

**Escola Universitària d'Enginyeria Tècnica
de Telecomunicació La Salle**

Treball Final de Grau

Grau en Enginyeria Multimèdia

**Desarrollo de un
motor de juegos
*from scratch***

Alumne

Joel López Romero

Professor Ponent

Alun Thomas Evans

ACTA DE L'EXAMEN DEL TREBALL FI DE CARRERA

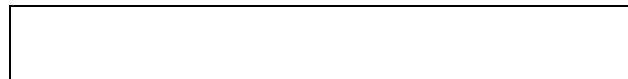
Reunit el Tribunal qualificador en el dia de la data, l'alumne

Joel López Romero

va exposar el seu Treball de Fi de Carrera, el qual va tractar sobre el tema següent:

Desarrollo de un motor de juegos *from scratch*

Acabada l'exposició i contestades per part de l'alumne les objeccions formulades pels Srs. membres del tribunal, aquest valorà l'esmentat Treball amb la qualificació de



Barcelona,

VOCAL DEL TRIBUNAL

VOCAL DEL TRIBUNAL

PRESIDENT DEL TRIBUNAL

Resumen

Los estudiantes y desarrolladores cuentan con miles de herramientas de desarrollo de aplicaciones y *software*. Incluso en el sector de los videojuegos y la animación, las empresas utilizan programas para los que es necesario formar a los futuros trabajadores. Sin embargo, es crucial que éstos no sólo aprendan a utilizar dichos sistemas de modelado o entornos de programación, sino que comprendan los procesos que subyacen al material que están generando y aprendan a “pensar” qué procedimientos y funciones siguen las máquinas para obtener un producto final (como una película 3D o un videojuego).

Para ello, en este trabajo se ha realizado una guía exhaustiva de cómo programar dos entornos 3D (dos motores gráficos distintos) para poder ser utilizados en cualquier producción interactiva, además de una comparativa entre ambos a nivel de rendimiento general, escalabilidad y limitaciones técnicas. De este modo, los estudiantes o desarrolladores podrán analizar, estudiar y aprender qué operaciones convierten unos y ceros en vértices que puedan proyectarse en una pantalla y ofrecer *feedback* visual al espectador o jugador de un mundo creado completamente de cero.

El primer motor gráfico consiste en un *raycaster* o simulación de un entorno 3D, y se implementa en JavaScript, con base en HTML5, pudiendo abrirse en un entorno web dedicado como *Github Pages*. Éste se puede desarrollar en cualquier IDE como *Visual Studio Code* o incluso *Notepad++*.

Por otro lado, el segundo motor gráfico es un *engine* de gráficos en 3D, programado en Java, para poder funcionar como un ejecutable de escritorio. El proyecto original está implementado en el *software* NetBeans IDE, en su versión 8.2.

Para asegurar el conocimiento adquirido en este trabajo, al lector se le ofrece, además de dichas guías de desarrollo, el contenido teórico que subyace a las operaciones necesarias, antes de comenzar a programar. Con toda la información ofrecida en este trabajo, los estudiantes habrán podido tratar los gráficos por ordenador desde primera fila, y estarán listos para aprender a utilizar cualquier herramienta de desarrollo de aplicaciones, comprendiendo su funcionamiento interno.

Resum

Els estudiants i desenvolupadors compten amb milers d'eines de desenvolupament d'aplicacions i programari. Fins i tot en el sector dels videojocs i l'animació, les empreses utilitzen programes per als quals és necessari formar els futurs treballadors. No obstant això, és crucial que aquests no només aprenguin a utilitzar aquests sistemes de modelatge o entorns de programació, sinó que comprenguin els processos subjacentes al material que estan generant i aprenguin a "pensar" què procediments i funcions segueixen les màquines per obtenir un producte final d'aquest caire (com una pel·lícula 3D o un videojoc).

Per això, en aquest treball s'ha realitzat una guia exhaustiva de com programar dos entorns 3D (dos motors gràfics diferents) per poder ser utilitzats en qualsevol producció interactiva, a més d'una comparativa entre tots dos a nivell de rendiment general, escalabilitat i limitacions tècniques. D'aquesta manera, els estudiants o desenvolupadors podran analitzar, estudiar i aprendre quines operacions converteixen uns i zeros en vèrtexs que puguin projectar-se en una pantalla i oferir *feedback* visual a l'espectador o jugador d'un món creat completament de zero.

El primer motor gràfic consisteix en un *raycaster* o simulació d'un entorn 3D, i s'implementa en JavaScript, amb base en HTML5, podent obrir-se en un entorn web dedicat com *GitHub Pages*. Aquest es pot desenvolupar en qualsevol IDE com *Visual Studio Code* o fins i tot *Notepad++*.

D'altra banda, el segon motor gràfic és un *engine* de gràfics en 3D, programat en Java, per poder funcionar com un executable d'escriptori. El projecte original està implementat en el programari NetBeans IDE, en la versió 8.2.

Per assegurar el coneixement adquirit en aquest treball, al lector li és ofert, a més d'aquestes guies de desenvolupament, el contingut teòric subjacent a les operacions necessàries, abans de començar a programar. Amb tota la informació continguda en aquest treball, els estudiants hauran pogut tractar els gràfics per ordinador des de primera fila, i estaran a punt per aprendre a utilitzar qualsevol eina de desenvolupament d'aplicacions, comprendent el seu funcionament intern.

Abstract

Students and developers are provided with thousands of software and application development tools. Even in the videogames and animation sector, companies use programs for which it is necessary to train future workers. However, it is crucial that they not only learn to use such modeling systems or programming environments, but also understand the processes that underlie the material they are generating and learn to "think" what procedures and functions the machines follow to obtain a final product (like a 3D movie or video game).

To do this, an exhaustive guide has been made in this document, focusing on how to program two 3D environments (two different graphics engines coded from zero) to be used in any interactive production, as well as a comparison between them regarding general performance, scalability and technical limitations. This way, students or developers will be able to analyze, study and learn which operations convert ones and zeros into vertices that can be projected on a screen and offer visual feedback to the viewer or player of a world created completely from scratch.

The first graphics engine consists of a ray caster or simulation of a 3D environment, and is implemented in JavaScript, based on HTML5, and can be opened in a dedicated web environment such as GitHub Pages. This can be developed in any IDE like Visual Studio Code or even Notepad ++.

On the other hand, the second graphics engine is a "real" 3D graphics engine, programmed in Java, to function as a desktop executable. The original project is implemented in the NetBeans IDE software, version 8.2.

To ensure the knowledge acquired in this work, the reader is offered, in addition to these development guides, the theoretical content that underlies the necessary operations, before starting to program. With all the information gathered in this document, students will have been able to deal with computer graphics from scratch, and they will be ready to learn to use any application development tool, whilst understanding its inner processes.

Contenido

1	Introducción	15
1.1	La visión general.....	15
1.2	Y, ¿en los videojuegos?	16
1.3	Las herramientas y su peligro.....	16
1.4	Objetivo en escena.....	18
1.5	Desarrollo y desglose del proyecto	20
1.6	¡Logro desbloqueado!	21
2	Motor 2D: <i>Raycaster</i>	23
2.1	¿Qué es un <i>raycaster</i> ?	23
2.2	Orígenes	25
2.2.1	Alternate Reality: The City (1985)	26
2.2.2	Wolfenstein 3D (1992)	26
2.3	Nuestro <i>engine</i>	28
2.4	Funcionalidades (en orden de implementación)	29
2.5	Diagrama de archivos y clases.....	32
3	Implementación del <i>raycaster</i>	35
3.1	Ficheros	35
3.1.1	<i>P5.js</i>	35
3.1.2	<i>Index.html</i>	35
3.1.3	<i>Styles.css</i>	36
3.1.4	<i>Main.js</i>	36
3.2	Generación de un nivel	37
3.2.1	Cambios en <i>main.js</i>	37
3.2.2	Implementación de <i>mapController.js</i>	38
3.3	Movimiento y ángulo del jugador	41
3.3.1	Cambios en <i>main.js</i>	41
3.3.2	Primeras funciones de <i>utils.js</i>	41
3.3.3	Implementación de <i>playerController.js</i>	42
3.4	<i>Player input</i> y movimiento del jugador	46
3.4.1	Implementación de <i>keyController.js</i>	46
3.4.2	Colisiones del jugador	47
3.5	<i>Raycasting</i>	48

3.6	Colisión de rayos	51
3.6.1	Nueva función en <i>Map</i> (<i>mapController.js</i>).....	51
3.6.2	Nueva función en <i>utils.js</i>	51
3.6.3	Colisiones horizontales.....	52
3.6.4	Colisiones verticales	55
3.6.5	Unión de los dos tipos de colisión.....	57
3.7	Generación de una escena multinivel.....	58
3.8	Proyección de muros (<i>vLines</i>).....	59
3.8.1	Cambios en <i>main.js</i>	59
3.8.2	Implementación de <i>renderController.js</i>	60
3.8.3	Implementación del mini mapa	61
3.9	Controles de tipo libre (opcional)	62
3.10	Lector de imágenes	63
3.10.1	Implementación de <i>imageLoader.js</i>	63
3.10.2	Modificaciones en <i>main.js</i>	65
3.11	Generación del nivel por imagen	66
3.12	Texturizado de muros	68
3.12.1	Implementación de clase: <i>VLine</i>	68
3.12.2	Modificaciones en la clase <i>Render</i>	72
3.13	Coloreando <i>floor</i> y <i>ceiling</i>	73
3.13.1	Modificaciones en la clase <i>Render</i>	74
3.13.2	Últimas llamadas en <i>main.js</i>	75
4	Motor 3D: “3ngine”	77
4.1	Nuestro <i>engine</i>	77
4.2	Funcionalidades.....	78
4.3	Diagrama UML (clases).....	79
4.4	Diagrama UML (relacional).....	84
5	Implementación del 3ngine	87
5.1	Definición de la estructura del proyecto.....	87
5.1.1	Montar Modelo-Vista-Controlador.....	87
5.1.2	Contenido de la escena	91
5.2	Carga de una escena (cubo de testeo)	93
5.3	Transformaciones de vértices	95
5.3.1	Migración de 3D a 2D (proyección).....	96

5.3.2	Transformaciones del objeto	99
5.3.3	Transformaciones de la cámara	102
5.4	<i>Input</i> del usuario	105
5.4.1	Modificaciones en <i>CameraModel</i>	105
5.4.2	Atributos del <i>UserInputController</i>	108
5.4.3	<i>EngineController implements KeyListener</i>	108
5.4.4	Aplicar el <i>input</i>	108
5.5	Generación de triángulos	109
5.5.1	Variable de ordenación	110
5.5.2	Cálculo de “depthValue”	110
5.5.3	Método de ordenación.....	111
5.6	<i>Triangle clipping</i>	113
5.6.1	Procedimiento	114
5.6.2	Información necesaria para el <i>clipping</i>	115
5.6.3	<i>VertexIntersectsPlane</i>	115
5.6.4	<i>TriangleClipToPlane</i>	117
5.7	Rayo de luz	119
5.8	Carga de archivos 3D (formato <i>.obj</i>)	120
5.8.1	Formato <i>.obj</i>	121
5.8.2	Lectura del <i>.obj</i>	121
5.9	Optimización de triángulos	122
5.10	Métodos de renderizado.....	124
5.10.1	Completo (<i>full render</i>)	124
5.10.2	<i>Surface</i>	125
5.10.3	<i>Wireframe</i>	125
6	Comparación: 3ngine vs <i>Raycaster</i>	127
6.1	Parámetros del sistema (PC)	127
6.2	Parámetros del <i>Raycaster</i> (motor 2D)	128
6.3	Parámetros del 3ngine (motor 3D)	129
6.4	Rendimiento general.....	131
6.5	Optimización y limitaciones técnicas	133
6.6	Escalabilidad general.....	135
6.7	Experimentación de gráficos y efectos visuales.....	135
7	Conclusiones.....	137

7.1	Resultados obtenidos y conclusiones generales.....	137
7.2	Coste en horas del trabajo	138
7.3	Puntos débiles y limitaciones del trabajo	139
7.4	Puntos fuertes	139
7.5	Líneas de continuación.....	140
8	Índices de referencias	141
8.1	Referencias a tablas	141
8.2	Referencias a ecuaciones	141
8.3	Referencias a diagramas	141
8.4	Referencias a figuras	142
9	Bibliografía	145

Acrónimos

2D/3D: Dos/tres dimensiones respectivamente.

fps: (en minúsculas) *frames per second*. Número de imágenes (o *frames*) por segundo.

FOV: *Field of view*. Campo de visión. Puede hacer referencia a una cámara o al jugador.

FPS: (en mayúsculas) *First person shooter*. Juego de disparos en primera persona.

IDE: *Integrated Development Environment*. Entorno de desarrollo integrado.

OBJ: Formato de archivo de tipo .obj, para representar información geométrica tridimensional.

RPG: *Role-Playing Game*. Juego de rol.

TPS: *Third person shooter*. Juego de disparos en tercera persona.

TFG: Trabajo Final de Grado.

En este capítulo se listan los acrónimos que aparecen en el documento, por orden alfabético. Ya no se desplegarán de nuevo en el resto del documento. Si los acrónimos provienen de palabras escritas en un idioma diferente al del documento, se escribirá en cursiva.

1 Introducción

“Me parece haber sido sólo un niño jugando en la orilla del mar, divirtiéndose y buscando una piedra más lisa o una concha más bonita de lo normal, mientras el gran océano de la verdad yacía ante mis ojos con todo por descubrir”, **Isaac Newton** (1642 – 1727).

Con esta frase, el descubridor de la Ley de la gravitación universal dio a entender el enorme horizonte de posibilidades que se presentaba al ser humano, sobre todo en lo que a ciencia respecta. Casi tres siglos después de que falleciera el brillante científico, se puede ver aplicada esta misma frase al enorme campo que presenta el mundo de la **computación**, evolucionando a pasos de gigante por cada año que pasa.

Actualmente se cuenta con dispositivos extremadamente capaces a los que nos dirigimos como ordenadores personales, y cuya función es sorprendentemente simple: escuchar y devolver. Éstos realizan una (absolutamente arrolladora) serie de cálculos según el *input* que reciben por parte del usuario. No obstante, también existe la posibilidad de gestionar qué es lo que el usuario pretende simular en este entorno digital, preparando una secuencia de acciones y escogiendo qué efecto asociar a cada valor obtenido.

En otras palabras: **programar** una función de la máquina.

1.1 La visión general

Vivimos en un mundo extremadamente digitalizado e interconectado, y los usuarios nos hemos acostumbrado a una gran cantidad de acceso libre y contenido en esa red. Ahora los desarrolladores gozan de una fuente prácticamente inexorable de recursos dentro de las cuales entran esos programas.

Estas herramientas se pueden utilizar para agilizar en gran medida los procesos más tediosos, complejos o repetitivos que se deben ejecutar de manera recurrente, ya sea en el trabajo, para llevar a cabo una afición o incluso en el ocio.

Sin ir más lejos, para redactar un documento de texto se puede crear un documento de Word, el *software* de Office, que nos ofrece procesos automatizados como generar un índice a partir de los títulos que se han introducido en el archivo, o numerar las páginas de forma instantánea.

Pero no se debe olvidar que estas herramientas son, han sido y están siendo (por ahora) desarrolladas por programadores y *developers* (ya sea de *software* o de *hardware*). Actualmente existen ya miles de herramientas para que los usuarios puedan prescindir de dedicar tiempo a preparar un recurso, ofreciendo una plataforma configurable y previamente preparada donde pueden centrarse directamente en el foco. Por ejemplo, si un programador necesita un entorno para escribir un código, compilarlo y ejecutarlo, lo más probable es que pueda descargar una aplicación que permita todas esas acciones e incluso más. Quizá incluso de forma gratuita.

Esta manera tan cómoda de trabajar en nuestros sistemas informáticos puede verse en prácticamente cualquier ámbito que envuelva una tarea mecanizada y un ordenador. Desde los programas más habituales y generales (como el ya mencionado Word o sus hermanos cercanos, Excel, OneNote y PowerPoint), hasta aquellos con un objetivo más particular o definido: Adobe

Photoshop para la creación y edición de imágenes, Audacity en cuanto a audio y sonido y Premiere o After Effects (ambos también de Adobe) para edición de vídeo y efectos visuales, por nombrar sólo algunos de los más habituales entre estudiantes.

1.2 Y, ¿en los videojuegos?

Del mismo modo, los programadores de videojuegos ya no se ven en la necesidad de comenzar desde cero un proyecto que acabe convirtiéndose en el nuevo *Call of Duty* o *Street Fighter*. La gran mayoría de estos productos (independientemente del género al que pertenezcan) comparten una gran cantidad de similitudes en su etapa de desarrollo, así como en las funciones que se necesitan implementar: véase el uso de texturas, cámaras, un sistema de sonido, colisiones, interacción por parte del usuario (el ya mencionado *input*)... e incluso podrían seguir apareciendo otros si se profundizara aún más en la lógica que subyace a estos procesos. De estos, uno de los ejemplos más claros es el siguiente: el que una entidad se detenga al colisionar contra un muro es un comportamiento aplicable a cualquier *game object* móvil del juego, tanto si es nuestro jugador como un NPC o un enemigo controlado por la IA. Sabiendo esto, lo más eficiente es preparar un código que pueda ser útil para cualquiera de estas tres entidades.

Sin embargo, esta situación que puede aliviar tanto el código del proyecto para los desarrolladores no sólo es aplicable a los objetos que conforman un único videojuego. Ese mismo ejemplo en que una criatura se detiene frente a un muro, o el tipo de movimiento de una cámara en un entorno tridimensional se mantiene en cualquier simulación física que se vaya a programar (siempre que ésa sea la intención).

En pocas palabras, si todos los objetos tienen las mismas propiedades (textura, colisiones, *bounding box*...) y la cámara (o jugador, directamente) se va a mover igual en cualquier simulación realista que nuestro sistema vaya a calcular y renderizar por pantalla, es extremadamente más cómodo buscar y desarrollar una base común para todos estos proyectos que compartan similitudes tan notables. Una herramienta capaz de aligerar el proceso técnico más confuso o tedioso como el algoritmo de renderizado de un videojuego, la lectura de objetos tridimensionales y sus distintas texturas, creación de escenas y menús, o la configuración del audio o los controles del usuario se vuelve prácticamente necesaria si los desarrolladores quieren mantener un flujo de trabajo intenso, obviando esas fases del proceso y pasando directamente al “videojuego”.

Esta potentísima herramienta cuya función principal es la recién expuesta se conoce como motor. O **motor de videojuego** (más comúnmente referida como *engine* o *game engine*). En ocasiones, a estos programas también se los llama “plataforma de desarrollo”.

1.3 Las herramientas y su peligro

Pese a que se podría indagar más en el mastodóntico crecimiento de la industria del videojuego (en 2020, en todo el mundo, ésta facturó más que el cine y los deportes juntos en Estados Unidos[14]), por ahora el foco de atención se lo llevarán los mencionados *game engines* y su papel en la industria y de cara a los programadores (o estudiantes de programación, más concretamente). En la Tabla 1 se muestran algunos de los videojuegos más exitosos de 2020, tanto a nivel de ventas, como de crítica y/o influencia en la industria. De cada uno de ellos, también la empresa encargada del desarrollo y el *engine* utilizado para éste.

Tabla 1. Videojuegos más exitosos de 2020 [13], y sus motores.

Videojuego	Desarrolladora	Motor
Among us [11]	InnerSloth	Unity (2021.4.12)
Assassin's Creed: Valhalla [19]	Ubisoft (Montreal)	Anvil (AnvilNext 2.0)
Call of Duty: Warzone [12]	Infinity Ward, Raven Software	IW Engine (IW 8.0)
Cyberpunk 2077 [15]	CD Projekt RED	REDEngine 4
Demon's Souls[25] , Demon's Souls Remake [23]	From Software, Bluepoint Games	Havok, Bluepoint Engine
DOOM Eternal [5]	ID Software	id Tech 7
Fall Guys: Ultimate Knockout [10]	Mediatonic	Unity
Final Fantasy VII: Remake	Square Enix, Geomerics, Epic Games	Unreal Engine 4

Cualquier desarrollador puede encontrar a su total disposición varios motores o editores de videojuegos, la mayoría de éstos de forma completamente gratuita (como en el caso de Unity, Unreal Engine y CryEngine).

En la Tabla 2 se muestran otros juegos varios, también éxitos de generaciones pasadas, donde podemos ver que los motores también sirven para cualquier género al que pertenezca el producto, tanto si es de lucha como un videojuego de disparos en primera o tercera persona. Del mismo modo, se puede observar que un mismo motor puede ofrecer sus funciones para videojuegos de distinta categoría, como en el caso de Unreal Engine 4.

Tabla 2. Videojuegos varios por motor y género.

Videojuego	Género	Desarrolladora	Motor
Devil May Cry 5 [7]	Hack and slash	Capcom	RE Engine
DmC: Devil May Cry [2]	Hack and slash	Ninja Theory	Unreal Engine 3
Tekken 7 [6]	Lucha	Namco Bandai Games	Unreal Engine 4
Days Gone	TPS	SIE Bend Studio	Unreal Engine 4
GRIP: Combat Racing [16]	Carreras	Caged Element	Unreal Engine 4
Crysis [27]	FPS	Crytek	CryEngine 2
Dark Souls [4]	RPG	From Software	Havok
Cuphead [24]	Plataformas	StudioMDHR	Unity
Asphalt 9: Legends [3]	Carreras	Gameloft Barcelona	Jet Engine

Si bien es cierto que, objetivamente hablando, todos estos datos corroboran que esta cantidad de información, recursos y herramientas no es más que beneficiosa para los desarrolladores y, por ende, para las empresas o equipos donde trabajan, también debemos ser conscientes del riesgo que entraña, de cara a los estudiantes, el perder de vista cómo funcionan estos motores y cómo diseñar y programar estas funciones, **antes** de empezar a utilizarlos.

La **comodidad** y seguridad que ofrece el contar con todas estas funcionalidades y algoritmos ya desarrollados puede representar un **peligro** de cara al interesado a caer en la comodidad de

utilizar todo un mundo de posibilidades como si se tratara de una caja negra. De verse en esta situación, un estudiante podría perder el interés en aprender y, en el peor de los casos, sobrescribirlo del todo por un interés en terminar, lo cual repercutiría negativamente en su visión y pasión para atreverse a ver más allá y aprehender sobre su propia **evolución**.

En una frase: para seguir mejorando debemos ser conscientes de todos los pasos que hemos andado hasta llegar al hoy. No sólo utilizarlos porque están ahí y son gratuitos, sino entender cómo han llegado hasta ahí, cómo luchan y ofrecen **soluciones** a las funcionalidades que el estudiante o desarrollador debe implementar, pese a que ya existan y/o sean más óptimas que las que probablemente acabe programando un becario o perfil junior. En eso consiste la etapa del aprendizaje.

1.4 Objetivo en escena

Como último dato empresarial, cada compañía o desarrolladora de la industria de videojuegos (sobre todo las de gran escala como Capcom, Nintendo, Ubisoft, etc.) suele utilizar su propio *game engine* —a ver, en la Tabla 3— a la hora de producir sus entregas. De modo que, en caso de terminar trabajando en una de éstas, el usuario deberá aprender a manejar dicha herramienta en cuestión.

Tabla 3. Motores principales de las desarrolladoras más exitosas.

Desarrolladora	Motor
Ubisoft	Anvil Engine
Rockstar	RAGE ¹
Crytek	CryEngine
ID Software	id Tech
Capcom	RE Engine MT Framework
Valve Corporation	GoldSrc (id Tech modificado) Source Engine
DICE	Frostbite
Bluepoint Games	Bluepoint Engine
Infinity Ward	IW Engine
Treyarch	Treyarch NGL (basado en id Tech 3) [21] Demonware IW Engine

Sin embargo, los motores de videojuegos, o más concretamente sus **funcionalidades**, se mantienen como punto de inflexión sobre el cual enfocar el aprendizaje del estudiante de cara al **desarrollo de videojuegos**, particularmente en su faceta más técnica.

En este documento, por tanto, se ofrecerá una **guía de desarrollo** para que un alumno interesado en asignaturas relevantes en la industria del videojuego sea capaz de implementar sus propias plataformas de desarrollo, **focalizando** así su atención en dicho **aprendizaje** y evitando que los estudiantes con esa misma intención salten directamente a utilizar

¹ Rockstar Advanced Game Engine

herramientas como Unity o Unreal Engine antes de comprender qué cálculos o procesos se están ejecutando en segundo plano, tras el monitor.

De tal modo, al terminar, el usuario habrá sido capaz de **programar** no sólo uno sino **dos plataformas** de desarrollo sobre las cuales poder seguir evolucionando el sistema o directamente producir un videojuego de primera mano, **desde cero**, y completamente funcional. Además de haber adquirido los conocimientos necesarios para programar sus propias herramientas o funcionalidades tal y como funcionan (conceptualmente) en los *game engines*.

Antes de profundizar en qué plataformas se van a exponer y desarrollar a lo largo de la guía, nombraremos en conjunto las **funcionalidades** que se implementarán en el sistema:

- **Gestión y lectura de recursos.** Consiste en la lectura de los archivos que se utilizarán en el sistema, desde ficheros de texto hasta imágenes o documentos de tipo OBJ.
- **Carga de escena y/o niveles.** Comenzando por la lectura del archivo fuente que se utilizará como referencia para generar un nivel, hasta situar en medio al jugador. Todo este proceso (básico para ambos motores) es crucial a la hora de “montar” un entorno interactivo o simulación.
- **Cámara.** También se le puede referir como jugador, ya que funciona como foco para el renderizado de la escena y sus elementos. Se verán las transformaciones básicas completas a realizar: traslación y rotación en tres direcciones, así como proyección (método de ordenación de triángulos incluido, al llegar a la fase de renderizado del 3ngine).
- **Transformaciones, lógica y edición de entidades.** Del mismo modo que se modificarán los parámetros de posición y rotación de la cámara, también se desarrollarán las operaciones para transformar los objetos de la escena, mover, rotar y escalar.
- **Texturizado.** Tras la lectura de recursos se deberán mostrar en pantalla las texturas asignadas a sus correspondientes elementos.
- **Física de colisiones.** Será necesario implementar un sistema de colisiones con los objetos si el usuario decide seguir evolucionando la plataforma y obtener una experiencia disfrutable, además de jugable.
- **User input.** Los controles de jugador son (aunque conceptualmente sencillos) absolutamente necesarios para el correcto desarrollo del sistema, además de ofrecer una capacidad inmersiva directa durante la implementación y el testeo del proyecto. Por eso es crucial tener una gestión de los controles con *feedback* claro y directo.

Desgraciadamente, el perfil de audio es un tema mucho más concreto y depende del lenguaje de programación, así como de la aplicación que el usuario decida utilizar, pudiendo llegar a variar en su implementación o necesitar de librerías externas. Por este motivo, se excluye del desarrollo del motor (aunque el plan de lectura de recursos es igual de efectivo con archivos de sonido).

1.5 Desarrollo y desglose del proyecto

Teniendo ahora claras las funcionalidades que se expondrán y ejecutarán a lo largo de esta guía, se entrará en el tema principal: ¿qué son estos dos proyectos que se van a implementar? En pocas palabras, un ***raycaster*** en dos dimensiones (el sistema más básico para generar una simulación tridimensional de una escena, más adelante se verá cómo) y un ***motor*** en ***3D*** que llamaremos “***3ngine***” (centrado en la lectura de geometría real y objetos tridimensionales, y el renderizado de gráficos desde cero).

No se codificarán todas las funcionalidades en los dos motores, ya que de este modo podremos mantener un nivel óptimo de ejecución y no sobrecargar el código; pudiendo centrar nuestra atención y buscar la implementación que interese al usuario en cada caso, además de mantenernos dentro de las limitaciones técnicas de cada sistema.

La distribución de funciones entre ambos motores será la que reza en la Tabla 4:

Tabla 4. Lista de operaciones de cada motor a implementar.

<i>Raycaster</i>	<i>3ngine</i>
Lectura de recursos	Lectura de recursos
Carga de escena multinivel	Carga de escena (.obj)
Texturizado	Transformaciones de escena
<i>User input</i>	<i>User input</i>
Cámara (2D), parámetros variables, transformaciones	Cámara (3D), transformaciones, optimización
Colisiones	Modos de renderizado, ordenación de elementos

De ahora en adelante, el contenido principal del documento se dividirá en dos grandes bloques. En cada uno de ellos se expondrá en qué consiste exactamente el motor correspondiente, por qué se ha decidido programar éste en particular, y una visión mucho más profunda y técnica de las funcionalidades mencionadas previamente para el proyecto en cuestión (igual con los cálculos matemáticos que hagan falta realizar). Luego se pasará a la guía de desarrollo (apartados de “Implementación”), donde serán facilitadas todas las explicaciones y diagramas necesarios para enseñar en qué consiste lo que se quiere desarrollar, y el estudiante o interesado podrá seguir las explicaciones mientras desarrolla el proyecto, aprendiendo mientras programa y adquiere los conocimientos conceptuales y técnicos necesarios.

Por último (en lo que concierne a esta parte de la guía), cabe mencionar que el estudiante o lector del documento tendrá la total capacidad de llevar estos conocimientos al lenguaje de programación con que se sienta más cómodo, en caso de tener preferencias, dado que el código de la guía no es simplemente un ejemplo obligatorio que seguir, sino que pretende enseñar, como se ha comentado, los **conceptos** e ideas en que se basan los algoritmos a programar.

Tras la parte práctica del documento, se llevará a cabo una comparación entre ambos motores gráficos, a nivel de rendimiento, escalabilidad del sistema (qué se puede hacer con lo que se

ofrece en cada uno de ellos), así como las limitaciones técnicas o posibles evoluciones de cada uno de ellos.

Para terminar, se mostrará la tabla con el coste en horas del proyecto, agrupado en los principales bloques de investigación y desarrollo, y se explicarán los resultados obtenidos en el apartado de conclusiones, junto a los puntos fuertes y débiles del proyecto y otro vistazo más general a las posibles vías de continuidad de los proyectos.

1.6 ¡Logro desbloqueado!

Dado que en este documento se ofrecerán tanto la teoría necesaria como los pasos en la implementación de ambos *engines*, una vez el estudiante haya comprendido y puesto en práctica los conocimientos adquiridos en este trabajo, estará listo para pasar al desarrollo de sistemas tales como motores de videojuego, plataformas de creación de recursos (como 3DS Max o Blender) y aplicaciones multimedia interactivas. Además, podrá utilizar cualquier plataforma de desarrollo de videojuegos al máximo nivel, siendo consciente de las subrutinas que se llevan a cabo durante la ejecución de uno de estos programas, y siendo capaz de replicarlas a su modo de ver.

Al final del trabajo, en el apartado de conclusiones se hablará de forma más extendida y concreta de los resultados de ambas implementaciones digitales, así como de las capacidades adquiridas (o potenciadas) por el alumno en caso de haber sido capaz de aprehender los conceptos, algoritmos y cálculos tras los que radican las transformaciones que estas herramientas como Unity, Unreal Engine, Source o CryEngine ofrecen a los programadores y diseñadores de experiencias virtuales.

2 Motor 2D: *Raycaster*

Antes de pasar a la implementación directa del sistema, veremos qué propiedades convierten a un raycaster en un sistema de renderizado completamente válido como motor, y por qué se ha decidido programar uno como primer subproyecto.

2.1 ¿Qué es un *raycaster*?

El *raycasting* es un procedimiento que consiste en lanzar una determinada cantidad de rayos (o “vectores”) desde un punto origen en común, abriéndose en ángulo para formar un cono que defina un área. Esta función se utilizaba habitualmente como sistema para renderizar una simulación 3D de un espacio almacenado y gestionado en dos dimensiones, utilizando como mapeado de la escena una matriz de datos binarios (donde el valor ‘0’ sería espacio libre y el valor ‘1’ un muro, por ejemplo).

Una vez se ha generado una escena, el comportamiento del raycaster radica en la lógica de la cámara, cuya posición actúa como el punto convergente de origen de los vectores, y a partir del cual los rayos se abren y avanzan, formando un abanico (que funcionará como campo de visión del jugador, o FOV), esperando a colisionar contra un elemento de la escena o alcanzar una distancia máxima previamente considerada. Como indica la Figura 1, ésta es la primera fase del proceso.

Una vez la cámara ya ha emitido los rayos y éstos han llegado hasta la pared, comienza la fase dos: dado que tenemos la distancia que cada rayo ha recorrido (rayos del 1 al 8 en la Figura 1), se pintará su correspondiente línea vertical en pantalla —a la que, por comodidad, denominaremos “*vLine*”, de *vertical line*— cuya altura será inversamente proporcional a dicha distancia. De este modo cuando el jugador se encuentre frente a un muro, dado que la magnitud del vector (o rayo) será extremadamente corta, se renderizará una *vLine* enorme (*vLines* 4 y 5), mientras que, si éste está muy alejado (y, por ende, el vector

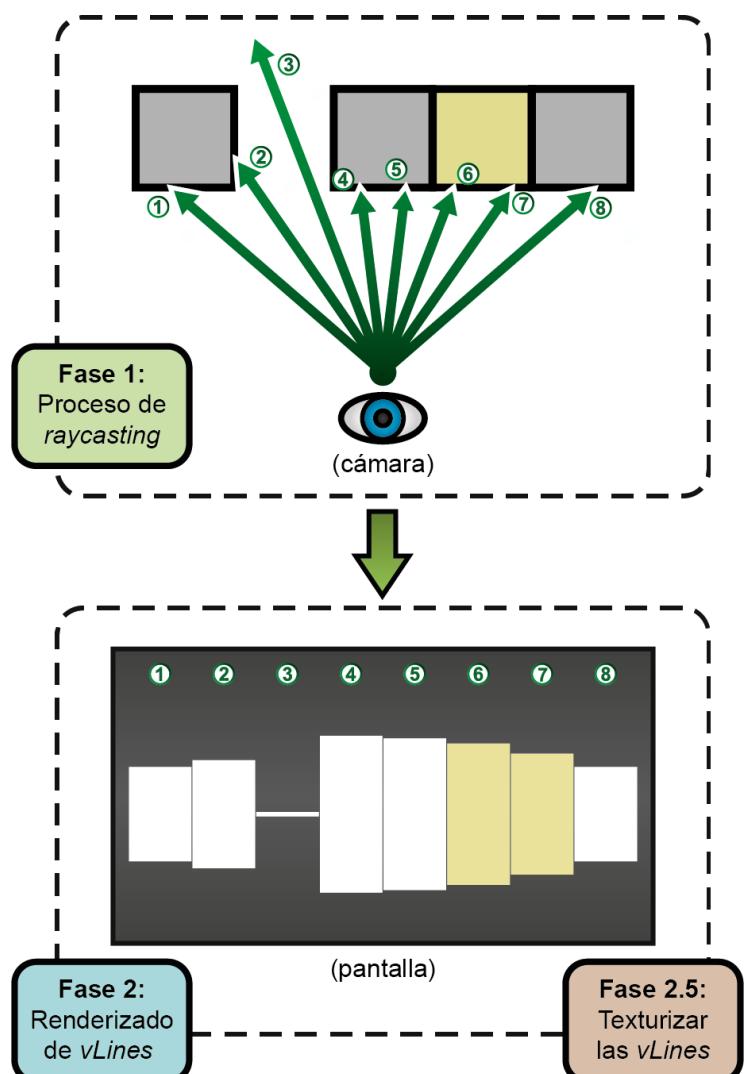


Figura 1. Representación gráfica del procedimiento de renderizado por *raycasting*, dividido en dos fases: emisión de rayos y dibujo y texturizado de *vLines* —una por cada rayo— (diagrama propio).

recorre mucha distancia), la *vLine* renderizada será mucho más baja, dando sensación de lejanía (*vLine* número 3).

Si bien es cierto que este sistema ofrece un método de renderizado extremadamente simple (aunque perfectamente capaz de soportar texturas, muros transparentes, e incluso aplicar luces), también está estrechamente ligado a tres limitaciones:

- **Los muros.** Todos los muros tendrán la misma altura, y no pueden renderizarse escenas con distintos niveles.
- **La escena.** Es necesario que el mapeado de la escena se distribuya en una cuadrícula (todos los cuadrados con la misma área), dado que gran parte de los cálculos matemáticos que envuelve la colisión de los rayos frente a los muros radica en trigonometría directa aplicada sobre los valores de dicha estructura (la longitud de los lados y el ángulo que forman contra los vectores que “salen” de la cámara).
- **La cámara.** Los controles del sistema pueden ser tanto de tipo tanque² como tipo normal (el estandarizado actualmente en todos los FPS).

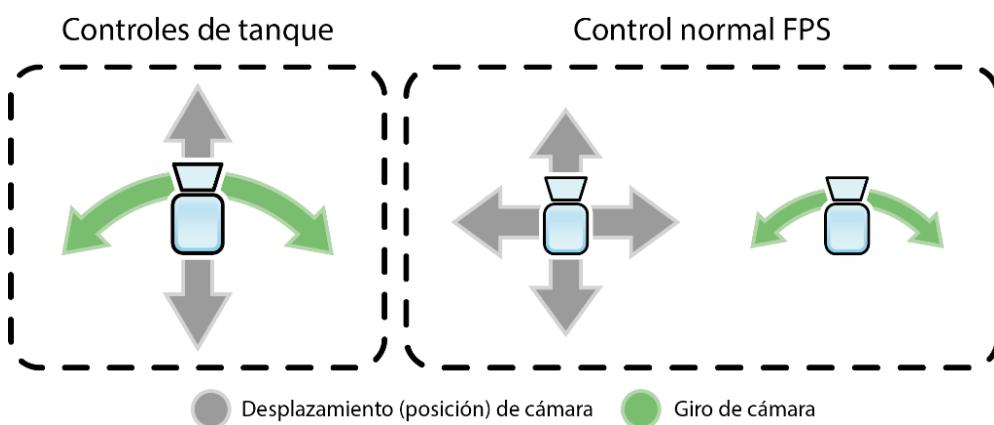


Figura 2. Tipos de control que se pueden implementar en un shooter. Aunque en el "Control normal FPS" aparezcan dos cámaras, son la misma, y ambos movimientos (desplazamiento y giro de cámara) se pueden hacer al unísono (diagrama propio).

Aunque ambos son válidos, nuestra implementación incluirá el control normal, que permite moverse en cualquier dirección horizontal (WASD en PC y stick analógico o pad direccional izquierdo —comúnmente denominado “cruceta”— en consolas) mirando en la misma dirección. Arriba, en la Figura 2 se puede distinguir la diferencia entre un tipo de controles y otro, y en la Figura 3, los botones mencionados en un controlador estándar de videojuegos.

² Los controles tipo tanque consisten en que el jugador únicamente pueda avanzar o retroceder en la dirección en la que está mirando, y girar la cámara a izquierda o derecha (*panning*). No podrá moverse hacia los lados.



Figura 3. Distinción entre pad direccional (o "cruceta"), y pads analógicos de un controlador manual de videojuegos estándar (diagrama propio).

Sin embargo, a la hora de girar la cámara los *raycasters* tienen una grave limitación a nivel de simulación 3D: **únicamente** se puede hacer ***panning* a izquierda o derecha**, no mirar arriba o abajo. Aunque puede simularse un efecto parecido al de *tilt* —es decir, inclinar la cámara— aplicando trigonometría adicional, éste no es un verdadero movimiento, sino una simulación del efecto óptico al recortar o desplazar las líneas que renderizaremos arriba o abajo.

Además, no entrará en el proyecto dicha implementación, dado que el consumo de recursos se dispararía aún más y podría generar incongruencias a la hora de aplicar texturas en los muros.

2.2 Orígenes

Especialmente en las primeras consolas de sobremesa como fueron la Atari, Amstrad CPC y demás, muchos desarrolladores de videojuegos utilizaban este tipo de renderizado gráfico para sus proyectos más atrevidos, dado que requiere de un consumo extremadamente menor al de las transformaciones y proyecciones que lleva a cabo un motor completamente 3D.

Ahora se mostrarán algunos de los juegos más emblemáticos que han empleado este tipo de simulación de entornos tridimensionales. De cada uno de ellos se hará un breve análisis, cuyos resultados se mostrarán en una tabla al terminar: el tipo de control de cara al jugador (tipo tanque, estándar u otro), si se permiten distintas alturas en el nivel, así como superficies transparentes (espacio entre rejas, por ejemplo) o translúcidas (cristales), si se añaden otros efectos en cuanto a gráficos (efectos de luz, humo, animaciones del entorno...) y el tipo de mapeado —estructura “libre”³, cuadrícula...

³ No está limitada a una estructura poligonal concreta, sino que los muros pueden tener curvas u otras estructuras propias generadas por funciones matemáticas.

2.2.1 Alternate Reality: The City (1985)⁴

Si bien es probable que éste sea el menos conocido de todos los que se listarán a continuación, *Alternate Reality: The City* fue el primer videojuego conocido que implementó el sistema de renderizado gráfico por *raycasting*. En él, el jugador se desplaza hacia delante o atrás en la dirección en la que está mirando, y **gira la cámara** a izquierda o derecha, pero **de 90 en 90 grados**.

Esto se debe a que es extremadamente **más óptimo** para el sistema que nuestro campo de visión “salte” entre norte, sur, este y oeste, antes que gestionar un FOV realista de cara a la cámara. De este modo, simplemente se debe calcular la distancia entre el jugador y los muros, pudiendo sobrescribir por valores predefinidos los cálculos trigonométricos que de otro modo sería necesario calcular.



Figura 4. Captura de pantalla de *Alternate Reality: The City* (captura propia desde emulador online, fuente: <https://www.myabandonware.com/game/alternate-reality-the-city-eh/play-eh>).

No obstante, al hacer avanzar o retroceder la cámara, la altura y las texturas de los muros **sí se van actualizando fluidamente** acorde a la distancia de los rayos lanzados por la cámara, dando una realista sensación de profundidad.

Como pequeño detalle, cabe destacar que, pese a avanzar suavemente cuando el jugador se va desplazando por el mapeado —que está estructurado en forma de cuadrícula—, al girar la cámara se le sitúa en el centro de la “casilla” en la que éste se encontraba antes de girar.

2.2.2 Wolfenstein 3D (1992)

Pese a que, como ya se ha presentado, no es el primer juego que renderizó gráficos mediante emisión de rayos, sí es el que hizo que dicho sistema ganara la fama que muchos *shooters* de la época ganaron en su día.

⁴ Se puede jugar a *Alternate Reality: The City* desde el navegador (de forma gratuita) en esta página web: <https://www.myabandonware.com/game/alternate-reality-the-city-eh/play-eh>

Wolfenstein 3D, desarrollado por ID Software, presentó un sistema de cámara que, a diferencia de *Alternate Reality: The City*, sí permitía al jugador girar la cámara hacia izquierda o derecha con suavidad, en vez de recurrir al “salto” de 90 grados.



Figura 5. Wolf3D (fuente:
<https://web.archive.org/web/20101123162338/http://idsoftware.com/games/wolfenstein/wolf3d/>)

No obstante, esta excelente simulación de un entorno en tres dimensiones no se limitó a renderizar gráficos o su experiencia jugable como FPS, sino que ofrecía otros pequeños “trucos” y efectos que agregaban aún más detalle a los entornos visuales: por si las cuidadas texturas no fueran suficientes, los rayos almacenaban la orientación del muro con el que chocaban.

De este modo, si la pared “mira” hacia arriba o abajo, se puede oscurecer la *vLine* que le corresponde, logrando una falsa sensación de iluminación en la escena, como muestra la Figura 6.

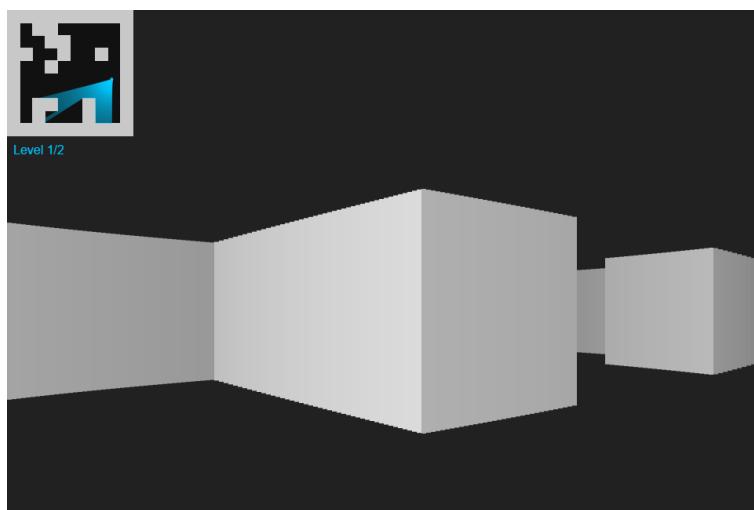


Figura 6. Falso efecto de iluminación en el renderizado de *vLines*, según la orientación de los muros.

Por si no fuera suficiente, también puede añadirse un efecto de degradado oscuro: cuanto más alejado esté el jugador de la pared (es decir, a mayor espacio recorran los rayos de la cámara), se ennegrece el color de la *vLine* (o de su textura), para enfatizar aún más en su lejanía.

Mediante este procedimiento se obtiene un efecto de degradado en función de la distancia de los rayos emitidos por la cámara, como se puede apreciar más fácilmente en los dos muros indicados en la Figura 7.

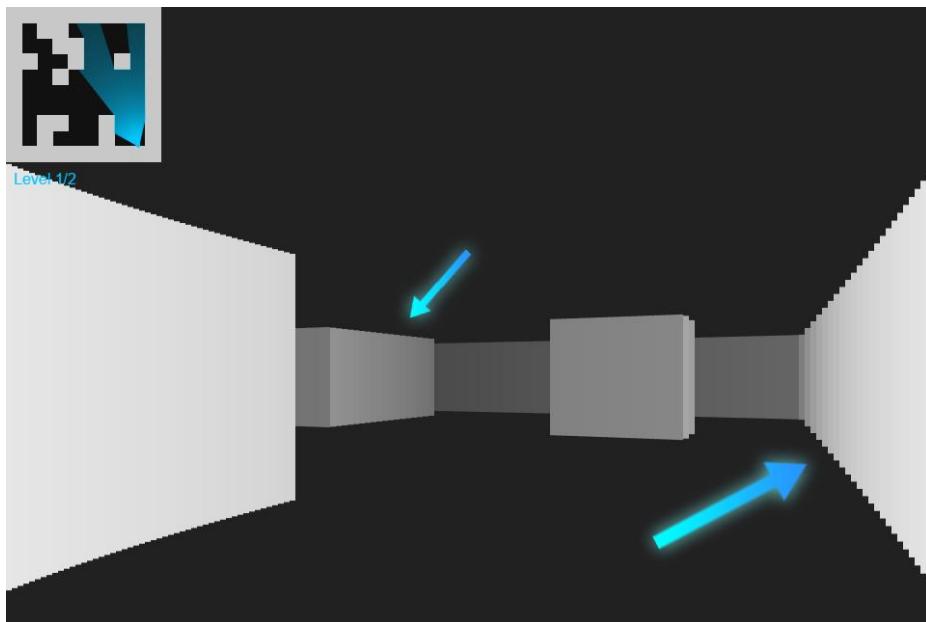


Figura 7. Falso degradado de iluminación en los muros, según la distancia de cada rayo al colisionar. A mayor distancia, *vLine* más oscura.

No obstante, combinar este último efecto de degradado con las texturas de las paredes tendría un gran impacto en el rendimiento y consumo de recursos del juego, por lo que se puede obviar del procedimiento.

