación de manera local.

Vue.js

Por último se muestra cómo crear un proyecto Vue desde cero, en caso de que sea necesario. Para este proyecto se empleó la versión 5.0.8.

Lo primero que hay que hacer es crear una carpeta, inicialmente vacía, donde se ubicarán todos los archivos: componentes, scripts, assets... etc. Y abrirla usando Visual Studio Code. Luego hay que acceder a la terminal que trae VS y escribir el siguiente comando.

```
npm create vue@latest
```

A continuación, nos ofrece la opción de introducir un nombre para el proyecto o dejarlo por defecto: vue-project. También pedirá permiso para instalar una serie de paquetes, lo único que hay que hacer es seleccionar “sí” o “no” en cada uno. En este caso el único que se instalará será Vue Router.

```
✓ Project name: ... <your-project-name>
✓ Add TypeScript? ... No / Yes
✓ Add JSX Support? ... No / Yes
✓ Add Vue Router for Single Page Application development? ... No / Yes
✓ Add Pinia for state management? ... No / Yes
✓ Add Vitest for Unit testing? ... No / Yes
✓ Add an End-to-End Testing Solution? ... No / Cypress / Playwright
✓ Add ESLint for code quality? ... No / Yes
```

✓ Add **Prettier** for code formatting? ... No / Yes

Luego pide que se ejecuten los siguientes comandos para situarnos en la carpeta del proyecto. Deberá aparecer en el inspector la jerarquía de archivos que aparece en la imagen 4.4-7.

```
> cd <your-project-name>  
> npm install
```

Dentro de la carpeta `src` se incluye todo el material necesario para desarrollar la página web. En `assets` se almacenan los modelos 3D e imágenes; en `components` están los scripts de las escenas; en `router` hay un script que permite configurar los enlaces para las vistas y en `views` se almacenan las vistas generadas a partir de los componentes.

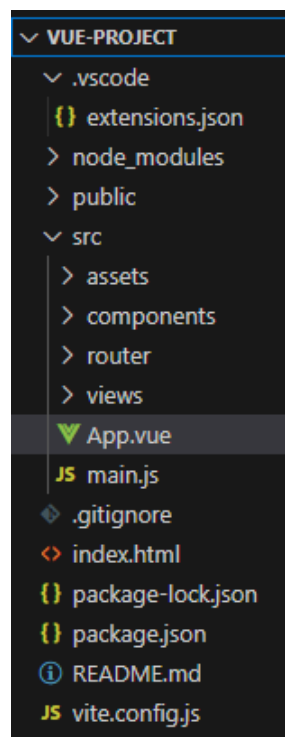


Ilustración A1-3: página de descarga de Node

Por último, se ejecuta el comando para arrancar el servidor local y visualizar la página desde el navegador, inicialmente se muestran unas vistas que vienen creadas por defecto a modo de prueba.


```
> npm run dev
```

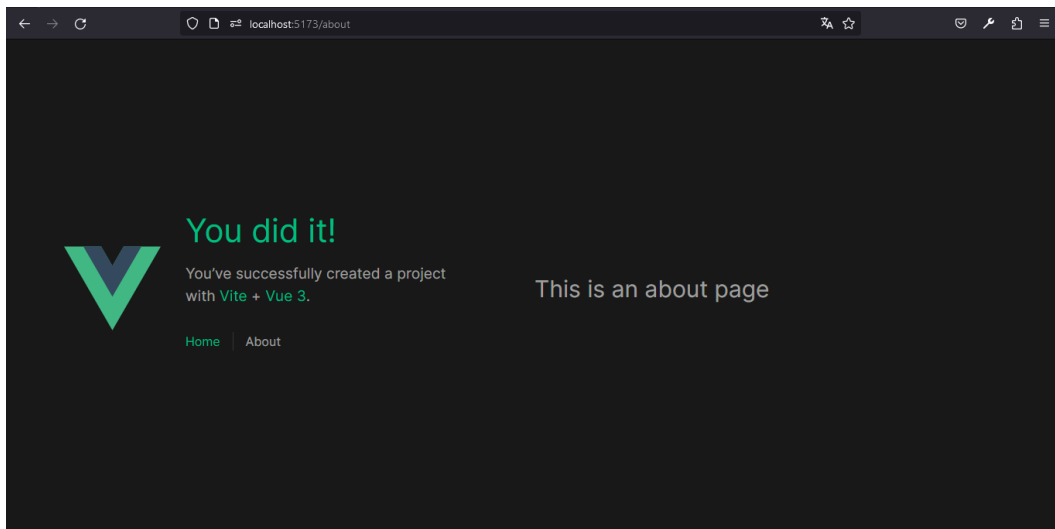


Ilustración A1-4: vista de prueba

GitHub

El enlace al repositorio de GitHub es el siguiente: <https://github.com/lrdf0001/TFG>

Apéndice II. Manual de usuario

En esta sección se explican cómo interactuar con las escenas, cuáles son los parámetros modificables y a qué objetos afectan. Esta sección está dividida en tantos apartados como escenas.