También cabe destacar que *Wolfenstein 3D* mostraba a los enemigos, ítems y otros elementos decorativos del escenario como *sprites* o animaciones planas siempre enfocando a la cámara (como el soldado enemigo de la Figura 5 o la lámpara que cuelga del techo).

Por último, ofrecía efectos interesantes como muros secretos que se desplazaban cuando el jugador los tocaba, y puertas que conectaban distintas habitaciones del mapeado, también en contraposición al sistema de *Alternate Reality: The City*, donde el sistema simplemente mostraba una imagen estática de una sala al atravesar las puertas o acceder a tiendas.

2.3 Nuestro engine

Ahora que se han mostrado las principales referencias en el campo del *raycasting*, se procederá a explicar las funcionalidades y bases del motor 2D a programar en esta primera mitad del documento.

Se puede ver e interactuar con el *raycaster* que implementaremos en el siguiente enlace: <https://emever.github.io/final-grade-raycaster.github.io/>

En la Figura 8 se muestra un posible resultado final del proyecto, aunque cada usuario podrá diseñar sus propios niveles, texturas y degradados de cielo y suelo (estos dos últimos habitualmente llamados *floor* y *ceiling*).

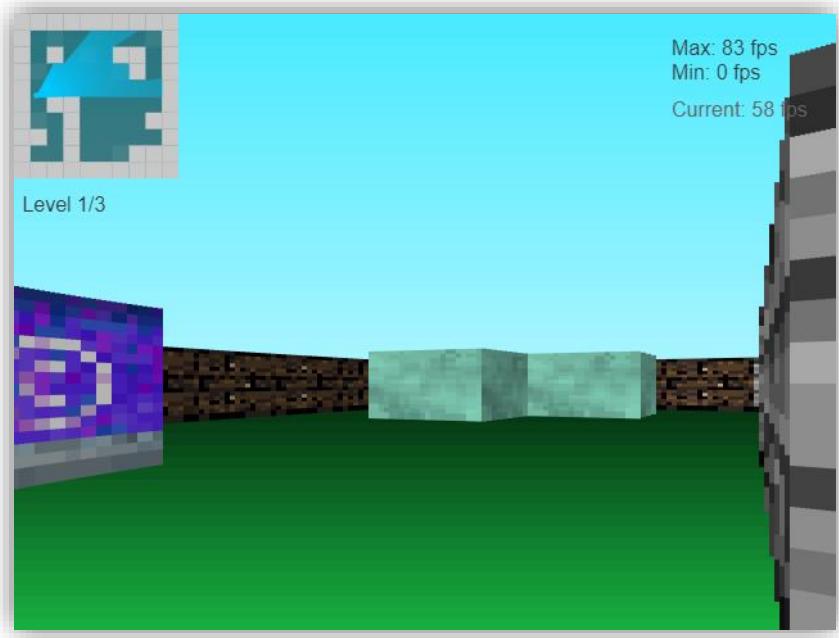


Figura 8. Versión final del raycaster a implementar en este proyecto, nivel 1 (captura propia del enlace <https://emevev.github.io/final-grade-raycaster.github.io/>)

A nivel de implementación, el código fuente está programado en web, empleando JavaScript para hacer todos los cálculos del motor y dibujar sobre un canvas. Se decidió así para comprobar los prometedores resultados que ofrece esta técnica de lógica y renderizado en un entorno completamente accesible y que todos los posibles usuarios o desarrolladores son capaces de explotar, dado que no se necesita de ningún programa determinado o *software* particular para su implementación: puede programarse desde el mismo bloc de notas de Windows, y abrirse en cualquier navegador.

Además, JavaScript fue el lenguaje de programación más demandado por las empresas tecnológicas en 2020 [9], así como uno de los considerados como más sencillos de aprender [8]. De este modo, incluso un perfil junior o estudiante con únicamente conocimientos básicos podrá codificar el proyecto y aprender mientras lo hace.

Ahora que se ha ofrecido un vistazo al aspecto del motor (Figura 8), profundizaremos en qué funciones llevará a cabo nuestro *engine*, más concretamente.

2.4 Funcionalidades (en orden de implementación)

Como se mostró anteriormente en la Figura 1, se puede considerar todo el proceso de implementación como un proyecto de tres fases, siendo la primera la **programación del raycast** de cara al jugador (o cámara), la segunda la **proyección de muros** en función de la distancia de los rayos, y la última la aplicación de **texturas** en dicha proyección.

Además, en cada una de dichas fases se ha dividido su desarrollo en procedimientos más concretos:

Fase 1: implementación del *raycaster*.

- **Generación de un nivel.** A partir de una matriz numérica, crearemos un nivel para el motor.

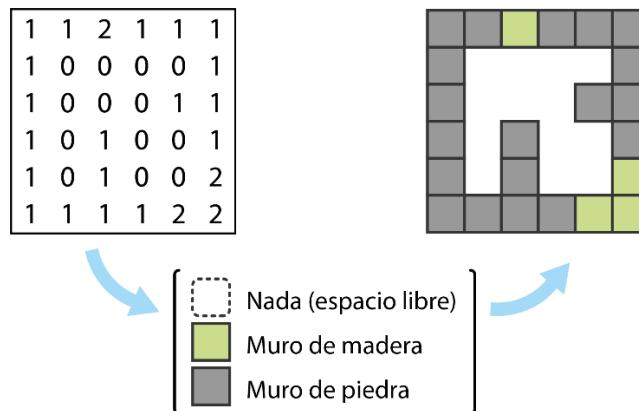


Diagrama 1. Proceso de creación de nivel desde una matriz numérica (diagrama propio).

- **Movimiento (y giro) del jugador.** Una vez ya se ve el mapa generado en el *canvas*, o pantalla, se creará la clase inicial del jugador (a la que más adelante se le irán añadiendo más atributos y funciones) y se colocará en el nivel. Al principio, tras hacer que el jugador pueda girar el ángulo en que está mirando, sus controles serán de tipo tanque.
- **Raycasting, y colisión de rayos.** Dado que el jugador ya puede girar y moverse, entrará en juego la implementación del campo de visión del jugador, y su colisión con los muros del nivel.
- **Generación de una escena multinivel.** Modificando ligeramente la clase “Map”, se generarán otros niveles en la misma escena, como si el jugador fuera subiendo o bajando de piso por un edificio, alternando de nivel.



Figura 9. *Raycaster* con lectura de varios niveles y colisión de rayos implementada. Resultado final de la fase 1 de implementación del *raycaster* (captura propia del *canvas* del proyecto).

Fase 2: proyección de los rayos y renderizado de “vLines”.

- **Proyección.** Mediante la distancia de los rayos emitidos por la cámara, se dibujarán en el *canvas* las “vLines”, simulando ser las paredes del nivel.
- **Controles de tipo libre.** Se implementará el sistema de control normal de los FPS actuales, siendo más cómodo e intuitivo para navegar por el nivel (visto en la Figura 2).

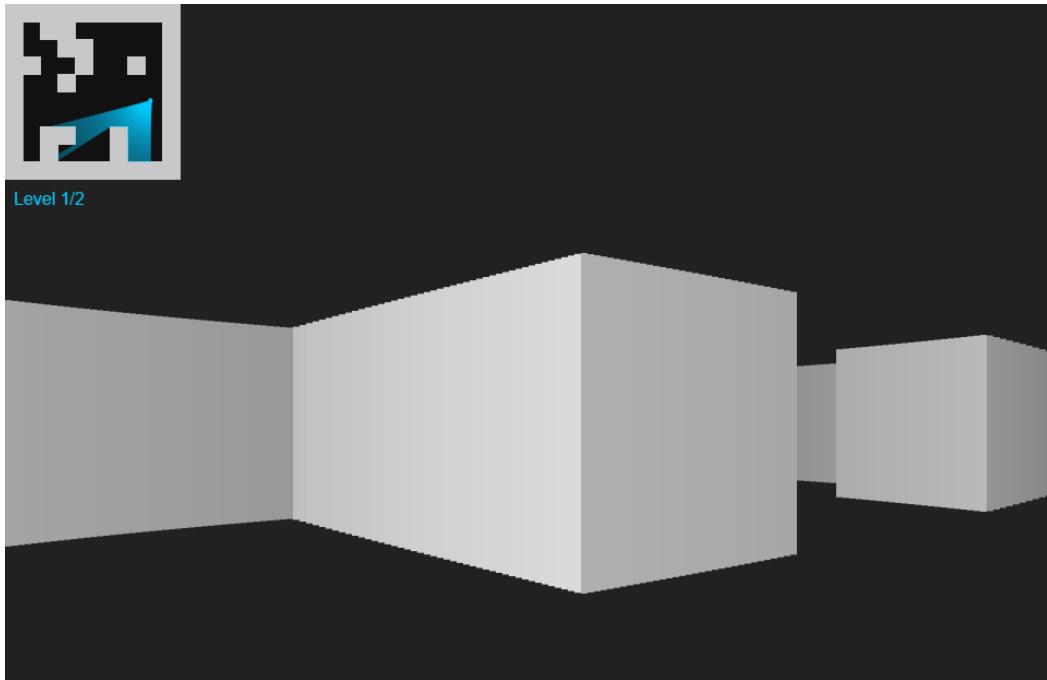


Figura 10. *Raycaster* con la proyección de muros terminada —renderizado de "vLines"—. Resultado de la fase 2 de implementación del *rayaster* (captura propia del *canvas* del proyecto).

Fase 3: tratado de imágenes.

- **Lector de imágenes.** Terminada la fase dos, el sistema ya estará generando las “vLines” para renderizar los muros y el proceso de *raycasting* ya estará terminado. Únicamente quedará pulir el aspecto visual. Por eso lo siguiente será crear la clase encargada de leer imágenes y almacenarlas en un formato que el código sea capaz de leer y asociar rápidamente, en vez del formato imagen estándar de JavaScript.
- **Generación de una escena multinivel por imágenes.** Aunque al principio se generaban los mapas a partir de matrices numéricas, ahora se aprovechará el lector de imágenes para generar niveles a partir de una imagen, asociando el color de cada píxel a un tipo determinado de pared.
- **Aplicar texturas en los muros.** Por último, se aplicarán las imágenes de textura a los muros y se colorearán el techo y el suelo, terminando así la implementación completa del motor gráfico 2D.

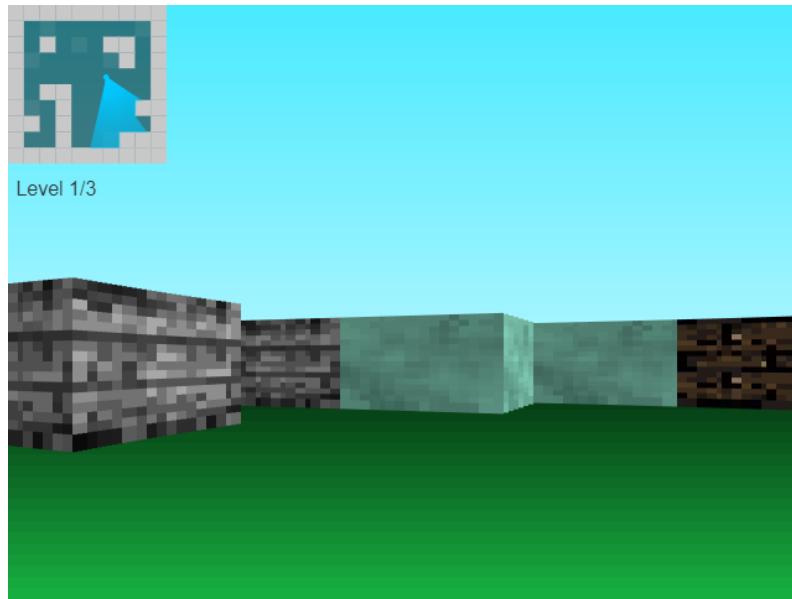


Figura 11. Resultado final de la fase 3 de la implementación del *raycaster* —con tres posibles texturas de ejemplo— (captura propia del *canvas* del proyecto)

Los ficheros con los que contará el *raycaster* en su versión final son los siguientes (en el siguiente punto se entrará en detalle de las funcionalidades de cada clase):

- **main.js**
- **renderController.js**
- **keyController.js**
- **mapController.js**
- **playerController.js**
- **imageLoader.js**
- **utils.js**
- **p5.js**
- **styles.css**
- **index.html**

2.5 Diagrama de archivos y clases

Para ahorrar algo de tiempo y pasar directamente al código del raycaster, utilizaremos la **librería P5.js** [26], que ofrece varias funciones lógicas indispensables ya preparadas para su uso, además de otras que se utilizarán para renderizar líneas y polígonos en pantalla.

Hay tres documentos de texto que no hará falta modificar una vez empieza el proceso de implementación: en el Diagrama 2 aparecen y se explica su principal función.

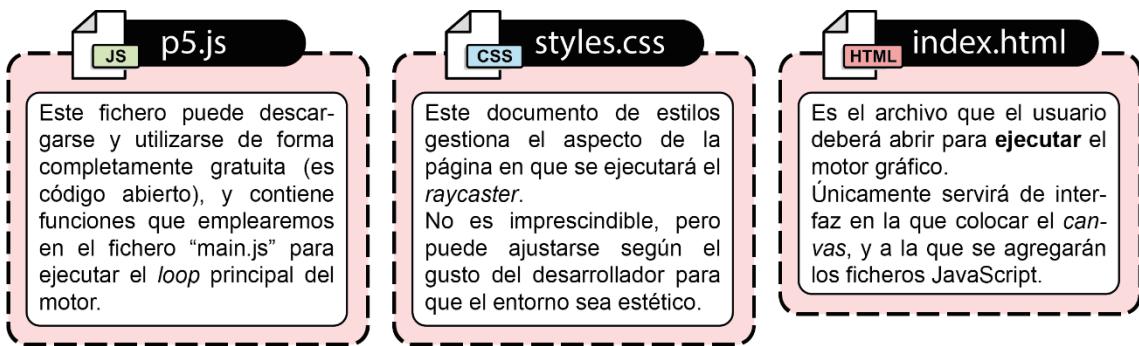


Diagrama 2. Documentos invariantes del proyecto: librería "p5.js", hoja de estilos "styles.css" y archivo ejecutable de navegador "index.html" (diagrama propio).

Como primer vistazo general al proyecto, en el Diagrama 3 se puede apreciar la estructura de archivos, así como las clases implementadas en cada fichero, y sus atributos y funciones.

Cabe destacar que los documentos “main.js”, “keyController.js” y “utils.js” no definen ninguna clase, sino que implementan funciones globales del proyecto. Para terminar, además de los ficheros descritos, también habrá una carpeta donde guardar las imágenes (las texturas y las que use usarán para generar cada nivel).

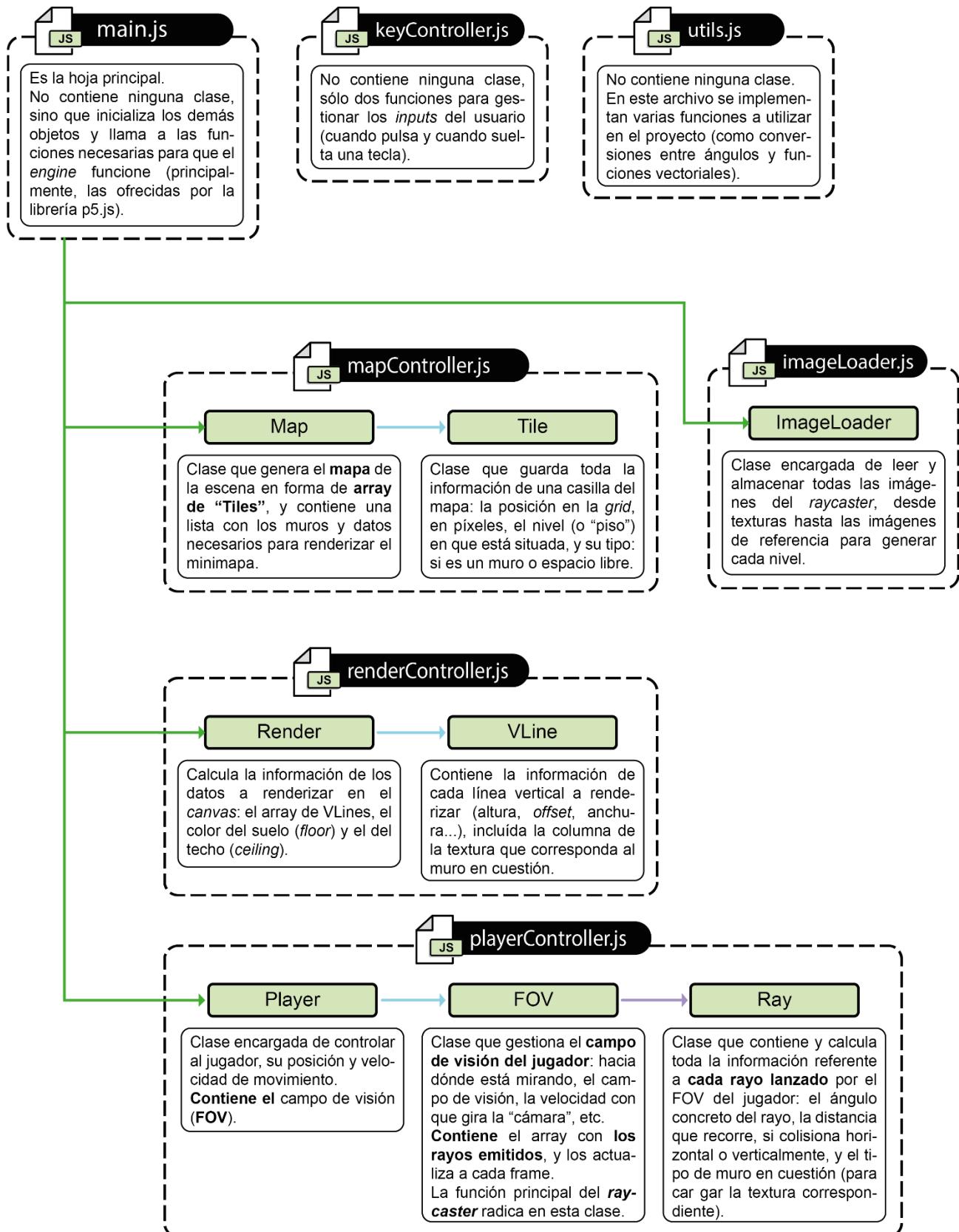


Diagrama 3. Esquema con los ficheros del proyecto y las clases definidas en cada uno de ellos, así como su función en el proyecto.

3 Implementación del *raycaster*

3.1 Ficheros

Dado que ya se ha decidido la estructura de archivos del proyecto, se pueden crear los ficheros al principio y asignarlos ya al “index.html”. Aunque más adelante se irán modificando cuando sea necesario, hay tres archivos que no será necesario alterar si los preparamos al principio.

Éstos son la librería P5.js, el “index.html” y la hoja de estilos “styles.css”. Además de ellos, en este apartado también se explicarán las funciones que se declararán en el “main.js”.

3.1.1 *P5.js*

Esta librería de funciones puede descargarse desde la página web de la comunidad de P5.js: <https://p5js.org/es/>. Es código abierto, libre de uso.

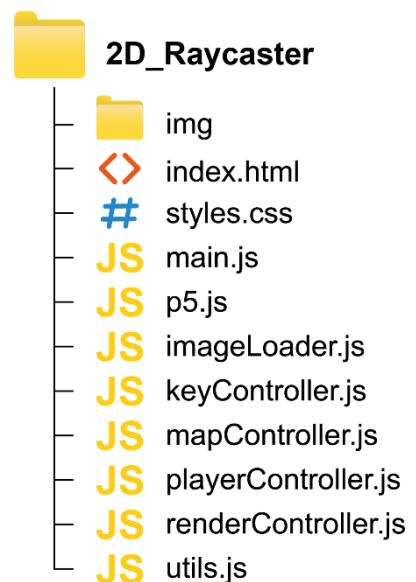
3.1.2 *Index.html*

El primer documento que se tendrá que escribir es el “index.html”: tras crear los demás archivos del proyecto (en el Diagrama 4 se puede ver la lista completa, con sus formatos) y asignarlos en el bloque *head*, necesitaremos agregar la etiqueta de *body*, como si fuera una página web corriente. Lo habitual sería crear un bloque que contuviera un *canvas* sobre el que renderizar el motor; sin embargo, la dejaremos sin contenido, porque la librería “p5.js” ya contiene una función que crea un *canvas* y lo introduce en el *body* automáticamente.

El orden de inclusión de los archivos en el *index.html* es importante, sobre todo en cuanto a las clases a crear y las variables globales que utilizaremos (como la cantidad de rayos a lanzar en el FOV del jugador, el espaciado entre ángulos, etc.). Si en “mapController.js” se hace una llamada a una variable definida en “renderController.js”, el sistema dará un error, dado que en el momento de lectura aún no existe en memoria.

Por ello se recomienda el orden presentado a continuación:

```
<html>
  <head>
    <link rel="stylesheet" href="styles.css"/>
    <script src="p5.js"></script>
    <script src="utils.js"></script>
    <script src="mapController.js"></script>
    <script src="keyController.js"></script>
    <script src="playerController.js"></script>
    <script src="renderController.js"></script>
    <script src="main.js"></script>
  </head>
```



```
<body>
</body>
</html>
```

3.1.3 Styles.css

La hoja de estilos sigue la estructura habitual, con la designación de los elementos a modificar visualmente y los códigos de color estándares (*P5.js* únicamente ofrece funciones a nivel de JavaScript).

En cuanto a la tonalidad a aplicar en el entorno del *engine*, un aspecto estético oscuro en el navegador es más cómodo para la vista que uno claro. Aunque cada desarrollador puede definir el que mejor le convenga, en la Figura 12 se muestra algunos posibles colores de referencia para el fondo del sistema:

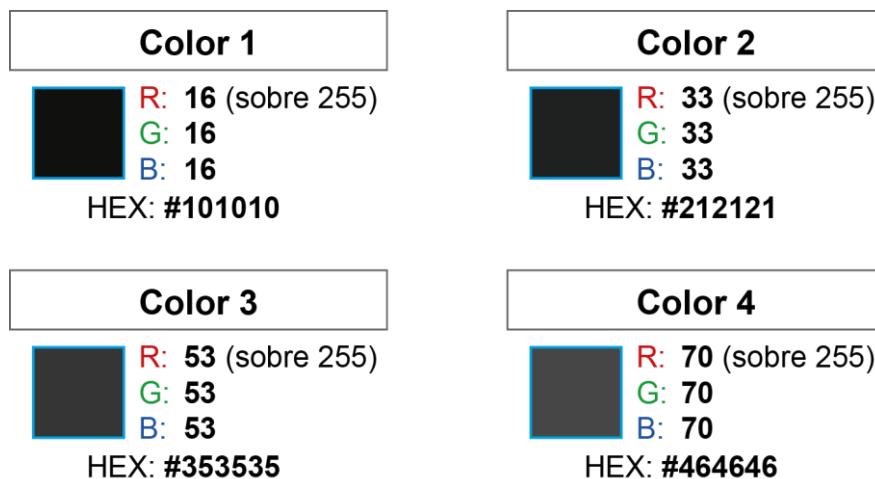


Figura 12. 4 códigos de color para el entorno del *raycaster*, tonalidades grises, de más oscuro a más claro.

De este modo, tanto el fondo de la página que se abrirá en el navegador como el del *canvas* (al que también se le ha añadido un fino borde blanco, para distinguirlo más fácilmente) son oscuros.

3.1.4 Main.js

El archivo “main.js” será quien contenga las instancias de las clases que se utilizarán en el proyecto; sin embargo, por ahora simplemente llamaremos a dos de las funciones que la librería *P5.js* ofrece: **setup** (función que se ejecuta al principio, al abrir el *index.html* con el navegador, y donde instanciaremos las variables principales como *Player*, *Map* y *Render*), y **draw** (función que utilizaremos a modo de *loop* principal del motor, ya que se ejecuta —por defecto— a una velocidad de 60 fps).

Además, definiremos una función **update** en la que actualizaremos las variables y la lógica del *engine*, a la que llamaremos en **draw** después de borrar el *canvas* y repintar el fondo, de modo que, a cada frame, primero se actualizarán las variables y luego se renderizará la escena.

El aspecto inicial del código de *main.js* debería ser similar al siguiente:

```
// MAIN.JS
// carga inicial del proyecto (se ejecuta al principio)
```

```

function setup() {}

// actualización de variables (60 fps)
function update() {}

// renderizado en el canvas (60 fps)
function draw() {
    // (1) limpiar el canvas al principio del loop
    clear("#212121");
    // (2) pintar fondo del canvas (gris oscuro)
    fill("#565656");

    // (3) llamar a la función donde se actualizan las variables
    update();
}

```

3.2 Generación de un nivel

El primer ítem de la lista de funcionalidades a desarrollar en este proyecto es la creación de un mapa (o “nivel”). Poder generar de manera intuitiva y rápida distintas escenas para un videojuego es un elemento imprescindible para conseguir resultados convenientes en un tiempo adecuado, y para ello se necesita un archivo origen como referencia (puede ser una matriz de números o caracteres, un mero documento de texto con la estructura escrita por zonas, una imagen...).

Por ahora se utilizará como base una matriz numérica, como se ha mencionado anteriormente y se ilustró en el Diagrama 1 (página 30). Este nivel tendrá una estructura de cuadrícula clásica, como la vista en *Wolfenstein 3D*, de modo que primero deberemos definir un tamaño para cada casilla del mapa (o *Tile*). Los valores más comunes en estos casos suelen ser múltiples de ocho: 16, 32, 64... El de ejemplo tendrá un tamaño de 32 píxeles de alto y ancho, y lo guardaremos en una constante llamada “*TILE_SIZE*”. Este valor lo definiremos en el fichero “*mapController.js*”, ya que está relacionados con el mapa.

3.2.1 Cambios en *main.js*

En el *main.js*, primero crearemos la variable global *Map* y la instanciaremos. Ésta será la encargada de gestionar la creación de la escena o nivel del *raycaster*. Después, en el *setup* llamaremos a la función “*createCanvas*” de la librería *p5.js*, que creará un canvas con la altura y anchura (en píxeles) que indiquemos por parámetros y llamaremos a una función de la clase *Map* que genere el nivel de la escena (“*objMap.init*”). Por último, en el *render* llamaremos a la función de renderizado del mapa.

Volviendo a la función “*createCanvas*”, el tamaño en píxeles del mapa (y, por tanto, del canvas) equivale al tamaño decidido para las casillas de la matriz multiplicado por las dimensiones de la matriz, como se muestra en el código, a continuación, y en el Diagrama 5 de forma más visual:

```
createCanvas(objMap.width * TILE_SIZE, objMap.height * TILE_SIZE);
```

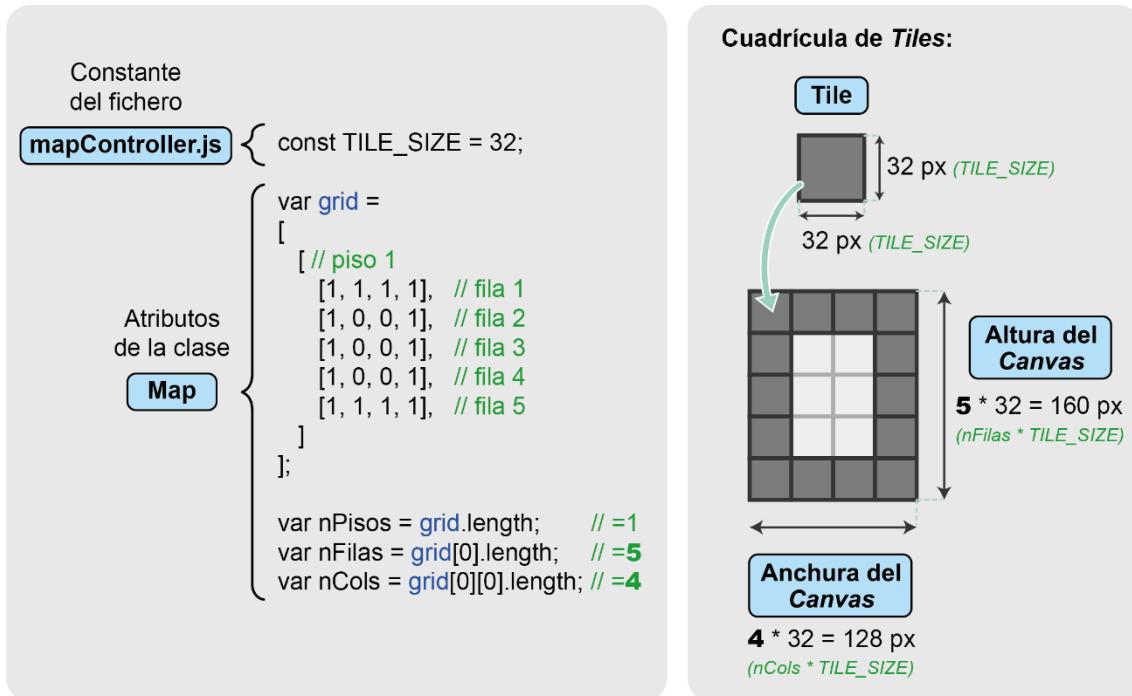


Diagrama 5. Cálculo del tamaño del *canvas* según la constante *TILE_SIZE* (diagrama propio).

La variable *grid* es la matriz numérica que representará el nivel de la escena.

3.2.2 Implementación de *mapController.js*

Después de decidir un tamaño para las casillas del mapa (*TILE_SIZE*), se implementarán las clases *Map* y *Tile*, ambas del fichero *mapController.js* y encargadas de gestionar el nivel del *engine* (y renderizar el mini mapa).

Es indispensable que la clase *Map* contenga una variable para almacenar la matriz numérica mencionada previamente (en el Diagrama 5, *grid*), ya que será el *array* que leerá el sistema y a raíz del cual generará la lista de casillas que formarán el mapa. También es útil almacenar tanta información como sea posible de los niveles a generar, como su altura y anchura (o bien en tamaño de filas y columnas de la matriz, o directamente en píxeles —o ambas), el número de niveles de la escena (de este modo se puede dejar preparado para la escena multinivel, más adelante).

Del mismo modo, es recomendable crear un *array* con todos los muros de la escena, de modo que sea más fácil ir a buscar los muros al *array* antes que ir comprobando las coordenadas en la matriz del mapa.

El desarrollador deberá implementar la función que **leerá la matriz numérica** (piso a piso, fila a fila y columna a columna) y **creará una lista de casillas** (que son las que utilizaremos a modo de mapa). Después, por cada índice de la lista añadirá una *Tile* al *array* de casillas de la clase *Map* (el atributo *tileset*). Cada una de estas *Tiles* contendrá toda la información necesaria, de modo que no será necesario ir accediendo constantemente a la *grid*, que es una matriz de tres dimensiones, sino que se podrá recorrer un *array* de una dimensión y acceder a sus atributos nivel, posición en la grid (horizontal y verticalmente), posición en píxeles (también horizontal y vertical) y tipo de casilla. En el Diagrama 6 se ilustra el proceso (de arriba abajo).

El paso 1 de dicho diagrama es el que ya se ha implementado en el constructor de la clase *Map*, mientras que el paso 2 es el que se lleva a cabo en la función “loadTileset”, de la misma clase.

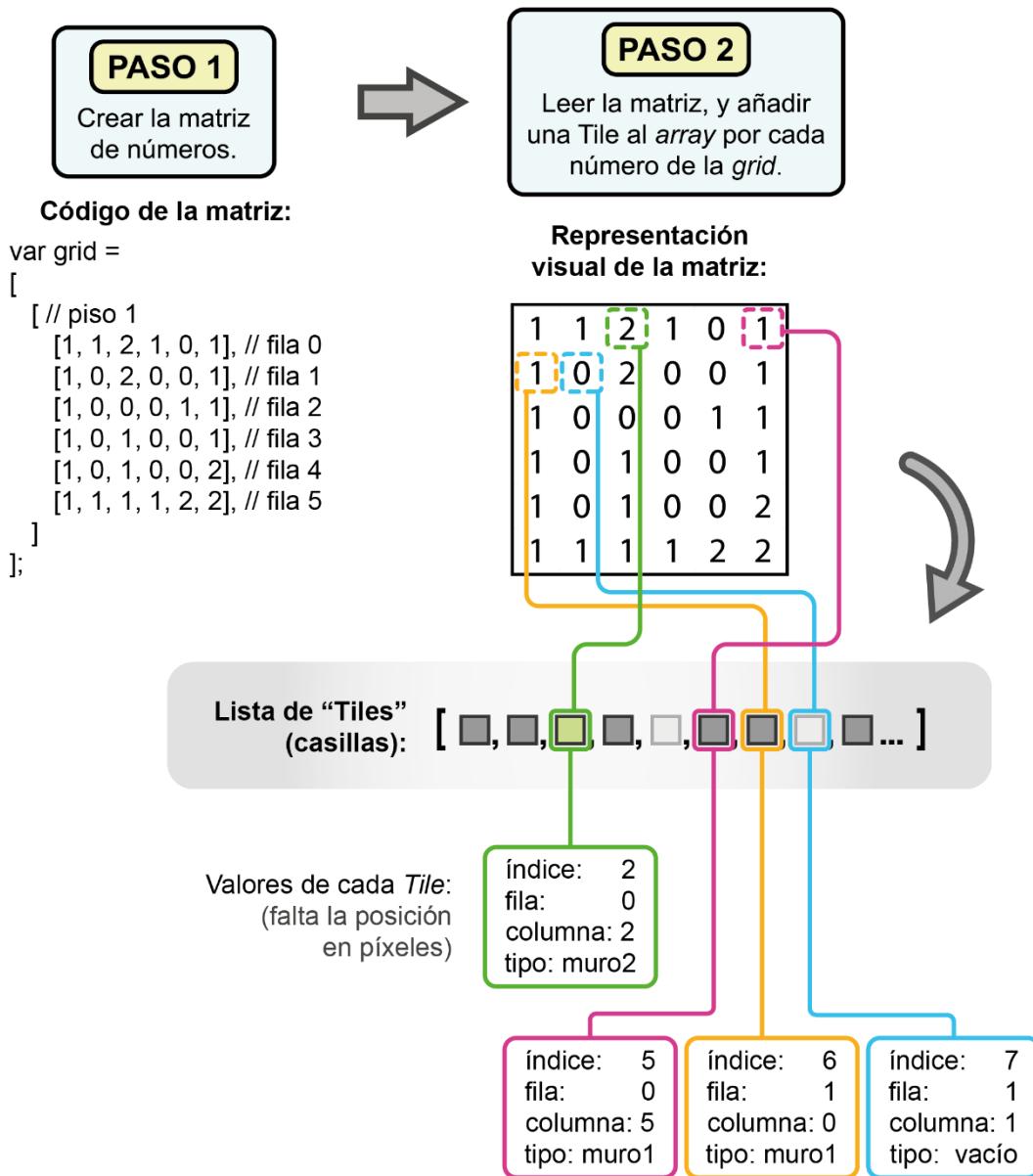


Diagrama 6. Generación del array de *Tiles* a partir de la *grid*. El paso 2 corresponde a la función *loadTileset*, de la clase *Map* (diagrama propio).

Por otro lado, “LoadWallset” rellena una lista con los muros del mapa (sólo añade al array las ya creadas, no crea un duplicado), a diferencia de la función “loadTileset”, que también crea casillas para los espacios en blanco del nivel. Esto será óptimo para cuando se deba detectar colisiones más adelante, pudiendo acceder a la lista de muros directamente, en vez de ir revisando qué ítems del array completo son sólidos (“wall”) y cuáles no (“none”).

El último método de la clase *Map* (por ahora) es la función *render*, que sólo recorre la lista de *Tiles*, y llama a la función *render* de cada casilla para mostrar una visual del mapa.

Hasta aquí la implementación de la clase *Map*. Ahora queda la clase *Tile*, cuyos atributos ya hemos descrito anteriormente. Cabe destacar que todos los **índices** (como las posibles posiciones, etc.) **empiezan siempre en el cero**. Por ejemplo, la variable “level”, a estas alturas siempre será cero, dado que aún no hemos implementado múltiples niveles en la misma escena.

Para el renderizado de las casillas se utilizará la función “fill”, que recibe el color con que se quiere pintar el próximo elemento a renderizar en el *canvas* (en este caso, los cuadrados que formarán el mapeado). Aunque admite varios formatos, en el proyecto se usa el estándar *rgba* (*red, green, blue, alpha*), con valores enteros entre 0 y 255 (inclusive). Cuando *alpha* tiene un valor de 255 significa que no debe tener transparencia, mientras que 0 significa que será completamente invisible.

En el sistema se puede utilizar este valor de transparencia para distinguir entre los muros que están al mismo nivel que el jugador y los que no, como se muestra en el código, a continuación:

```
if(this.class == "wall") fill(200,200,200,255);
else if(this.class == "none") fill(200,200,200, 0);
rect(
    this.xPos, // coordenada x inicial
    this.YPos, // coordenada y inicial
    TILE_SIZE, // anchura del rectángulo
    TILE_SIZE // altura del rectángulo
);
```

Después de decidir el color del objeto a dibujar en el *canvas*, se usa la función “rect”, también de *P5.js* para renderizar un rectángulo (o cuadrado, en este caso), en las coordenadas determinadas por parámetro, así como el tamaño, todo en píxeles.

Con esto termina el código para la creación del canvas y generación de un nivel. En la Figura 13 se observa un posible resultado, correspondiente a la grid indicada en la misma imagen.

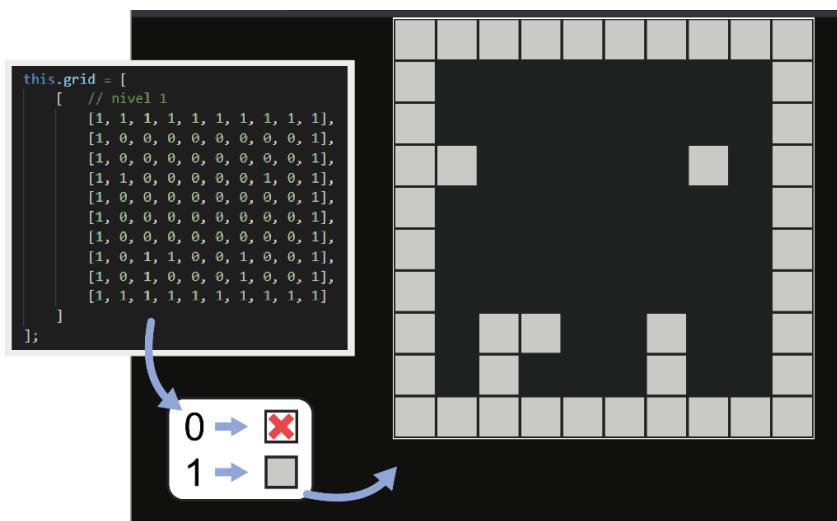


Figura 13. Resultado de la generación de un nivel mediante matriz numérica (capturas y diagrama propios).

3.3 Movimiento y ángulo del jugador

Después del mapa, la lógica del jugador es indispensable para el *engine*. Parecido a como se ha hecho con el anterior punto, primero veremos los cambios en el *main.js* y luego implementaremos la clase *Player*.

3.3.1 Cambios en *main.js*

En el main, vemos los cambios siguientes:

- Justo después de instanciar el mapa (“new Map();”) llamaremos al **constructor** de la clase **jugador** (*Player*). Los parámetros de la función equivalen a la posición en píxeles donde se situará inicialmente (en el ejemplo, 160 píxeles tanto en el eje de coordenadas horizontal como en el vertical), y el nivel inicial (dado que por ahora nuestra escena sólo cuenta con un nivel, será 0⁵).
- Como éste será el jugador controlado por el usuario, actualizaremos sus atributos, en la función *update* del *main.js*: su posición en píxeles, el nivel de la escena en que está (es decir, los valores enviados como parámetros) y el ángulo en que está mirando.
- La última llamada a añadir en el *main.js* es la del renderizado del jugador, en la función *draw*. Igual que con el mapa, se implementará un procedimiento encargado de pintar al jugador sobre el mapa (es importante llamar a las funciones en este orden: primero renderizar el mapa y luego el jugador, para que éste se pinte por encima, en el *canvas*).

3.3.2 Primeras funciones de *utils.js*

En la clase *Player* se empezarán a ver ángulos, y las funciones matemáticas que se llamarán en el código utilizarán los radianes como unidad de medida, de modo que será extremadamente útil implementar una función global, para poder invocarla en cualquier parte del código, y que se encargue de la conversión de ángulos sexagesimales (es decir, los que dividen una circunferencia en 360 partes iguales) a radianes (sistema en que un giro completo equivale a dos veces pi unidades).

La fórmula es la mostrada en la Ecuación 1, donde θ_{rads} son los radianes (el resultado final), θ_{360} son los grados sexagesimales que se quieren convertir y π es el número pi:

$$\theta_{rads} = \theta_{360} \times \frac{\pi}{180}$$

Ecuación 1. Conversión de grados sexagesimales a radianes.

La función a implementar en *utils.js* tendría este aspecto (recibe como parámetro el ángulo en sexagesimales, *angle*):

```
// utils.js
function degreesToRadians(angle) {
    return angle * Math.PI/180;
}
```

⁵ Cabe recordar que todos los índices empezarán en el cero.

Pero los cambios en *utils.js* no acaban ahí: como se acaba de mencionar, empezaremos a ver operaciones con ángulos durante la implementación del jugador, y, así como la conversión, también deberemos asegurar que los ángulos no superarán nunca los 360 grados (o 2 veces π , en radianes). Aunque esta operación no es exactamente una normalización, definiremos la función como “normalizeAngle”, y se encargará de corregir el ángulo que reciba por parámetro (ya en radianes) por su valor correcto.

```
function normalizeAngle(angle) {
    angle = angle % (2 * Math.PI);
    // si el angulo es negativo, sumamos un giro completo (2*pi):
    if (angle < 0)
        angle = (2 * Math.PI) + angle;
    return angle;
}
// fin de utils.js
```

El “%” en JavaScript actúa como el módulo de la división. Utilizamos el módulo para obtener el residuo de la división entre 2 π y el ángulo que intentamos corregir (“angle”). Después comprobamos si el ángulo es negativo y, en caso afirmativo, le sumamos un giro completo para obtener el valor que le correspondía al anterior, pero en positivo.

En la Tabla 5 se puede ver el proceso que lleva a cabo la función “normalizeAngle” por columnas, paso a paso y para varias posibles combinaciones de valores:

Tabla 5. Proceso de corrección de ángulos (por columna), función de *utils.js*.

RECIBE: Valor de angle (parámetro)	PASO 1: Módulo entre 2 π “% (2*Math.PI)”	PASO 2: ¿Es negativo?	A DEVOLVER: Resultado final
0 radianes	0	No	0 radianes
1 radián	1	No	1 radián
2π radianes	0	No	0 radianes
3π radianes	π	No	π radianes
-1 radián	-1	Sí (+ 2π)	5.283185 radianes
5π radianes	π	No	π radianes
- 3π radianes	- π	Sí (+ 2π)	π radianes

3.3.3 Implementación de *playerController.js*

“PlayerController.js” es el fichero donde se definirán las clases que envuelven toda la lógica del jugador (incluido su campo de visión o FOV, que es lo que actúa como *raycaster*).

En este mismo fichero implementaremos, por ahora, dos clases distintas:

- **Player.** Es la clase que gestiona la posición y el movimiento del jugador (y, por tanto, también su velocidad de movimiento), y contiene además una variable de clase FOV, que será el campo de visión de la cámara.
- **FOV (field of view).** Es el campo de visión del jugador, y entre sus atributos se encuentran el origen desde el cual se “abre” el FOV (que es el jugador), el ángulo

concreto en que éste está mirando (“angleView”) y el *array* de rayos emitidos por la cámara (éste se implementará en el siguiente apartado, *raycasting*, en la página 46). También contiene la velocidad de giro del ángulo de la cámara.

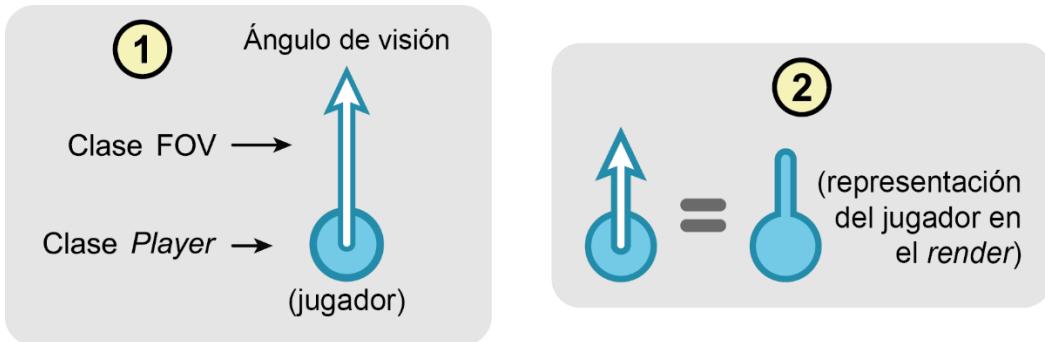


Diagrama 7. Bloque 1: Aspecto del jugador (definido en la clase *Player*) con el ángulo hacia el que mira (clase *FOV*). Bloque 2: aspecto del jugador y su ángulo en el renderizado (diagramas propios).

En el Diagrama 7 se puede observar (en la izquierda) la función de cada clase de una manera más gráfica y cómo se representarán al jugador y su ángulo de ahora en adelante (a la derecha).

El valor del ángulo del FOV (es decir, el ángulo en que está mirando el jugador) se considerará como se muestra en el Diagrama 8: en radianes, y comenzando desde el eje horizontal derecho, incrementándose en sentido opuesto a las agujas del reloj.

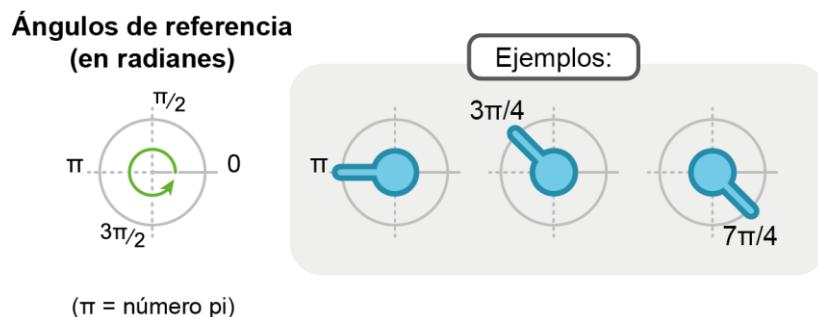


Diagrama 8. Ángulo de visión del jugador (en radianes, como la variable definida), con el sentido del giro, y tres ejemplos.

También es importante decidir una velocidad adecuada para el giro del FOV del jugador (aunque por ahora el FOV sólo contenga el ángulo del jugador y ningún *array* de rayos). En el código de ejemplo, el usuario estará mirando hacia arriba (ya que, aunque el valor de la variable “angleView” sea negativo, en JavaScript el eje Y del *canvas* está invertido, lo que hace que los valores inferiores estén arriba y los superiores abajo; como se ilustra en la Figura 14).

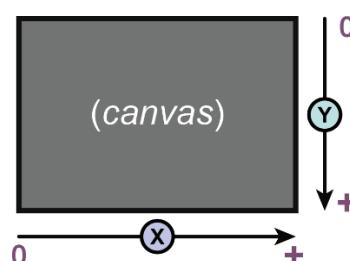


Figura 14. Incremento del eje de abscisas y de ordenadas en un canvas de JavaScript (diagrama propio).

Para marcar si el jugador gira a izquierda o derecha, se actualizará la variable “turnDirection” con un 1 o -1 cuando el usuario pulse la tecla correspondiente (de lo contrario, cuando el jugador no esté girando estará a cero), de modo que en la función *update* podemos simplemente multiplicar la velocidad de giro (“fovSpeed”) por este valor y el ángulo de la cámara cambiará correctamente hacia donde el usuario haya pulsado de forma directa, sin necesitar de condicionales para gestionar el *input*, como se muestra en el código a continuación (es la función *update* de la clase *FOV*):

```
update() {
    this.angleView += this.turnDirection * this.fovSpeed;
}
```

Además del constructor del *FOV*, se implementará su función *render*, que pintará una línea en la dirección en que el usuario esté mirando. Para definir su color se utiliza la función *stroke*, que recibe los valores *red*, *green* y *blue*, mediante enteros del 0 al 255; y, para renderizarla por el *canvas* está la función *line*, también de *P5.js*, que recibe como parámetros las coordenadas x e y orígenes del vector, y las coordenadas x e y destino.

Para calcular las coordenadas destino del vector que indica el ángulo de visión del jugador debemos utilizar el coseno (para x) y seno (para y) de dicho ángulo, como se indica en el Diagrama 9 (como dato adicional, los dos ejemplos mostrados en el diagrama pertenecen al set de ejemplos del Diagrama 8).

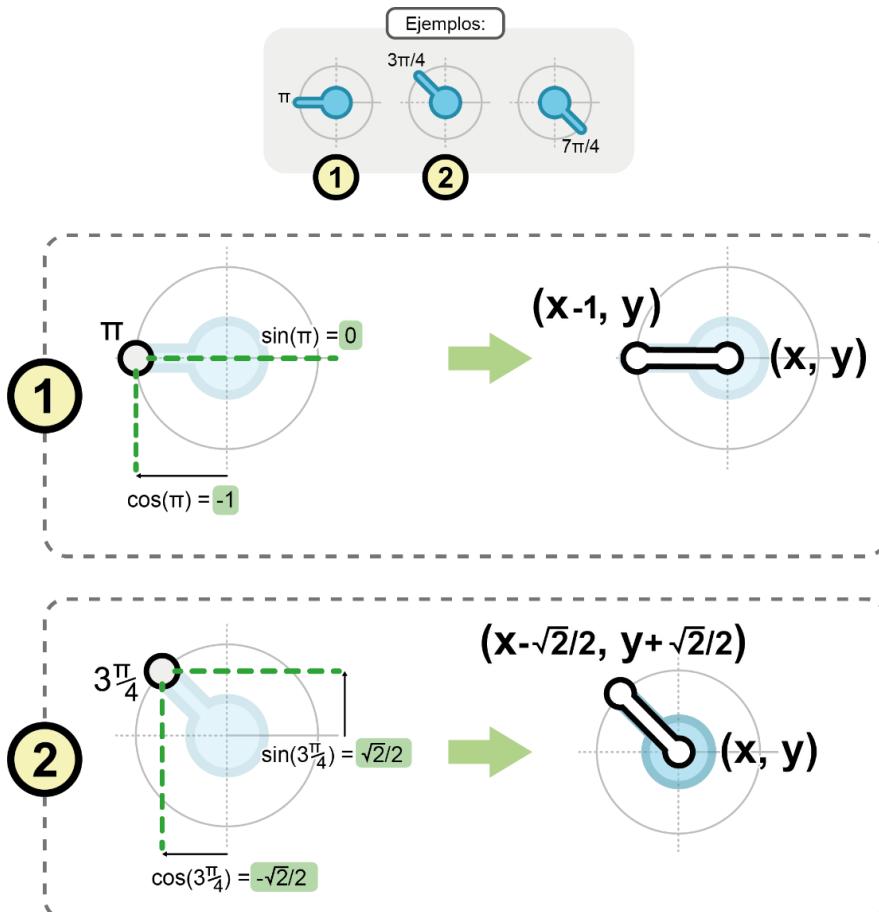


Diagrama 9. Cálculo trigonométrico del ángulo de visión del jugador (e incremento en las coordenadas destino del vector).

Después de definir la clase `FOV`, lo siguiente es implementar la clase `Player`, que gestionará la posición del jugador o cámara y su movimiento. Es importante no olvidar que `Player` contiene también el atributo `FOV`, campo de visión, de la clase recién implementada.

Parecido a como se hizo con el giro de la cámara, usaremos el mismo estilo de variables 0 (cuando el jugador no se mueve), 1 y -1 para el movimiento del jugador, para ahorrar condicionales en el `update`. De este modo, simplemente podemos actualizar el `FOV` del jugador y luego llamar a la función que mueve al usuario: “`movePlayer`” (el orden también es indispensable, de lo contrario el sistema estaría calculando el `FOV` de la cámara y luego modificando su posición).

```
update() {
    // actualizamos la posición del jugador
    this.movePlayer()
    // actualizamos los rayos del fov del jugador (posición, etc.)
    this.fov.update();
}
```

También coincide el concepto de la trigonometría para mover al jugador en función del ángulo en que mira. De este modo, cuando el usuario quiera avanzar, el sistema incrementará el valor de las coordenadas en función del seno o coseno, la velocidad de movimiento del jugador y la dirección en que avanza (si hacia delante o hacia atrás).

```
movePlayer() {
    this.x += this.walkDirection * this.movSpeed *
Math.cos(this.fov.angleView);
    this.y += this.walkDirection * this.movSpeed *
Math.sin(this.fov.angleView);
}
```

Puede verse que, dado que por defecto si el jugador está quieto el valor de “`walkDirection`” es cero, el incremento a aplicar en `x` e `y` se vuelve cero y éste no se desplaza.

Para terminar con la clase `Player`, sólo queda hacer la función de renderizado (donde también llamamos a la función `render` del `FOV` del usuario). Al usar la función “`circle`” de `P5.js` para pintar al jugador, se puede abrir el `index.html` para obtener un resultado como el de la Figura 15.

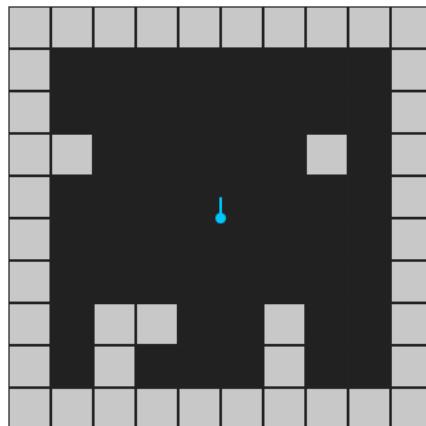


Figura 15. Renderizado del `canvas` tras implementar la clase `Player` y `FOV`, aún sin array de rayos (captura propia).

En la Figura 15 puede verse, además del mapa que ya se implementó en el anterior apartado, el círculo de color que representa al jugador y la línea que indica el ángulo en que está mirando.

3.4 *Player input* y movimiento del jugador

Ahora que ya se ha implementado la parte básica del jugador, pasaremos a definir la clase que gestionará los controles del usuario, de modo que el usuario pueda moverse por la escena. Después programaremos la colisión del jugador contra los muros del nivel.

Como recordatorio, como se ha comentado anteriormente, a estas alturas de la implementación los **controles** del jugador serán de **tipo tanque**.

3.4.1 Implementación de *keyController.js*

En el fichero “keyController.js” no se implementará ninguna clase, sino que se llamará a dos funciones de la librería *P5.js* para detectar si el usuario ha pulsado alguna tecla (“keyPressed”) o, por el contrario, la ha soltado (“keyReleased”).

Primero almacenaremos en constantes los valores identificativos de las teclas que se utilizarán en el sistema de control. En el ejemplo:

```
const KeyW = 87;
const KeyA = 65;
const KeyS = 83;
const KeyD = 68;
```

- “W” para avanzar en la dirección en que está mirando el jugador.
- “S” para retroceder en la dirección en que está mirando el jugador.
- “A” para girar la cámara hacia la izquierda.
- “D” para girar la cámara hacia la derecha.

```
function keyPressed() {
    if (keyCode == KeyW) objPlayer.walkDirection = 1;
    else if (keyCode == KeyS) objPlayer.walkDirection = -1;
    else if (keyCode == KeyD) objPlayer.fov.turnDirection = 1;
    else if (keyCode == KeyA) objPlayer.fov.turnDirection = -1;
}
```

Puede verse que los valores que pasamos a “walkDirection” y “turnDirection” son los que se decidieron previamente, en “playerController” (1 para avanzar y girar a la derecha, y -1 para retroceder y girar a la izquierda).

Del mismo modo, en la función “keyReleased” se igualarán todas esas variables a cero (dado que ya no es necesario mover al jugador o girar la cámara).

Ahora el desarrollador puede abrir *index.html* y mover al jugador por el nivel en función del ángulo en que mira, aunque aún no se han gestionado las colisiones con los muros de la escena, como se puede ver en la captura 2 de la Figura 16.

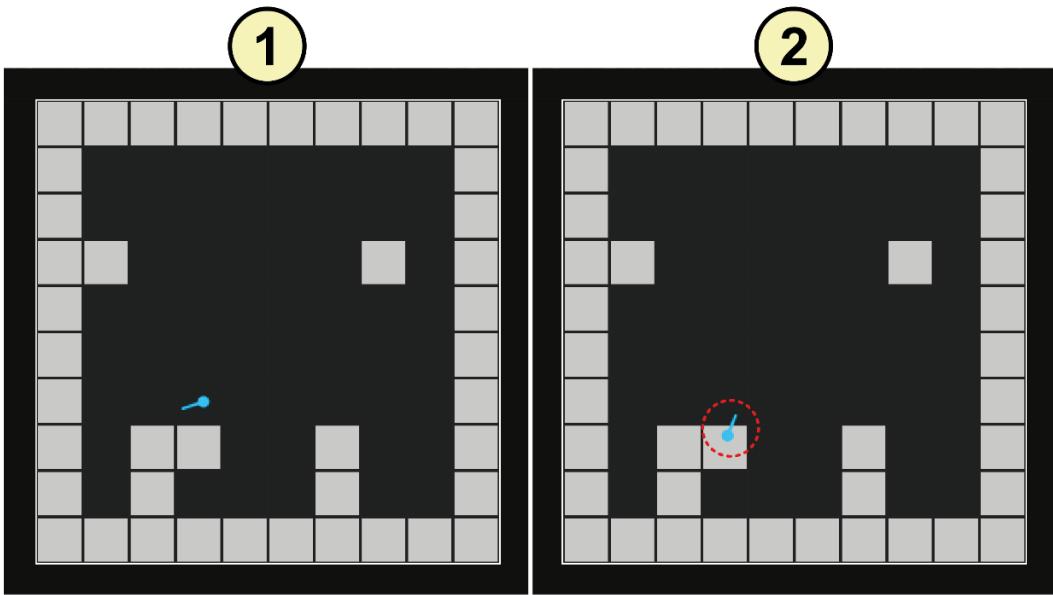


Figura 16. Clase *Player* con controles de usuario para el movimiento y giro de cámara. Sin colisiones implementadas (capturas propias).

3.4.2 Colisiones del jugador

Para poder gestionar la colisión del jugador con los muros del nivel debemos añadir una condición a la función “movePlayer”: en vez de modificar directamente las variables de posición del jugador (“x” e “y”), calcularemos la nueva posición —“dX” y “dY” en el proyecto— mediante el mismo código de antes, pero sólo actualizaremos al jugador con las nuevas “x” e “y” tras comprobar que en dichas coordenadas no hay un muro con una nueva función que añadiremos a la clase *Map*, “hasWallAt”.

Además, se comprobará si hay un muro en la nueva posición por separado: primero en “x” y luego en “y” (aunque aquí el orden sí es indiferente). Así, el jugador puede desplazarse parcialmente, aunque encuentre un muro de cara. Los casos se muestran en el Diagrama 10:

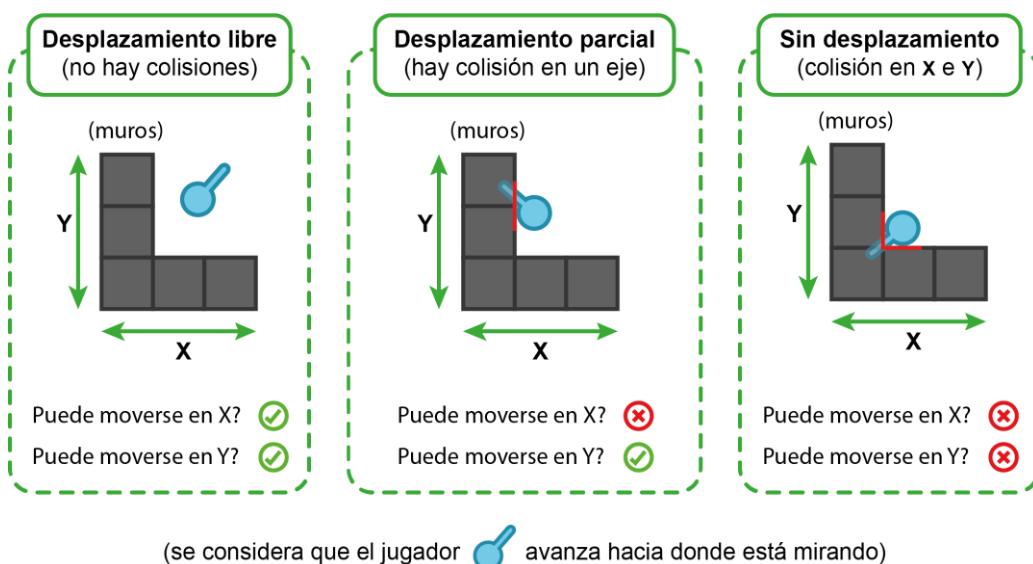
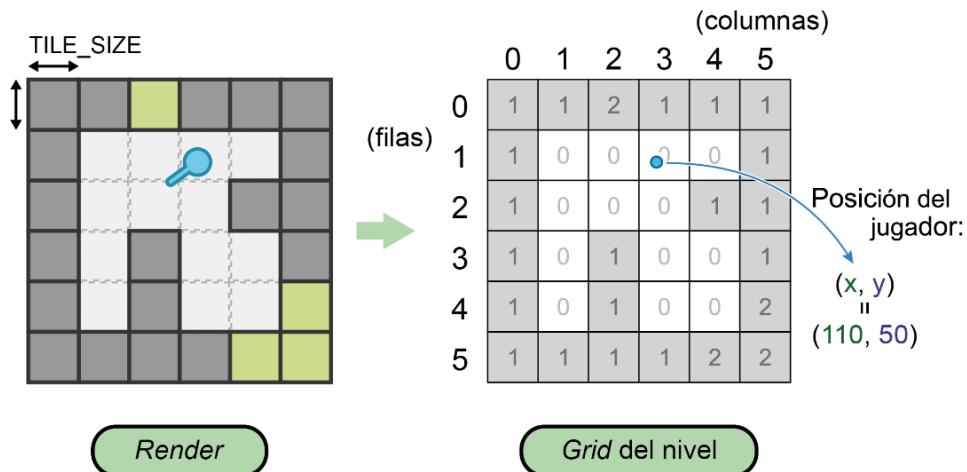


Diagrama 10. Distinción entre colisiones en "X" e "Y" o ambas.

Por último, en la clase *Map* se deben definir las funciones “*hasWallAtX*” y “*hasWallAtY*”, a las que se ha llamado en “*movePlayer*”. Éstas reciben por parámetro la posición del jugador en píxeles dentro del *canvas* (“*dx*” para el eje x y “*dy*” para el eje y), y el nivel en que se encuentra (por ahora será cero, ya que sólo hemos generado un nivel), y detectarán si en la posición indicada por parámetros hay un muro.

El proceso matemático para obtener la posición se describe en el Diagrama 11, y consiste en dividir la posición en píxeles por el tamaño decidido para las celdas (“*TILE_SIZE*”) y extraer los decimales del resultado.



- (X) $\text{player.x} / \text{TILE_SIZE} \rightarrow 110 / 32 = 3,4375 \rightarrow \text{Math.floor}(3,4375) = \text{Columna 3}$
- (Y) $\text{player.y} / \text{TILE_SIZE} \rightarrow 50 / 32 = 1,5625 \rightarrow \text{Math.floor}(1,5625) = \text{Fila 1}$

Diagrama 11. Obtener filas y columnas de la *grid* desde la posición en píxeles del *canvas*.

Cabe mencionar que la función devolverá el número correspondiente a la matriz en las coordenadas calculadas.

Como cualquier valor distinto de cero (es decir, *true*) será un muro, si al comprobar el valor en el *if* de la función “*movePlayer*” (de la clase *Player*, actualizada anteriormente) éste es un *false*, significa que no hay un muro en esa posición. Sin embargo, más adelante se necesitará indicar el valor del muro contra el que ha colisionado para cargar la textura correspondiente.

3.5 Raycasting

Primero se definirán las tres constantes a utilizar como valores para el campo de visión del jugador. Éstas son el campo de visión que cubre la cámara (FOV), el número de rayos emitidos por ella (“FOV_NUM_RAYS”) y una última que se calculará a partir de estas dos, y es el espacio entre rayos (“FOV_ANGLE_SPACING”).

```
// playerController.js

// constantes globales
const FOV = degreesToRadians(75);
const FOV_NUM_RAYS = 64;
const FOV_ANGLE_SPACING = FOV / (FOV_NUM_RAYS - 1);
```

Es importante tener en cuenta que el número de rayos *casteados* por la cámara repercutirá proporcionalmente en el rendimiento del sistema. Cuantos más rayos lance el jugador, más calidad de resolución en los muros, pero más cálculos deberá hacer el sistema y más recursos consumirá el *engine*. Por eso es importante que el desarrollador decida un valor que permita renderizar los muros con la suficiente calidad en el *canvas*, pero acorde a la capacidad de su ordenador, como se muestra en la Figura 17.

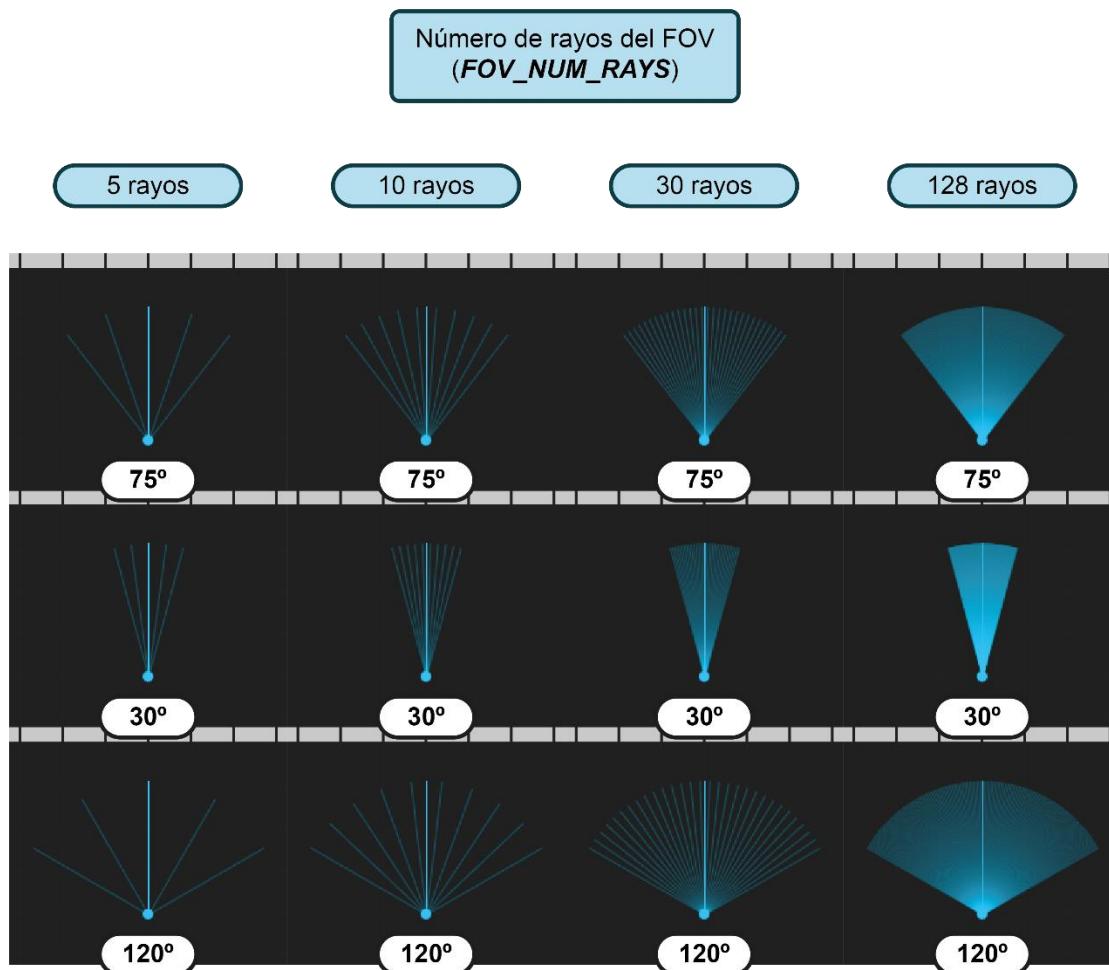


Figura 17. Representación comparativa del jugador con varios FOVs (en grados sexagesimales), y con distinta cantidad de rayos o "FOV_NUM_RAYS" (diagrama propio).

Una vez el desarrollador haya decidido un número de rayos y un ángulo de visión a probar, se prosigue con la implementación de la clase *Ray*, rayo, en el mismo documento ("playerController.js").

En el constructor de la clase *Ray* no es necesario un atributo que almacene el origen del vector, dado que éste siempre será la posición del jugador (y, al ser una variable global del proyecto, podremos leerla cuando lo necesitemos).

La función *render* de *Ray* es prácticamente igual a la de *render* del FOV del jugador, ya que el punto de origen es el mismo: la posición de la cámara. Sin embargo, pese a que la trigonometría para calcular la posición final de la línea es la misma, esta vez multiplicamos por una distancia cualquiera elegida por el desarrollador.

Eso es todo respecto a la clase *Ray*, a estas alturas. Ahora se debe agregar a la clase *FOV* el atributo *array* de rayos y la función que llamaremos, una vez por *loop*, para crear los rayos del campo de visión: “*castFOVRays*”.

La función contendrá un bucle en que se añadirá al *array* definido en el constructor un nuevo rayo, con su valor de dirección actualizado mediante índice multiplicado por el espaciado de los rayos. También es importante inicializar siempre el *array* al principio de la llamada, para borrar los rayos anteriores. De lo contrario el *array* se iría llenando indefinidamente. Éstos se van creando y metiendo en el *array* de izquierda a derecha (desde el punto de vista del jugador). Por eso antes del bucle inicializamos una variable “*rayAngle*” equivalente al ángulo central del campo de visión del jugador menos la mitad de dicho campo.

```
castFOVRays() {
    this.rays = [];

    let rayAngle = this.angleView - FOV/2;
    for (let iRay = 0; iRay < FOV_NUM_RAYS; iRay++) {
        this.rays.push(new Ray(rayAngle));
        rayAngle += FOV_ANGLE_SPACING;
    }
}
```

En el *update* del *FOV* será donde se llamará a la función que acaba de implementarse, después de actualizar el ángulo en que mira el jugador. El orden también es importante aquí, dado que, si primero creáramos los rayos y luego modificásemos el ángulo del jugador, nuestro *FOV* iría con un retraso de 1 frame respecto a nuestro *input*, porque estaríamos usando valores del anterior ciclo del *loop* al llenar el *array* de rayos.

Tras actualizar también la función *render* de *FOV*, nuestro *engine 2D* debería tener un aspecto parecido al de la Figura 18. Los valores de las constantes para este *raycast* concreto son los siguientes: un campo de visión (“*FOV*”) de 75 grados sexagesimales —deben convertirse a radianes— y 128 rayos generados (“*FOV_NUM_RAYS*”).

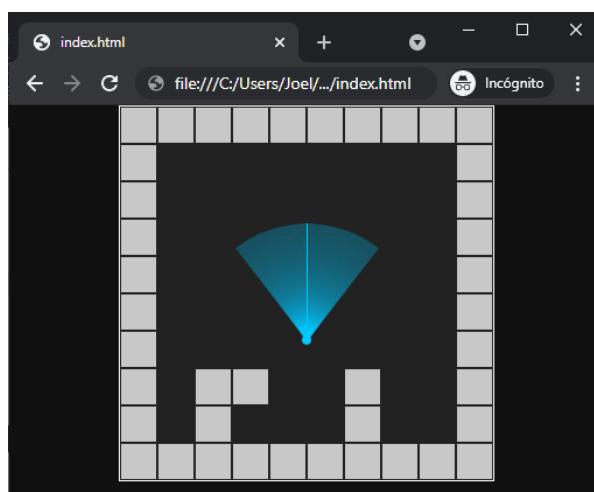


Figura 18. Captura del *raycasting* implementado, con un radio de 100 píxeles, 75° de *FOV* y 128 rayos; sin colisiones (captura propia).

Sin embargo, queda el aspecto fundamental (y más complicado) del *raycasting*: la detección de colisiones. Como puede verse en la Figura 19, si acercamos al jugador a un muro y lo “miramos”, nuestro campo de visión atraviesa los elementos.

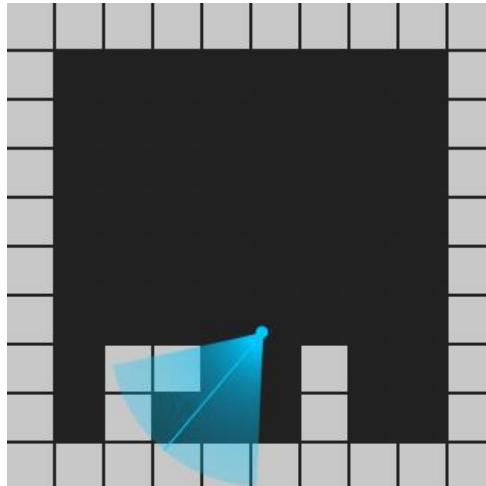


Figura 19. Resultado del *raycaster* sin colisiones: el FOV atraviesa cualquier elemento del nivel (captura propia).

3.6 Colisión de rayos

Para gestionar la colisión de los rayos del FOV del jugador deberemos implementar una nueva función para obtener el tipo de *Tile* que hay en la *grid* del nivel en una posición determinada, y luego añadir (por separado) la detección de colisiones de cada *Ray* a lo largo del eje X — colisiones horizontales — y a lo largo del eje Y — colisiones verticales.

También definiremos otra nueva función en nuestra librería *utils.js* para obtener la magnitud de un vector dada la distancia vertical y horizontal que recorre.

3.6.1 Nueva función en *Map* (*mapController.js*)

Primero se implementará este nuevo método en la clase *Map* que nos devolverá si en una posición dada (cuyo valor será en píxeles) hay un muro o es un espacio libre de colisiones. Recibirá por parámetros las coordenadas en el eje de abscisas y de ordenadas, y el nivel en que se encuentra el jugador (ya que cuando lleguemos al sistema multinivel debemos tener en cuenta el “piso” en que se encuentra el jugador en la escena).

En esta función sí que se **comprobarán** las coordenadas “x” e “y” **a la vez**, a diferencia de las anteriores “*hasWallAtX*” y “*hasWallAtY*”. Además, para seguir con el mismo estilo de nomenclatura, se llamará “*hasWallAt*”.

Necesitaremos llamar a esta función para detectar la colisión de los rayos emitidos por la cámara (los que generamos en la parte de *raycasting*).

3.6.2 Nueva función en *utils.js*

Ésta nueva función, a la que llamaremos “*getDistFromSideLength*”, recibe la distancia horizontal y vertical y devuelve su magnitud (es decir, calcula su raíz cuadrada). Como ocurrió con el seno y el coseno anteriormente, deberemos llamar a la operación “*sqrt*” de la librería *Math* de JavaScript.

```
function getDistFromSideLength(dx, dy) {
    return Math.sqrt(dx*dx + dy*dy);
}
```

Y con estas dos nuevas funciones, ya se puede implementar la colisión de los rayos.

3.6.3 Colisiones horizontales

Hay distintas maneras de detectar estas colisiones, pero una de las más eficientes es sacar partido de la estructura de cuadrícula de nuestra escena. Independientemente del orden, el desarrollador debe detectar colisiones a lo largo del “x” y del eje “y” por separado (pese a ello, se deberá ir calculando la posición final de cada rayo en ambas coordenadas).

En esta guía implementaremos primero la detección de colisiones horizontales.

Para definir ya la nomenclatura a utilizar, así como ilustrar el próximo proceso de cálculo, en la Figura 20 se puede encontrar una escena ficticia del proyecto en que el jugador está mirando al muro (render). Abajo se segmenta la distancia total recorrida por el ángulo de visión central de la cámara (“xDist” y “yDist”) en varios valores: “xInit” e “yInit” —distancia recorrida horizontal y verticalmente hasta cruzarse con el primer cambio de fila y/o columna de la *grid* del mapa— y “xStep” y “yStep” —longitud constante que marca cada cuántos píxeles el rayo colisionará contra la *grid* del mapa.

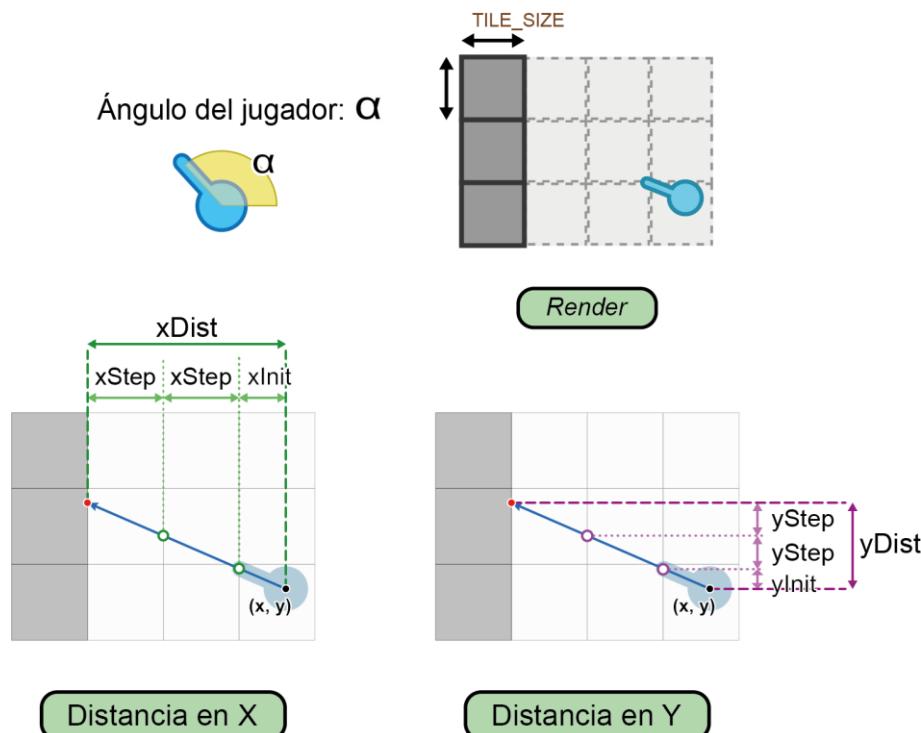


Figura 20. Ilustración de un posible renderizado de la escena, con la distancia recorrida por un rayo segmentada en variables (prioridad respecto a las colisiones en “x”) y por cada eje de coordenadas (diagrama propio).

Ahora que se ha mostrado una representación gráfica de las variables que se utilizarán en el proceso de detección de colisiones horizontales, en el Diagrama 12. Proceso ordenado de la detección de colisiones horizontales del *raycaster*, con las operaciones matemáticas

correspondientes. ¡Error! No se encuentra el origen de la referencia. se explica el proceso paso a paso, junto al código a programar.

PROCESO DE CÁLCULO DE LAS COLISIONES HORIZONTALES:

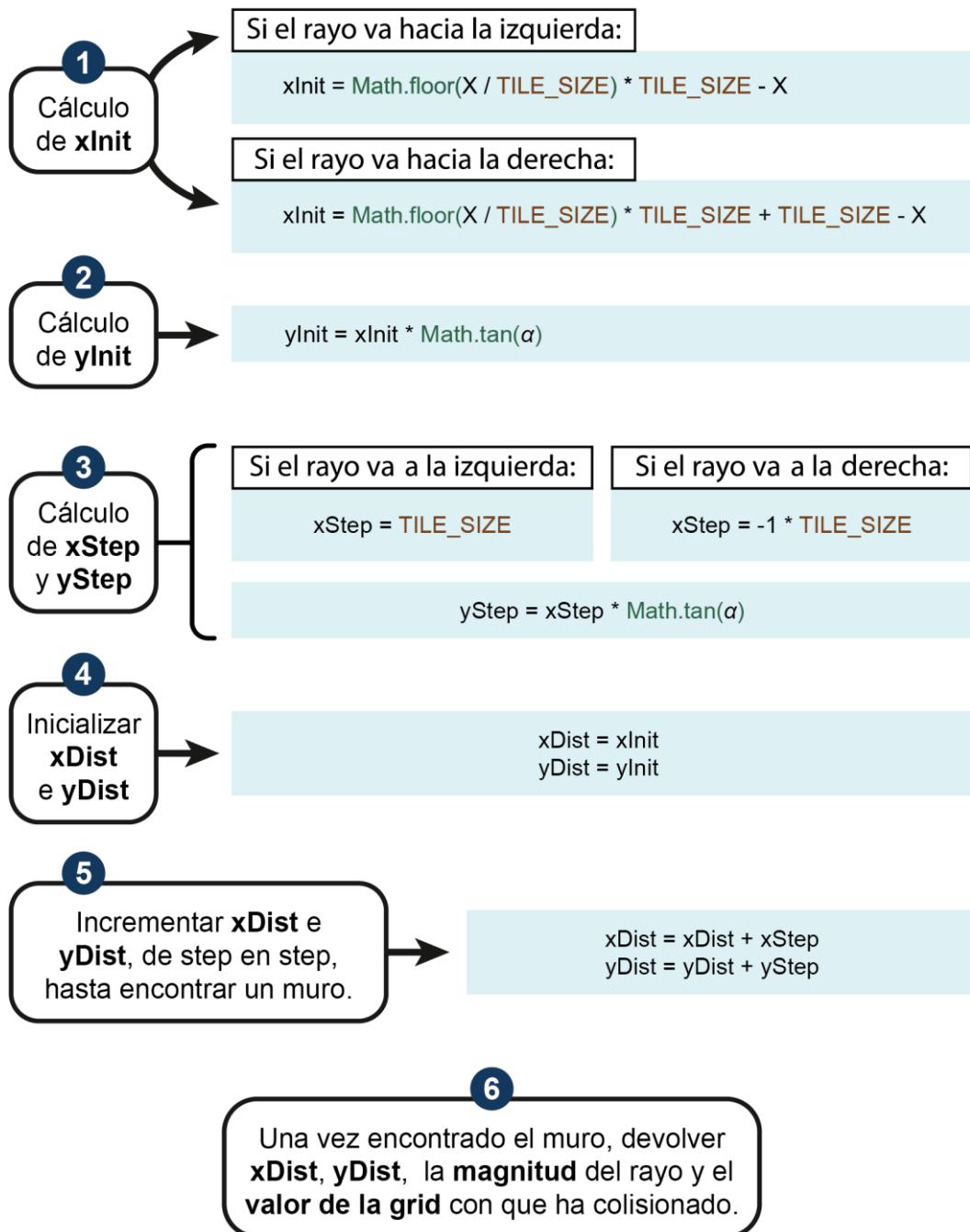


Diagrama 12. Proceso ordenado de la detección de colisiones horizontales del *raycaster*, con las operaciones matemáticas correspondientes.

La función que gestionará este proceso se llama “`checkXWallHit`”, ya que ésta sólo calcula las coordenadas y distancias en píxeles de los puntos en que el rayo golpea una nueva *Tile* de la **grid** a lo largo del eje de abscisas. Este método lo programaremos en la clase *Ray*.

Como se puede ver a continuación, la función devuelve un array con cuatro valores: la coordenada “X” en píxeles de la colisión del rayo con el primer muro, la misma en “Y”, la distancia total recorrida por el rayo (desde el jugador hasta el muro encontrado, calculada mediante la nueva función de *utils*, “*getDistFromSideLength*”) y el valor del muro contra el que ha chocado el rayo (*collision*). Para devolver todos los valores juntos desde la misma función deberán almacenarse en un *array*, como se muestra en el código de ejemplo:

```
return [
    xDist,
    yDist,
    getDistFromSideLength(xDist, yDist),
    collision
];
```

Una vez la función ya está programada, la llamaremos desde otro método en esta misma clase: ***cast***. Aquí, asociaremos los resultados de la función “*checkXWallhit*” a los atributos del rayo en cuestión (se puede aprovechar el formato de *array* también para asignar los valores, no sólo al devolverlos, como se muestra a continuación).

```
cast() {
    let xWallhitResult = this.checkXWallhit();
    [this.dX, this.dY, this.distance, this.rayImpactsOn] =
xWallhitResult;
}
```

Ya para terminar, para aplicar esta funcionalidad debemos llamar a ***cast*** cuando generamos los rayos del FOV del jugador, en “*castFOVRays*”, método ya implementado de la clase *FieldOfView*.

Al ejecutar el código, después de preparar todas las llamadas para gestionar la colisión en X, los rayos del FOV del jugador deberían comportarse de manera igual que la vista en la Figura 21. En ella se puede apreciar cómo los rayos se detienen al colisionar con los muros que se encuentran a lo largo del eje “x” de la grid, pero ignoran los muros que encuentran al avanzar en “y”.

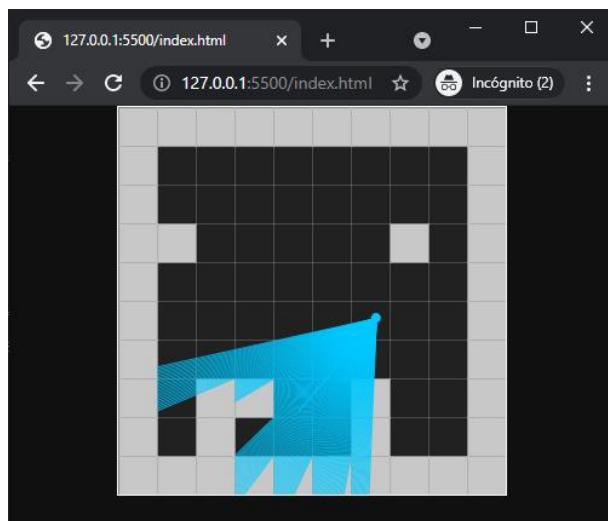


Figura 21. Aspecto del *raycaster* con la colisión de rayos sólo en “X” implementada (captura propia).

3.6.4 Colisiones verticales

El proceso para calcular las colisiones verticales es extremadamente parecido al anterior, sólo que ahora el valor constante equivalente a “TILE_SIZE” no será el salto en el eje “x”, sino en el “y”, como se muestra en la Figura 22:

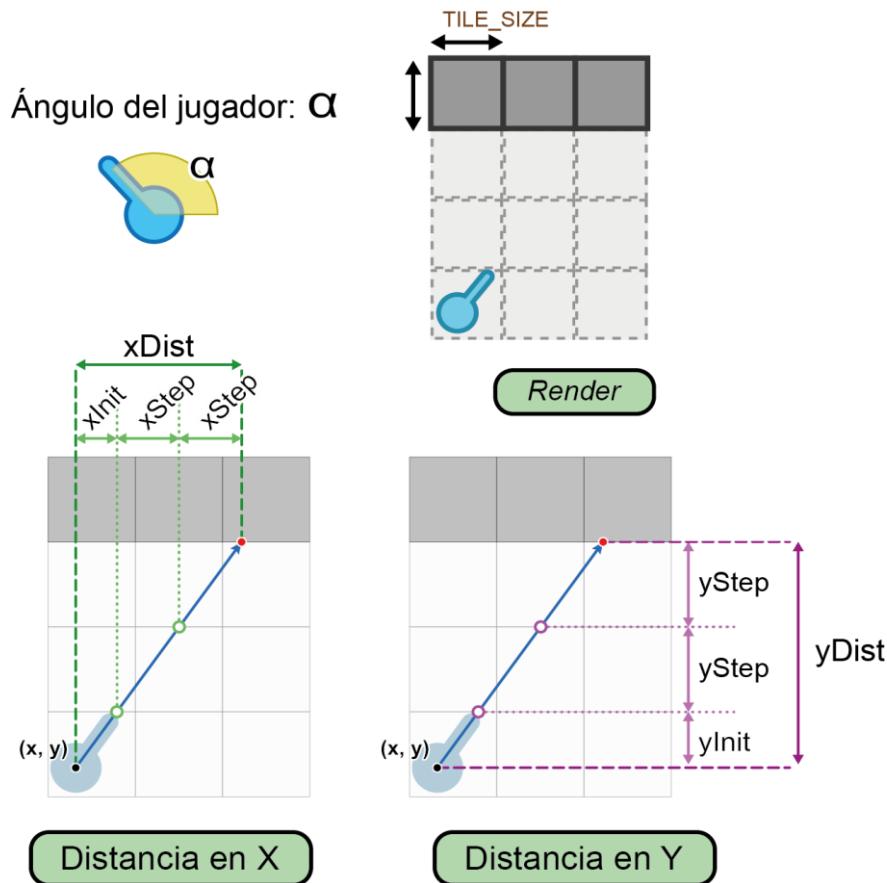


Figura 22. Ilustración de un posible renderizado de la escena, con la distancia recorrida por un rayo segmentada en variables (prioridad respecto a las colisiones en “Y”) y por cada eje de coordenadas (diagrama propio).

Se puede ver que la variable “yStep” será ahora la que almacene la constante “TILE_SIZE”, lo que significa que la prioridad de los cálculos recaerá sobre el eje de ordenadas, y serán los valores en el eje de abscisas los que dependerán de la tangente del ángulo del jugador.

A continuación (como en el anterior subapartado), se presenta el Diagrama 13, dónde están organizados los puntos a programar en la nueva función para la clase *Ray*, “checkYWallHit”.

PROCESO DE CÁLCULO DE LAS COLISIONES VERTICALES:

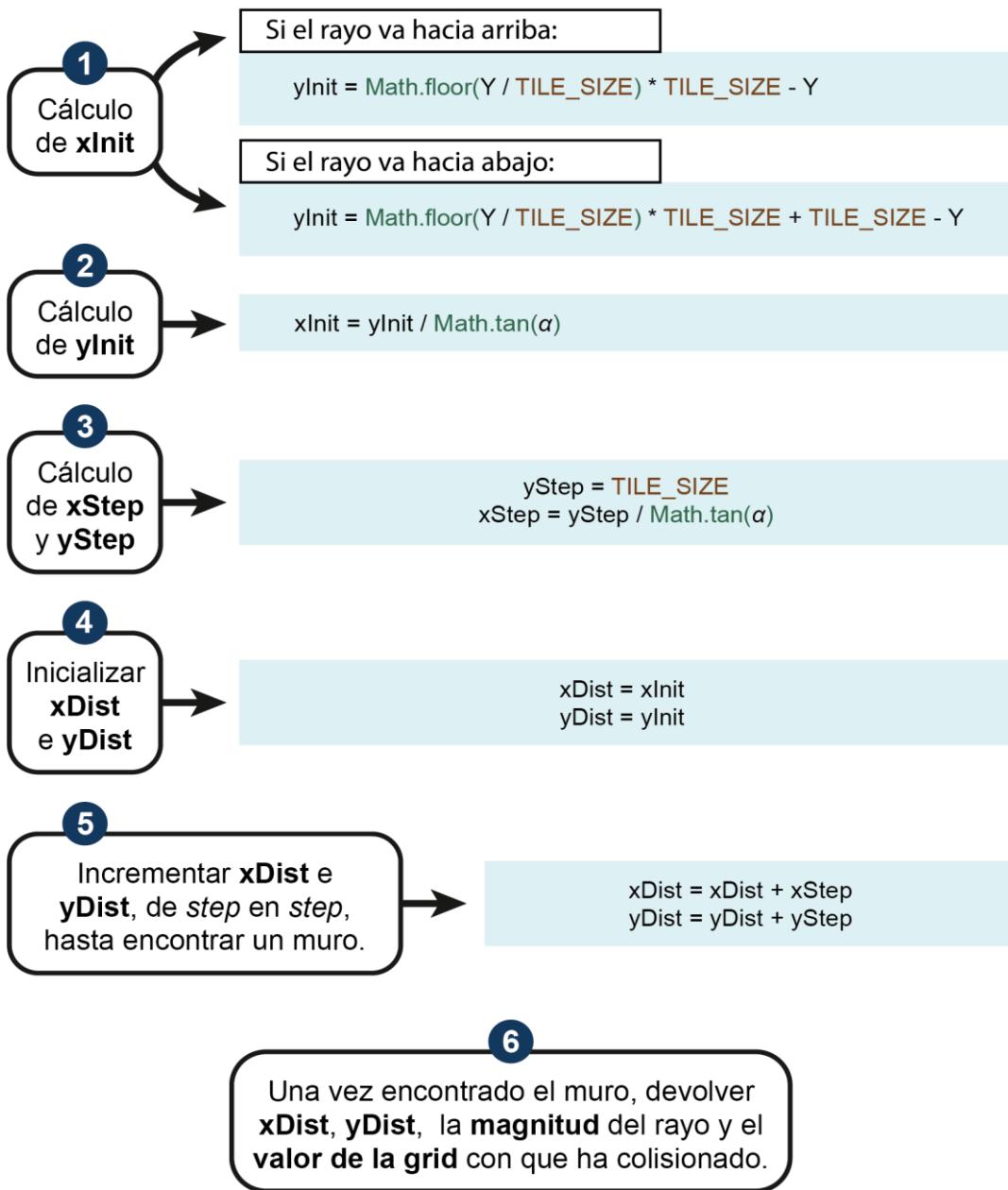


Diagrama 13. Proceso ordenado de la detección de colisiones verticales del *raycaster*, con las operaciones matemáticas correspondientes.

Como dato adicional puede verse que, para detectar si el rayo avanza hacia arriba o abajo (distinción que será necesaria para las coordenadas a calcular) se empleará el seno del ángulo de visión del jugador; comportamiento similar al empleado en las colisiones horizontales, para las cuales se calculaba el coseno para indicar si el rayo avanzaba hacia la izquierda o la derecha. En el código de ejemplo se ve cómo:

```
let rayGoesUp = (Math.sin(this.direction) < 0)? true : false;
```

Tras haber implementado este método, llamado “checkYWallhit”, podemos sustituir la anterior llamada a “checkXWallHit” en la función *cast* de la clase *Ray* por la nueva, para comprobar su funcionamiento. Cabe recordar que la nueva función también devolverá esas mismas cuatro variables que se asignarán a los atributos del rayo en cuestión, como en el código de ejemplo:

```
cast() {
    let yWallhitResult = this.checkYWallhit();
    //let xWallhitResult = this.checkXWallhit(); // (antes)
    [this.dX, this.dY, this.distance, this.rayImpactsOn] =
yWallhitResult;

}
```

El comportamiento de los rayos al renderizar el canvas debería ser el mostrado en la Figura 23: en ella se puede ver cómo los vectores se detienen al colisionar con un muro en el eje vertical de la grid, pero ignoran cualquier colisión horizontal.