La aplicación web está disponible en el siguiente enlace:
<https://tfg-lrdf0001-ujaenes.vercel.app/>

Transformaciones

Esta escena es la primera escena que se muestra al entrar en la aplicación. Si se despliegan los controles se puede ver los siguientes elementos:



Ilustración AII-1: controles escena 1

Se puede encontrar 3 submenús, que representan las operaciones de transformación escalado, rotación y traslación, más abajo dos botones para aplicar

cambios y restaurar la escena al estado inicial, y una checkbox para visualizar el cubo en modo *wireframe*.

Las transformaciones se pueden aplicar de dos maneras distintas:

- A. Se introducen valores en los 3 submenús y luego pulsar el botón “aplicar”, de esta manera se aplican las transformaciones en orden clásico: $M = T \cdot R \cdot S$
- B. O añadir valores solo en una operación y darle a aplicar. De este modo se pueden aplicar las operaciones en cualquier orden.

Luz Ambiental

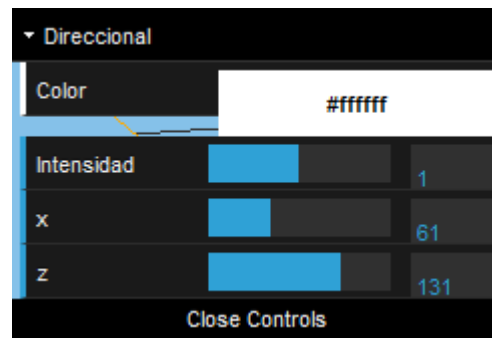
Para la escena de la luz ambiental es más sencillo, se tienen dos desplegados desde los que se puede modificar la intensidad y cambiar el color de cada luz. Disminuyendo la intensidad a 0 se desactiva la luz en cuestión.



Ilustración All-2: controles escena 2

Luz Direccional

Los controles de esta escena son bastante parecidos a la anterior, solo que se añade la posibilidad de rotar la luz respecto al eje X y al eje Z, esto hace que cambie su posición y se aprecie el desplazamiento de las sombras.

*Ilustración All-3: controles escena 3*

Luz Puntual

En esta escena el control añadido permite modificar la atenuación de la luz, esto afecta al alcance de sus rayos en todas las direcciones. Y al igual que en las otras se puede modificar la intensidad y el color.

*Ilustración All-4: controles escena 4*

Foco

Esta luz es algo más compleja que las anteriores ya que su comportamiento depende de más parámetros. Al igual que las otras se puede modificar la intensidad y el color. Además se puede alterar la atenuación, cuanto más bajo sea este valor, menos alcance tienen los rayos de luz. También se puede aumentar o disminuir el ángulo de apertura. Y la penumbra, el degradado de color que se forma en los límites del ángulo.

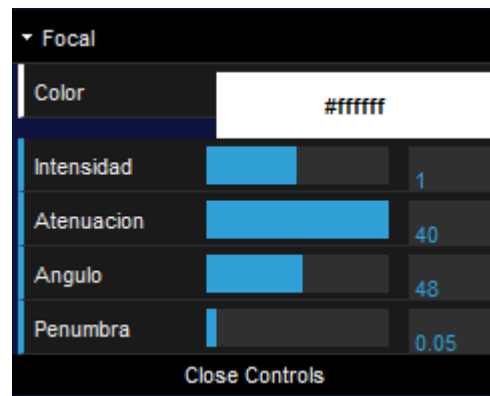


Ilustración All-5: controles escena 5

Selección

En esta escena no hay ningún desplegable con controles. Se trata de seleccionar objetos con el ratón haciendo click en cualquier punto dentro del canvas. Al seleccionar un objeto, este parpadea y rota según el desplazamiento del ratón. Si se tiene un objeto seleccionado y el puntero abandona el área del canvas, el objeto dejará de estar seleccionado.

Grafo de escena

En este desplegable se puede encontrar 3 submenús en los que se pueden rotar los puntos articulables del brazo robótico. Los ángulos están representados en grados, el primer punto se puede rotar en los 3 ejes (X, Y y Z), el segundo respecto X y Z, y el tercero respecto al eje Y. Si se marca la casilla de “Animar” el brazo se moverá automáticamente hasta cierta posición.



Ilustración All-6: controles escena 7

Mallas de triángulos

Como se ha explicado anteriormente en esta escena se presenta una esfera junto con su wireframe visible. La cual se puede agrandar modificando su radio, aumentar o disminuir las secciones en longitud y latitud. Y seccionar la esfera de manera vertical u horizontal.



Ilustración All-7: controles escena 8

Aplicación de texturas

En este menú, haciendo click en las diferentes opciones se puede modificar las funciones de magnificación y minificación de la textura que se aplica en el plano de la escena, y también se puede cambiar la imagen de la textura. Para saber más acerca de cada opción ver apartado 4.4.1 de la sección de desarrollo.

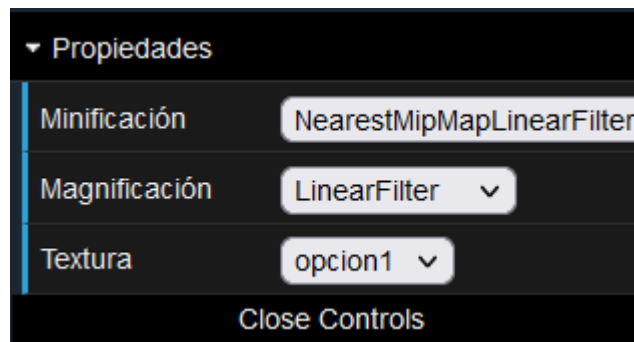


Ilustración All-8: controles escena 9

Bump Mapping

En esta escena se puede alternar entre un cubo y una esfera, a las que se le pueden aplicar dos texturas diferentes, una simulando unos ladrillos y otra de unas rocas. Al alterar el nivel de detalle se puede aplicar el efecto de bump mapping con mayor o menor intensidad.

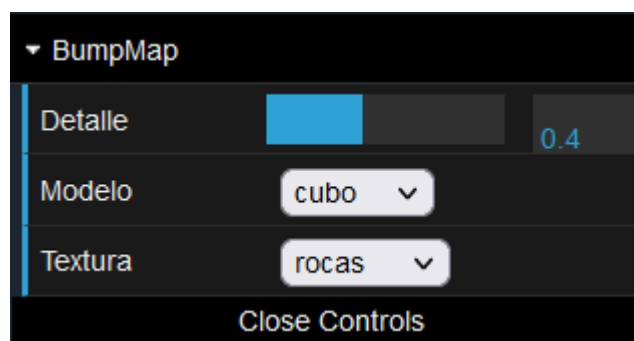


Ilustración All-9: controles escena 10

Tipos de reflexión

En esta escena se presenta una modelo que parece una gran goma elástica retorcida, a la cual se le puede modificar su geometría, es decir, su tamaño y el

número de triángulos añadiendo más divisiones tanto radiales como segmentos. Por otro lado se puede alterar su aspecto y ver cómo influye un mayor nivel de especularidad y de rugosidad, y la combinación de ambos. También se puede aplicar una textura preconfigurada que simula un suelo de madera. Activando la opción de “animar”, el modelo rota automáticamente.

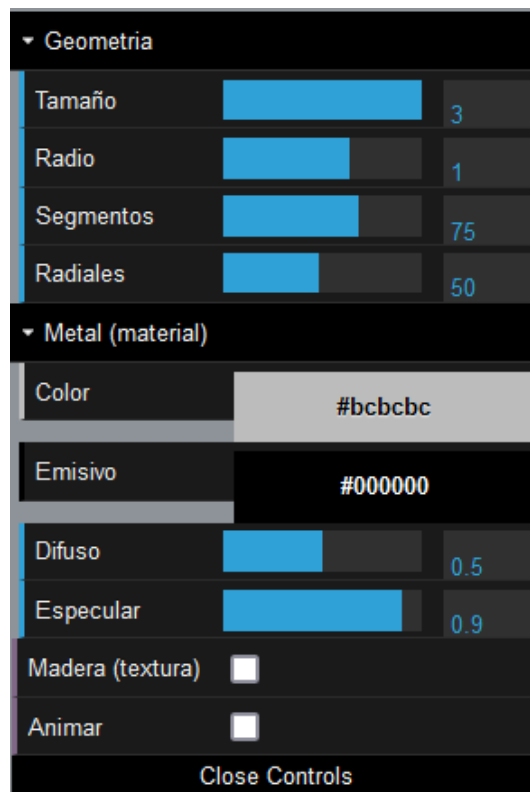


Ilustración All-4: controles escena 11

Apéndice III. Descripción contenido entregado

Los archivos entregados en este TFG son los siguientes:

- **Herramienta_para_la_docencia_de_Informática_Gráfica_utilizando_WebGL_TFG.pdf.** La actual memoria del proyecto.
- **Proyecto_Vue_TFG.zip.** Archivo comprimido con el código y assets.
- **Vídeo de demostración.** Incluido en el archivo comprimido.