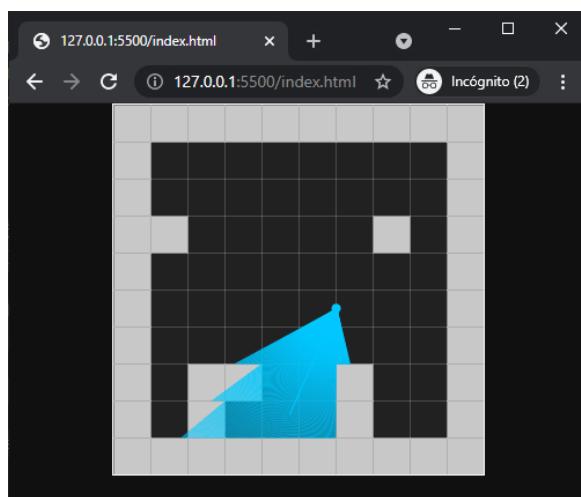


Figura 23. Aspecto del *raycaster* con la colisión de rayos sólo en "Y" implementada (captura propia).

3.6.5 Unión de los dos tipos de colisión

Dado que el engine ya cuenta con los dos procedimientos para gestionar la colisión de los rayos, ahora sólo queda combinarlos para que funcionen a la vez. La operación es simple, se hacen las dos llamadas en “*cast*” (clase *Ray*), pero sólo aplicaremos las variables “*xDist*”, “*yDist*” y distancia recorrida por el rayo de aquella de las dos que haya recorrido menos distancia: es decir, el rayo se detendrá al colisionar con el primer muro que encuentre, que será el que recorra la menor distancia. Del mismo modo, actualizaremos “*rayImpactsOn*” con el valor de la grid con que el rayo ha colisionado.

Además, se puede aprovechar la detección de la colisión para saber si el rayo ha colisionado horizontal o verticalmente. De este modo se añadirá un efecto de sombreado en la proyección de los muros, más adelante.

Al ejecutar el *index.html* tras aplicar estos cambios, el *raycaster* es completamente funcional y la colisión de rayos estará terminada al completo. En la Figura 24 se puede apreciar el resultado:

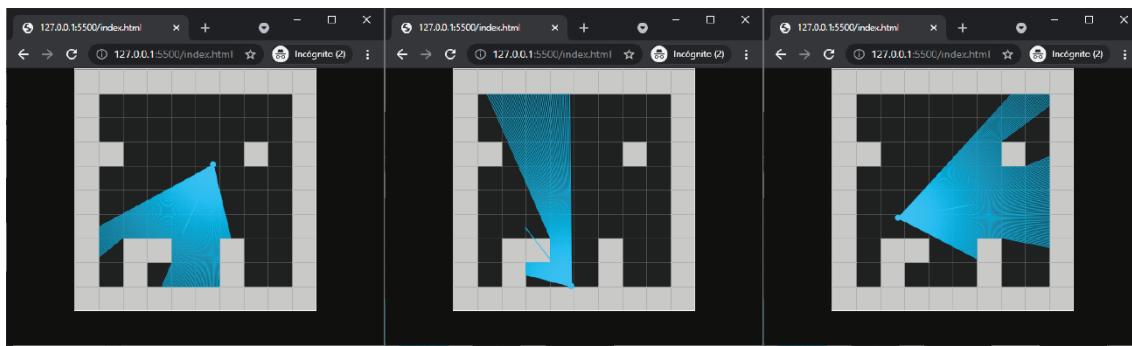


Figura 24. Resultado de la implementación de las colisiones del *raycaster*, tanto en "x" como en "y" (capturas propias).

3.7 Generación de una escena multinivel

Dado que desde un principio se contó con este punto, el código está prácticamente adaptado ya para soportar una escena multinivel. Simplemente se puede agregar otra matriz numérica al *array grid* de la clase *Map*.

Una vez definido este nuevo nivel, se gestionarán los controles del jugador para poder subir o bajar de “piso” en la escena. Es decir, lo siguiente es modificar “keyController.js”:

- Se definen dos constantes más con el valor numérico identificativo de los botones arriba y debajo de las flechas direccional, que son 38 y 40 respectivamente.

```
// keyController.js
// nuevas constantes
const KeyArrowUp = 38; // subir 1 nivel de altura
const KeyArrowDown = 40; // bajar 1 nivel de altura
```

- Añadimos la lógica de los dos inputs en la función “keyPressed” de este mismo fichero. Ésta reside en incrementar o decrementar en 1 el valor del nivel del jugador (ya se definió un atributo en la clase *Player* para controlar este valor), siempre que no supere el número de niveles de la escena o ya esté en el más bajo.

Lo siguiente será mostrar en el canvas algún tipo de información que marque en qué nivel se encuentra el jugador. Asimismo, también podría darse cierto *feedback* de la lista de niveles disponibles. El formato más cómodo es directamente mostrar un *string* de texto con el habitual formato “Nivel 1 de 2” o “Nivel 1/2”. Para ello también debemos modificar ligeramente algunas otras partes del proyecto:

- En la función *setup* del fichero “main.js”, incrementamos la altura del *canvas* entre 50 y 70 unidades (a continuación, en el código de ejemplo, 64 píxeles).

```
createCanvas(objMap.width*TILE_SIZE, objMap.height * TILE_SIZE + 64);
```

- En la función “render” de la clase *Map* (fichero “mapController.js”) generamos la cadena de texto, a partir de los atributos de *Player* y *Map*. Necesitaremos llamar a las funciones “textSize” (para definir el tamaño de los caracteres) y “text” (para escribir en el *canvas*) de la librería *p5.js*.

- También, en caso de no controlarlo aún, deberá añadirse la lógica por parte del mapa que consiste en renderizar únicamente las *Tiles* del nivel de la escena en que esté el jugador.

En la Figura 25 se puede ver el resultado de la ejecución del programa. La escena está formada por dos niveles de los cuales, como se podía apreciar anteriormente en la grid numérica definida, el primero no ha cambiado ninguna Tile, y el segundo está vacío excepto por los bordes del piso.

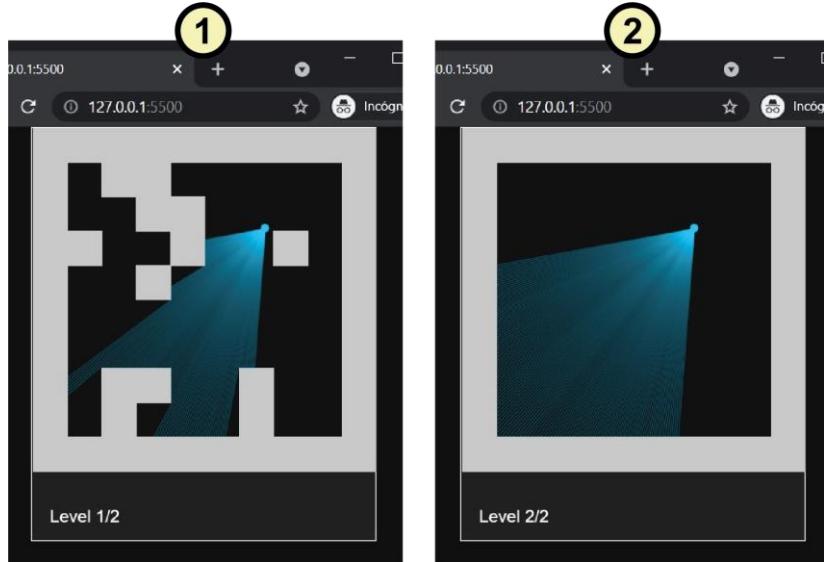


Figura 25. Comparación de dos capturas de la misma escena, (la 1 el primer nivel, la 2 el segundo). El jugador ha cambiado de una a otra sin modificar su posición (capturas propias).

3.8 Proyección de muros (*vLines*)

Llegado este punto, el *raycaster* en dos dimensiones ya ha tomado forma y la colisión de los rayos permite el paso a la renderización de los muros, dado que toda la información necesaria de cada rayo está ya almacenada en la clase *FOV* del jugador (quien contiene el array de vectores).

Para proyectar los muros, el sistema pintará una línea vertical (que será un rectángulo de un ancho determinado) en el *canvas* por cada rayo lanzado por el campo de visión de la cámara, cuya altura será inversamente proporcional a la distancia recorrida por dicho rayo. Tras ello, si centramos el rectángulo en el *canvas*, la sensación al mover al jugador por cada nivel de la escena simulará el efecto en tres dimensiones deseado.

3.8.1 Cambios en *main.js*

Primero, se definirá una nueva clase encargada de gestionar el renderizado de estas “*vLines*” para dibujar los muros. Esta se llamará “*Render*”.

El *canvas* ahora obtendrá sus dimensiones de dos constantes que se definirán en el próximo fichero a desarrollar (por ahora simplemente se les llama en el constructor del *canvas*), en vez de utilizar la grid del nivel multiplicada por el tamaño de las *Tiles*, como se hacía anteriormente.

```
createCanvas(VIEWPORT_WIDTH, VIEWPORT_HEIGHT);
```

La función *setup* del *main.js* no cambia, pero sí *update* y *draw*, donde llamaremos a dos funciones de esta nueva clase.

El método “*update*” de *Render* (“*objRender*”, en el código) se encarga de almacenar en un nuevo array los datos necesarios (que se calcularon en el FOV del jugador, por cada rayo) para cargar la proyección de los muros.

Por otro lado, el método “*loadProjection*”, al que se llamará en *draw*, para ejecutarse a cada vuelta del *loop*, lee dicha información cargada en el *update* y renderiza los muros (antes de llamar a las funciones *render* del jugador y el mapa).

3.8.2 Implementación de *renderController.js*

Aquí es donde se definirán las nuevas constantes para el tamaño del *canvas* (en el código, llamado “*viewport*”), así como el ancho de las “*vLines*” que renderizaremos, ya que se puede calcular al principio de la ejecución. Éste último depende del ancho que se haya decidido para el *canvas* y el número de rayos lanzados por el FOV (a más rayos, más “*vLines*”).

```
// renderController.js
const VIEWPORT_WIDTH = 960;
const VIEWPORT_HEIGHT = 640;
const RENDER_VLINES_SCALING = 1;
const RENDER_VLINES_WIDTH = VIEWPORT_WIDTH / FOV_NUM_RAYS /
RENDER_VLINES_SCALING;
```

En la clase *Render* se guardarán como atributos el punto medio desde el cual empezar a dibujar las “*vLines*” (en *Wolfenstein 3D* era el centro de la pantalla; y aunque en este caso se ha dejado aproximadamente a un tercio de distancia de la base inferior del *canvas*, depende enteramente del desarrollador dónde situar el umbral del horizonte).

Las demás variables son *arrays* que guardan los valores de altura, *offset* (posición desde la base a la que dibujar los rectángulos) y el color de cada rayo. En el método *update* siempre deberán vaciarse antes de calcular la altura y el color de la línea vertical (que luego se añadirá al *array*).

Hay que hacer un último cálculo para el cálculo de la altura de la línea de la proyección. De lo contrario, aparecerá un efecto de ojo de pez generado por la diferencia de la hipotenusa entre la cámara y el muro a lo largo del FOV del jugador.

```
calculateVLineHeight(i) {
    let distToProjectionPlane = VIEWPORT_WIDTH/2 / Math.tan(FOV/2);
    let height = (TILE_SIZE / objPlayer.fov.rays[i].distance) *
distToProjectionPlane;
    return height;
}
```

Para calcular el color también utilizamos la distancia del rayo, de modo que cuanto más cerca quede el muro, más claro sea el rectángulo que le corresponde (como máximo, 255; el mínimo a gusto del desarrollador). Además de ello, aprovechamos el atributo que se calculó en el rayo y que indicaba si la colisión del rayo era vertical o no, para escalar el color al 80%, dando sensación de sombra en el renderizado.

```

calculateVLineColor(i) {
    let answer = 255 - (objPlayer.fov.rays[i].distance *
255/objPlayer.fov.maxDistance);
    if (objPlayer.fov.rays[i].rayHitsVertically) answer *= 0.8;
    return answer;
}

```

Como se mencionó previamente, queda por implementar “loadProjection”, el método que dibujará los rectángulos para el *canvas*. Para éste se pueden utilizar las mismas funciones ofrecidas por *P5.js* y que se usaron previamente para renderizar el mapa: *fill* y *rect*.

Sin embargo, aún no servirá ejecutar el código, ya que falta una última modificación a aplicar. Aprovecharemos que anteriormente renderizamos la escena del nivel en 2D y con el jugador y su FOV, para ahora usarlo como mini mapa: como si se tratara de un *shooter* actual.

3.8.3 Implementación del mini mapa

Para convertir nuestro anterior render del *canvas* (la *grid* del nivel donde se encuentra el jugador) en un minimapa, simplemente hay que multiplicar algunas de las fórmulas utilizadas al renderizar por un factor de escalado inferior a uno: por ejemplo, de 0.5 unidades —es decir, reduciendo el anterior tamaño a la mitad.

- Primero, definir la constante con dicho factor en “mapController.js”.

```
const MAP_SCALING = .5; //factor de reescalado del mapa
```

- *Render* de la clase *Map*. Deberán modificarse las coordenadas en que se avisaba al jugador del nivel de la escena en que está (el texto).
- *Render* de la clase *Tile*. Tanto las coordenadas de cada casilla como sus dimensiones deberán multiplicarse por este factor.
- En el *render* de la clase *Ray*, las coordenadas de cada rayo deben “reescalarse” como se muestra en el código:

```

render() {
    stroke('rgba(0,200,255,0.3)');
    strokeWeight(1);
    line(MAP_SCALING * objPlayer.x,
        MAP_SCALING * objPlayer.y,
        MAP_SCALING * (objPlayer.x + this.dX),
        MAP_SCALING * (objPlayer.y + this.dY)
    );
}

```

- Por último, también el método *render* de la clase *Player*: las coordenadas del círculo que representa al jugador en el mini mapa también hay que multiplicarlas por el factor.

Tras modificar las funciones de renderizado del *canvas* para escalar el mini mapa e implementar el fichero “*renderController.js*”, al ejecutar el *index.html* debería verse un resultado parecido al de la Figura 26.

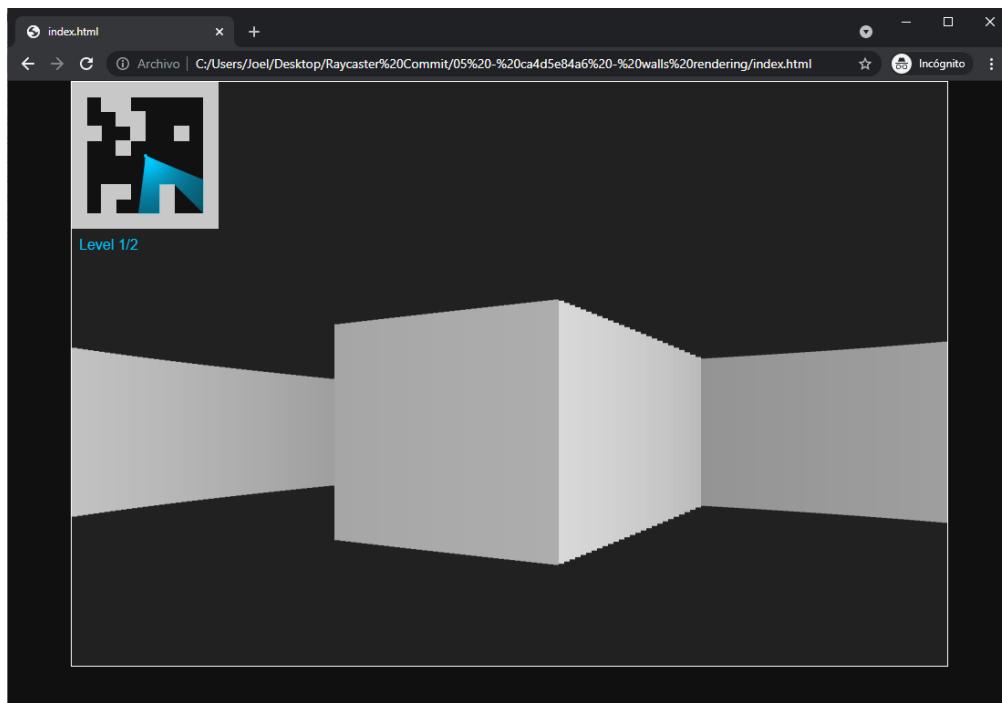


Figura 26. Resultado de la proyección de muros y mini mapa (captura propia).

Ya se ha implementado correctamente la proyección de los muros, así como el mini mapa en la esquina superior izquierda, como en un *shooter* moderno.

3.9 Controles de tipo libre (opcional)

Para hacer el movimiento del *raycaster* más intuitivo, a continuación, se mostrará cómo implementar el sistema de controles de tipo libre, a diferencia del tipo tanque que se programó inicialmente, en el apartado *Player input* y movimiento del jugador (página 46). Desarrollar este tipo de controles es completamente opcional por parte del programador del *engine*, ya que no influye en el resultado visual o en la lógica global del sistema.

- En el constructor de la clase *Player* se necesitará otra variable de gestión de movimiento, como se hizo para avanzar o retroceder. En este caso, el nuevo atributo deberá indicar si el jugador está quieto (por defecto, valor 0), o se mueve a izquierda (-1) o derecha (1).
- Una vez definido dicho atributo, se modificará la función de movimiento del jugador: “movePlayer”, también de la clase *Player*. Asimismo, para avanzar a izquierda o derecha, se deberá incrementar “x” o “y” en función del ángulo del jugador (e incrementando dicho ángulo en 90 grados sexagesimales, para obtener su izquierda o derecha). Después, multiplicamos por coseno y seno en función del eje en que modificar la posición del jugador.
- Para terminar, debemos incluir el control de usuario para moverse mediante el nuevo sistema en el “keyController.js”. En los nuevos controles, el jugador se moverá libremente mediante las teclas “W”, “A”, “S” y “D”, y girará a izquierda o derecha con “Q” y “E” respectivamente (y también con las flechas direccionales izquierda o derecha del teclado).

Visualmente no ha habido ninguna modificación, pero sí que, al ejecutar, el desarrollador ya ha modificado los controles. De nuevo, el código implementado con las teclas es solo un ejemplo, corre a cargo del programador decidir el tipo de movimiento, así como las teclas para éste.

3.10 Lector de imágenes

Tras el renderizado básico de los muros, llega la última fase de desarrollo del *raycaster*, este motor en 2D. Para leer las imágenes para las texturas, así como generar los niveles en función de mapas de color se necesita un procedimiento completo de lectura y gestión de imágenes en el *engine*.

En el Diagrama 14 se muestra una posible lista de imágenes del proyecto, así como un posible formato de nomenclatura intuitivo a seguir.

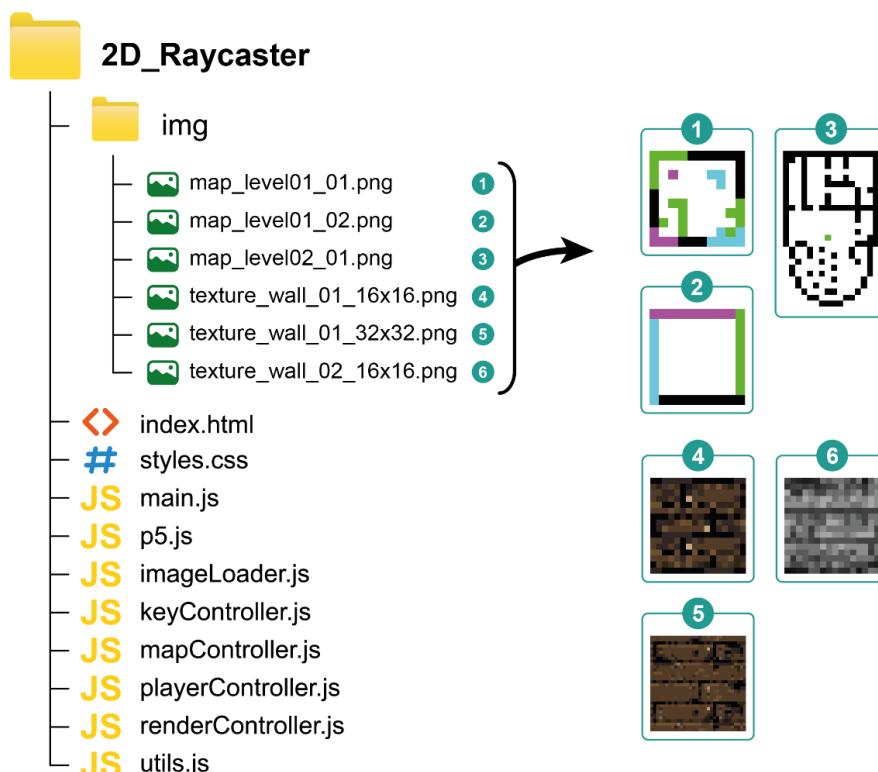


Diagrama 14. Distribución de los archivos del proyecto, ahora con imágenes en la carpeta “img” (y una captura de muestra de cada una).

En JavaScript, la carga de imágenes puede ser complicada si no se utilizan librerías externas especializadas para ello, pero ya se aclaró en un principio que la única biblioteca adicional a la que accederíamos sería *P5.js*, de modo que el código está adaptado a dicho uso.

3.10.1 Implementación de *imageLoader.js*

Para leer las imágenes del proyecto, en el documento “*imageLoader.js*” implementaremos la clase *ImageLoader*, encargada de leer dichos archivos, obtener la información necesaria y convertirlos a un formato que sea entendible para el usuario.

Para ello, crearemos un *canvas* (que será el mismo utilizado en el *main*, ya que automáticamente *P5.js* ya lo asigna por defecto al llamar a “*createCanvas*”) del tamaño de la imagen a leer, la

renderizaremos por pantalla y sacaremos una captura del resultado para poder extraer el color de los píxeles mostrados —y así poder generar una matriz de colores.

La clase *ImageLoader* tendrá una lista con todas las imágenes leídas por el *engine*, y otros dos *arrays* con aquellas imágenes de la anterior lista que sean mapas de nivel y texturas respectivamente. Cada **imagen** contará con los siguientes **atributos**:

- *index*: índice numérico de la imagen
- *data*: imagen en formato P5.js (no lo usaremos directamente)
- *pixels*: matriz de colores (uno por píxel) (éste sí lo usaremos)
- *function*: si es una textura o un mapa a color del nivel
- *path*: ruta relativa de la imagen ("img/...")
- *height*: altura de la imagen
- *width*: anchura de la imagen

A continuación, en el Diagrama 15 se muestra el proceso de lectura de las imágenes estructuradamente, antes de pasar al código.

PROCESO DE CARGA DE IMÁGENES (DESDE EL CANVAS):

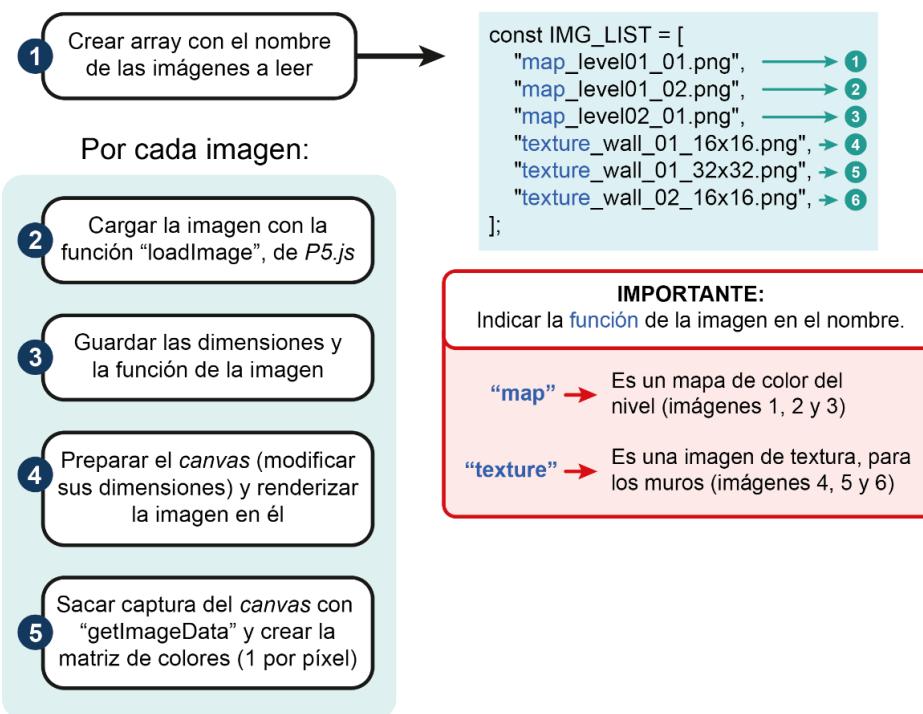


Diagrama 15. Proceso de lectura de imágenes, en 5 pasos; y ejemplo de nomenclatura de éstas.

El código está dividido en varios métodos para llevar a cabo todos los pasos mostrados en el proceso de carga de imágenes:

- **init**: método que crea el *canvas* y carga las imágenes en el *array* (la lectura inicial de imágenes, mediante la función "loadImage" de P5.js. También almacena el tipo de función de la imagen (si es una textura o un mapa de nivel).

- ***loadImagesPixels***: por cada imagen de la lista almacenada en la constante “IMG_LIST”, actualiza las dimensiones del *canvas* con las de la imagen, dibuja la imagen en él (con la función ***image***, de la librería *P5.js*), y llama a la función que carga la matriz de colores de los píxeles de la imagen (“*dataToPixelArray*”).
- ***dataToPixelArray***: esta función lee el color de cada píxel de la imagen y lo añade a la matriz (en caso de ser un mapa de nivel, deberemos aproximar antes el color —con “*getAproxColor*”, el último método a implementar— por si hubiera cambios no deseados al capturar la imagen renderizada en el *canvas*).
- ***getAproxColor***: recibe los cuatro dígitos que definen el color leído (RGBA) y devuelve el color con los valores RGBA aproximados por dígitos concretos que se respetarán al pintar las imágenes de los mapas de cada nivel, de modo que al ser leídos por el sistema no se cometa ningún error de lectura o los valores indicados no se encuentren. A continuación, se ofrecen los valores aproximados a los que se transforman los colores al capturar el *canvas*:

```
let newValues = []; // nuevos valores
// por cada valor, buscamos su aproximación
for(let iValue = 0; iValue<valuesArray.length; iValue++) {
    let aux = valuesArray[iValue];
    if (aux < 25) newValues.push(0);
    else if (aux >= 25 && aux < 50) newValues.push(25);
    else if (aux >= 50 && aux < 75) newValues.push(50);
    else if (aux >= 75 && aux < 100) newValues.push(75);
    else if (aux >= 100 && aux < 125) newValues.push(100);
    else if (aux >= 125 && aux < 150) newValues.push(125);
    else if (aux >= 150 && aux < 175) newValues.push(150);
    else if (aux >= 175 && aux < 200) newValues.push(175);
    else if (aux >= 200 && aux < 225) newValues.push(200);
    else if (aux >= 225 && aux <= 255) newValues.push(255);
}
```

- ***distributeImages***: por último, también se puede implementar un método que lea el *array* de imágenes generado anteriormente, y asignará las imágenes a las dos listas mencionadas que corresponda: al *array* con los mapas de cada nivel (imágenes 1, 2 y 3 de ejemplo del Diagrama 14, en la página 63Diagrama 15), o al *array* con las texturas de los distintos tipos de muro (imágenes 4, 5 y 6 del mismo).

Sin embargo, el *engine* aún no utiliza esta clase recién definida. Para usarla, debemos añadir las llamadas correspondientes al fichero principal del proyecto: *main.js*.

3.10.2 Modificaciones en *main.js*

Crearemos la variable de clase “*ImageLoader*”, recién definida, y haremos varias llamadas en *setup* y otra función ofrecida por *P5.js*, “*preload*”, que también se ejecutará una sola vez al principio, pero antes de “*setup*”.

```
var imageLoader; // ésta es la nueva variable
```

```

var objMap;
var objPlayer;
var objRender;

// nueva función (instanciaremos "ImageLoader" y lo inicializaremos)
function preload() {
    imageLoader = new ImageLoader();
    imageLoader.init();
}

```

Dado que en “preload” hemos creado ya el array de imágenes (aunque por ahora está vacío), en *setup*, antes que cualquier otra llamada, leeremos las imágenes (*loadImagesPixels*) y las distribuiremos (*distributeImages*) entre los dos *arrays* que definimos, el de las texturas y el de los mapas de nivel.

Y, con esta nueva clase y los cambios pertinentes en *main.js*, ya está implementada la lectura de imágenes para el *engine*, aunque aún no hacemos uso de esta funcionalidad.

3.11 Generación del nivel por imagen

El primer apartado en que utilizaremos la recién implementada lectura de imágenes será para generar los niveles en función del color. En el Diagrama 16 se narra (de forma más teórica) el proceso que implementaremos para la carga de nivel.

PROCESO DE CARGA DE ESCENA MULTINIVEL POR IMÁGENES: (a implementar en la clase *Map*)

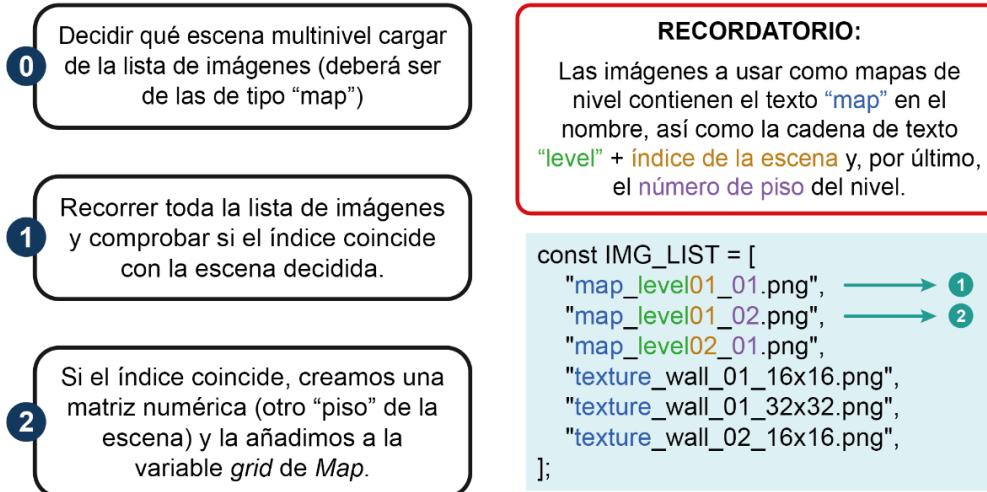


Diagrama 16. Proceso para cargar una escena multinivel mediante imágenes (con indicaciones sobre la nomenclatura de los archivos de imagen).

Como ya se ha explicado, el proceso consiste simplemente en añadir un paso previo en que se crea la matriz numérica automáticamente al leer la imagen del nivel que se quiere cargar. En el Diagrama 17 se muestra el concepto del proceso, usando como ejemplo la escena “01”, formada por dos niveles.

GENERANDO LA ESCENA 01 POR IMÁGENES:

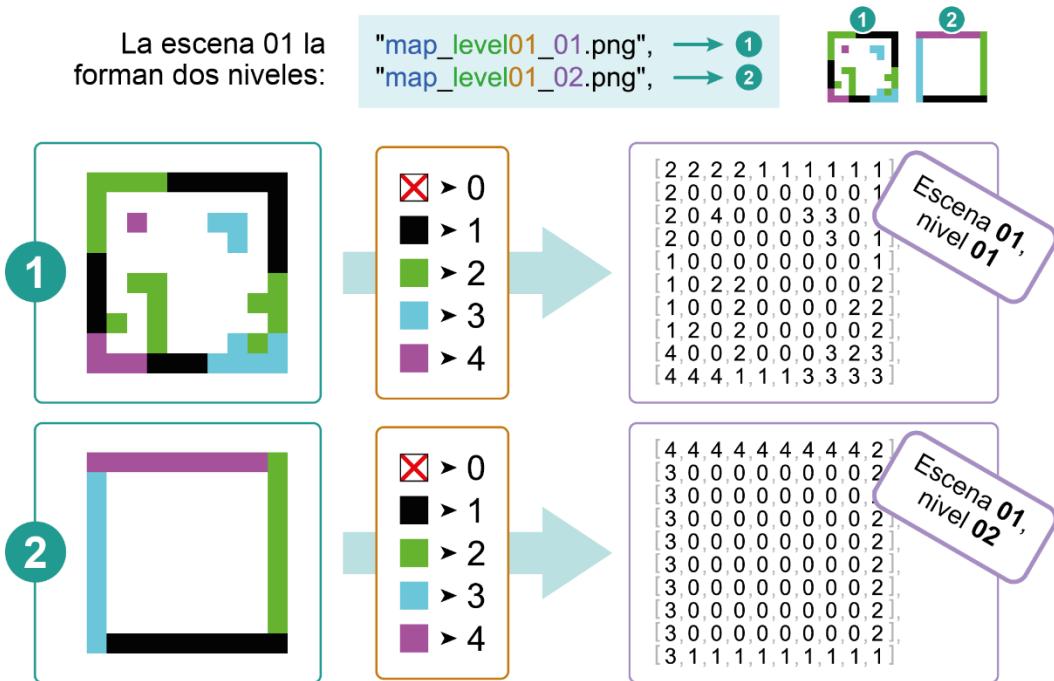


Diagrama 17. Proceso de conversión a matriz numérica desde el mapa de color de una escena multinivel (usando como ejemplo "map_level01").

De este modo, estaremos automatizando el primer proceso de generación del nivel, que era inicializar a mano la matriz numérica del mapa (el atributo *grid* de la clase *Map*). No necesitaremos hacer ninguna otra modificación para la generación de mapas, una vez añadamos el código presentado a continuación, ya que los siguientes pasos, como generar las *Tiles* y obtener la información necesaria (como su tipo y las posiciones) permanecen exactamente igual a como se implementaron previamente.

La nueva función a agregar al proyecto y que llevará a cabo este proceso es *levelImageToGrid*. Ésta recorrerá todo el *array* de imágenes, y generará la matriz numérica de aquellas imágenes cuya función sea la de "mapgrid" —el atributo *function* al que se hace referencia es uno de los asignados a nuestras imágenes durante la carga inicial, procedimiento programado en el anterior punto—. Al comparar el nombre de la imagen que corresponda, comenzará a leer píxel a píxel la variable *pixels*, que contiene el color leído de ésta y, si el valor coincide con uno de los deseados para algún muro, devolverá el número que toque agregar a la grid del mapa.

Con estos cambios y adiciones, el raycaster ya carga la escena en función de las imágenes que encuentre en la carpeta "img" del proyecto (siempre y cuando cumplan con la nomenclatura decidida por el desarrollador, como podría ser la mostrada de ejemplo).

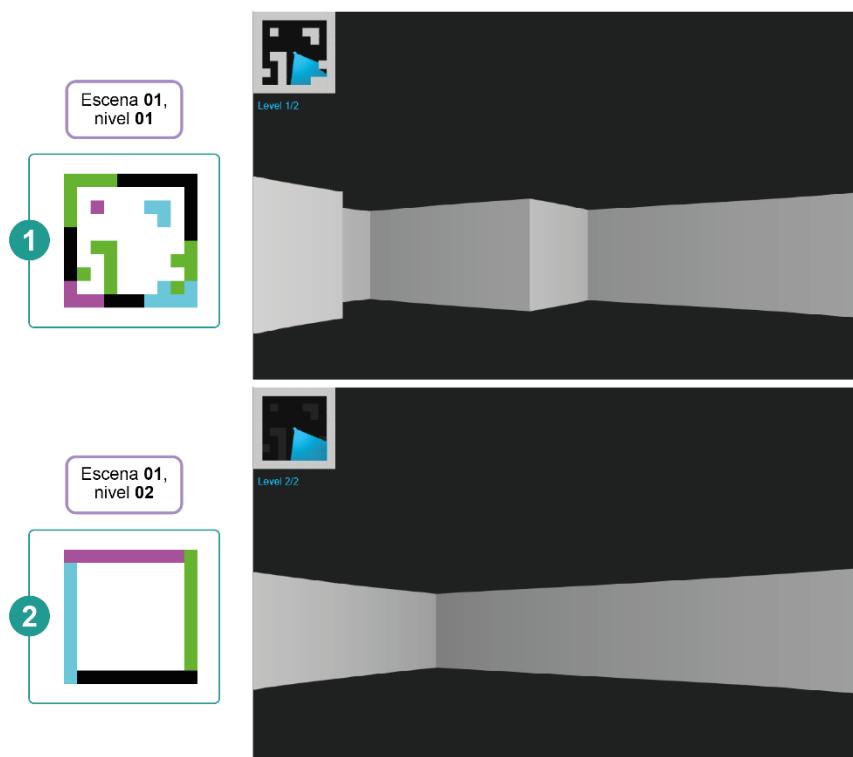


Figura 27. Captura del resultado de la generación de niveles por imagen (capturas propias).

En la Figura 27 se pueden apreciar ambos niveles, cargados satisfactoriamente a raíz de las respectivas imágenes de la izquierda (ambas de resolución 10x10 píxeles). Aunque aún no se pueden apreciar las distinciones entre los muros, sí detectan automáticamente colisiones en todos ellos (dado que la condición para que haya una colisión por parte del jugador es que encuentre un valor distinto de cero —es decir, en nuestro caso, 1, 2, 3 o 4— en la *grid*).

3.12 Texturizado de muros

Esta funcionalidad puede llegar a ser confusa a la hora de su implementación, de modo que el apartado constará de varios diagramas explicativos para desarrollar su cálculo.

3.12.1 Implementación de clase: *VLine*

En el último proceso de codificación de la clase *Render*, encargada de la proyección de muros, se definieron una serie de variables de tipo *array* para almacenar los valores de cada rayo. En este apartado se modificará dicha implementación, ahora sí, creando un solo *array* de valores de una nueva clase llamada *VLine* que almacenará todos los valores necesarios para la proyección de muros y su carga de texturas (y se programará en *renderController.js*).

A nivel conceptual, una “*vLine*” equivale a la columna de píxeles de la textura que corresponde a uno de los rayos del FOV del jugador. Además, cada píxel de dicha columna será un rectángulo almacenado en un *array* de dicha línea.

En primer lugar, en el Diagrama 18 se muestra cómo a cada muro se le aplica una distinta textura, y qué aspecto simularía ésta respecto a la cámara del jugador (especialmente en las ilustraciones de “vista isométrica”).

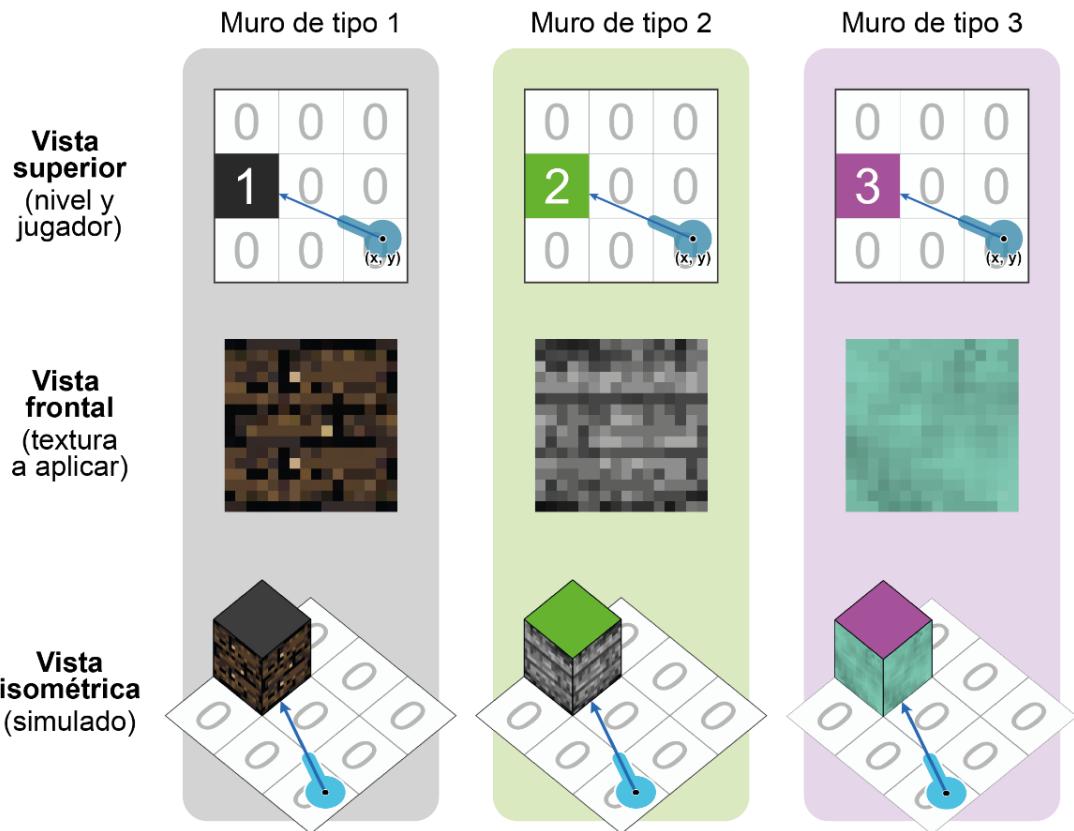


Diagrama 18. Representación de las texturas a aplicar en función del muro de la *grid*, así como el aspecto visual de éstas acorde a la posición de la cámara.

Dado que cada textura se asocia a uno o más números de la *grid* que genera las escenas, y tomando como referencia el segundo muro del Diagrama 18, en el Diagrama 19 puede verse el resultado al repetir el proceso rayo a rayo (y, por ende, línea a línea): **Error! No se encuentra el origen de la referencia..**

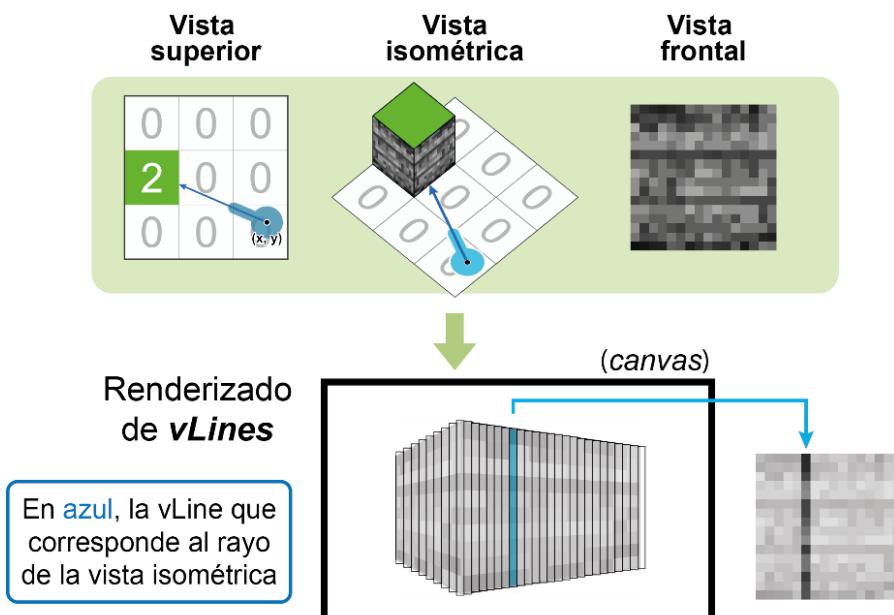


Diagrama 19. Ejemplo de obtención de columna de píxeles de la textura en función del rayo emitido por el jugador.

Dado que cada *VLine* cuenta con valores distintos, en el Diagrama 20 se muestra la visión de los atributos, seguido de una lista con las explicaciones pertinentes, antes de pasar al código.

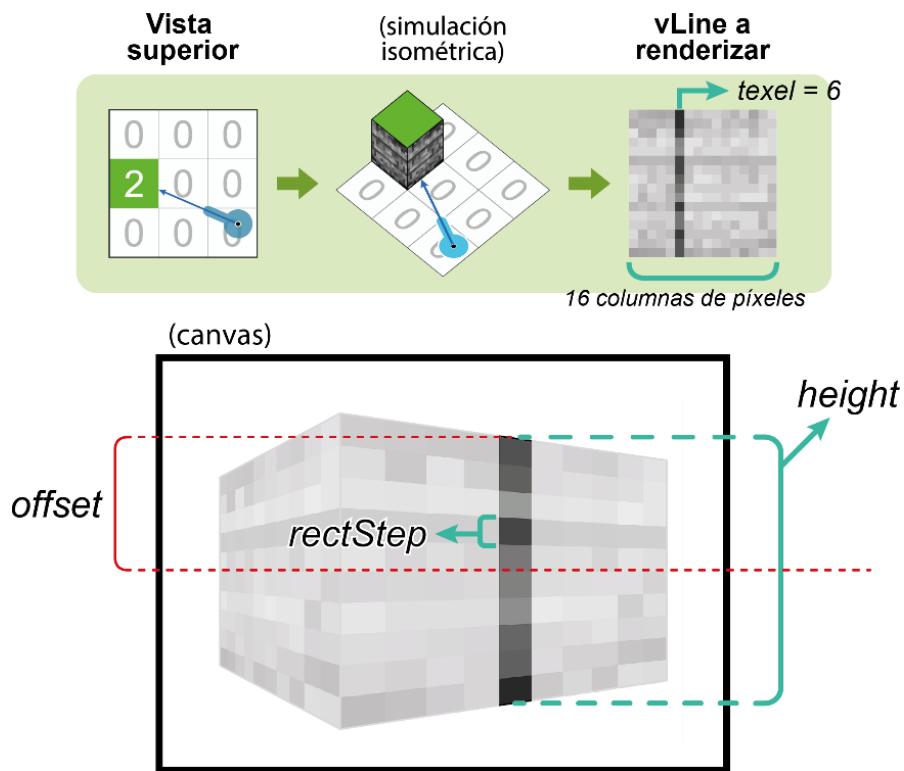


Diagrama 20. Descripción visual de los atributos de la clase *VLine*, con una simulación del canvas.

Las variables indicadas son las siguientes:

- **Index:** equivale al índice del rayo que la *VLine* está representando (índice del rayo dentro del FOV del jugador).
- **Height:** altura de la *VLine*.
- **Offset:** es igual a la mitad de la altura de la línea, y equivale a la distancia a partir de la cual deberá renderizarse (sumado al valor del horizonte de la clase *Render*).
- **TexelColumn** (columna de “texelado”⁶): columna de píxeles de la textura que la *vLine* renderizará.
- **TextureRects:** array de rectángulos que conforman la línea (cada rectángulo o bloque de color será un píxel de la textura).
- **RectStep:** altura de dichos rectángulos.

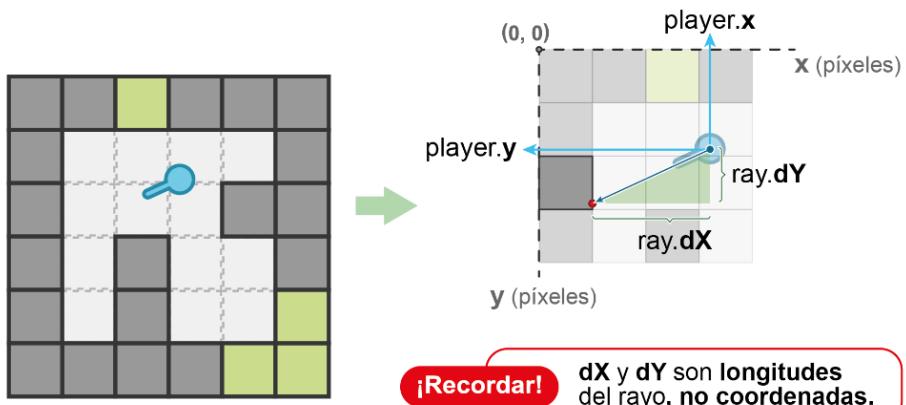
Tras definir los atributos de la clase, se implementará primero la función “*setTexelColumn*”, que es la encargada de calcular qué columna de píxeles (*texelColumn*) de la textura debe asociarse a la *VLine*. Para ello, se hará uso de la distancia recorrida por el rayo, la posición del jugador y el tamaño de las *Tiles*, tal y como se explica en el Diagrama 21.

El concepto tras las operaciones es similar al de la obtención de una columna de la grid en función del tamaño de las *Tiles*: equivale al módulo de la división por *_TILE_SIZE*.

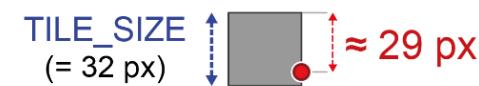
⁶ “Texel” es un acrónimo combinado entre **textura** y **píxel**.

PASO 1

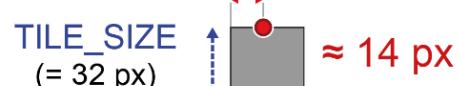
Calcular píxel de colisión
del rayo (respecto a *TILE_SIZE*)



Si la colisión es **horizontal**:
(como en el ejemplo)



Si la colisión fuera **vertical**:



Coordenada (en píxeles) en que se encuentra la colisión.

Diagrama 21. Distinción e ilustración de variables para el cálculo del *Texel* de la colisión (primer paso).

Una vez obtenido el píxel de la colisión del rayo en rangos de 0 al *TILE_SIZE* definido inicialmente, deberemos mapear este valor, y escalarlo a las dimensiones de la textura a aplicar en el muro correspondiente. En el Diagrama 22 se pone como ejemplo un resultado en caso de aplicar una textura de 16 píxeles cuadrados a la misma colisión vista en el Diagrama 21.

Para esta operación, se utilizará la función *map* ofrecida por la librería *P5.js*, y le pasaremos como parámetros los siguientes valores:

- “**valor_a_mapear**”: 29 píxeles.
- “**nI**” (valor inicial del rango de referencia): 0 píxeles.
- “**nF**” (valor final del rango de referencia): equivale a *TILE_SIZE* y son 32 píxeles.
- “**mI**” (valor inicial del rango mapeado): 0 píxeles.
- “**mF**” (valor final del rango mapeado): equivale al ancho de la textura.

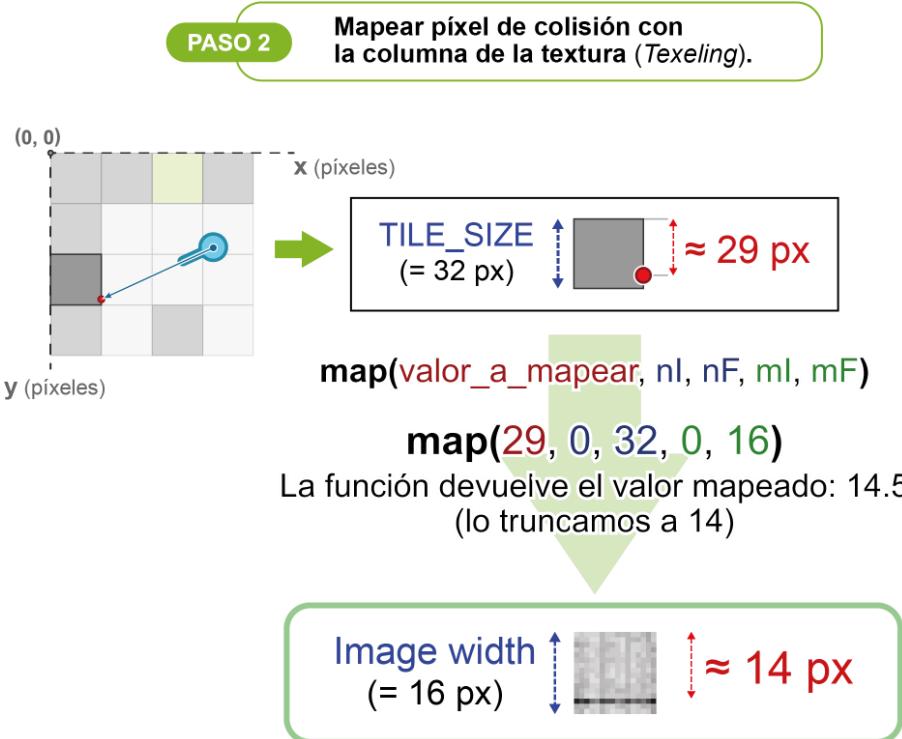


Diagrama 22. Proceso de mapeado del *Texel* mediante la función *map*, de la librería *P5.js*.

Con esto, ya se ha obtenido la fila o columna de píxeles de la textura que corresponde a la *VLine* de la proyección.

A continuación, “*setRectangleTexturing*”, llena el array “*textureRects*” con los colores de los rectángulos que representarán cada píxel de la columna de la textura, uno encima de otro (verticalmente, en el ejemplo, las imágenes son de 16 píxeles cuadrados, de modo que el array tendrá un tamaño de 16 colores).

El último método por implementar en esta clase es la función de renderizado, que pinta el número de rectángulos correspondiente en las coordenadas que tocan, en función del valor de altura de cada bloque de color y su altura. Recibe como parámetro el valor de umbral del horizonte, a partir del cual cuya mitad servirá para centrar la *VLine*, como se mostró en el Diagrama 20 (página 70).

Tras esto, la clase *VLine* ya contiene todos los elementos y funciones necesarias para ejecutar el texturizado de los muros de forma satisfactoria, pero ahora se debe adaptar la clase *Render* (también en el documento “*renderController.js*”) para tratar con las nuevas variables y métodos.

3.12.2 Modificaciones en la clase *Render*

Ya no se necesitan tres arrays distintos con la información obtenida inicialmente en esta clase, sino que ahora se puede simplemente organizar todo en un mero array de clase *VLine*, y el valor del horizonte, según el cual centraremos las *VLines* en el proceso de renderizado de éstas.

```
class Render {
    constructor() {
        // el umbral del horizonte esta a 3/5 de distancia de la base
```

```

        this.horizonThreshold = 3/5;
        this.vLines = [];
    }

```

Ahora en *update* se calculará la altura de cada *VLine* (como se hacía anteriormente), se creará la *VLine* y se calculará su columna de texelado y los colores de los rectángulos (llamadas a *setTexelColumn* y *setRectangleTexturing*). Tras ello, se añade la *VLine* al atributo de tipo array.

No es necesario hacer ningún cambio en *main.js*, dado que ya llamábamos a “*loadProjection*” para renderizar la proyección de los muros y simplemente hemos actualizado los métodos.

Lo que significa que, al ejecutar el *index.html* del proyecto, deberíamos obtener un resultado parecido al de la Figura 28, donde pueden apreciarse las texturas de los tres tipos distintos de muro, pero aún sin pintar el fondo del *canvas* ni el suelo (comúnmente denominados *floor* y *ceiling*).

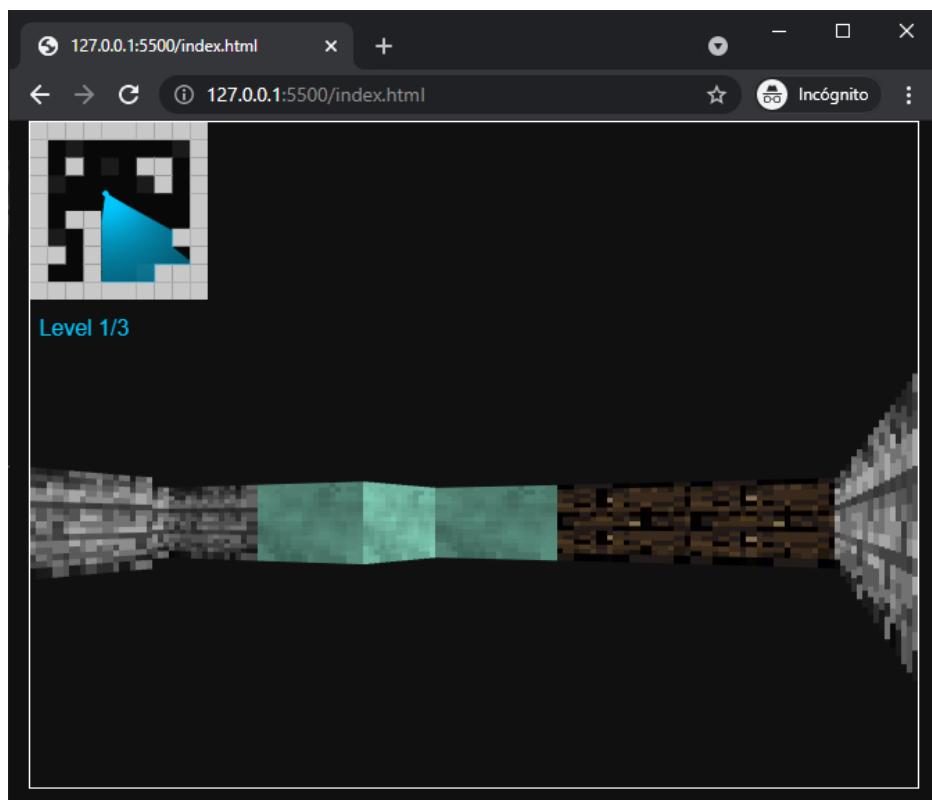


Figura 28. Resultado de ejemplo de texturizar los muros proyectados por el *raycaster* (captura propia).

3.13 Coloreando *floor* y *ceiling*

La última adición visual que queda por añadir al *engine* son los rectángulos que servirán para representar el suelo (*floor*) y el “techo” (*ceiling*) del nivel.

Como se mostró en la teoría del documento, algunos de los primeros *raycasters* que utilizaron este sistema simplemente pintaban un rectángulo detrás de la proyección de los muros (por ejemplo, en *Wolfenstein 3D*), del color que hubieran decidido para el suelo o el cielo. Sin embargo, en nuestro caso iremos un paso más allá y usaremos una serie de operaciones parecidas a las de la generación de texturas, pero esta vez para crear un simple degradado para el suelo.

Para implementar esta última funcionalidad deberemos añadir algunas llamadas a *main.js*, así como programar algunos otros métodos también en la clase *Render* (documento *renderController.js*).

3.13.1 Modificaciones en la clase *Render*

Primero se añadirán dos constantes nuevas en el fichero *renderController.js*, que indicarán el número de colores que el desarrollador decida para el degradado. Por cada uno de estos colores se renderizará un rectángulo encima de otro, cubriendo el trozo de *canvas* que corresponda (parte inferior para el *floor* y superior para el *ceiling*).

```
// nuevas constantes:  
const FLOOR_NUM_HLINES = 16; // número de colores para el suelo  
const CEILING_NUM_HLINES = 16; // número de colores para el techo
```

En el ejemplo se ha optado por 16 colores para ambos, que son relativamente pocos pero más que suficientes para dar una sensación cómoda de degradado.

Tras la definición de las constantes, se definirán algunos nuevos atributos en la clase *Render*, cuya función es la expuesta a continuación:

- *floorGradient*: array de colores para el suelo.
- *floorHLinesSpacing*: altura de cada rectángulo horizontal del degradado (*hLines* viene de *horizontal lines*) para el suelo.
- *ceilingGradient*: array de colores para el techo.
- *ceilingHLinesSpacing*: altura de cada rectángulo horizontal del degradado (*hLines* viene de *horizontal lines*) para el techo.

También habrá que implementar las funciones encargadas de inicializar el array de colores de cada degradado (*setFloor* y *setCeiling*), y los otros dos métodos encargados de renderizar los rectángulos de color para éste (*renderFloor* y *renderCeiling*).

Para generar los colores intermedios del degradado, se deciden el color inicial y final, y se mapea el índice del rectángulo que toque pintar dentro del loop. El factor resultante de dicha operación puede usarse como parámetro de “*lerpColor*”: una función también ofrecida por la librería *P5.js* que interpola entre los dos colores y devuelve el valor resultante de dicho rango en la posición indicada por la variable *interpolation*. A continuación, se muestra el código (donde “c1” y “c2” son los dos colores):

```
for (let i=0; i<FLOOR_NUM_HLINES; i++) {  
    let interpolation = map(i, 0, FLOOR_NUM_HLINES, 0, 1);  
    this.floorGradient.push(lerpColor(c1,c2,interpolation));  
}
```

Este proceso de mapeo e interpolación de los colores es el mismo tanto para el techo como para el suelo: lo único que cambia son las coordenadas en las que habrá que dibujar los rectángulos del degradado a la hora de renderizarlos en el *canvas*.

Ya no es necesario implementar más operaciones ni métodos para nuestro *game engine*. Sólo queda llamar a estas funciones en *main.js*.

3.13.2 Últimas llamadas en *main.js*

Las funciones *setFloor* y *setCeiling* se pueden llamar nada más instanciar la clase *Render*, en *setup*, ya que no necesitamos calcular información adicional y los degradados no cambian durante la ejecución.

Tras añadir dichas llamadas, sólo queda renderizarlos (al principio de la función *draw*, de modo que, al proyectar los muros y el mini mapa, los degradados queden al fondo):

```
function draw() {
    clear("#212121");
    fill("#565656");
    update();

    objRender.renderCeiling();
    objRender.renderFloor();
    objRender.loadProjection();
    objMap.render();
    objPlayer.render();
}
```

Tras esto, el *raycaster* 2D está completamente terminado, y se han realizado todas las funcionalidades planteadas para el *engine*. Al ejecutar el proyecto por última vez, con todas las funcionalidades implementadas, el resultado obtenido es el siguiente:

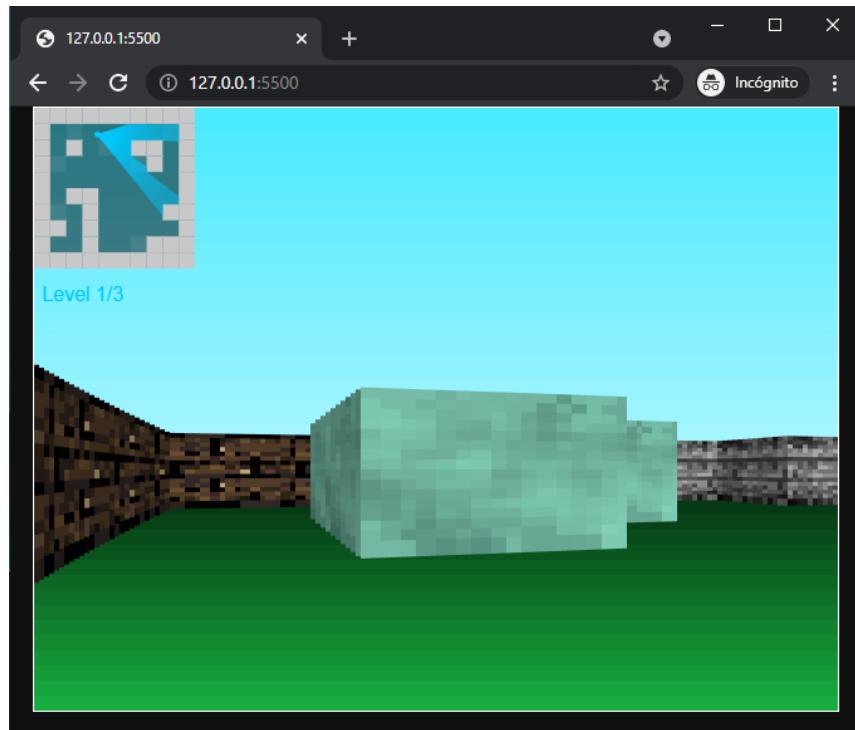


Figura 29. Resultado final de la implementación del *raycaster engine* (captura propia).

4 Motor 3D: “3ngine”

El segundo bloque de desarrollo de esta guía envuelve la implementación de una base para cualquier *game engine* en tres dimensiones. En este caso, no se centrará el desarrollo en ninguna referencia previa concreta (a diferencia del *raycaster*, cuyas funcionalidades iban estrechamente ligadas al sistema implementado en *Wolfenstein 3D*), sino que simplemente se explicarán en profundidad las funcionalidades y métodos del proyecto.

4.1 Nuestro *engine*

El motor que se implementará tras haber terminado el sistema *raycaster* del apartado anterior consiste en una mezcla entre motor de videojuego y sistema de renderizado.

Para llevar a cabo la organización del proyecto se ha establecido que siga la convención MVC (estilo Modelo-Vista-Controlador), residiendo la lógica del sistema en el paquete “Controlador” (*Controller*), los tipos de variables y clases no primitivas en “Modelo” (*Model*), así como la gestión visual del engine y el renderizado y llamadas de gráficos en “Vista” (*View package*). Además de éstos, habrá otros dos paquetes adicionales: otro del estilo *Model*, pero con las clases básicas para la creación de elementos 3D (triángulos, vértices y mallas), y el último, llamado *Utils*, que incluirá una clase —o más, en función del estilo del desarrollador— para las operaciones matemáticas que deberán invocarse en distintas partes del código, y que contendrá una gran serie de métodos estáticos para dichos cálculos.

A nivel de funcionalidades para el usuario, el “3ngine” permitirá la lectura y carga de archivos 3D de formato *.obj*, pero no el texturizado de los triángulos que conforman su superficie. En vez de ello, se dará una lección teórica y práctica de cómo funcionan las transformaciones que se llevan a cabo desde que un vértice se encuentra en un espacio virtual hasta llegar al punto que los jugadores ven por pantalla. De este modo, se ofrece una solución completamente modificable por el desarrollador y a la que se pueden añadir efectos físicos a los elementos de la escena (gravedad, movimiento, rotaciones...) a su gusto.

El proyecto de ejemplo estará programado en **Java**, por varios motivos:

- Es extremadamente más cómodo programar este proyecto en un lenguaje que de por sí esté **orientado a objetos**, ya que los elementos como vértices, triángulos, mallas (*meshes*)... tendrán sus propios métodos y atributos.
- Ya que en el *raycaster* se utilizó JavaScript, que es un lenguaje web; para cubrir el espectro completo, Java permitirá al desarrollador programar este *engine* como una **aplicación independiente** de escritorio, además de compartir la misma estructura en el código que en JavaScript.

Al terminar la implementación del motor, se podrá generar un archivo *.jar* para usarlo como ejecutable, que funciona de forma más intuitiva.

4.2 Funcionalidades

La organización más simple para el desarrollo de los métodos decididos inicialmente para este *engine* es la siguiente (de forma cronológica respecto a su implementación):

- **Definición de la estructura del proyecto.** Se dará una posible muestra de gestión de los archivos del proyecto, en función de las clases decididas para el *engine* (que también se pueden ver en el diagrama UML, en el siguiente apartado).
- **Carga de una escena (cubo de *testing*).** Servirá como ejemplo de una figura simple, cuyos vértices y triángulos estarán situados y conectados a mano por el desarrollador.
- **Transformaciones y proyección de vértices.** En este punto se mostrarán, por orden de ejecución, todas las operaciones que subyacen al proceso de renderizado de un objeto, desde sus vértices en el espacio virtual definido hasta las dos coordenadas del píxel en que se dibujará al terminar el proceso.
- **User input.** Gestionaremos los controles de usuario, para que éste pueda moverse por la escena libremente.
- **Generación de triángulos y método de ordenación.** Hasta este punto los objetos se renderizaban sin malla: sólo puntos y líneas conectando los vértices. Tras la carga de objetos se mostrará cómo pintar la superficie de los polígonos de la malla y se mostrará un método de ordenación de los triángulos como referencia.
- **Creación de luces lineales.** Para dar una muestra de las posibilidades del sistema, se añadirá una fuente de luz a la escena, con una dirección definida, y que generará un efecto de sombreado en los triángulos de la malla.
- **Carga de archivos (.obj).** Por último, al conseguir todas las operaciones de transformación y proyección de los vértices (y, por ende, triángulos) del cubo, en otro subapartado se explicará la estructura de los archivos de formato .obj, uno de los más frecuentes para el modelado de mallas en 3D. Se enseñará a leerlos y a crear las escenas en función de dichos documentos, cargando un archivo de este tipo por escena.
- **Otros métodos de renderizado (wireframe y mezclas).** Ya que en el proyecto se almacenará todo tipo de información sobre los vértices y triángulos que conforman la superficie de los objetos, se podrán modificar los parámetros de renderizado para utilizarlos como sistema de renderizado o análisis de estructuras: no sólo como un *game engine* en 3D.

Al terminar la implementación del *engine* 3D (y, en modo de renderizado *wireframe*), se podrá obtener un resultado similar al mostrado en la Figura 30 (aunque la información del tiempo de ejecución y renderizado del motor no es necesaria en absoluto, da una idea de las posibles utilidades del motor como *software* de análisis de optimización de objetos 3D).

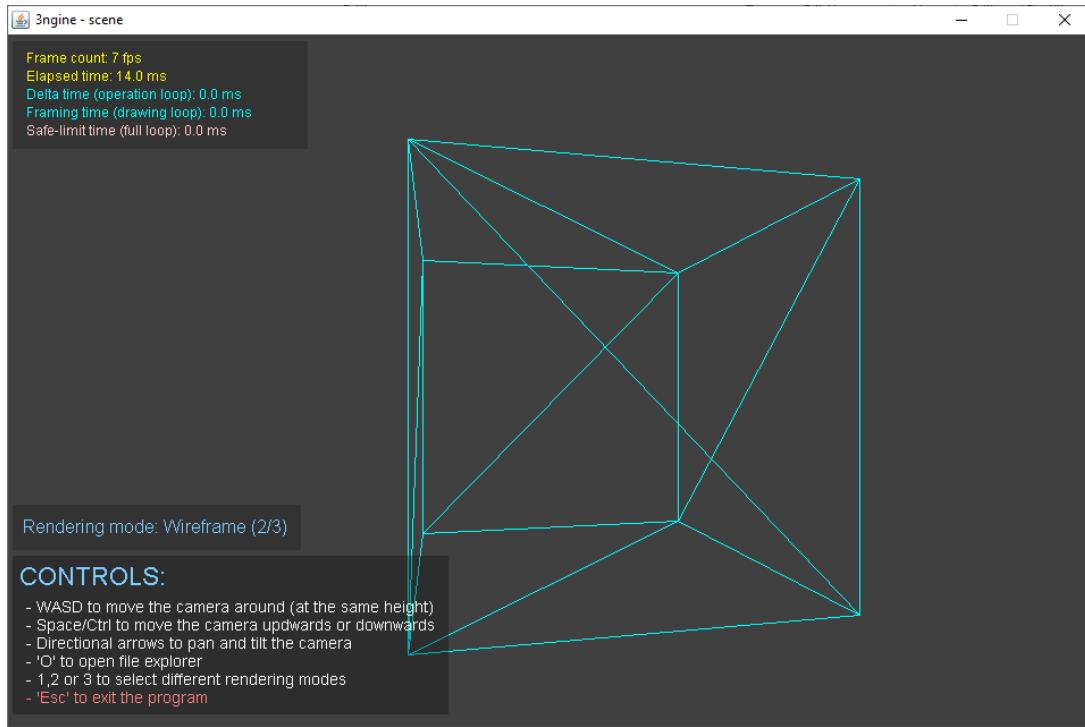


Figura 30. Renderizado en modo *wireframe* con información adicional de testeo y tiempo de cálculo y proyección de un cubo (captura propia del 3ngine).

4.3 Diagrama UML (clases)

A continuación, se presenta el diagrama UML de una posible organización de clases para el *engine*. Como se mencionó anteriormente, está dividido en cinco *packages* de Java (dos de ellos de tipo *Model*), y, aunque las clases sí están relacionadas entre ellas, primero, en los siguientes diagramas se enseñan en violeta dichas variables en cuestión, en vez de unirlas con líneas, ya que habría demasiado texto y no sería intuitivo (más adelante se ofrecerá un diagrama relacional de las clases donde se aclarará dicha estructura de forma más simple).

El primer paquete de clases del proyecto que se puede ver en el Diagrama 23 consiste en las clases que contienen la información necesaria para cargar la estructura de los objetos 3D: la superficie del objeto (*Mesh*), y los vértices y triángulos para crearla.

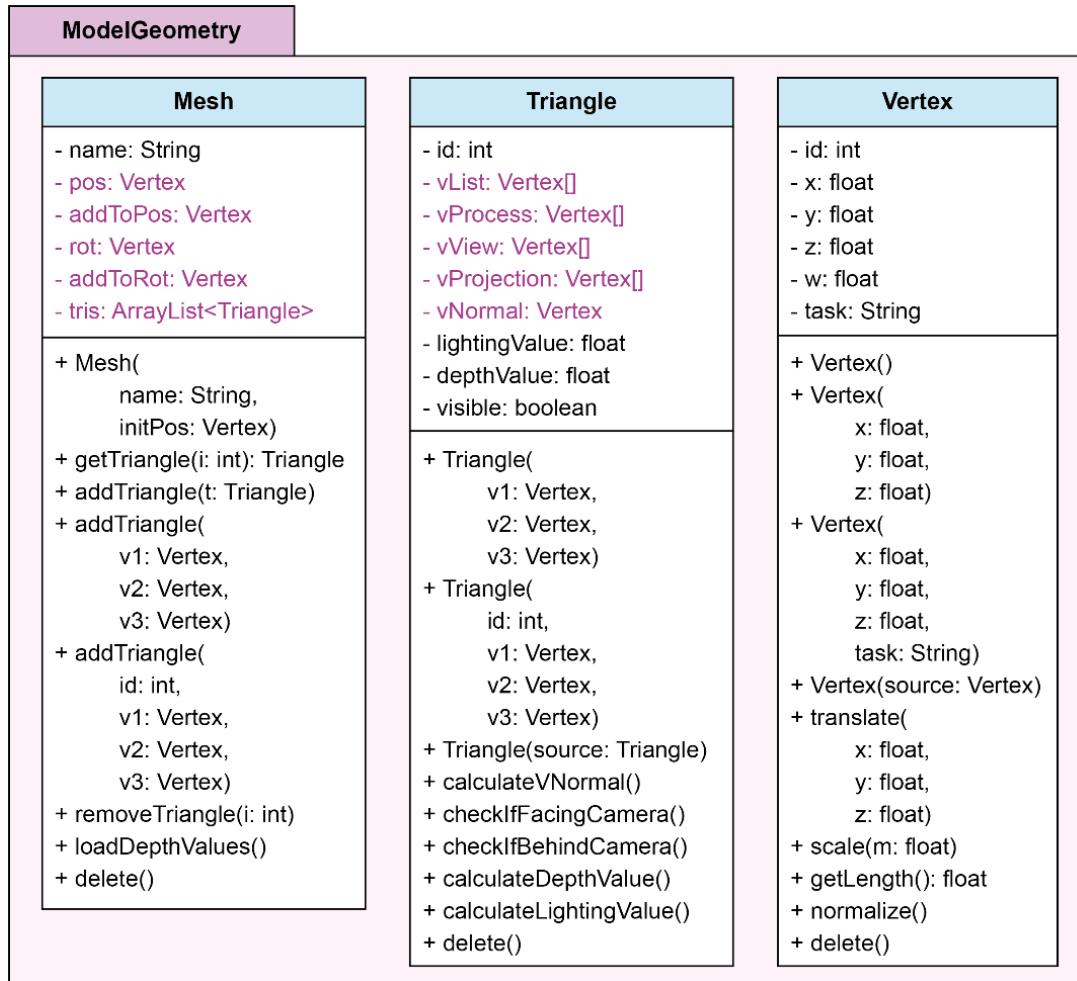


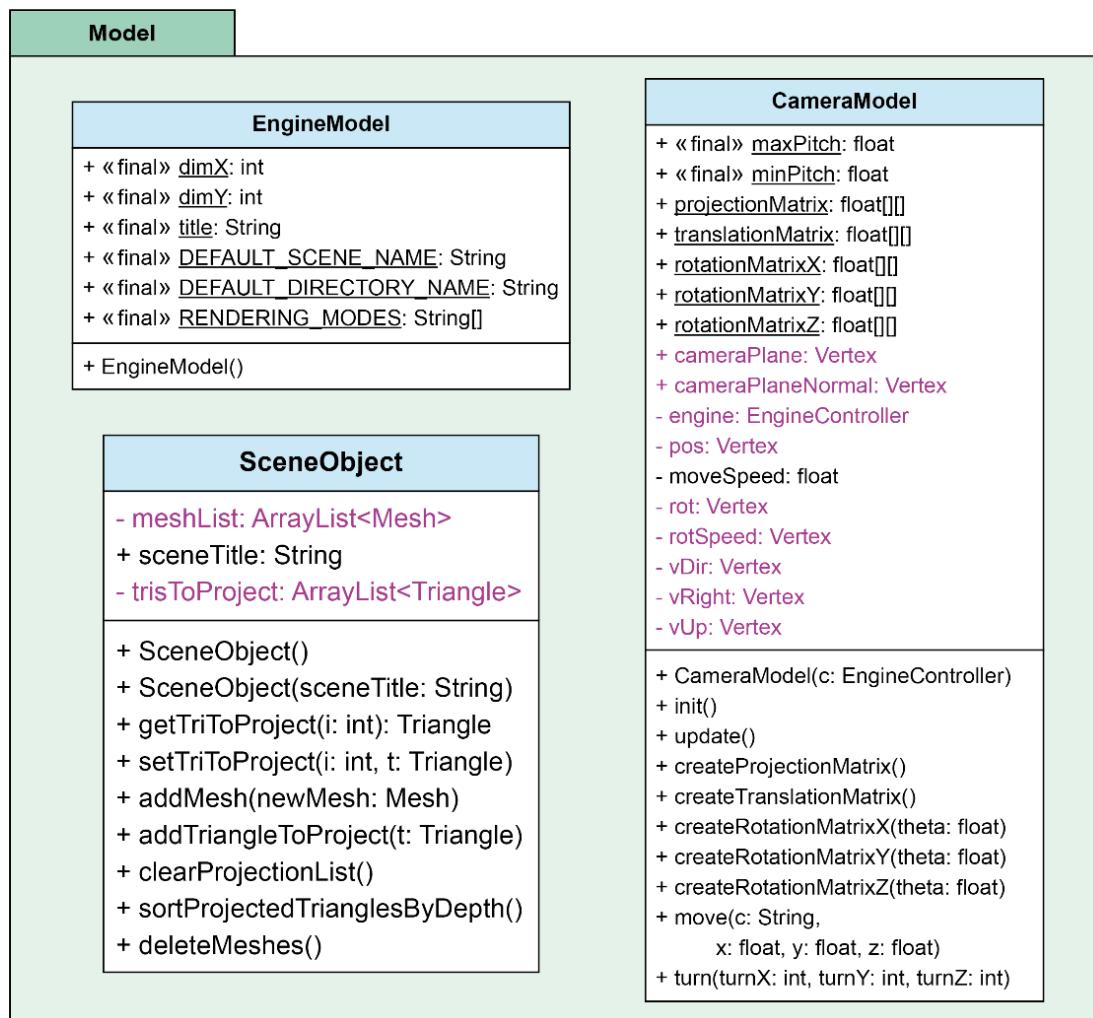
Diagrama 23. Diagrama UML, package **ModelGeometry**, con las clases necesarias para crear la estructura de un objeto 3D.

Este paquete de clases contiene:

- **Mesh**: el tipo malla (*mesh*), que será el objeto que se renderizará en la escena. Está formado por un *array* de triángulos (*triangle*) y algunos atributos de tipo vértice (*vertex*) para reutilizar los tipos, y que consisten en la posición, una modificación de la posición de la malla (*addToPos*), e igual con la rotación, por si el desarrollador quiere hacer que el ítem en cuestión gire en cualquiera de los tres ejes de coordenadas.
- **Triangle**: son los triángulos que forman la superficie de la malla. Tienen un identificador, por si se quisiera utilizar la información en algún momento, un valor de luz (para cuando se implemente la luz en la escena), otro de profundidad y un *booleano* que indicará si el triángulo es visible por parte de la cámara. En cuanto a los tres vértices que conforman el triángulo, cabe destacar que contiene cuatro versiones de éstos:
 - **vList** son los vértices originales de la forma, sin aplicar ninguna transformación.
 - **vProcess** son esos mismos vértices tras aplicar transformaciones a los objetos: traslaciones, rotación, etc.
 - **vView** es el resultado de calcular la nueva posición de los vértices tras las modificaciones de la cámara (es decir, en función de la posición y los ángulos de visión del jugador).
 - **vProjection** es el último valor de los vértices, y es su valor tras proyectarlos a la pantalla mediante la *projection matrix* de la cámara.

- **Vertex**: son los vértices que conforman los triángulos (y, por ende, también la malla del objeto). Contienen un identificador, del mismo modo que los mencionados previamente, además de las coordenadas en el espacio virtual del *engine*, un valor adicional ("w") que se empleará en las matrices, a la hora de llevar a cabo las multiplicaciones para los cálculos y transformaciones de éstos; y una cadena de caracteres para describir la función del vértice si fuera necesario.

El segundo paquete, mostrado en el Diagrama 24, consiste en las tres clases que conforman la estructura base de elementos necesarios para el funcionamiento del *engine*, y contiene el *EngineModel*, con las dimensiones e información básica del motor, la escena a cargar (que contendrá una *Mesh*) y la cámara, para renderizar dicha escena.



Tras tener los dos paquetes con las clases modelo definidos, se puede mostrar el proceso técnico de fondo del *engine*, con el *loop* principal del motor y la clase que gestiona los objetos y llama a las transformaciones del motor. También se incluyen en este *package* la clase de controles e *input* del usuario y la que implementa la lectura de objetos 3D en formato .obj.

A diferencia de las relaciones entre clases, en el Diagrama 25 sí que se muestra (además de lo mencionado previamente) cómo la clase *EngineController* implementa la interfaz de métodos

KeyListener, sobrescribiendo las funciones que detectan la pulsación de las teclas de control, además de la clase *EngineLoopThread*, que es el hilo ejecutor principal del motor y es una extensión de la superclase *Thread*.

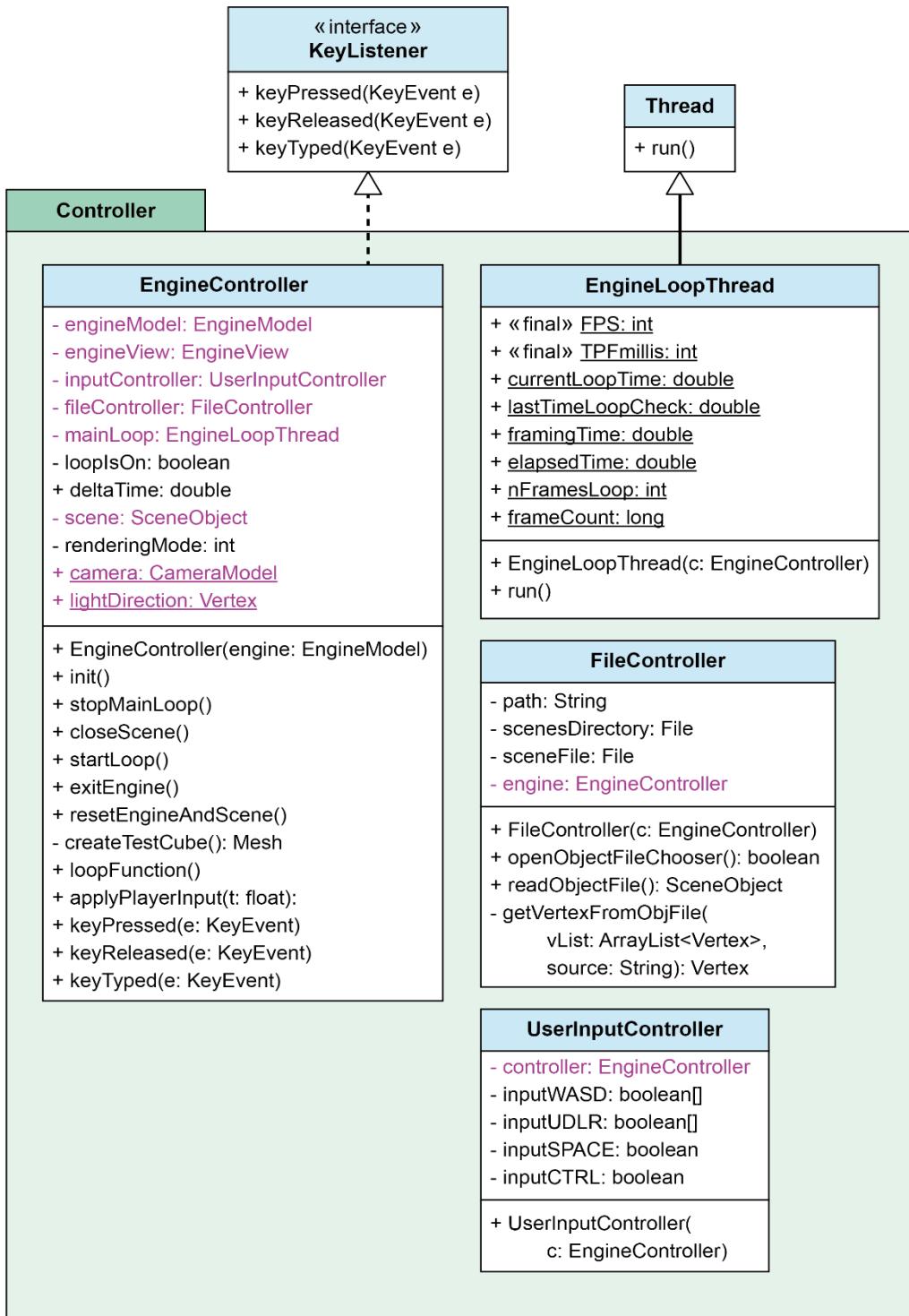


Diagrama 25. Diagrama UML, *package Controller*, para la gestión de elementos y procesos del *engine*.

Por último, en el Diagrama 26 se muestran la clase *EngineView*, agregada al paquete *View* y que se encarga de los elementos gráficos del motor (además de los atributos que de por sí necesita la ventana que se abre al ejecutar el programa); una clase adicional repleta de métodos varios de cálculo (principalmente transformaciones y operaciones matemáticas varias como

multiplicación de matrices y vectores, sumas, restas y divisiones de vértices, etc.), *UtilsMath*; y la clase *Main*, que crea y llama al *engineController* y *engineModel* (e inicia el *loop* principal del proyecto) para dar comienzo al proceso del *engine*.

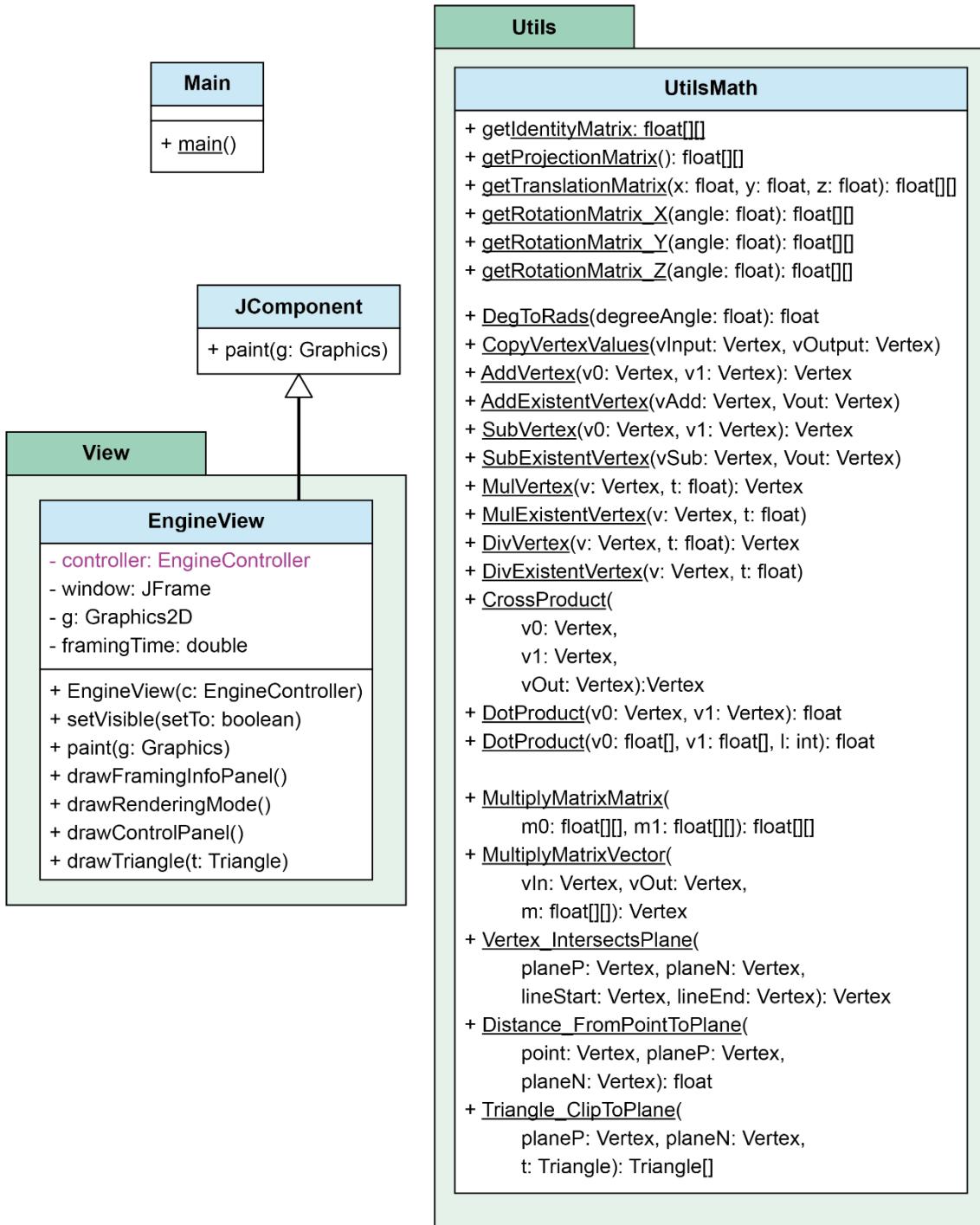


Diagrama 26. Diagrama UML, *package View* y *Utils*, para renderizar los gráficos y crear la ventana del *engine*, y ofrecer una clase con métodos de cálculos matemáticos varios.

Como ocurrió con algunas clases del *package Controller*, en este caso *View* también extiende de la clase *JComponent*, de la cual sobrescribe el método *paint*, para dibujar en la ventana (mediante el parámetro *Graphics*).

4.4 Diagrama UML (relacional)

Dado que en el anterior punto ya se cubrió la parte de contenido técnico de cada clase, así como el listado de métodos y atributos de éstas, seguidamente se ofrece una versión del diagrama UML centrada en mostrar cómo se relacionan las clases definidas anteriormente. De este modo el desarrollador puede decidir por qué clases le conviene comenzar a implementar el código, pudiendo evitar una gran cantidad de tiempo y posibles errores durante el proceso de *backtracking*.

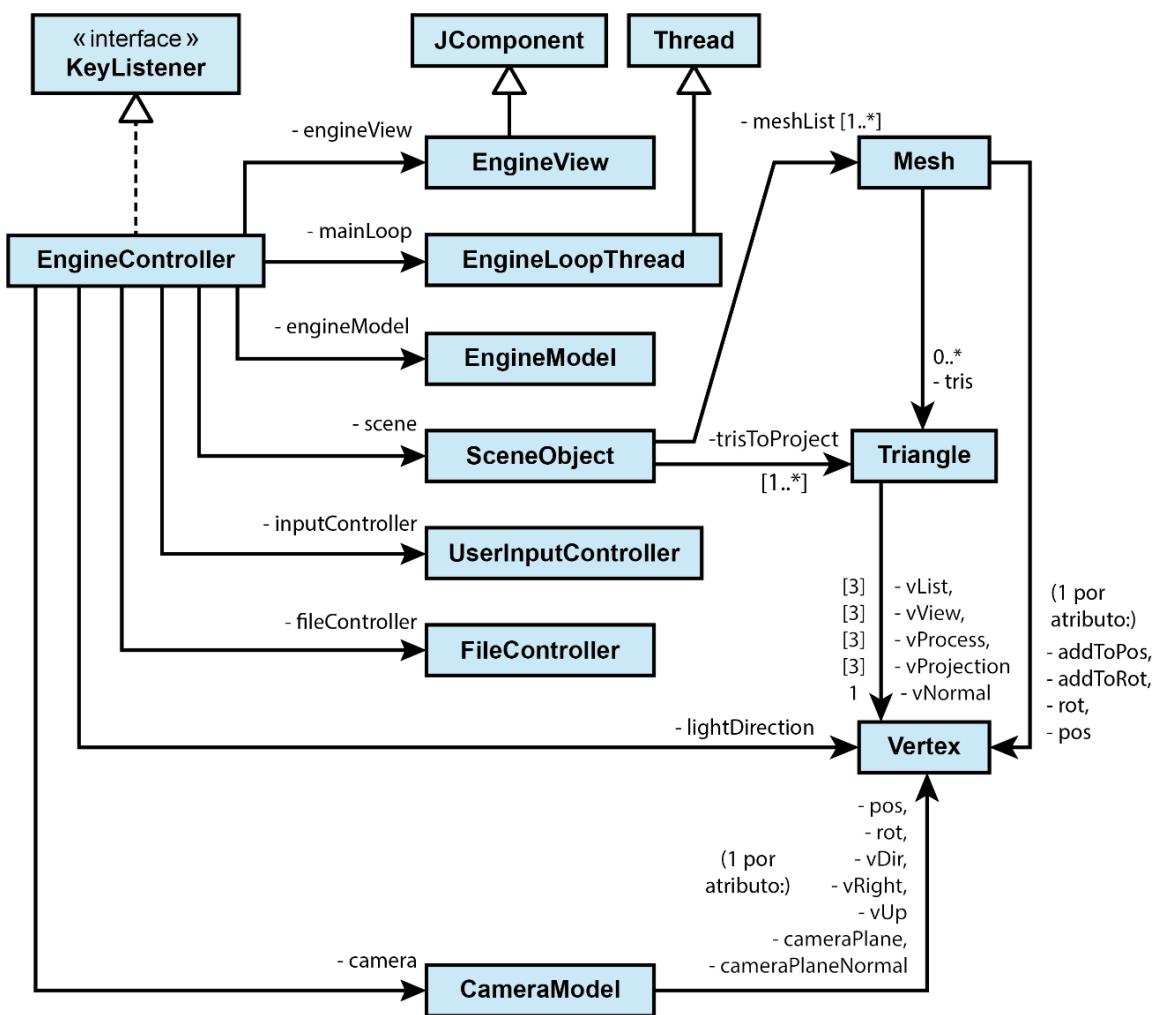


Diagrama 27. Diagrama relacional del 3ngine, con la distribución de clases asignada a los controladores, así como herencias o implementaciones externas y un ejemplo de su multiplicidad (puede haber variaciones según la voluntad o preferencias del desarrollador).

Una posible distribución de clases (con tantas reutilizaciones como se permiten, en particular de la clase **Vertex**) se muestra en el Diagrama 27, así como el concepto general de dicha organización: cada elemento a implementar tiene una función única e intransferible y, de este modo, se podrá sacar el máximo partido a la consecuente estructura del proyecto y conseguir que no haya relaciones incongruentes entre clases.

Pese a no ser mostrado en el diagrama, es recomendable que cada clase del *package View*, así como el *FileController*, *CameraModel* y *SceneObject* también contengan una referencia al *EngineController*.

Como último dato sobre el esquema, si el atributo no marca ninguna multiplicidad en su relación entre clases, por defecto es de multiplicidad 1 (*engineView*, *scene*, *mainLoop*, *engineModel*, *fileController*, *lightDirection*...).

5 Implementación del 3ngine

De nuevo, se dividirá la implementación del motor en varias fases:

5.1 Definición de la estructura del proyecto.

En esta primera parte se creará el proyecto con las clases básicas para contener la información del *engine*, así como una escena vacía. Es importante destacar que el orden cronológico más seguro para implementar las clases siguiendo este estilo es el siguiente: clases del paquete “modelo”, paquete “controlador”, y paquete “vista”.

5.1.1 Montar Modelo-Vista-Controlador.

A continuación, se muestra el Diagrama 28 como ejemplo para organizar las clases iniciales (de este subapartado) de forma más intuitiva:

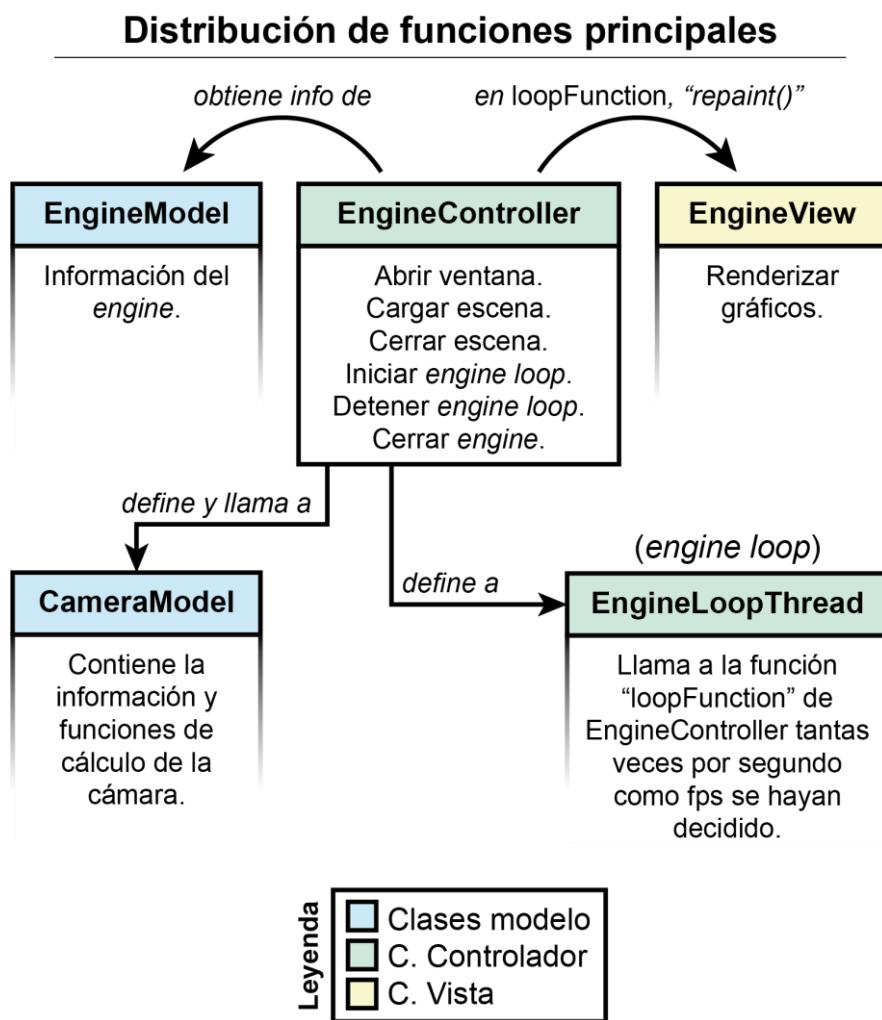


Diagrama 28. Modelo MVC básico inicial para el 3ngine, distribuido por clases y funciones.

Las primeras clases que el desarrollador deberá definir, por tanto, son las del *package Model*. Para crear un proyecto básico capaz de abrir una ventana y renderizar gráficos bastará con *EngineModel*. En el 3ngine se han definido como atributos varios datos sobre el motor, como las dimensiones de la ventana, el título de ésta y los distintos tipos de renderizado con los que

contará el *engine* (todos estos datos pueden apreciarse en el diagrama UML de clases presentado en el anterior apartado).

Según futuras funcionalidades que el desarrollador quiera implementar, éste debería incluir la información que no dependa de ninguna otra fuente en esta clase modelo.

Además del modelo del 3ngine, también se puede crear el *CameraModel*, que contiene los atributos para la funcionalidad de la cámara en el sistema (posición, velocidad de movimiento, rotación y velocidad de rotación, el vector hacia donde mira...). También se definen en dicha clase las matrices de tipo *float* que se utilizarán para los cálculos de las transformaciones de los vértices (“projectionMatrix” —la matriz con los valores para calcular la proyección de los vértices de la malla en la ventana—, “translationMatrix” —matriz para calcular el desplazamiento de los objetos en el entorno 3D virtual— y una “rotationMatrix” —para rotar los elementos de la escena— por cada eje de coordenadas). Sin embargo, aún no se hará uso de todos estos atributos, ya que la transformación de vértices será más adelante.

El controlador básico que gestiona y/o utiliza la información cedida por estas clases de tipo modelo será *EngineController*. También es el encargado de inicializar la escena del motor (cámara, mallas, luz, etc.) y contiene el método que nuestro hilo de ejecución repetirá en bucle (como también se muestra en el UML, su nombre es “loopFunction”), una vez por *frame*, para dibujar en la ventana de gráficos.

Sin embargo, la llamada a esta “loopFunction” se hace en la otra clase principal del *package Controller*: *EngineLoopThread*. Ésta se mantiene como hilo de ejecución principal del motor. En caso de que el desarrollador haya decidido (por ejemplo), una frecuencia de muestreo de 60 fps, llamará a la función mencionada previamente 60 veces por segundo.

Dado que este paso intermedio no es habitual en el estilo de arquitectura MVC, en el Diagrama 29 se muestra visualmente el procedimiento para el hilo de ejecución principal del motor.

Ejecución del *loop* del 3ngine (con llamadas a funciones):

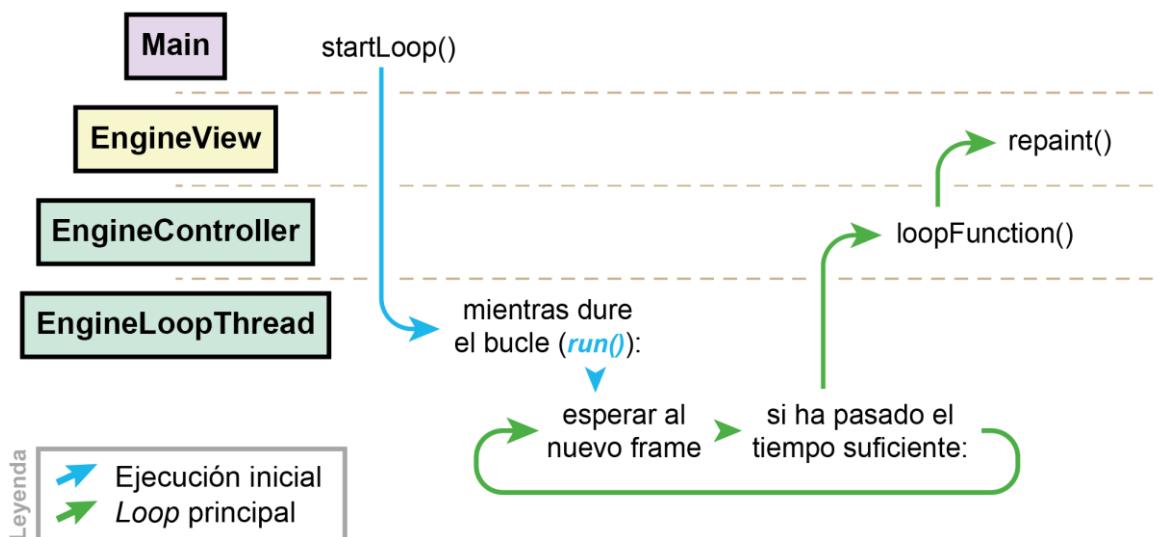


Diagrama 29. Loop principal del 3ngine, agrupado por clases y llamadas a funciones.

De este modo, siempre que un usuario abra una escena, *EngineController* cargará los elementos de la escena y el atributo de tipo *EngineLoopThread* empezará a hacer los cálculos en bucle para renderizar los elementos por pantalla.

En cuanto al loop principal del motor (este *EngineLoopThread*), se puede implementar a estas alturas iniciales del proyecto sin necesitar ninguna otra modificación más adelante, de modo que a continuación se muestra la operativa básica para su desarrollo.

La clase deberá extender de *Thread* para poder utilizar el método *run* y así iniciar otro hilo de ejecución, como se muestra a continuación.

```
// definiendo la clase
public class EngineLoopThread extends Thread {
```

En el método *run* deberemos calcular, si no se ha hecho previamente —en el constructor, por ejemplo—, el tiempo que el programa debe esperar hasta llamar a la función *loopFunction* del controlador (lo que significa que, obligatoriamente, esta clase debe contener un atributo de acceso a la clase *EngineController* del motor).

Para calcular dicho tiempo se puede hacer uso de “currentTimeMillis”, que devuelve la cantidad de segundos que han pasado desde el 1 de enero de 1970 hasta la llamada a dicho método. Al llamar dos veces a esta función y restar ambos tiempos obtenemos la diferencia de segundos entre la primera y la segunda llamada. Tras ello sólo queda comparar si esa cantidad de segundos ha superado el tiempo estimado que el programa debe esperar (cuyo valor dependerá de los fps que el desarrollador decida para el *software*).

En el código se muestra un ejemplo si el desarrollador decidiera un muestreo de 60fps para la ejecución (la variable en que lo almacena es “timeToPass”, y está en milisegundos).

```
@Override
public void run() {
    double time0 = System.currentTimeMillis();
    int FPS = 60;
    int timeToPass = 1000/FPS;
```

Cabe mencionar que debemos acceder a dos bucles: el primero de ellos se mantendrá activo mientras el programa se esté ejecutando (hasta que el usuario cierre el motor o la escena), *booleano* indicado en “*isLoopOn*”, variable pública estática del controlador. El segundo bucle será el que irá sumando el tiempo que ha pasado por cada llamada a “*currentTimeMillis*” hasta que los segundos sean iguales o superiores al “*timeToPass*”, mencionado previamente.

```
while (this.controller.isLoopOn()) {
    double time1 = System.currentTimeMillis();

    if (time1 - time0 >= timeToPass) {
        // si entra, debemos cargar el nuevo "frame"
        time0 = System.currentTimeMillis();
        this.controller.loopFunction();
    }
}
```

```
    } // cierra el bucle de la escena
} // cierra la definición del método "run"
} // cierra la definición de la clase EngineLoopThread
```

Con este código termina la implementación del *loop* principal del *engine* (el método *loopFunction* será quien lleve a cabo todas las operaciones de los vértices y, al final, llame a la función *repaint*, del *EngineView*, para que dibuje el resultado en la ventana).

Sin embargo, para que el motor reconozca la función *repaint* en la clase vista, ésta debe extender de “*JComponent*”, y sobrescribir el método “*paint*” (que recibirá como parámetro una variable de tipo *Graphics*). Todas estas indicaciones ya se mostraban en el diagrama de clases UML del anterior apartado.

Para el renderizado de gráficos en Java, se emplearán las funciones ofrecidas por la clase *Graphics2D* para dibujar líneas o formas poligonales en la ventana. Nada de imágenes ni mapas de color preestablecidos. Estas funciones se llaman, después de hacer un *cast* para cambiar el tipo de contexto gráfico, desde el objeto de tipo *Graphics* ofrecido como parámetro de dicho método *paint*.

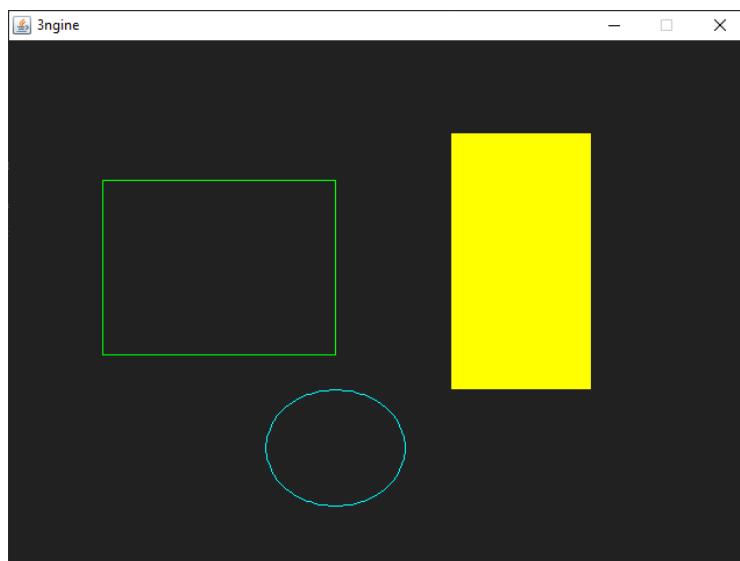


Figura 31. Muestra de ventana con figuras sencillas, utilizando `java.awt.Graphics` (captura propia).

A continuación, se listan algunas funciones que se pueden usar en la clase *EngineView* para poder dibujar formas geométricas como las de la Figura 31:

- **Graphics2D.setBackground()** necesita como parámetro un color que usará como fondo de la ventana. Debería ser la primera función a llamar en el proceso de *render*.
- **Graphics2D.clearRect()** limpia la ventana en las coordenadas indicadas como parámetro, y en un área rectangular del tamaño indicado por el desarrollador.
- **Graphics2D.setColor()** necesita como parámetro un color decidido por el desarrollador, y que utilizará al pintar todos los objetos que se dibujen tras este uso.
- **Graphics2D.fillRect()** es parecido a la estructura de “*clearRect*”, pero, en vez de borrar, pinta un rectángulo del último color establecido por “*setColor*” en las coordenadas y de tamaño indicados por parámetro.

- **Graphics2D.drawRect()** es igual que “fillRect”, solo que no pinta el interior del rectángulo, sólo su perímetro o contorno.
- **Graphics2D.drawOval()** dibuja el perímetro de un círculo (que puede ser un óvalo), con los mismos parámetros que “fillRect” y “drawRect”.

Sin embargo, como se ha mencionado, para poder emplear estas funciones antes deberemos hacer un *cast*, convirtiendo el parámetro de tipo *Graphics* a *Graphics2D*, como se muestra en el código, a continuación:

```
@Override
public void paint(Graphics g) {
    Graphics2D g2 = (Graphics2D) g;

    g2.setBackground(new Color(0x212121));
    g2.clearRect(0, 0, this.windowWidth, this.windowHeight);

    g2.setColor(Color.YELLOW);
    g2.fillRect(380, 80, 120, 220);
```

La anterior función pintará el rectángulo en las coordenadas (380, 80), de 120 píxeles de ancho y 220 de alto, de color amarillo. El mismo que puede apreciarse a la derecha, en la Figura 31.

```
g2.setColor(Color.GREEN);
g2.drawRect(80, 120, 200, 150);
g2.setColor(Color.CYAN);
g2.drawOval(220, 300, 120, 100);
}
```

Como se puede apreciar en “setBackground” y “setColor”, hay varias maneras de asignar un color a los elementos: puede indicarse el valor de los factores RGB (y *Alpha*, si se quisiera agregar transparencia), o utilizar los colores estáticos ofrecidos por la clase *Color*, igual que sucedía con JavaScript.

5.1.2 Contenido de la escena

Tras poder abrir una ventana de gráficos y llamarla exitosamente desde el hilo de ejecución principal del *engine loop*, se definirá la estructura de la escena y la distribución de sus elementos en cada clase.

A estas alturas es cuando deberían desarrollarse las clases *Model* centradas en la geometría de los objetos; a saber: mallas (nuestra clase *Mesh* —conocidos comúnmente como “objetos” o “GameObject” en algunos motores y editores de videojuegos como Unity—), triángulos (clase *Triangle*) y vértices (clase *Vertex*).

El otro elemento indispensable para el renderizado de la escena sería la cámara (clase *CameraModel*, el mencionado en el anterior subapartado) y la luz (aunque una variable de tipo *Vertex* servirá para ésta última). Sin embargo, a estas alturas aún no es necesario implementar ninguna de estas dos funcionalidades.

Pese a que ya se mostró la distribución de elementos en los diagramas UML, en el Diagrama 30 se reincide en la estructura a programar con los atributos imprescindibles para conformar la escena del *engine*.

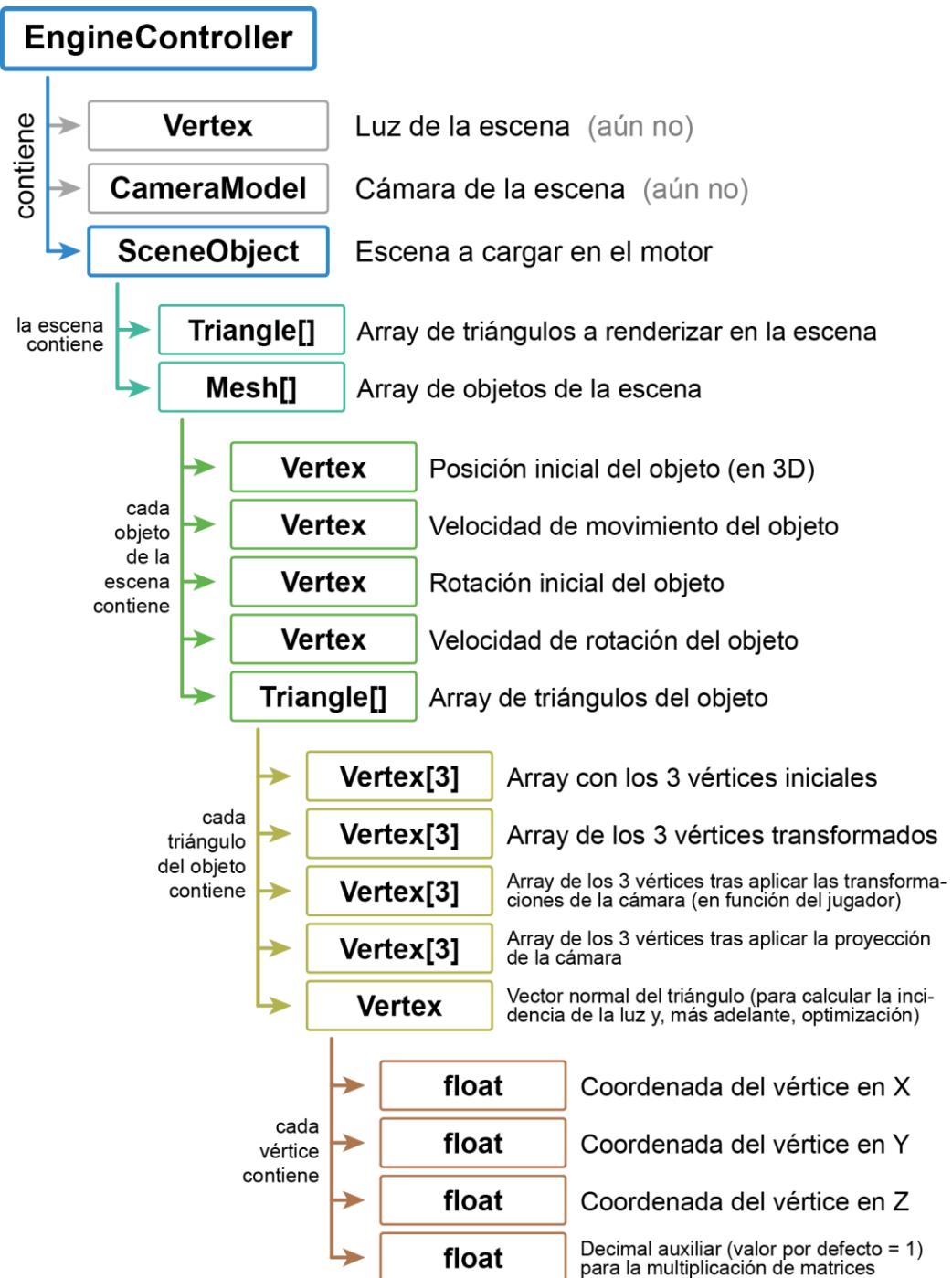


Diagrama 30. Estructura básica de una escena del EngineController, con la clase y descripción de sus atributos.

Para disipar posibles dudas sobre los dos atributos que la escena contiene, el concepto tras ambas variables es que el array de objetos de la escena contenga todos los vértices de todos los triángulos de todos los objetos, pero, una vez calculadas las transformaciones a aplicar a los vértices, el programa añade al array de triángulos a renderizar **únicamente los triángulos que la cámara es capaz de ver** (en breve, porque si la cámara no enfoca al triángulo, significa que éste

no es visible por parte del jugador, de modo que no es necesario perder recursos en dibujar dicho elemento por pantalla).

En el proyecto se han implementado también las funciones para eliminar vértices, triángulos y objetos, de modo que el usuario pueda abrir distintas escenas sin cerrar la instancia del programa (método `stopMainLoop` y `closeScene` de `EngineController`). Simplemente se detiene el bucle del método `run` (recordemos, el de la clase `EngineLoopThread`) y se iguala a “null” el atributo correspondiente del `EngineController`, así como la luz (tipo vértice) y la cámara (`CameraModel`). Tras igualar a null también (y por último) la variable `SceneObject`, se crea un nuevo `EngineLoopThread` y una nueva escena, con todos los elementos descritos previamente (por eso es recomendable implementar un método “init” en `EngineController`, al que se pueda llamar para crear una escena).

5.2 Carga de una escena (cubo de testeo)

En el método “init” del `EngineController` se ejecutan las siguientes instrucciones (ordenadas cronológicamente), en lo referente a la generación de escenas:

- Se crea una escena en blanco, y se le asigna un título (el título no es necesario, pero es recomendable darle un nombre a la ventana, como mínimo).
- Crea una cámara y la sitúa en una posición determinada (coordenadas **X e Y iguales a 0**, pero **Z superior a cero**, para no confundir a la hora de proyectar vértices inicialmente).
- Crea una luz vectorial para la escena (aunque aún no se tenga en consideración para el renderizado), y normalizar el vector.
- Añadir el o los objetos que correspondan a la escena.

Una de las formas más simples para hacer pruebas y que se puede modelar escribiendo los vértices uno a uno es un cubo, de modo que es recomendable empezar por definir uno (es importante añadirlo a la lista de mallas —`SceneObject.meshList`— de la escena, de lo contrario es como si no existiera).

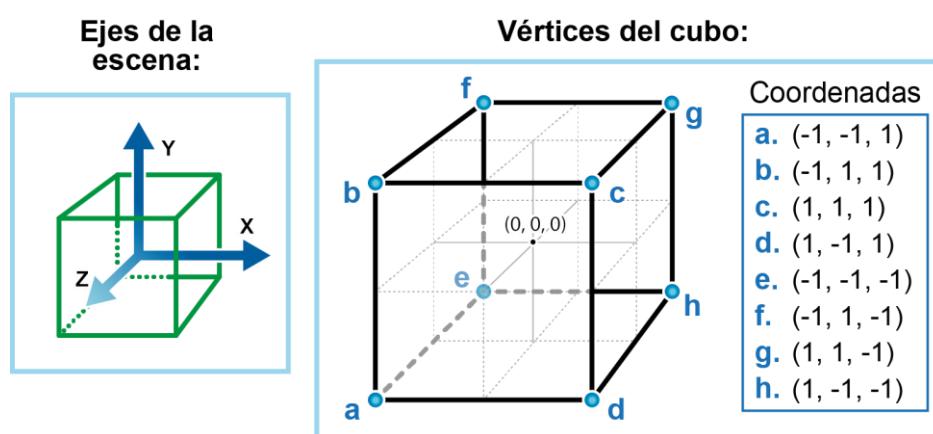


Diagrama 31. Estructura del cubo (vértices, coordenadas y ejes).

En el 3ngine, este método se define como “`createTestCube`”, y genera las seis caras mediante dos triángulos para cada una de ellas. A la hora de crear los triángulos es importante mantener el mismo sentido, de modo que los vectores normales de cada triángulo enfoquen hacia afuera. Toda esta información se ilustra más intuitivamente en el Diagrama 31 (los ejes de coordenadas

y vértices, así como sus posiciones) y el Diagrama 32 (triángulos de cada cara y la dirección en la que avanzará el vector normal de cada uno de ellos cuando los calculemos, más adelante).

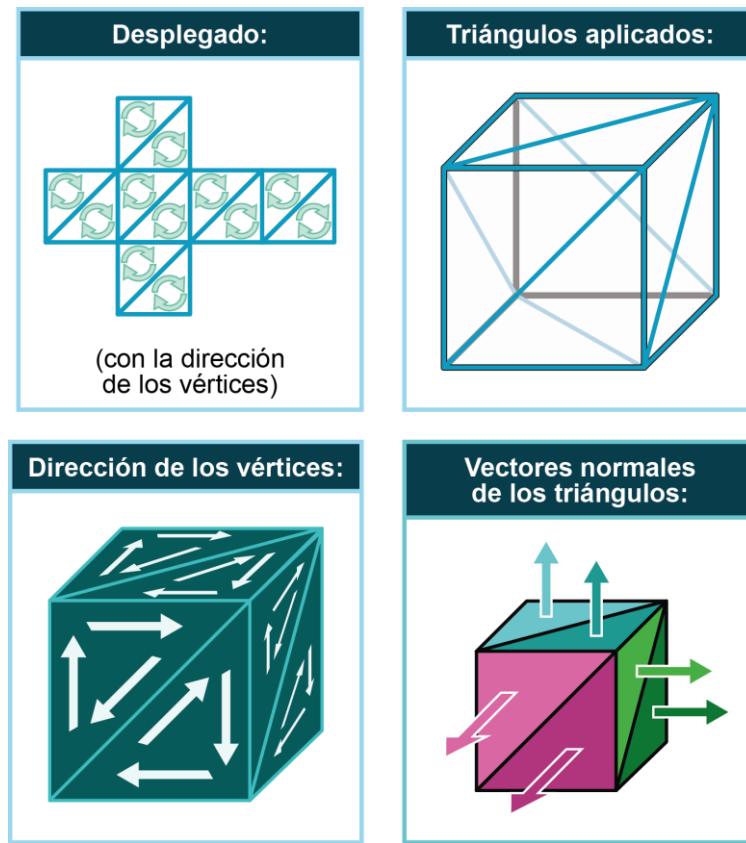


Diagrama 32. Triángulos del cubo, orientación para definir los vértices y vectores normales resultantes.

Dado que las clases modelo de geometría ya están hechas, podemos pasar directamente al método para crear el cubo, que consiste en crear los ocho vértices que forman el polígono, preferiblemente agrupándolos por cara, como se muestra en el Diagrama 33:

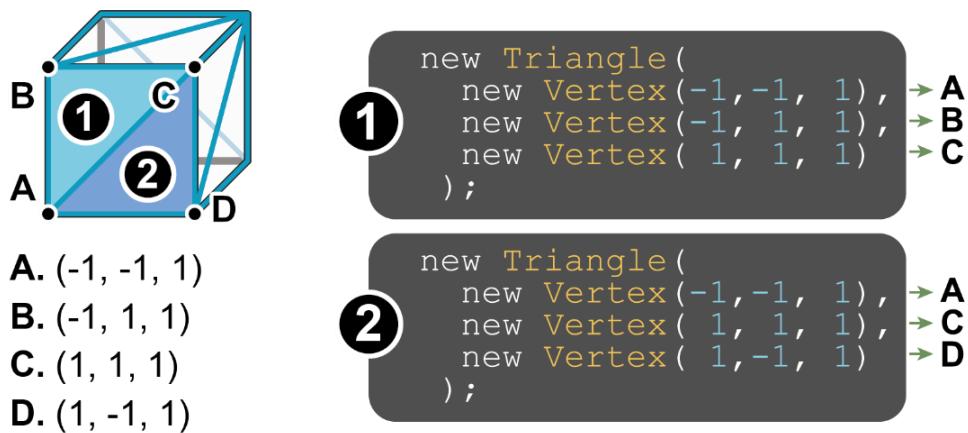


Diagrama 33. Creación de triángulos de una cara del cubo de testeо.

Después de crear los doce triángulos (son seis caras, cada una de las cuales está formada por dos de ellos), hay que crear el cubo, que es de clase *Mesh*; y añadirlos a su atributo array de triángulos. Por último, en el método de creación de la escena, agregamos el cubo al *array* de

Meshes de la variable *SceneModel*, y con esto ya se ha generado un cubo virtual para nuestro *engine*.

5.3 Transformaciones de vértices

Tras haber creado y añadido un cubo a la escena, se implementarán las operaciones del entorno virtual 3D para poder representarlo en la pantalla, que es un entorno 2D. Esta serie de cálculos, listada en el Diagrama 34, se programará en el método “loopFunction”, el llamado por el hilo de ejecución tantas veces por segundo como *frames* decida el desarrollador, y es **imprescindible que sigan el orden indicado** en el organigrama. Hay **3 fases**, según su **función**:

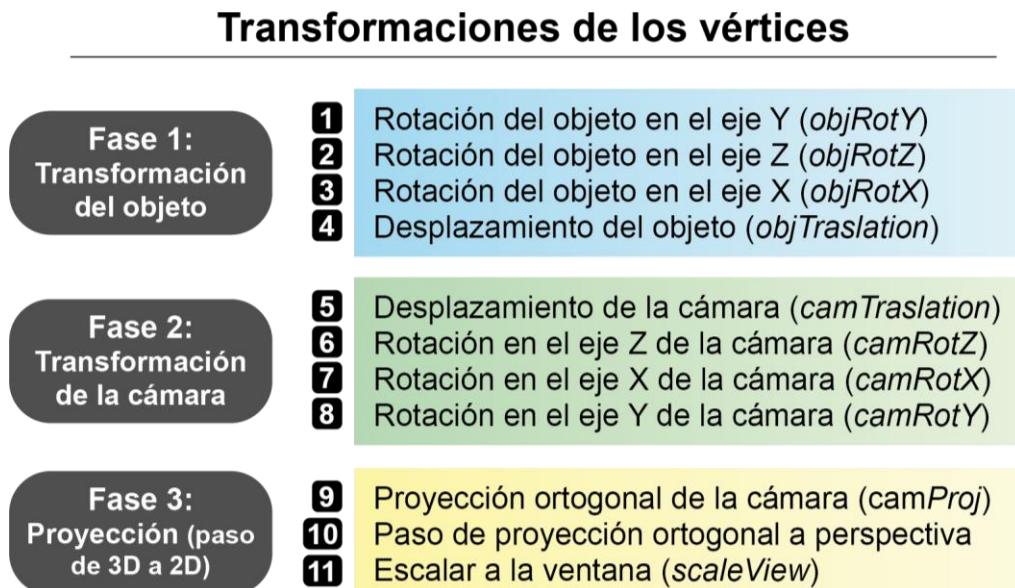


Diagrama 34. Transformaciones a aplicar a los vértices de cada objeto de la escena, ordenados según la ejecución, agrupados en 3 fases y con la nomenclatura recomendada para el código.

Todos estos cálculos se llevan a cabo mediante multiplicaciones concatenadas de matrices: cada una de las operaciones a realizar tiene una **matriz que se multiplica por los vértices del objeto**, agregando las modificaciones una a una en cada (a excepción del 10º paso, que son solo dos divisiones).

Sin embargo, antes de comenzar podemos tener en cuenta varios factores que permiten optimizar extremadamente este tedioso y exigente proceso de cálculo:

- **¿La cámara es dinámica?** En caso de que la cámara del *engine* sea estática, no hay necesidad de aplicar los cálculos referidos al jugador (los pasos del 5 al 8 se pueden ignorar).
- **¿Los objetos de la escena son dinámicos?** Si, por otra parte, hay algún objeto de la escena que no vaya a moverse (por ejemplo, un edificio), no es necesario hacer las operaciones de modificación de los vértices de dicho objeto, pudiendo pasar directamente al paso 5, saltando los cuatro primeros cálculos en función de objeto.
- **Matrices de transformaciones genéricas.** En vez de ir concatenando las operaciones una detrás de otra (es decir, multiplicar los vértices del objeto por cada matriz, una a una), se pueden multiplicar las matrices de cada paso (manteniendo el orden mostrado en el

Diagrama 34), generando una matriz genérica para cada una de las tres fases del proceso.

- **“Precalcular” las operaciones.** Dado que las operaciones van a tener que repetirse para cada vértice de cada triángulo de cada objeto de la escena, generar las 10 matrices cada vez que se deba realizar un cálculo para un vértice es inviable para el rendimiento del *engine*, de modo que es indispensable analizar el código y la aplicación que se está desarrollando para ver qué operaciones pueden realizarse al principio de “loopFunction” (por ejemplo, la matriz de proyección siempre será la misma, por lo que puede inicializarse y añadirse al *CameraModel* al principio).

Por otra parte, dado que en este apartado empezará la carga de operaciones, es recomendable programar los métodos que multiplican un vector de 4 valores por una matriz (también de dimensión 4). Además, en caso de querer implementar las matrices de transformaciones genéricas, también será necesario un método que multiplique dos matrices de dimensión 4. En el 3ngine, estas funciones se encuentran en *UtilsMath*, la única clase del package *Utils*.

5.3.1 Migración de 3D a 2D (proyección)

La primera operación del proceso que se llevará a cabo será la proyección de los vértices del objeto de la escena en la pantalla, de modo que se pueda al fin visualizar el cubo que se creó y añadió a la escena en el anterior punto.

Antes de comenzar a operar, en esta fase de proyección es importante no olvidar el atributo **w** de los vértices del 3ngine (recordar que las tres dimensiones son “x”, “y” y “z”; y también esta variable etiquetada como “w”, un tipo *float* que por defecto tendrá valor 1 y servirá para operar con las matrices).

1

Coordenadas ortogonales del vértice (matriz de proyección)

```
// VARIABLES NECESARIAS
float sX = 1.28f; // ancho del sensor de la cámara
float sY = 0.72f; // altura del sensor de la cámara
float fL = 1f; // distancia focal de la cámara
float vX = fL * width / 2 * sX; // factor de conversión en X
float vY = fL * height / 2 * sY; // factor de conversión en Y
```

Vértice a proyectar	Matriz de proyección
Coordenadas ortogonales del vértice	$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \times \begin{bmatrix} vX & 0 & 0 & 0 \\ 0 & vY & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Diagrama 35. Primer paso de la proyección de vértices de los objetos de la escena (paso 9 del proceso completo).

En el Diagrama 35 se puede ver la primera operación del proceso —que es en verdad el noveno paso del proceso completo de transformación—. Para ésta, debe crearse la **matriz de proyección** (es recomendable almacenarla en *CameraModel*, para no perder la información a cada vuelta del bucle). La matriz de proyección **adapta** los tres valores (*x*, *y*, *z*) de cada vértice a las dos dimensiones de la pantalla mediante dos factores de conversión: “*vX*” y “*vY*”.

Estas variables modifican el efecto visual de la proyección a raíz de la **distancia focal** deseada por el desarrollador y un **sensor** imaginario para escalar los vértices transformados a la proporción y el tamaño de la ventana.

Esta matriz también invierte la *Z* que recibe de las coordenadas originales del vértice (mediante el -1, tercer valor de la tercera columna). Además, la **distancia focal** —variable “*fL*”— influye en la posición de los vértices simulando un efecto de *zoom*. A mayor valor, mayor será el efecto.

Del mismo modo es importante que la **proporción del sensor** sea la **misma que la de la ventana**, de lo contrario la imagen se escalará de forma incorrecta. El tamaño de éste, sin embargo, puede variar según el gusto del desarrollador, aunque entonces la distancia focal deberá adaptarse para que la proyección de los vértices sea realista.

Lo más recomendable es que el **tamaño del sensor** (tanto en *X* como en *Y* sean valores **entre 0,5 y 1,5**), de este modo la distancia focal también variará en pocas décimas, siendo más sencilla de detectar la correcta.

2) Coordenadas del vértice con perspectiva

```
// Cálculo de la perspectiva (xP e yP)
float xP = x / z;
float yP = y / z;
// Crear otro Vertex con los nuevos valores
Vertex vProjection = new Vertex(xP, yP, of);
```

$$\begin{bmatrix} xP \\ yP \\ 0 \\ 1 \end{bmatrix}$$

3) Escalado final del vértice a las dimensiones de la ventana

```
// VARIABLES NECESARIAS
float width = 1280f; // ancho de la ventana (píxeles)
float height = 720f; // altura de la ventana (píxeles)
```

$$\begin{array}{l} \text{Vértice final, con escalado} \\ \left[\begin{array}{c} x \\ y \\ 0 \\ 1 \end{array} \right] = \left[\begin{array}{c} xP \\ yP \\ 0 \\ 1 \end{array} \right] \times \left[\begin{array}{cccc} 1 & 0 & 0 & \frac{\text{width}}{2} \\ 0 & -1 & 0 & \frac{\text{height}}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right] \end{array}$$

Diagrama 36. Pasos 2 y 3 de la fase de proyección de los vértices (10 y 11 del proceso completo).

Sin embargo, la proyección explicada previamente hace una conversión de los vértices a dos dimensiones de forma ortogonal, lo que significa que no se tiene en cuenta la profundidad de los elementos en la escena. Para que el renderizado de los objetos tenga en cuenta la perspectiva, debemos dividir las coordenadas **X** e **Y** obtenidas tras la proyección por el valor de **Z** (ya que la distancia al vértice será inversa a sus dimensiones). Este es el paso dos del Diagrama 36.

Tras esto ya se ha añadido perspectiva a la proyección: sólo queda el tercer paso, centrar el resultado en la cámara. Para ello, la matriz de escalado añade un offset equivalente a la mitad del tamaño de ancho y altura de la ventana a las coordenadas **X** e **Y** del vértice respectivamente, para centrar los elementos, y, de paso, invierte las coordenadas en **Y** (dado que, como se comentó previamente, también en Java las coordenadas del eje de ordenadas están invertidas).

Una vez terminados estos tres pasos, el *engine* ya cuenta con la información suficiente como para mostrar por la ventana del programa los objetos de la escena. Cabe destacar que ésta es la última operación del método “*loopFunction*”, justo antes de llamar a la función *repaint* para que dibuje los gráficos en la ventana; sin embargo, es recomendable comenzar el proceso por aquí porque es la primera función que permite visualizar los resultados de las transformaciones hasta ahora.

En *repaint*, para pintar el perímetro de los triángulos se utiliza la función *drawPolygon*, de **Graphics2D**. Ésta recibe un array con todas las coordenadas en *X* de los vértices del polígono cerrado, otro con las coordenadas en *Y*, y el número de vértices de la figura.

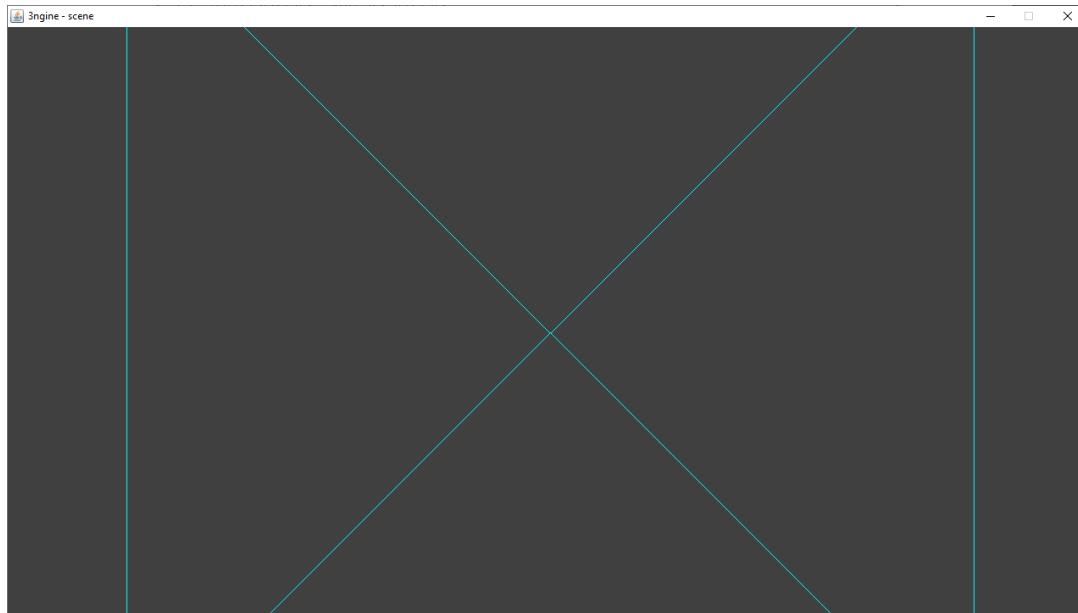


Figura 32. Resultado de la proyección del cubo. Posiciones iniciales del cubo y la cámara: (0,0,0) (captura propia).

En la Figura 32 puede apreciarse el resultado de proyectar el cubo de testeo en la cámara. Dado que aún no se han implementado más transformaciones, ambos están en la posición central del entorno virtual, por lo que la cámara se encuentra dentro del cubo. Por eso el renderizado muestra líneas que aún no parecen un cubo. Tras poder desplazar el cubo y tomar en cuenta la

posición de la cámara se podrá alejar uno de los dos (o ambos), y apreciar la figura de forma más coherente.

5.3.2 Transformaciones del objeto

Las transformaciones en los vértices del objeto representaban los cuatro primeros pasos del proceso completo (por lo que se implementan antes de la proyección y escalado de los vértices del anterior apartado, que eran los pasos 9 y 10).

Para comenzar con las transformaciones del objeto, se mostrarán las matrices de rotación empleadas para cada eje, así como el efecto de aplicar dicha matriz a los vértices, en el Diagrama 37.

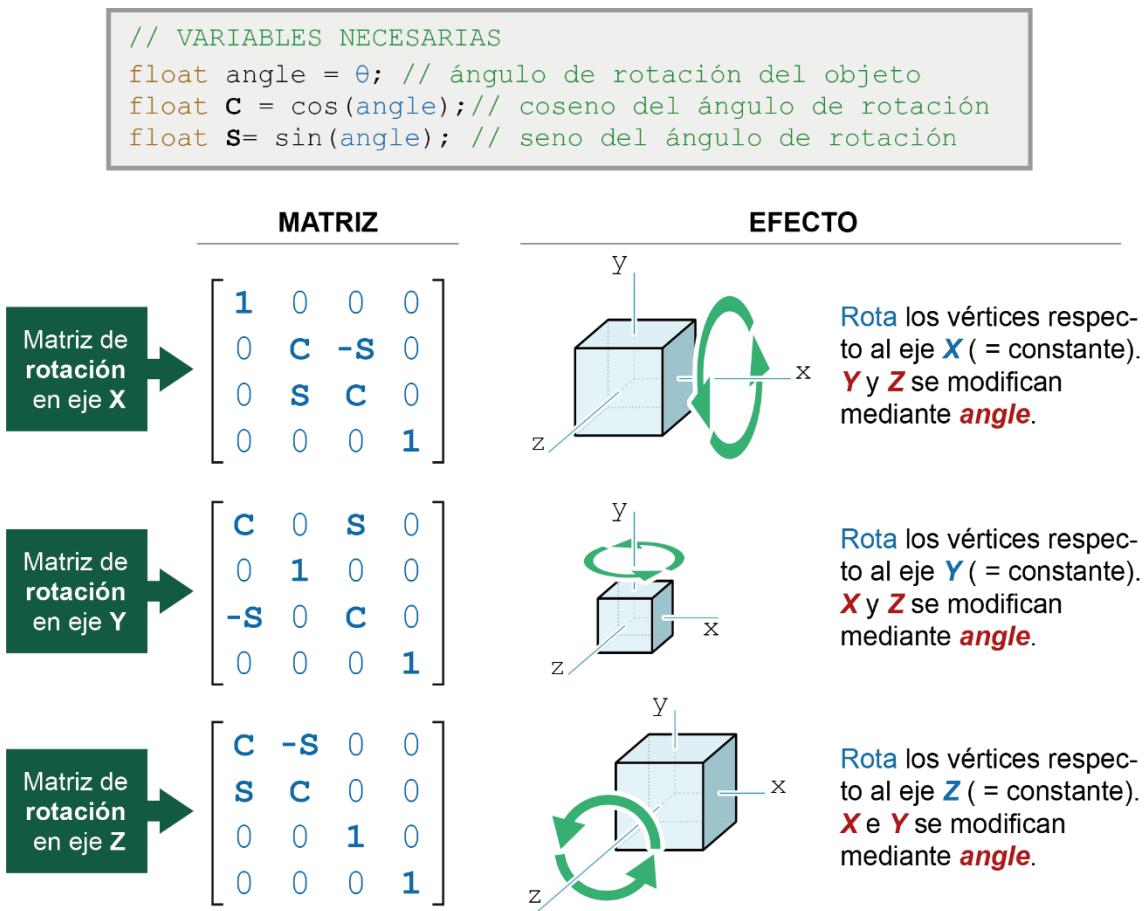


Diagrama 37. Matrices de rotación para los vértices del objeto, y su efecto en el cubo de testeо.

Para las operaciones trigonométricas necesarias para definir estas matrices será necesario el ángulo de rotación del objeto en cada eje (dado que empleamos la clase *Vertex* para almacenar la rotación a añadir al objeto —“addToRot”, siendo “rot” la rotación inicial—, en una misma variable se pueden guardar las rotaciones de cada eje por separado, aplicando más tarde el ángulo que corresponda según haya decidido el desarrollador), como se indica en el recuadro de **variables necesarias**.

Es importante destacar que los ángulos que reciben las funciones trigonométricas de Java están en radianes (igual que ocurría con el *raycaster* en *JavaScript*), de modo que se debe implementar

el método en *UtilsMath* para realizar la conversión (la misma que se mostró en el anterior *engine*).

Después de aplicar las rotaciones del objeto sólo queda el paso de la traslación de sus vértices (paso 4 de 10 del proceso, y último de la fase de transformaciones de los vértices). Para éste se utilizará la matriz de traslación, que tiene una importante distinción a tener en cuenta, y cuyos términos se muestran en el Diagrama 38.



Diagrama 38. Matriz de traslación para el desplazamiento de vértices.

A continuación, en el Diagrama 39, se muestra un ejemplo del cálculo de la traslación con el resultado, para aclarar el aviso del signo invertido de la matriz.

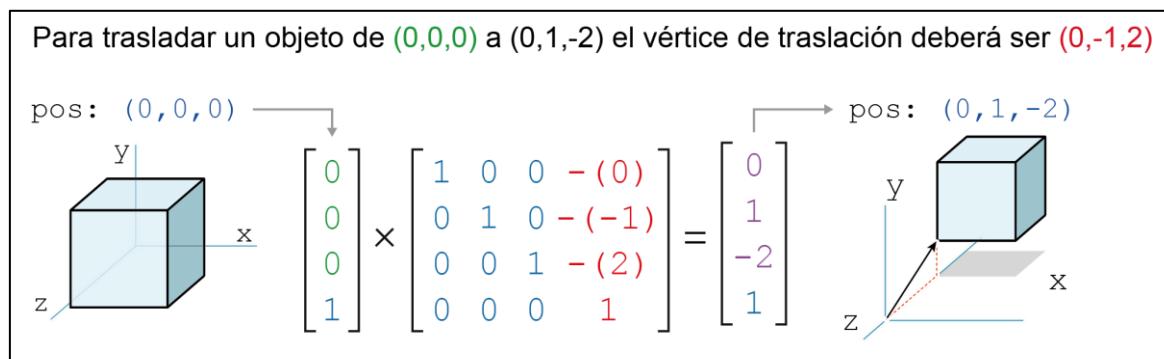


Diagrama 39. Ejemplo de traslación de los vértices de un cubo.

Una vez definida también la matriz de traslación para el desplazamiento de los vértices, podemos pasar al cálculo completo de la fase de transformación del objeto, que equivale al indicado en el Diagrama 40. En él se enseña la operación de cálculo de la matriz genérica, que es la que se usará para multiplicar los vértices del objeto, realizando todas las rotaciones y la traslación mediante una sola matriz.

```
// VARIABLES NECESARIAS
float α = 0; // ángulo de rotación en el eje X (radianes)
float θ = 0; // ángulo de rotación en el eje Y (radianes)
float φ = 0; // ángulo de rotación en el eje Z (radianes)
float cX = cos(α), sX = sen(α); // seno y coseno de α
float cY = cos(θ), sY = sen(θ); // seno y coseno de θ
float cZ = cos(φ), sZ = sen(φ); // seno y coseno de φ
float dX = 0; // traslación a aplicar en X (pixeles)
float dY = 0; // traslación a aplicar en Y (pixeles)
float dZ = 0; // traslación a aplicar en Z (pixeles)
```

$$\begin{array}{c} \text{Matriz genérica} \\ \downarrow \\ \left[\begin{array}{cccc} x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \\ w_1 & w_2 & w_3 & w_4 \end{array} \right] \end{array} = \begin{array}{c} \text{Matriz de rotación en Z} \\ \downarrow \\ \left[\begin{array}{cccc} cX & -sX & 0 & 0 \\ sX & cX & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right] \end{array} \times \begin{array}{c} \text{Matriz de rotación en Y} \\ \downarrow \\ \left[\begin{array}{cccc} cX & 0 & sX & 0 \\ 0 & 1 & 0 & 0 \\ -sX & 0 & cX & 0 \\ 0 & 0 & 0 & 1 \end{array} \right] \end{array} \times \begin{array}{c} \text{Matriz de rotación en X} \\ \downarrow \\ \left[\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & cX & -sX & 0 \\ 0 & sX & cX & 0 \\ 0 & 0 & 0 & 1 \end{array} \right] \end{array} \times \begin{array}{c} \text{Matriz de traslación} \\ \downarrow \\ \left[\begin{array}{ccccc} 1 & 0 & 0 & -dX \\ 0 & 1 & 0 & -dY \\ 0 & 0 & 1 & -dZ \\ 0 & 0 & 0 & 1 \end{array} \right] \end{array}$$

Diagrama 40. Proceso de cálculo de la matriz genérica para la fase de transformación de los vértices.

Una vez calculada esta matriz genérica, simplemente queda multiplicar los vértices iniciales del cubo por esta matriz, **antes del cálculo de la proyección**, como se indica en el Diagrama 41.

$$\begin{array}{c} \text{Coordenadas iniciales del vértice} \\ \downarrow \\ \left[\begin{array}{c} X \\ Y \\ Z \\ 1 \end{array} \right] \end{array} = \begin{array}{c} \text{Coordenadas del vértice transformado} \\ \downarrow \\ \left[\begin{array}{c} x \\ y \\ z \\ 1 \end{array} \right] \end{array} \times \begin{array}{c} \text{Matriz genérica} \\ \downarrow \\ \left[\begin{array}{cccc} x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \\ w_1 & w_2 & w_3 & w_4 \end{array} \right] \end{array}$$

Diagrama 41. Multiplicación de los vértices iniciales del objeto por la matriz genérica de transformación.

Tras esta serie de cálculos el motor gráfico es capaz de desplazar los objetos que haya en la escena. Por ejemplo, se puede mover el cubo inicial 2 o más unidades hacia atrás para verlo en su totalidad, como se aprecia en la Figura 33 (también se ha añadido rotación en todos los ejes).

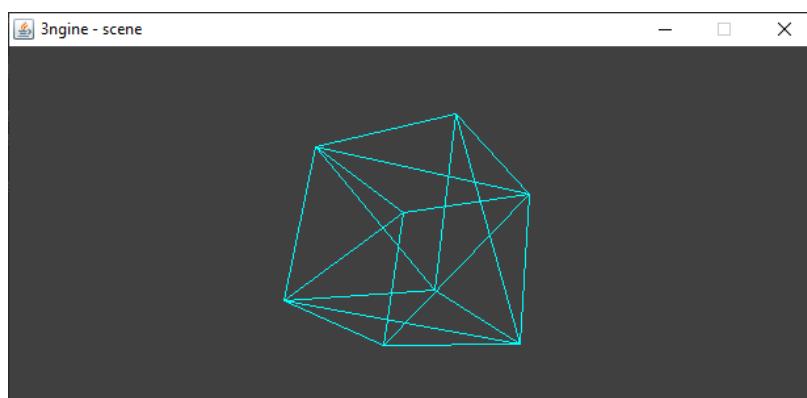


Figura 33. Resultado del motor gráfico tras añadir rotación y desplazamiento inicial al cubo (captura propia).

5.3.3 Transformaciones de la cámara

La última serie de cálculos que quedan por implementar para la fase de proyección y transformación de *meshes* del *engine* es la que permite al jugador desplazarse por la escena de forma realista en cualquiera de las 3 dimensiones, así como girar la cámara.

En el 3ngine, se opta por **simular** un falso efecto de movimiento de la **cámara** ya que, en verdad, lo que ocurre en el programa es que la **escena estará rotando y moviéndose de forma opuesta al jugador**, como se describe en la Figura 34, donde se puede ver que la cámara mantiene su posición.

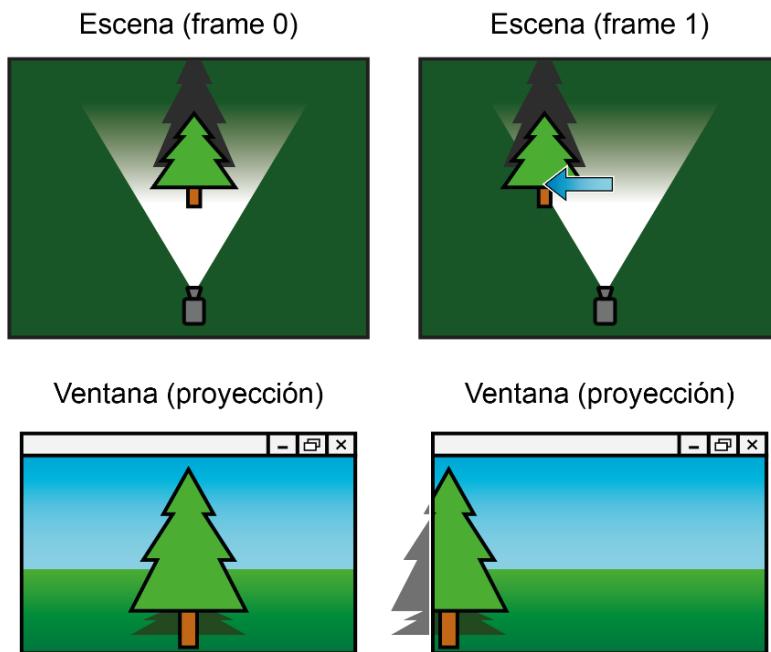


Figura 34. Representación visual del efecto falso de movimiento de cámara (ilustración propia).

En la imagen se puede apreciar cómo se puede simular el movimiento (y el giro) de la cámara invirtiendo las operaciones que ya se han definido previamente. Mover la cámara a la derecha ofrecerá el mismo resultado a nivel visual que desplazar toda la escena hacia la izquierda, como se ve en el *frame 1* de la escena.

Por eso, dado que se volverán a transformar los vértices, se pueden reutilizar las matrices anteriores. Únicamente es necesario definir una posición y un ángulo de rotación para la cámara (3ngine utiliza *CameraModel*, almacenando en la clase las matrices empleadas para el movimiento, la rotación y proyección de la cámara). Además, en caso de que nuestra cámara sea dinámica, como se mencionó previamente, será necesario actualizar estas matrices a cada *frame*.

Para esta segunda fase de las transformaciones de los vértices se utilizan exactamente las mismas matrices que en la fase 1, la única diferencia es que ahora los ángulos de rotación no son del objeto, sino de la cámara; y, en la matriz de traslación, los valores *dX*, *dY*, y *dZ* son la posición de la cámara (que, esta vez, no es necesario cambiar de signos, ya que el efecto óptico es inverso al movimiento de la cámara). En el Diagrama 42 se ofrecen las matrices a definir para

la cámara del *engine*, así como el efecto visual que producirá en el renderizado de los vértices y las variables que se necesitan para los elementos de la matriz.

```
// VARIABLES NECESARIAS
float dX = 0; // posición en X de la cámara (entorno 3D)
float dY = 0; // posición en Y de la cámara (entorno 3D)
float dZ = 0; // posición en Z de la cámara (entorno 3D)
float α = 0; // ángulo de la cámara en X (radianes)
float θ = 0; // ángulo de la cámara en Y (radianes)
float φ = 0; // ángulo de la cámara en Z (radianes)
float cX = cos(α), sX = sen(α); // seno y coseno de α
float cY = cos(θ), sY = sen(θ); // seno y coseno de θ
float cZ = cos(φ), sZ = sen(φ); // seno y coseno de φ
```

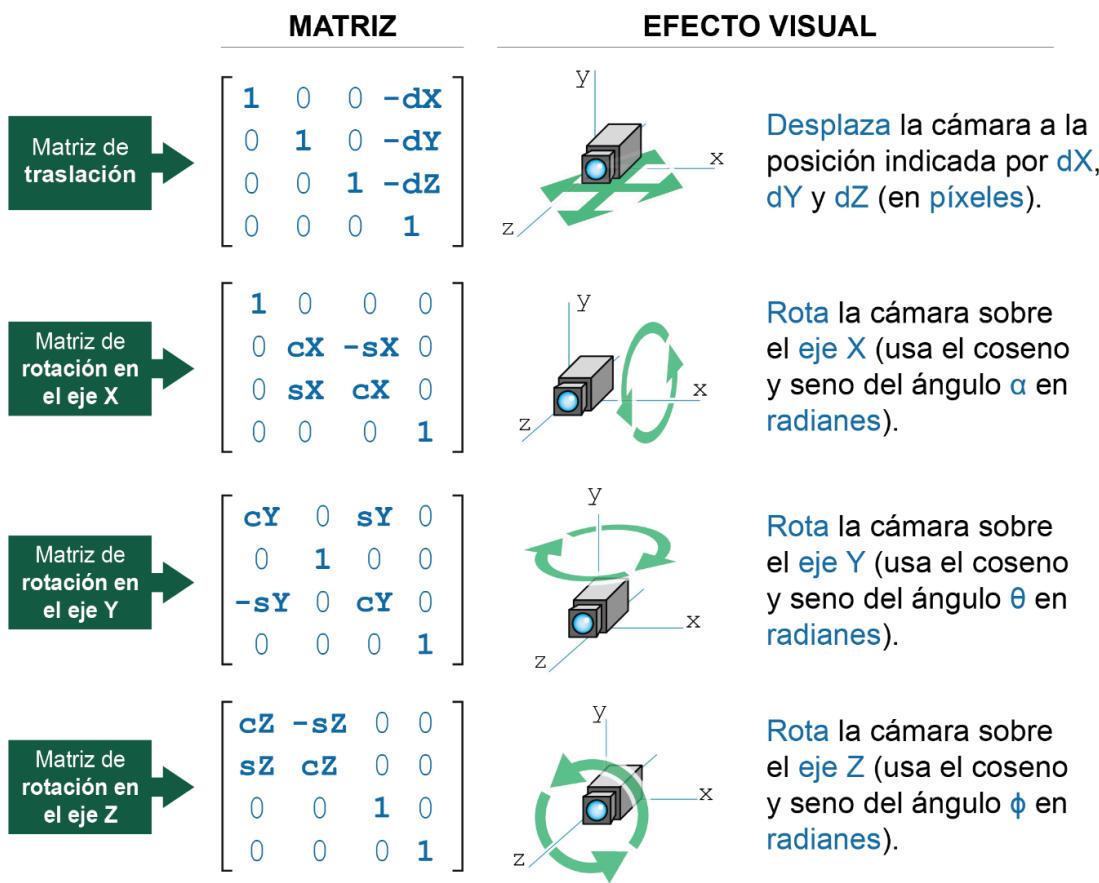


Diagrama 42. Matrices de transformación de los vértices respecto a la cámara, con cada efecto, y las variables necesarias para su implementación.

De nuevo, sólo queda multiplicar las matrices para obtener la matriz genérica de transformaciones de la cámara, y luego multiplicamos los vértices transformados en la primera fase (es decir, los obtenidos tras modificar la posición y las rotaciones del objeto) por esta matriz. De este modo se lleva a cabo el proceso explicado anteriormente: estaremos aplicando una nueva transformación a todos los vértices de la escena para simular un efecto de dinamismo de la cámara.

En el Diagrama 43 se da un vistazo a esta concatenación de operaciones. Esta vez el orden de la serie de cálculos ha cambiado, situando como la primera del cálculo la traslación de cámara (al mover el vértice antes de rotarlo, dado que el punto de referencia sobre el cual rotan los objetos

es el (0,0,0), no se desplaza de forma relativa a la cámara, sino a la escena, utilizando la posición de la cámara como *offset*).

$$\begin{array}{c}
 \text{Matriz} \\
 \text{genérica} \\
 \hline
 \end{array}
 =
 \begin{array}{c}
 \text{Traslación} \\
 \text{de cámara} \\
 \hline
 \end{array}
 \times
 \begin{array}{c}
 \text{Rotación de la} \\
 \text{cámara en Z} \\
 \hline
 \end{array}
 \times
 \begin{array}{c}
 \text{Rotación de la} \\
 \text{cámara en Y} \\
 \hline
 \end{array}
 \times
 \begin{array}{c}
 \text{Rotación de la} \\
 \text{cámara en X} \\
 \hline
 \end{array}$$

$$\begin{bmatrix}
 x_1 & x_2 & x_3 & x_4 \\
 y_1 & y_2 & y_3 & y_4 \\
 z_1 & z_2 & z_3 & z_4 \\
 w_1 & w_2 & w_3 & w_4
 \end{bmatrix}
 =
 \begin{bmatrix}
 1 & 0 & 0 & -dx \\
 0 & 1 & 0 & -dy \\
 0 & 0 & 1 & -dz \\
 0 & 0 & 0 & 1
 \end{bmatrix}
 \times
 \begin{bmatrix}
 cX & -sX & 0 & 0 \\
 sX & cX & 0 & 0 \\
 0 & 0 & 1 & 0 \\
 0 & 0 & 0 & 1
 \end{bmatrix}
 \times
 \begin{bmatrix}
 cX & 0 & sX & 0 \\
 0 & 1 & 0 & 0 \\
 -sX & 0 & cX & 0 \\
 0 & 0 & 0 & 1
 \end{bmatrix}
 \times
 \begin{bmatrix}
 1 & 0 & 0 & 0 \\
 0 & cX & -sX & 0 \\
 0 & sX & cX & 0 \\
 0 & 0 & 0 & 1
 \end{bmatrix}$$

Diagrama 43. Cálculo de la matriz genérica para la transformación de la cámara.

Si se rotaran primero los vértices, y luego se multiplicasen por la matriz de traslación, estaríamos rotando nuestra cámara sobre el punto (0,0,0), en vez rotar la escena sobre nuestra cámara.

Tras obtener la matriz genérica, simplemente se multiplican los vértices transformados en la anterior fase del proceso por ella, y el resultado de la operación (que será otro vértice de dimensión 4) es el que se proyectará (fase 3 del proceso) en la ventana.

Con un correcto orden de las operaciones, puede obtenerse el resultado de la Figura 35 mediante los mismos parámetros de modificación (el cubo se encuentra en las coordenadas (0,0,0), y no se le ha aplicado rotación, esta vez).

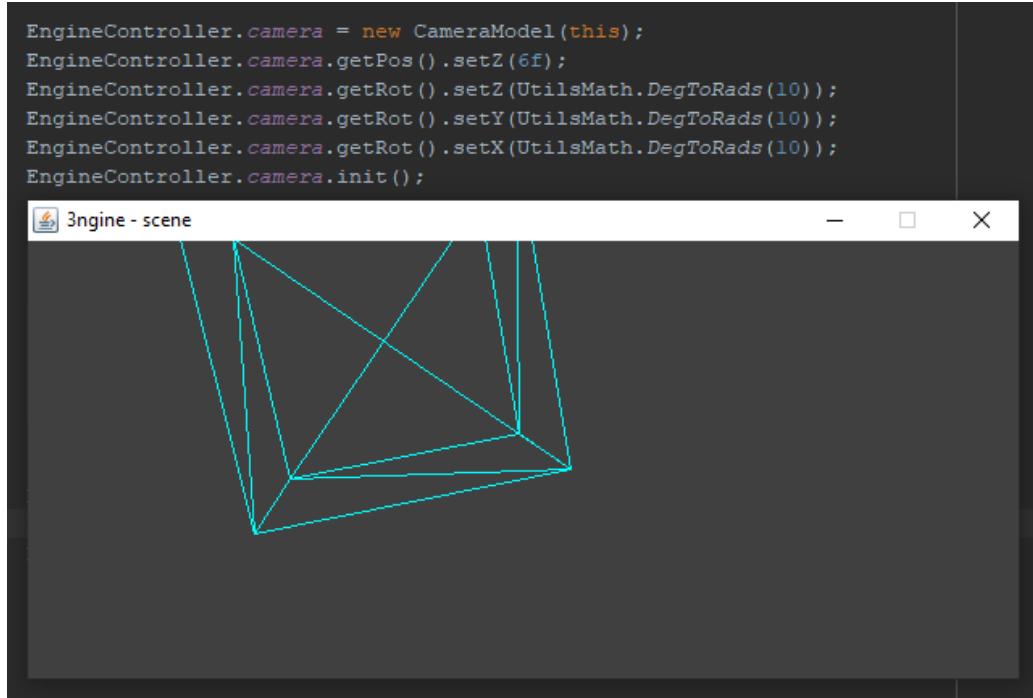


Figura 35. Resultado de aplicar transformaciones de cámara al cubo de testeo (que está inmóvil en el centro de la escena), con los valores modificados incluidos (captura propia).

El aspecto final del *render* puede variar en función del tamaño del sensor escogido para la cámara del *engine*, la distancia focal de la proyección y/o las coordenadas de los vértices del cubo del desarrollador.

5.4 Input del usuario

Una vez el *engine* ya es capaz de transformar y proyectar los vértices, el siguiente paso es gestionar la entrada de comandos por parte del usuario, de forma que éste pueda mover la cámara y avanzar a su antojo.

En esta fase entra en juego la clase *UserInputController*, del package *Controller*. Ésta contiene una serie de atributos de tipo booleano que indican *true* si el usuario está pulsando la tecla que les corresponde, y *false* en caso contrario.

De modo que, para programar esta funcionalidad se llevan a cabo 3 pasos (en este orden):

- Detectar el input del usuario mediante la interfaz *KeyListener* (hay que importarla desde el paquete *Java.awt.event*). En el 3ngine, los métodos de la interfaz se sobrescriben en la clase *EngineController*.
- Cambiar los booleanos de *UserInputController* según la entrada del teclado del usuario.
- Modificar las variables que correspondan en función de dichos booleanos (como la posición u orientación de la cámara), al principio del hilo de ejecución principal del bucle (al principio de “loopFunction”, antes de empezar el proceso de transformación de los vértices del anterior apartado).

En la Tabla 6 se muestra un esquema de los controles del 3ngine para un teclado QWERTY convencional. En gris (los últimos cuatro) aún no es necesario implementarlos, dado que son funcionalidades que aún están por desarrollarse.

Tabla 6. Controles de teclado del 3ngine.

Tecla	Función
W	Avanzar en la dirección de la cámara
S	Retroceder en la dirección de la cámara
A	<i>Strafe</i> (desplazamiento) a la izquierda de la cámara
D	<i>Strafe</i> (desplazamiento) a la derecha de la cámara
Barra espaciadora	Elevarse (volar hacia arriba)
Control (izquierda)	Volar hacia abajo
Flechas direccionales	Girar la cámara hacia arriba, abajo, izquierda o derecha.
O	Abrir otro objeto (cargar nueva escena)
1	Modo de renderizado completo
2	Modo de renderizado wireframe
3	Modo de renderizado de solo superficies

Sin embargo, antes de comenzar a implementar estas adiciones al código, necesitaremos tres variables nuevas en la clase *CameraModel* para poder mover la cámara por la escena según el *input* del usuario.

5.4.1 Modificaciones en *CameraModel*

A estas alturas, dado el anterior apartado, *CameraModel* contiene tres ángulos de rotación (uno por cada eje de coordenadas). El ángulo de rotación es suficiente para modificar la dirección en que mira la cámara, ya que se puede modificar dicho ángulo directamente; sin embargo, para

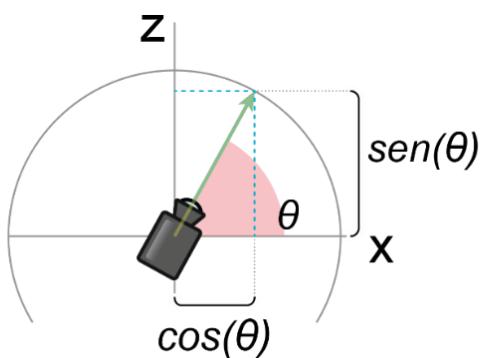
poder mover la cámara sí necesitamos conocer la dirección en que está mirando la cámara. Y, dado que el jugador también podrá volar hacia arriba y abajo y realizar *strafing* a los lados, esto añade otros dos vectores más a la lista de vectores necesarios, siendo ésta:

- **vDir**. Vector que representa la dirección en que está mirando la cámara (aunque sólo en X y Z, dado que, independientemente de la magnitud del vector en Y, al avanzar, el usuario solo se desplazará horizontalmente por la escena).
- **vUp**. Vector que siempre apunta hacia arriba (este vector puede definirse una sola vez, ya que no cambiará nunca durante la ejecución). Se utiliza para calcular el “vDir”, y se tiene en cuenta para mover la cámara hacia arriba o abajo.
- **vRight**. Vector que señala a la derecha de “vDir”. Se utiliza para efectuar el *strafing* de la cámara a izquierda y derecha.

El primero de los vectores para crear puede ser el “**vUp**”, que simplemente indica la dirección en que avanza el eje de coordenadas Y. Como no se va a modificar nunca, se puede definir al principio (en el constructor de *CameraModel*), y su valor es: **(0, 1, 0)**. Es decir, señala siempre hacia arriba.

Después, el vector dirección de la cámara. Éste funciona exactamente igual que el ángulo de jugador del *raycaster*, en el anterior motor desarrollado: calculando la magnitud del vector en X y Z mediante el coseno y el seno del ángulo de rotación de la cámara en Y (más un *offset*), como en el Diagrama 44.

El *offset* mencionado consiste en restarle medio pi (radianes) al ángulo, y esto se debe a que, pese a que las coordenadas del suelo se expanden a lo largo del eje X y Z, la proyección de la cámara se está llevando a cabo desde el eje Z (no olvidar que, en la fase de proyección, el atributo por el cual se dividen las coordenadas X e Y es Z). De modo que hay que aplicar al ángulo la diferencia en grados entre el eje Z y el X, que es de 90 grados sexagesimales (-90 grados, ya que el signo de la rotación de la cámara debe invertirse).



$$\mathbf{vDir} = (\cos(\theta - \text{PI}/2), 0, \sin(\theta - \text{PI}/2));$$

Diagrama 44. Proceso de cálculo del vector de dirección de la cámara (**vDir**), en función del ángulo de rotación de la cámara, así como el *offset* a agregar.

Debido a este cálculo y, dado que no es necesaria la magnitud del atributo Y en el vector direccional de la cámara, ésta puede dejarse siempre a cero, siendo necesario únicamente realizar dos modificaciones a “vDir” en cada *frame*.

Por otra parte, Y se mantiene como 0 (como hemos comentado, no es necesario también el valor de Y porque, al desplazarnos hacia la dirección en que mire el jugador, se avanzará manteniendo la misma coordenada en Y, como se exemplifica en el Diagrama 45, independientemente del *tilt* de la cámara del jugador). Es decir, como en un FPS.

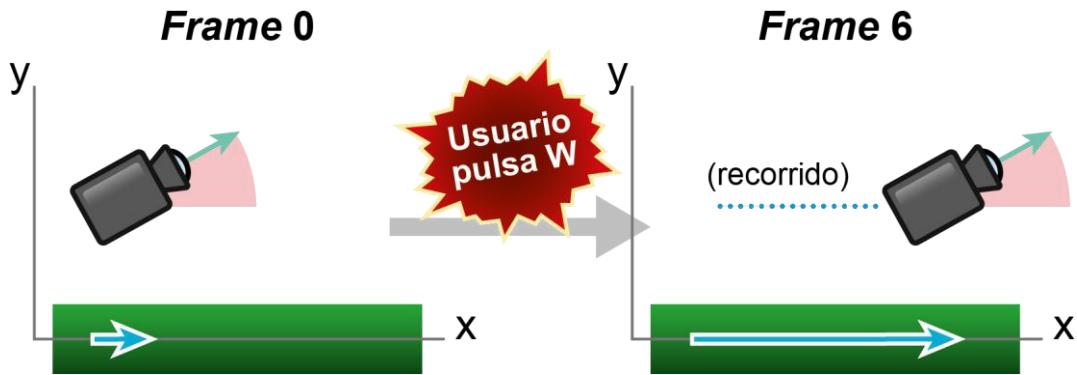
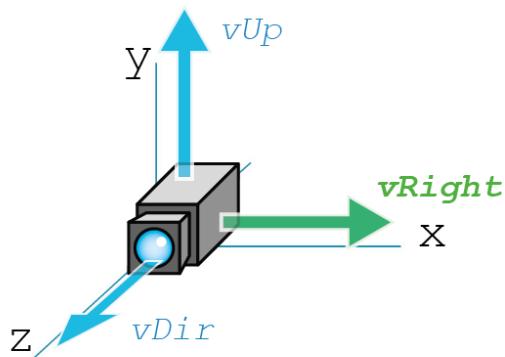


Diagrama 45. Desplazamiento de la cámara a lo largo del eje X (sin cambios en el eje Y), pese al ángulo de la cámara.

La coordenada en Y de la (posición de la) cámara únicamente varía cuando el usuario pulsa la barra espaciadora o la tecla “control” izquierda, no cuando se desliza a través de X y Z mediante “W”, “A”, “S” o “D”.

Por último, se calcula el vector que señala a la derecha de “vDir”. Para obtener este vector hay que realizar el **producto vectorial entre vDir y vUp**, ya que el resultado es el vector ortogonal a ambos (esquema ilustrativo en el Diagrama 46). También debe calcularse a cada vuelta del bucle, pero después de calcular “vDir”.



$$\text{vRight} = \text{crossProduct}(\text{vDir}, \text{vUp});$$

Diagrama 46. Cálculo del vector “vRight”, para *CameraModel*.

En el 3ngine, el método para calcular el producto vectorial entre 2 vectores (aunque la clase sea *Vertex*, en concepto siguen siendo 3 magnitudes, por lo que es completamente válida) se encuentra también en *UtilsMath*. A continuación, se muestra cómo calcular el producto vectorial entre dos vectores:

```
// Para obtener el vRight, v1 debe ser vDir, y v2, vUp
public static Vertex crossProduct(Vertex v1, Vertex v2) {
    Vertex vOut = new Vertex(0f, 0f, 0f);
    vOut.setX(v1.getY() * v2.getZ() - v1.getZ() * v2.getY());
    vOut.setY(v1.getZ() * v2.getX() - v1.getX() * v2.getZ());
    vOut.setZ(v1.getX() * v2.getY() - v1.getY() * v2.getX());
    return vOut;
}
```

Por último, también es importante normalizar los vectores *vDir* y *vRight* justo después de calcular cada uno (este es un método de la clase *Vertex*). Para ello solo hay que calcular la magnitud del vector (o vértice, mejor dicho), y luego dividir las *x*, *y* y *z* del *Vertex* por dicho valor.

5.4.2 Atributos del *UserInputController*

La clase *UserInputController* es la que contiene todos los booleanos (uno por cada tecla) que indican qué tecla está siendo presionada por parte del usuario al inicio de cada *frame*. De este modo, cuando el usuario pulsa una tecla, el *EngineController* actualiza el booleano que corresponda.

5.4.3 *EngineController implements KeyListener*

Como se lee en el título, la clase *EngineController* de nuestro sistema de renderizado debe implementar *keyListener*, y sobrescribir los métodos *keyPressed*, *keyReleased* y *keyTyped* (aunque éste último está en blanco: no se utiliza). Como ocurre con el método *paint* del *EngineView*, es importante poner el “*@Override*” antes de declarar cada uno de los métodos en el *EngineController*.

Tanto en el *keyPressed* como en el *keyReleased* del 3ngine se hace un *switch* para detectar qué tecla ha enviado una señal por el parámetro *KeyEvent*. Entonces, el atributo de *UserInputController* que le corresponda se actualiza (*true* si el usuario ha pulsado —es decir, en *keyPressed*—, y *false* si ha soltado la tecla —*keyTyped*—).

5.4.4 Aplicar el *input*

Ya están definidas todas las variables que indican el *input* del usuario (en *UserInputController*), así como los métodos que actualizan estas variables (*keyPressed* y *keyReleased*). Sólo queda implementar el efecto que estos *inputs* tienen en nuestro código.

Para ello, en el método “*loopFunction*” —el mismo en que se lleva a cabo la transformación de los vértices de la escena, las transformaciones de la cámara y su proyección—, antes de nada, en el 3ngine se ha llamado a un nuevo método también definido en *EngineController* que mueve y gira la cámara.

Esta función es “*applyPlayerInput*”.

Para mover la cámara, en *applyPlayerInput* se llama a la función “*move*” de *CameraModel* que, según la tecla pulsada crea un vértice igual a *vDir* o *vRight* (en caso de retroceder —ha pulsado “S”— en vez de avanzar, o ir a la izquierda de la cámara —ha pulsado “A”— en vez de la derecha, multiplica el vértice por -1 para invertir los términos del vector. Después, multiplica el vector por el *elapsedTime* (que equivale al tiempo que dura cada *frame*, en función de los fps que haya

elegido el desarrollador), para mantener la misma velocidad de movimiento en caso de variar los fps del programa.

Por último, suma el vector resultante de la multiplicación (y el cambio de signo, de ser necesario) a la anterior posición de la cámara (es decir, hace un *translate* de la posición de la cámara).

Además de ésta, *applyPlayerInput* llama al método “**turn**”, también **de CameraModel**, que incrementa el ángulo de rotación de la cámara en el eje que corresponda a la tecla pulsada (si inclina o alza la cámara arriba o abajo, se cambia la rotación de X —preferiblemente hasta un límite— ya que es el eje sobre el que rota la cámara; si gira a izquierda o derecha, se cambia la rotación de Y, por el mismo motivo —aunque sin límite—). Como ocurre con la traslación de la cámara, se suma a la rotación actual la velocidad de rotación de la cámara, con el signo invertido según la orientación del giro, y multiplicado por el *elapsedTime* del *frame*.

Por último, antes de terminar la ejecución del método *applyPlayerInput*, si alguno de los tres ángulos de la cámara es mayor o menor que el doble de pi o el doble de pi negativo respectivamente, la rotación se reinicia a cero de nuevo.

5.5 Generación de triángulos

En el anterior proceso de transformación de los vértices, justo tras proyectarlos se llama a la función de *repaint* de *EngineView*, que pinta los tres vértices de cada triángulo del objeto de la escena.

Sin embargo, en el 3ngine hay un paso previo. Una vez calculados los vértices proyectados, manda los triángulos a un array llamado *trisToProject* que se encuentra en la variable escena del engine (*SceneObject*). Al llamar a la función *repaint*, *EngineView* accede a este array y pinta los triángulos que encuentra en él. Hasta este punto, se había utilizado la función *drawPolygon* para dicha tarea, método que pinta el perímetro de (en este caso) todos los triángulos del cubo —es decir, el cubo se renderizaba en estilo *wireframe*—. Sin embargo, para pintar el interior del triángulo se usa el método *fillPolygon*, que recibe exactamente los mismos parámetros (array de coordenadas en X de los vértices del polígono, otro array con las coordenadas en Y de los mismos, y el número de vértices que conforman la figura —en este caso 3—). Podemos llamar a ambos métodos para conseguir el estilo de renderizado “completo”, donde se aprecian los vértices que forman los triángulos y, a la vez, su área.

Sin embargo, al ejecutar el código puede verse que falta un proceso esencial en la fórmula. En la Figura 36, el antes de usar *drawPolygon* se estableció como color el azul, y antes del *fillPolygon*, el negro. Gracias al color de los triángulos puede apreciarse que no se renderizan de forma ordenada.

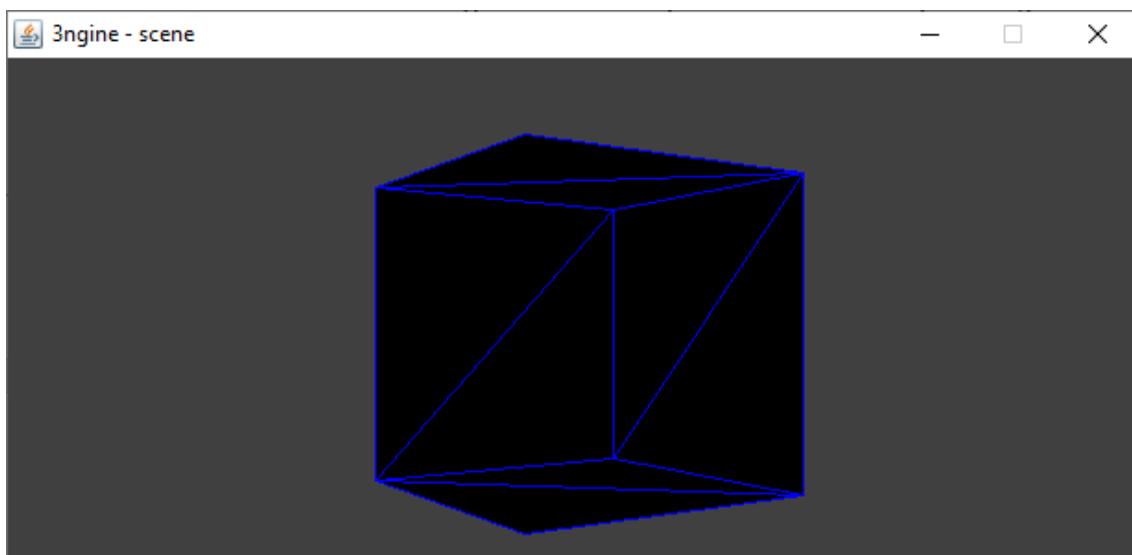


Figura 36. Captura de pantalla del renderizado de triángulos del cubo (preordenación).

Este proceso de ordenación que hasta ahora se ha obviado dado que no influía en el renderizado de tipo *wireframe* (solo las líneas azules), es crucial para cualquier renderizado de superficies que se deba implementar en el *engine*, independientemente de su última finalidad, ya que es una indicación precisa de los modelos que se envían al *software* de renderizado.

5.5.1 Variable de ordenación

Antes de comenzar a ordenar los triángulos, es necesario tener claro sobre qué se van a ordenar. Los triángulos de la escena deberían renderizarse del modo descrito a continuación: primero pintar los triángulos más alejados de la cámara, e ir renderizando los demás triángulos, unos sobre otros, de modo que los más cercanos a la cámara son los que acaban pintándose por encima de los demás. Por ello, la variable de la cual depende la ordenación de los triángulos es la distancia de su centro respecto a la cámara.

Para realizar esta operación, debemos llevar a cabo dicho cálculo para todos los triángulos que haga falta proyectar en la escena. El 3ngine almacena este resultado en el atributo “**depthValue**” de los triángulos (será un número decimal, no es necesario obtener el resultado en forma de vector).

5.5.2 Cálculo de “**depthValue**”

Para calcular la distancia entre los triángulos y la cámara, primero hay que encontrar el punto central del triángulo. Para ello, hacemos la **media aritmética de cada coordenada de los vértices** del triángulo, como se muestra en la Ecuación 2.

$$\bar{x} = \frac{x_1 + x_2 + x_3}{3}$$

Ecuación 2. Media aritmética de las coordenadas X de los tres vectores de un triángulo.

En la fórmula, x_1 , x_2 y x_3 son las coordenadas en el eje “X” de los 3 vértices del triángulo, y \bar{x} es la media de las tres coordenadas. Este proceso se repite para las otras tres coordenadas para **obtener el centro del triángulo**. Una vez las tenemos, creamos un nuevo *Vertex*, cuyas coordenadas sea este centro del triángulo.

Por último, sólo debemos **restar la posición de la cámara al nuevo *Vertex*** para obtener el vector que viaja desde el centro del triángulo hasta la cámara; y calcular su magnitud.

En el 3ngine, el método para restar dos vértices (o vectores) se encuentra en el *package UtilsMath*, donde también hay funciones de suma, multiplicación y división de vectores; y se llama ***subVertex***. Simplemente resta las coordenadas del primer vector con las del segundo y devuelve un nuevo *Vertex* con el resultado (si el usuario quiere enviar como parámetro un *Vertex* ya creado para el resultado, puede utilizar la función *subExistentVertex*).

Por último, el método “*getLength*” (en la clase *Vertex*) **calcula la magnitud del vector** mediante el teorema de Pitágoras.

Es extremadamente importante remarcar que, para este cálculo, **los valores** de los vértices **que** estamos usando para el cálculo de la profundidad son los de ***vProcess*** (es decir, debemos hacer este cálculo justo después de la transformación de los vértices de la escena —primera fase del proceso completo de transformación de los vértices—, y antes de aplicar las transformaciones de la cámara —segunda fase—).

5.5.3 Método de ordenación

Una vez hecho el cálculo anterior, al final de “*loopFunction*”, antes de renderizar los gráficos, se hace una llamada al método de ordenación para los triángulos a proyectar en la escena.

Este método, implementado en la clase *SceneObject*, lee el atributo que contiene la lista de triángulos de la escena (de los cuales ahora también contamos con su variable *depthValue*), y ejecuta un método de **ordenamiento del array por selección**.

Por cada ítem del *array* (es decir, por cada triángulo a proyectar), repite esta serie de operaciones en este orden:

- Considera como máximo el primer valor leído (el triángulo actual).
- Accede al siguiente triángulo de la lista y lee su *depthValue*. Si éste es mayor que el previo (en caso de ser el primero, si es mayor que el triángulo actual), lo considera como nuevo máximo y almacena su índice en la lista.
- Repite el paso anterior para todos los triángulos que quedan por ordenar en la lista de proyección, de modo que al terminar se ha obtenido el máximo *depthValue* de los triángulos que quedaban por comprobar y se conoce su posición.
- Se intercambia la posición de dicho máximo con la del triángulo actual.
- Consideramos como triángulo actual el siguiente de la lista y repetimos los anteriores pasos.

Este proceso es suficientemente óptimo, ya que evitamos una relectura excesiva de la lista de triángulos, gracias al almacenamiento del máximo y su índice en la lista. De este modo, a medida que el método sigue y quedan menos triángulos, el procedimiento se va volviendo más rápido, dado que cada vez quedan menos elementos para ordenar.

En el Diagrama 47 se muestra el proceso visualmente:

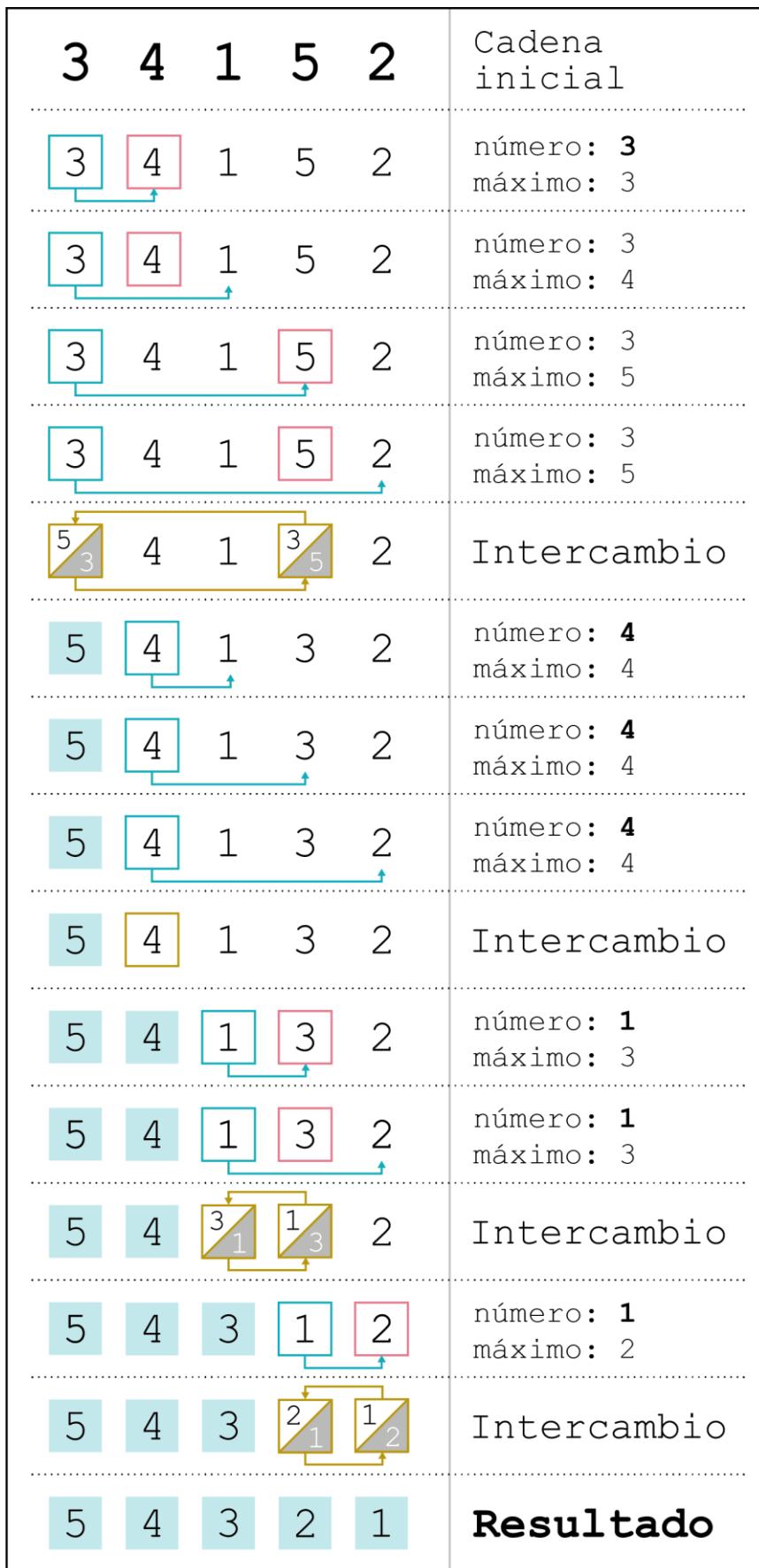


Diagrama 47. Método de ordenación por selección, para los triángulos a proyectar en la escena.

Tras implementar este método de ordenación en el 3ngine, el resultado es el de la Figura 37, donde puede apreciarse que los triángulos están ahora ordenados de menor a mayor distancia a la cámara.

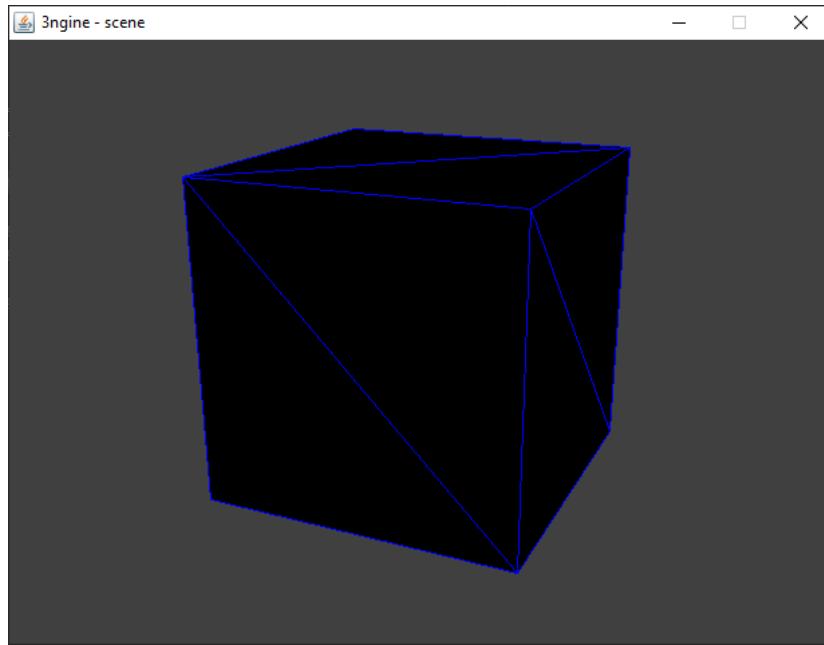


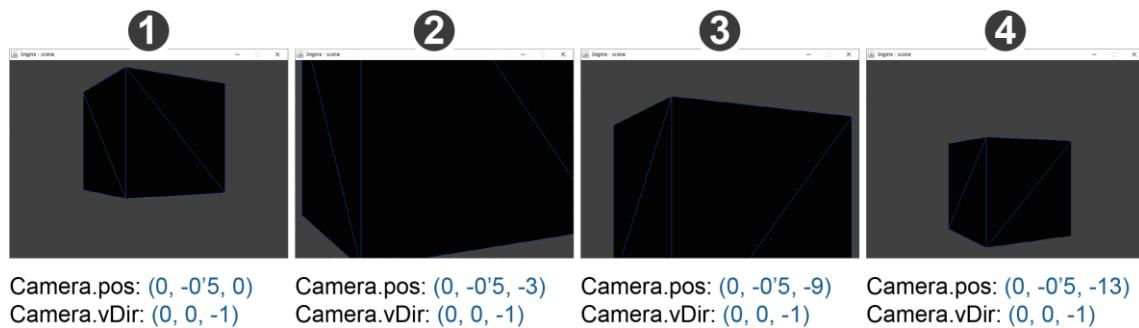
Figura 37. Render del cubo de prueba con los triángulos ordenados (captura propia).

5.6 Triangle clipping

Antes de seguir con la luz y la carga de objetos, hay un error que se debe solucionar a la hora de proyectar los triángulos. Si alguno de los vértices se sitúa detrás de la cámara, la matriz de proyección genera un inverso de las coordenadas al otro lado de ésta, generando un efecto óptico similar al de un cubo falso detrás de la cámara. Por eso, si el usuario se acerca demasiado y/o los vértices se sitúan detrás de él, éste puede ver un duplicado del cubo que no debería ser visible.

Es decir, el cubo que aparece en las capturas 3 y 4 de la Figura 38 en verdad no existe.

Renderizado del engine: El “cubo falso”



El cubo se encuentra estático, en la posición inicial (0, 0, 6).
La cámara siempre avanza hacia Z = -1.

Figura 38. Captura del error de proyección usando el cubo de prueba.

Este “falso cubo” es el resultado del proceso de proyección de los vértices, cuando éstos están detrás de la cámara. Se produce un renderizado del cubo que sí que existe, pero invertido (de ahí el ligero efecto de reflejo de la captura entre la primera toma y la cuarta, pese a que la altura de la cámara no ha cambiado).

Para solucionar este problema, se utiliza un método que recorta los triángulos cuando éstos chocan con el plano de proyección de la cámara.

5.6.1 Procedimiento

En el Diagrama 48 se muestra de forma visual en qué consiste este “recorte” de los triángulos, generado en el plano de la cámara (que habrá que definir).

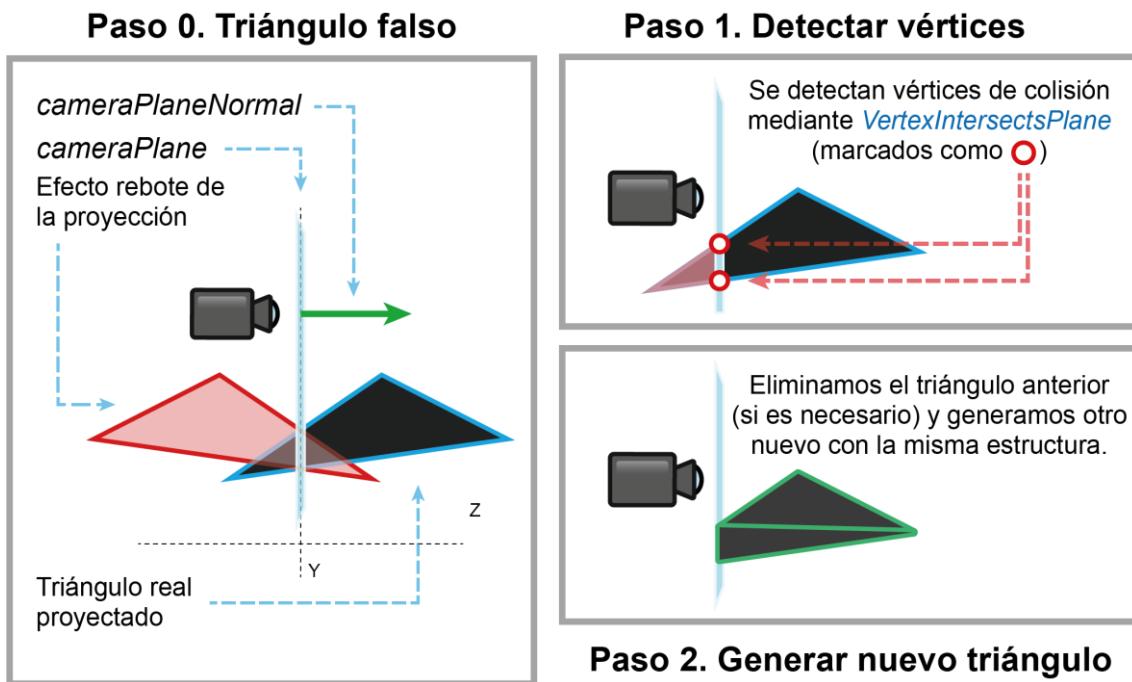


Diagrama 48. Procedimiento de “triangle clipping”, para solucionar la proyección falsa de triángulos.

Son necesarios dos nuevos métodos para realizar este *clipping*:

- **`VertexIntersectsPlane`** devuelve la coordenada en que un vector atraviesa un plano (recibe las coordenadas en que se encuentra el plano en forma de vértice, y las coordenadas de inicio y fin del vector que lo atraviesa).
- **`TriangleClipToPlane`** es la función que llama a `VertexIntersectsPlane`, y recorta los triángulos. Ésta recibe como parámetro el triángulo que hay que recortar, y devuelve un nuevo *array* con los triángulos que habrá que proyectar en la escena (un *array* porque un triángulo puede dividirse en varios al recortarse, y será necesario devolverlos todos para la lista de proyección).

En general, el cambio que se va a hacer en el programa es simple: antes, nada más calcular todas las transformaciones de los vértices, cada triángulo se mandaba a la lista de proyección para ser renderizado en la ventana. Ahora, después de calcular todas las transformaciones de los vértices, comprobaremos si alguno de ellos se está proyectando por detrás de la cámara (por ende, generando el anterior error) y, en caso afirmativo, en vez de mandarlo a la lista de

triángulos a proyectar, se manda a la función *TriangleClipToPlane*, y enviamos el *array* resultante de dicha función a la lista.

Como último dato antes de avanzar al desarrollo de este apartado en el proyecto, todas estas funciones mencionadas están en la clase *UtilsMath*, junto a las demás operaciones aritméticas de vectores.

5.6.2 Información necesaria para el *clipping*

El **producto escalar entre dos vectores normalizados** equivale a la fórmula de la Ecuación 3, donde v_1 , v_2 y v_3 representa cada módulo del vector. Igual para \vec{u} .

$$\vec{v} \cdot \vec{u} = v_1 \cdot u_1 + v_2 \cdot u_2 + v_3 \cdot u_3$$

Ecuación 3. Producto escalar entre dos vectores normalizados.

Ahora se avanza a la implementación del método *VertexIntersectsPlane*. Para esto son necesarios dos atributos más en la cámara del *engine* (*CameraModel*). Uno de ellos es un *Vertex* que señala la posición en que se encuentra el plano de la cámara —*cameraPlane*—, y otro *Vertex* que indica el vector normal del plano —*cameraPlaneNormal*. A nivel técnico, estos indican en qué eje de ordenadas señala el plano de la cámara y a qué distancia de la cámara los triángulos deberán empezar a recortarse. Estos atributos se definen al principio y **no es necesario actualizarlos**, ya que son relativos a la proyección que los vértices sufrirán frente a la cámara. No es físicamente necesario moverlos junto a la cámara **porque**, recordemos, **nuestra cámara no se desplaza por el entorno, sino que el entorno se transforma de manera inversa**.

Para hacer pruebas durante el recorte de los triángulos, un valor cómodo para el *cameraPlane* es el **(0,0, -1)**. Esto significa que los triángulos empezarán a “clipearse” cuando se acerquen a más de una unidad de la cámara. Por su parte, *cameraPlaneNormal* siempre tendrá el valor **(0,0,1)**, señalando a lo largo del eje Z, que es sobre el que se hace la proyección de la cámara.

Es importante remarcar que los vectores deben estar normalizados.

5.6.3 *VertexIntersectsPlane*

Ya tenemos la información suficiente para implementar la función que calcula el vértice de colisión del vector contra el plano de la cámara.

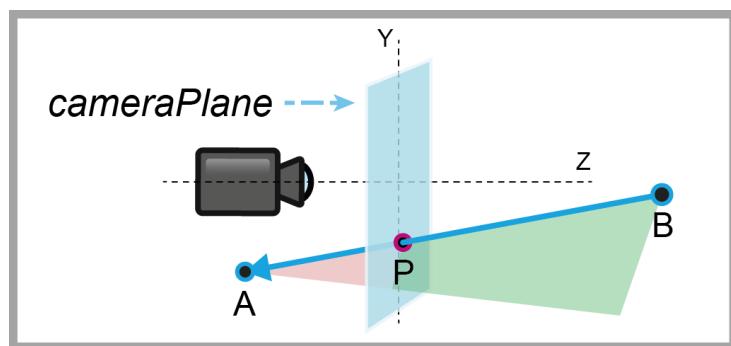


Diagrama 49. Representación de un vector que atraviesa el plano de la cámara.

Para ilustrar el proceso se hará uso del Diagrama 49, donde se puede apreciar un vector que viaja del vértice B al A (éste representa un lado del triángulo a recortar).

El punto de interés a calcular es la coordenada de colisión de este vector \overrightarrow{BA} con el plano de la cámara (*cameraPlane*): el vértice **P**.

Sabemos que:

$$\begin{cases} \overrightarrow{BA} = (x_{BA}, y_{BA}, z_{BA}) = (x_A - x_B, y_A - y_B, z_A - z_B) \\ P = (x_P, y_P, z_P) \end{cases}$$

Ecuación 4. Vectores y punto necesario para el cálculo de la intersección del vector en el plano de la cámara.

Y podemos considerar el vector \overrightarrow{BP} como una fracción del vector \overrightarrow{BA} . Es decir:

$$\overrightarrow{BP} = \overrightarrow{BA} \cdot t$$

Ecuación 5. Relación entre el vector del punto B a P y el vector de B a A.

Lo cual genera la siguiente ecuación de tercer grado:

$$\begin{cases} x_P - x_B = x_{BA} \cdot t \\ y_P - y_B = y_{BA} \cdot t \\ z_P - z_B = z_{BA} \cdot t \end{cases}$$

Ecuación 6. Sistema de ecuaciones a resolver para obtener la coordenada de colisión del vector al plano.

Donde no conocemos el valor del factor t , ni dos de las coordenadas de **P** (x_P , y_P y z_P). Sin embargo, sí que contamos con información suficiente para operar la ecuación, dado que sabemos que la línea que “interseca” al *cameraPlane* colisionará con éste en *cameraPlane.z*.

Por tanto, el valor de t es calculable de la siguiente manera:

$$t = \frac{z_P - z_B}{z_{BA}}$$

Ecuación 7. Cálculo del factor t de escalado de vértices entre el vector BP y BA.

Una vez obtenido t , simplemente se sobrescribe para resolver las ecuaciones restantes de la ecuación de tercer grado, obteniendo las coordenadas de colisión del vector restantes (x_P e y_P).

A nivel de programación, el método seguirá la siguiente forma: los parámetros de la función son los mencionados previamente, y consisten en la posición del plano respecto a la cámara (en el 3ngine, **(0, 0, -0.1)**), y los dos vértices del vector que atraviesa el plano (el inicial y el final).

```
public static Vertex Vertex_IntersectsPlane(Vertex planeP,
                                             Vertex vStart, Vertex vEnd) {
```

Recordemos que, para hacer la resta de dos vectores, en *UtilsMath* hay funciones de cálculo de este tipo.

```
// Cómo obtener el punto de colisión del vector con el plano
Vertex BA = new Vertex(UtilsMath.SubVertex(vEnd, vStart));
float t = (planeP.getZ() - vStart.getZ()) / BA.getZ(); // t
float Cx = BA.getX() * t + vStart.getX(); // collision in X
float Cy = BA.getY() * t + vStart.getY(); // collision in Y
```

```
Vertex collision = new Vertex(Cx, Cy, planeP.getZ());
```

Tras esta operación, la función devuelve *collision*, que ya son las coordenadas de la colisión del vector en el plano.

5.6.4 TriangleClipToPlane

En el 3ngine, esta función se llama en el “loopFunction”, una vez por cada triángulo a proyectar. Es la función encargada de leer los triángulos transformados (antes de su proyección), y, por cada uno de ellos, devolver el *array* con los triángulos resultantes. Es importante que devuelva un *array* ya que, en función del caso, podría ser necesario hasta crear dos triángulos.

En la “loopFunction”, después de las transformaciones de los vértices y la cámara (antes de la fase de proyección), es necesario comprobar qué vértices de cada triángulo se encuentran detrás del plano de la cámara (es decir, sobre qué vértices es necesario recortar los triángulos). Para esto, simplemente se comprueba si la coordenada Z de los vértices es inferior a la Z del plano de la cámara (*cameraPlane*): en caso afirmativo, significa que el vértice se encuentra frente a la cámara y, por tanto, es visible. Por otro lado, si la Z del plano de la cámara es superior a la Z del vértice del triángulo en cuestión, significa que este vértice se encuentra detrás de la cámara y, por tanto, es necesario recortar el triángulo. Si la Z es la misma significa que el vértice está en la misma profundidad que el plano de la cámara.

Por ello, es recomendable crear dos *arrays* de *Vertex* e introducir en uno de ellos los vértices del triángulo que estén delante de la cámara (*insidePoints*, en el 3ngine), y en el otro array (*outsidePoints*) los que estén detrás del plano de proyección y no deban verse.

Una vez los tres vértices del triángulo están distribuidos en *insidePoints* y *outsidePoints*, pueden darse hasta cuatro casos distintos a considerar. En el Diagrama 50 se ofrece una muestra visual de cada uno de ellos.

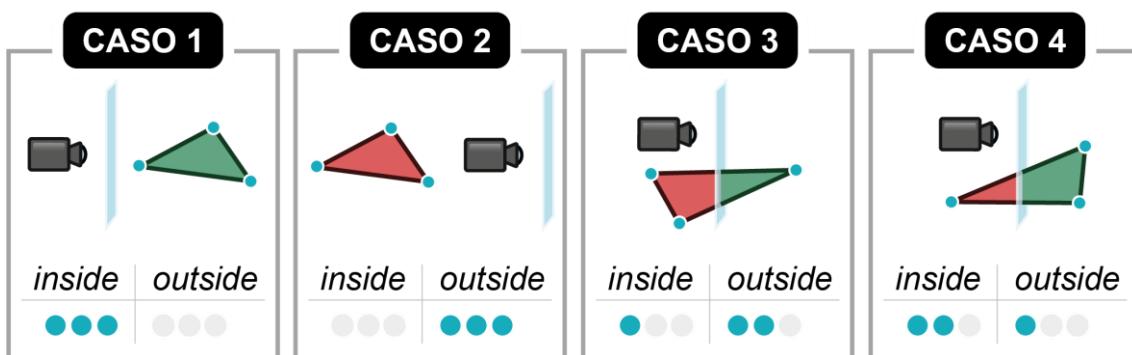


Diagrama 50. Distintos casos que pueden darse en el *triangle clipping*, con una representación visual de cada uno de ellos y cuántos vértices se encuentran frente al plano de proyección (*inside*) y los que no (*outside*).

En el caso 1 no es necesario hacer ninguna transformación: el triángulo se manda a la lista de triángulos a proyectar sin ningún cambio, ya que todos los vértices se encuentran frente al plano de la cámara.

En el caso 2, todos los vértices se encuentran por detrás de la cámara (es decir, ninguno debería ser visible), por lo que no lo mandamos a la lista de proyección.

Para el caso 3 hay que llamar dos veces a la función *VertexIntersectsPlane*, para obtener los puntos en que dos lados del triángulo colisionan con el plano de la cámara. Tras obtener los vértices de colisión como resultado, creamos un nuevo triángulo con el vértice que se encontraba dentro del plano inicialmente, y los otros dos obtenidos de la intersección del lado con el plano de la cámara.

En cuanto al caso 4, inicialmente el proceso es el mismo: llamar a *VertexIntersectsPlane* para las dos colisiones con el plano de la cámara. Sin embargo, al recortar el triángulo obtendríamos un *quad*. Es decir, un polígono de cuatro vértices, en vez de un triángulo. Esto significa que la función *TriangleClipToPlane* deberá crear entonces dos triángulos nuevos para formar el *quad*, como en el Diagrama 51.

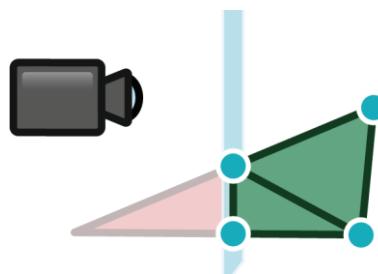


Diagrama 51. Formando un *quad* a partir de dos triángulos.

Al final, enviamos los dos triángulos a la lista de triángulos a proyectar.

Es importante recordar que, al crear los triángulos, debe mantenerse la orientación de los vértices deseada para obtener los vectores normales que interesen al desarrollador.

Al terminar la implementación de este método, el cubo de testeo se recortará si la cámara se acerca demasiado a éste. Para un efecto más exagerado se puede desplazar el plano de la cámara: por ejemplo, en la figuraXXX se observa el efecto cuando el plano de la cámara se encuentra en la posición **(0, 0, -1)** en vez de **(0, 0, -0.1)**, que es el valor por defecto.

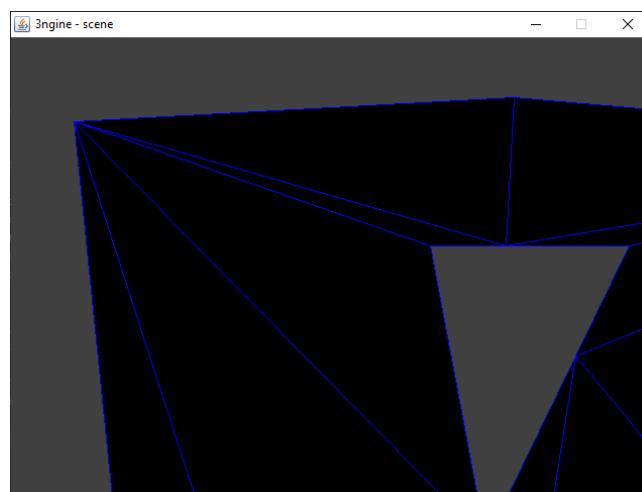


Figura 39. Captura del cubo de testeo con los triángulos recortándose en una de las esquinas. Los lados de cada triángulo permiten ver el efecto de *clipping*.

5.7 Rayo de luz

El rayo de luz del 3ngine es direccional. Es decir, se crea un **Vertex** (que actúa como vector) que **indica la dirección** en que la luz viaja en la escena (que es **(0, -0.5, -1)**, antes de normalizarla, en caso del 3ngine) y, mediante el producto escalar entre los vectores normales del triángulo y el vector de luz, asigna a cada triángulo un color en función de la incidencia de la luz.

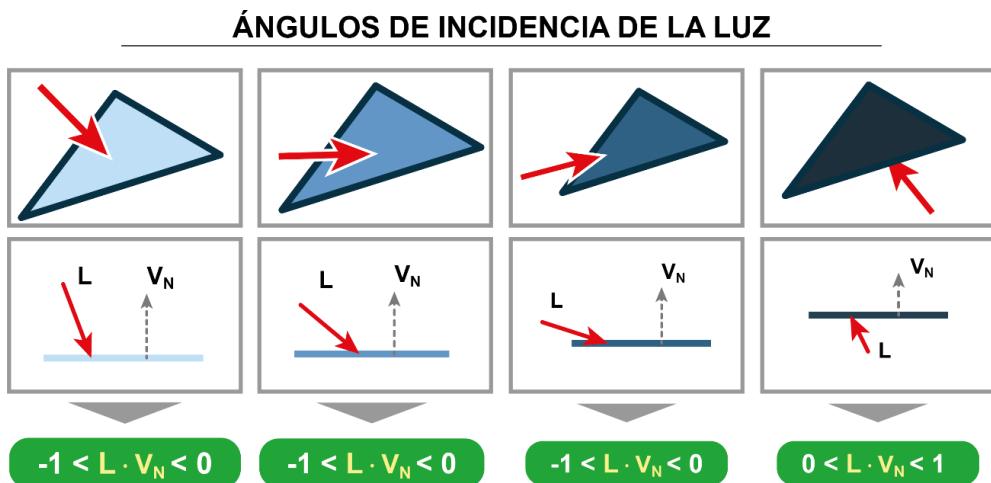


Diagrama 52. Representación gráfica del rayo de luz incidiendo sobre un triángulo.

En el Diagrama 52 se indica el efecto. También se puede ver los valores entre los cuales el producto escalar se situará según el rayo de luz que el desarrollador cree para la escena y el vector normal de cada triángulo. En la Ecuación 8 está la fórmula a implementar para la función “**dotProduct**” del 3ngine (también en el *package UtilsMath*), aunque, para que esta fórmula sea correcta, tanto la luz como el vector normal deben estar normalizados.

$$L \cdot V_N = L_X \cdot V_{N_X} + L_Y \cdot V_{N_Y} + L_Z \cdot V_{N_Z}$$

Ecuación 8. Producto escalar entre el vector L (luz) y el vector V_N (vector normal del triángulo).

Dado que el programador será quién decide la dirección y sentido del vector de luz, sólo se desconoce el vector normal de los triángulos de la escena. Para obtenerlo, se calcula el producto vectorial del triángulo en función de dos de los lados de éste (aquí es donde entraba en juego la importancia de la orientación de los vértices al crear los triángulos).

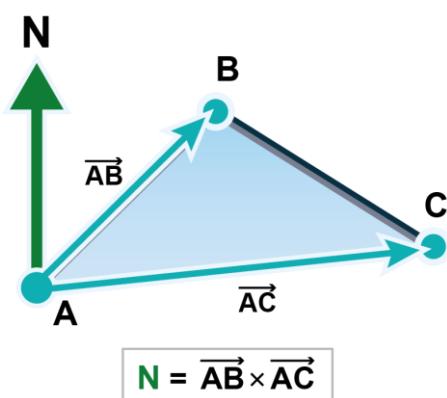


Diagrama 53. Cálculo del vector normal de los triángulos en función de dos de los lados.

Es indispensable remarcar que **la normal de los triángulos debe calcularse antes de las transformaciones de la cámara**. De lo contrario obtendríamos un efecto visual similar a si la luz se moviera con la cámara.

Tras calcular el producto vectorial para obtener la normal de cada triángulo, y luego el producto escalar para obtener la incidencia de la luz, éste último se guarda en la clase *Triangle* como *float*, y, al pintar los triángulos, se mapean los valores para que, a menor valor, más blanco se pinte el triángulo; hasta el cero, que sería el negro. El resultado puede apreciarse a continuación, en la Figura 40.

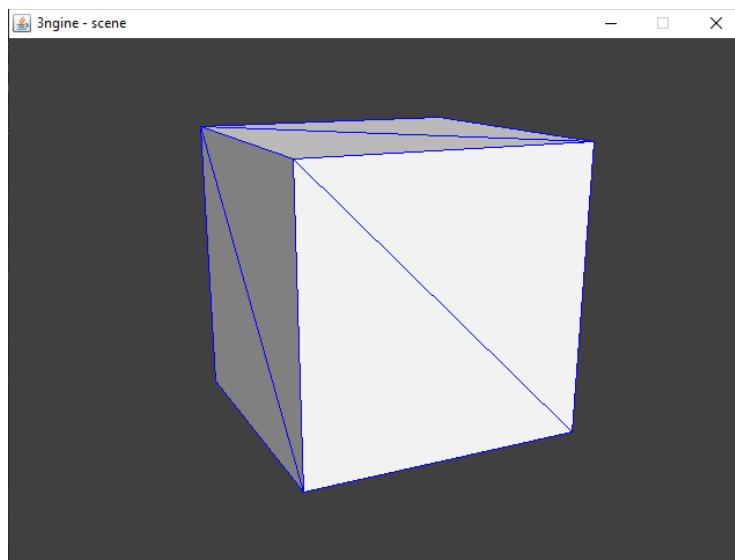


Figura 40. Renderizado del cubo con la luz direccional de la escena (0, -0.5, -1) (captura propia).

5.8 Carga de archivos 3D (formato .obj)

Para dejar descansar el cubo de testeo, ahora se procederá a la funcionalidad de lectura de archivos 3D en formato *OBJ*. Los métodos necesarios para este proceso están implementados en la clase *FileController*.

En el 3ngine, se añade un nuevo control: si el usuario pulsa la letra ‘O’ se llama a la función *openObjectFileChooser* (que devuelve un booleano que indica si el usuario ha decidido abrir otro archivo o no). Ésta crea un **JFileChooser** que espera a que el usuario escoja un documento para abrir. Gracias al método *setFileFilter*, se añade un filtro que oculta cualquier archivo que no sea una carpeta (o directorio) o un **.obj**. Tras eso, si el usuario ha clicado en un archivo para abrirlo, el **JFileChooser** lo obtiene y almacena la *File* en el atributo “*newFile*” de *FileController*. Después devuelve un *true* como resultado de la función. De lo contrario, devuelve un *false*.

Al terminar la ejecución de *openObjectFileChooser*, si ésta ha devuelto “cierto” como resultado, se llama a la función *resetEngineAndScene* de *EngineController*, que detiene el anterior hilo de ejecución del motor gráfico, cierra la escena (borra todos los triángulos y la información del objeto abierto previamente) y lee el objeto que encuentra en *newFile*, de *FileController*, mediante el método **readObjectFile** (que, si todo sale bien, devolverá la nueva escena, con todos los vértices y triángulos formados, a partir de la **lectura del .obj**). Si no ha habido ningún error, crea una nueva cámara, luz y un nuevo hilo de ejecución (*EngineLoopThread*) y empieza a

transformar los vértices. Para ahorrar métodos, **si la escena está vacía crea el anterior cubo de testeo**; si no (como es el caso actual), sólo carga el objeto leído en *readObjectFile*.

5.8.1 Formato .obj

Antes de comenzar, se analizará la estructura de un *.obj* estándar. Este tipo de archivos son documentos de texto con una estructura organizada por etiquetas, como en la Figura 41.

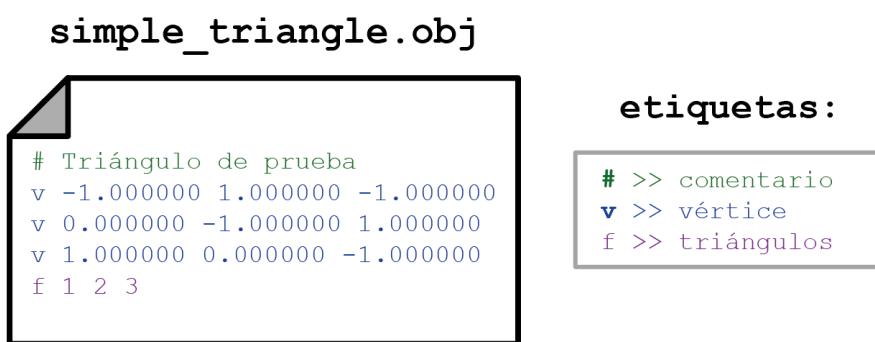


Figura 41. Estructura de un *.obj* de ejemplo.

Aunque hay otras etiquetas, como “vn” para los vectores normales de cada vértice, no los usaremos (ya se ha implementado nuestro propio sistema de generación de normales). Por otro lado, los términos numéricos que acompañan a la etiqueta “v” representan la posición de cada vértice, mientras que los tres números de “f” indican los vértices que forman un triángulo (es decir, habrá tantos vértices como “v” en el documento, y tantos triángulos como “f”).

Podemos crear este documento de prueba y pasar a su lectura: el resultado será un simple triángulo cuya normal enfocará a la cámara.

5.8.2 Lectura del .obj

La lectura del archivo corre a cargo de la función *readObjectFile*, mencionada previamente. En ella, el 3ngine crea una variable de tipo *Scanner*, y le asigna el *newFile*, que contiene el archivo *OBJ* que el jugador eligió antes, en el *JFileChooser*. También crea un *array* vacío de *Vertex*, y otro de *Triangle*, que añadirá a un *SceneObject* (y lo devolverá como resultado de la función).

Para leer el documento, utiliza la función *hasNextLine* del *Scanner* y, mientras no haya terminado de leer el documento (es decir, mientras ésta devuelva *true*), leerá la línea con el método *nextLine*, y partirá el *String* resultante en varios en función de los espacios de la línea.

Al haberlos partido, se sabe que el primer segmento equivaldrá a la etiqueta de la línea, de modo que, en función del carácter, se actuará de un modo u otro:

- Si la etiqueta es “#”, la línea es un comentario y no es necesario hacer nada con ella.
- Si es “v”, crea un *Vertex* con las coordenadas indicadas y lo añade al *array* de vértices. Para convertir el *String* segmentado en un *float*, se utiliza la función *parseFloat*, de la clase *wrapper Float*.
- Si es “f”, crea un *Triangle* a partir de los vértices indicados. Para ello, dado que estamos leyendo el documento en el mismo orden en que añadimos los vértices al *array*, se puede simplemente acceder al índice del array que pone en el segmento. Para convertir

el *String* en un entero, también utilizamos una clase *wrapper*, sólo que, esta vez, es *Integer*. El método para ello es *parseInt*.

Tras leer el documento, cerramos el *Scanner* con el método *close*, se crea una nueva *SceneObject* y una nueva *Mesh*. Entonces, se añade la lista de triángulos a la malla, y la malla se añade a la lista de objetos de la escena.

Al final, el método devuelve la escena, y todos los vértices y la información ya están completos, dado que las normales y el *lightValue* de los triángulos se calculan a cada vuelta del bucle principal, en el “loopFunction”.

En la Figura 42 se muestra el proceso de llamada para abrir el triángulo de prueba que se ofrecía en la Figura 41.

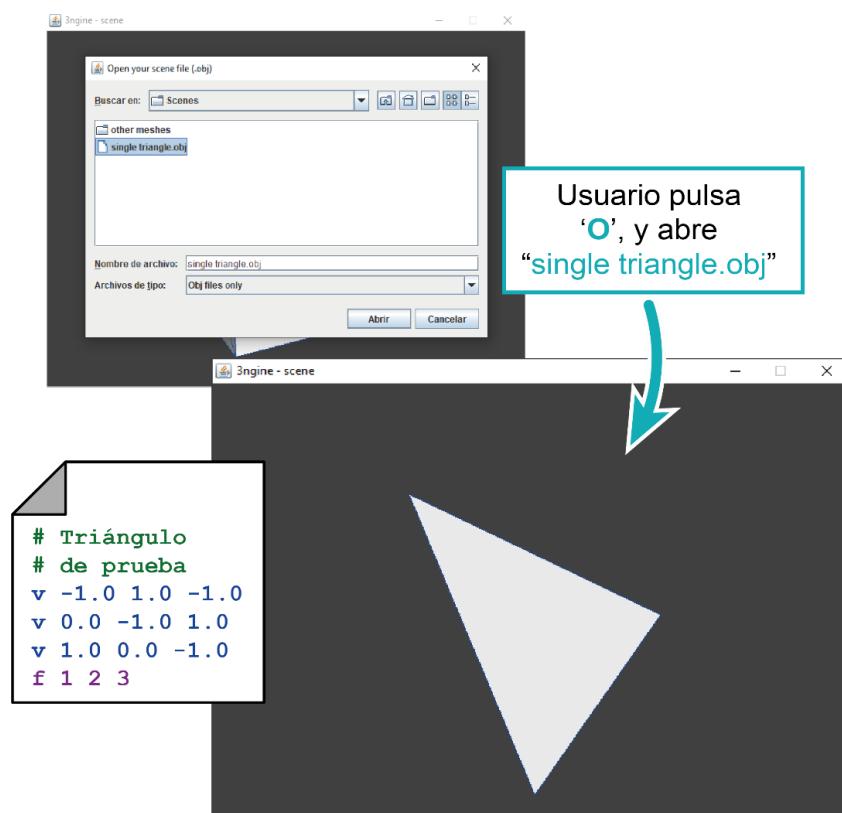


Figura 42. Captura resultante de la lectura del triángulo de ejemplo.

5.9 Optimización de triángulos

Una parte crucial del motor gráfico es la optimización de los cálculos que se llevan a cabo. Varias de las transformaciones que los vértices sufren son indispensables para conocer su posición y variaciones, pero hay un efecto que se puede tener en cuenta para optimizar el cálculo de los triángulos: **no es necesario renderizar los triángulos que no enfocan a la cámara, ya que** estos, al darse dicha situación, **no son visibles**. Sin ir más lejos, **en la Figura 40**, sólo tres de las caras del cubo (es decir, seis triángulos) son visibles por la cámara. Esto significa **no es necesario proyectar ni renderizar la mitad de los vértices y triángulos de la mesh**.

Para saber si los triángulos del objeto enfocan a la cámara (y, por tanto, si son visibles), se utiliza un método muy similar al empleado para calcular la incidencia de la luz sobre los triángulos. Sin

embargo, esta vez, en vez de utilizar el vector de la luz, se calcula el producto escalar entre un vector que viaja desde cualquier vértice del triángulo hacia la cámara (al que llamaremos *cameraRay*) y la normal del triángulo.

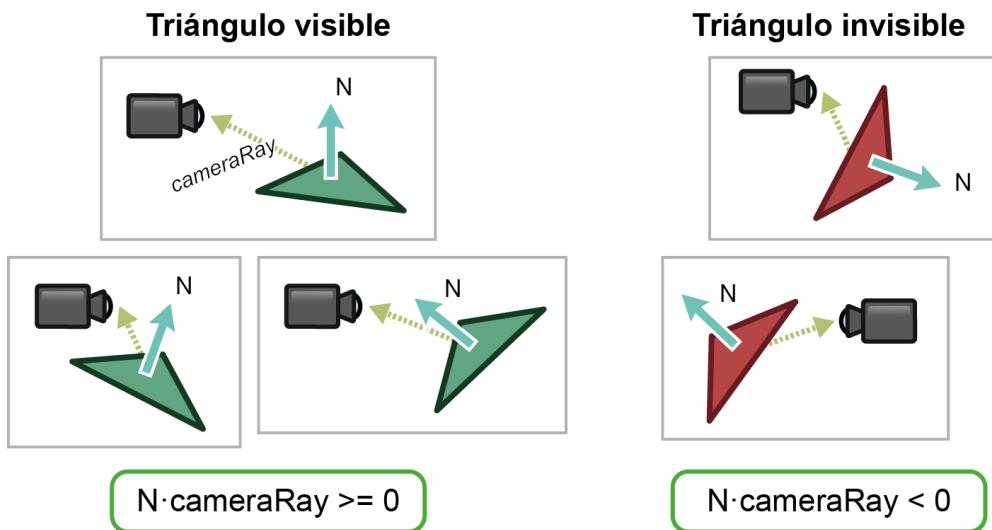


Diagrama 54. Distinción entre triángulos que enfocan a la cámara y los que no mediante el producto escalar entre *cameraRay* y el vector normal (N) del triángulo.

En el 3ngine, el método llamado *checkIfFacingCamera* (de la clase *Triangle*) lleva a cabo esta comprobación. Sin embargo, en el proyecto también puede considerarse que el vector *cameraRay* viaja en sentido opuesto (de la cámara al triángulo), y entonces los triángulos serían visibles cuando el producto escalar entre la normal del triángulo (previamente calculada) y dicho rayo sea inferior a 0, y no superior o igual, como se muestra en el Diagrama 54.

Este cálculo hay que hacerlo después de aplicar las transformaciones de los vértices. Se puede llamar a dicho método incluso antes de aplicar las transformaciones de la cámara, aunque suene incoherente, ya que la posición de la cámara (que es el único atributo de la cámara que interviene en la operación) se actualiza al principio del *loop*, cuando el usuario pulsa la tecla.

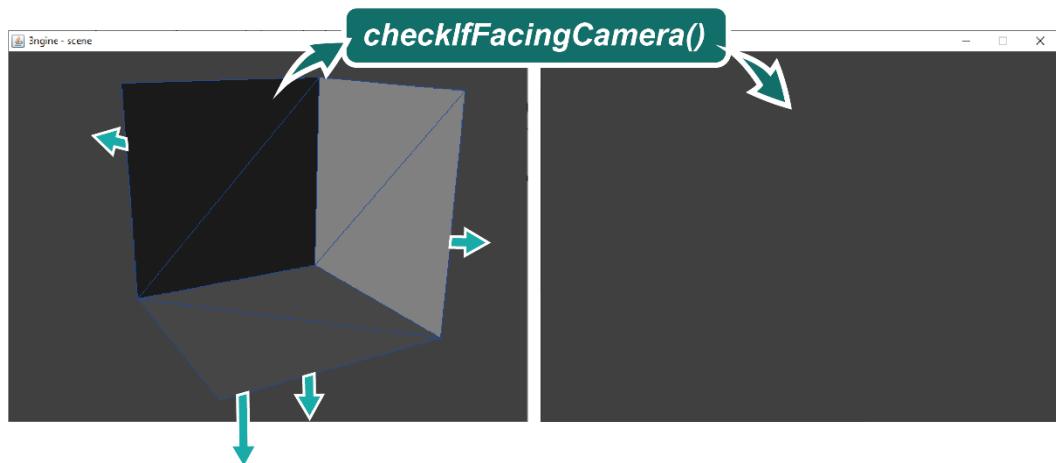


Diagrama 55. Efecto al activar *checkIfFacingCamera* (capturas propias del render).

En el Diagrama 55 se muestra el efecto del método implementado con seis de los doce triángulos del cubo. Dado que el sentido de *cameraRay* es opuesto a las normales de todos los triángulos

mostrados en la captura, si activamos el método ninguno de ellos se verá hasta que movamos la cámara a otra posición, de frente a los tres lados del cubo.

5.10 Métodos de renderizado

Más que una nueva funcionalidad, es el resultado de todos los cálculos realizados hasta la fecha. Tras conseguir la proyección de los vértices, la transformación del objeto y la cámara, el *clipping* y ordenación de los triángulos y los procesos adicionales de luz, optimización del *engine* y lectura de ficheros *.obj*, el usuario puede pasar a hacer pruebas con varios tipos de renderizado.

En el 3ngine, se puede alternar entre tres tipos distintos de renderizado, pulsando el 1, 2 y 3 del teclado. Éstos cambian el renderizado a modo *surface*, *wireframe*, y completo (*full render*) respectivamente, aunque, a nivel técnico, simplemente ejecutan o no las operaciones correspondientes según el modo de renderizado indicado (mediante condicionales *if* en el código).

Para escribir el texto en el canvas, el *engine* utiliza la función *drawString*, que recibe el texto y las posiciones en "x" e "y" (en píxeles) donde escribirlo.

5.10.1 Completo (*full render*)

El renderizado completo de los objetos de la escena es el presentado durante las últimas fases del proyecto (en la Figura 43 se muestra la Tetera de Utah, un estándar del modelado 3D [20] para gráficos por ordenador).

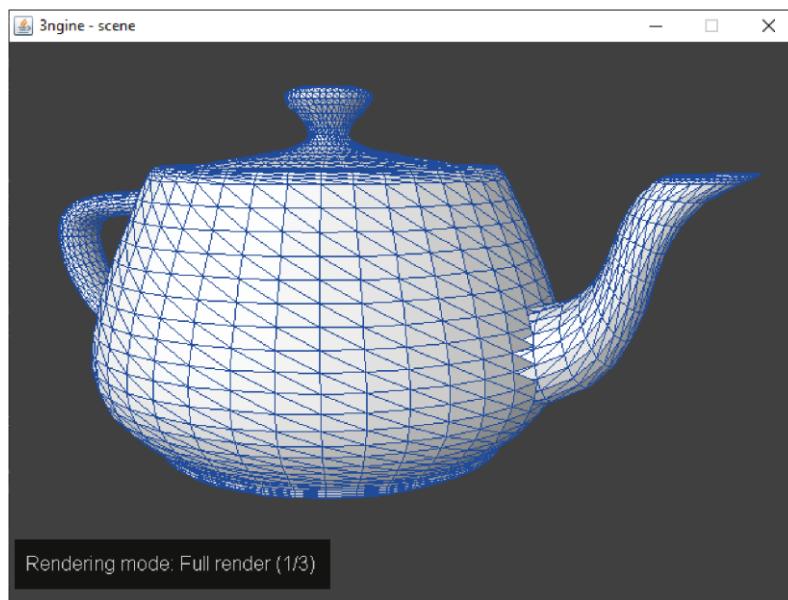


Figura 43. Renderizado de la Tetera de Utah, en modo *full render* (captura propia).

Este tipo de renderizado aplica todas las funcionalidades descritas a lo largo del proyecto: además de las transformaciones de los vértices (que son comunes a todos los tipos de renderizado), ordenación de los triángulos y su optimización (es decir, comprobar si éstos enfocan a la cámara e ignorar los que no lo hagan).

5.10.2 Surface

Para mostrar únicamente la superficie de los triángulos deben hacerse las mismas comprobaciones y cálculos que en el caso del *full render*. Sin embargo, si a la hora de renderizar no se pinta el contorno de los triángulos se obtiene un efecto de realismo por capas gracias a la incidencia de la luz, mediante *flat lightning*, como en la Figura 44.

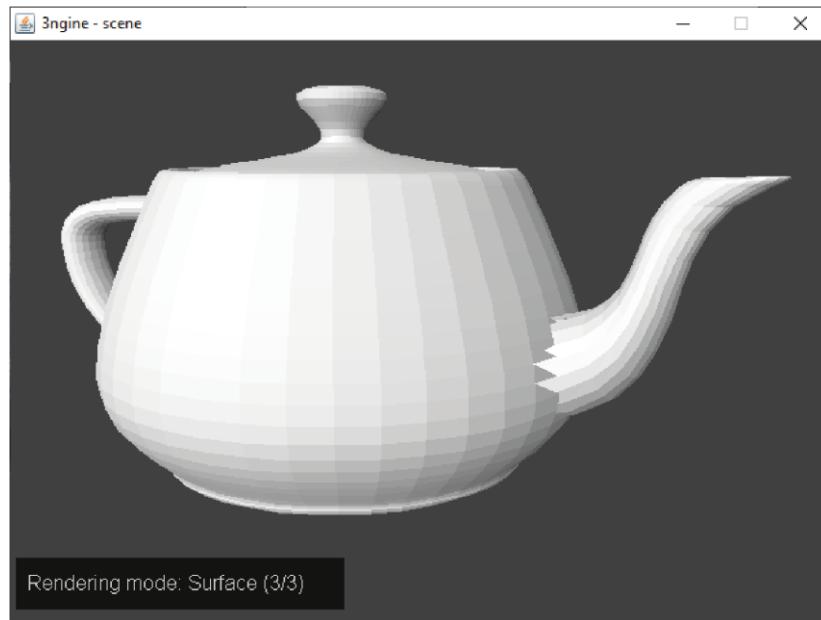


Figura 44. Renderizado de la Tetera de Utah, en modo *surface* (captura propia).

5.10.3 Wireframe

El modo *wireframe*, como su nombre indica, el renderizado muestra el “cableado” de los triángulos, pero no su superficie. Este tipo de renderizado es sumamente útil para analizar la estructura de los modelos, ya que permite visualizar todos los vértices del objeto a la vez en cada *frame*.

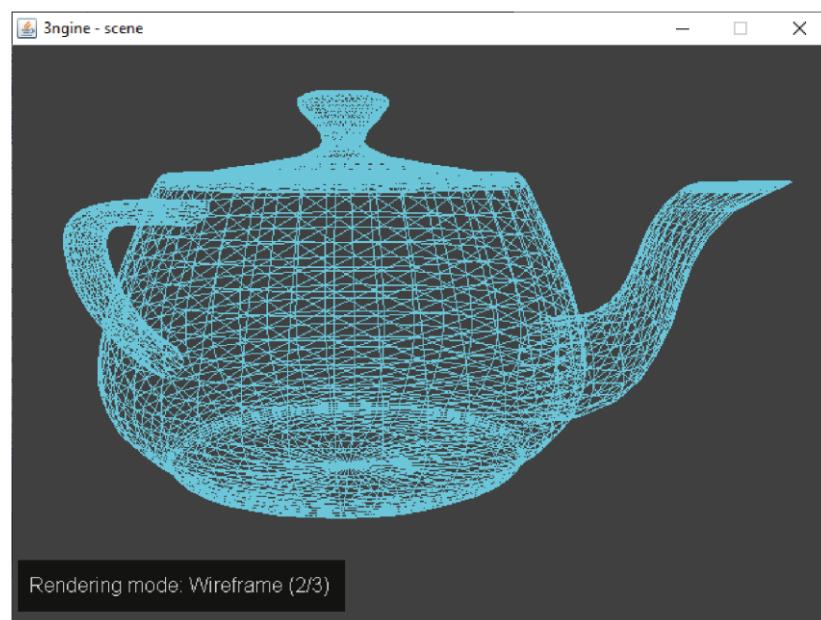


Figura 45. Renderizado de la Tetera de Utah, en modo *wireframe* (captura propia).

Aunque pueda resultar en una gran cantidad de cálculos a nivel de procesado, dado que no es necesario ordenar los triángulos ni comprobar sus normales o su incidencia con la luz de la escena, simplemente se basa en la transformación de los vértices de la escena, por lo que es incluso más óptimo que los dos anteriores tipos de renderizado.

6 Comparación: 3ngine vs Raycaster

Antes de realizar la comparación debe tenerse en cuenta que, a nivel técnico o sobre las especificaciones de cada sistema, éstos son demasiado distintos (incluida su plataforma de ejecución) como para buscar un aliciente común equivalente a analizar entre ambos.

Por este motivo, primero se establecen los parámetros del ordenador en que se ejecutan los motores, luego los parámetros cada *engine* (que, a grandes rasgos, pueden considerarse las propiedades de la cámara —o características del renderizado— de éstos), y luego se pasará al análisis comparativo entre ambos.

6.1 Parámetros del sistema (PC)

Los parámetros del equipo en que se ejecutan ambos motores son los mostrados en la Figura 46.

Especificaciones del dispositivo		Especificaciones de Windows	
Nombre del dispositivo	DESKTOP-FR99E0H	Edición	Windows 10 Home
Procesador	Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz 3.19 GHz	Versión	20H2
RAM instalada	16,0 GB	Instalado el	21/03/2021
Identificador de dispositivo	[REDACTED]	Compilación del sistema operativo	19042.1165
Id. del producto	[REDACTED]	Experiencia	Windows Feature Experience Pack 120.2212.3530.0
Tipo de sistema	Sistema operativo de 64 bits, procesador basado en x64		
Lápiz y entrada táctil	Compatibilidad con entrada manuscrita		

Figura 46. Propiedades del sistema (sin identificación de dispositivo ni de producto) (captura propia del menú de "Configuración" de Windows).

En cuanto al adaptador de pantalla, la tarjeta gráfica es una NVIDIA GeForce GTX 1060, de 6GB de RAM dedicada, y la versión 27.21.14.6647 del controlador, como se puede apreciar en la Figura 47.



Figura 47. Tarjeta gráfica del sistema (captura del "Administrador de Dispositivos" de Windows, "Adaptadores de pantalla").

6.2 Parámetros del *Raycaster* (motor 2D)

Para la comparativa, se considerarán parte del rendimiento todas las funcionalidades implementadas, de modo que, además de la emisión de rayos y la proyección de los muros del mapa (la matriz numérica que más tarde se convertiría en imagen a color en baja resolución), también se tendrá en cuenta el renderizado de texturas, que acarrea el mayor impacto en cuanto a consumo de recursos durante la ejecución del programa.

A nivel más técnico, cabe recordar que el *raycaster* genera una simulación de un entorno 3D mediante el *casteo* de rayos y sus colisiones. De este modo, y aprovechando que ya se configuró de manera que el usuario pudiera decidir a placer cuántos rayos emitir y el ángulo de visión de la cámara, los parámetros recomendados en este aspecto son los siguientes:

- Número de rayos: **160**. Para evitar *glitches* visuales, es recomendable que el número de rayos a emitir sea un múltiple de 8. También es importante recordar que, a mayor número de rayos, menor será el rendimiento global del sistema, ya que los cálculos se acentuarán progresivamente. 160 rayos permiten una clara distinción del nivel a cualquier distancia de los muros, así como una reconocible visualización de las texturas.
- Ángulo de visión (FOV): **70 grados sexagesimales**. Al incrementar demasiado el ángulo de visión, el efecto ojo de pez generado por las operaciones trigonométricas a izquierda y derecha de la cámara aumenta, y, por el lado opuesto, al establecer un ángulo de visión demasiado estrecho, la cámara genera un *zoom-in* que tampoco es realista. De este modo, 70 grados sexagesimales dan un resultado eficiente.
- Resolución de las texturas: **16x16 píxeles**. Por último, el *Raycaster* también consumirá más recursos cuanto mayor sea la resolución de las texturas del usuario, ya que deberá crear más rectángulos de color por cada píxel de la textura asociado a los muros.

A nivel de entorno de ejecución, el *engine* se está ejecutando en una ventana de incógnito de Google Chrome, en su versión oficial 92.0.4515.159 para máquinas de 64 bits, como se puede apreciar en la Figura 48; y con la configuración por defecto de navegador.

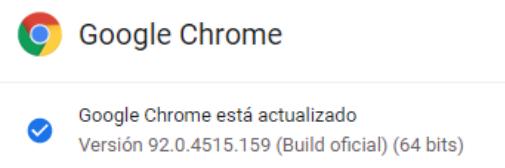


Figura 48. Captura de información de Google Chrome.

A continuación, en la Figura 49 se muestra una captura de pantalla del *engine* 2D o “Raycaster” con esta configuración visual. Además, para ello, se ha añadido una función de cálculo de los *fps* del sistema (ésta calcula una estimación de dicha velocidad invirtiendo el tiempo entre *frame* y *frame*). Indica el máximo número de *fps* alcanzado, el mínimo, y el actual (el último tiempo de *frame* registrado por el sistema).

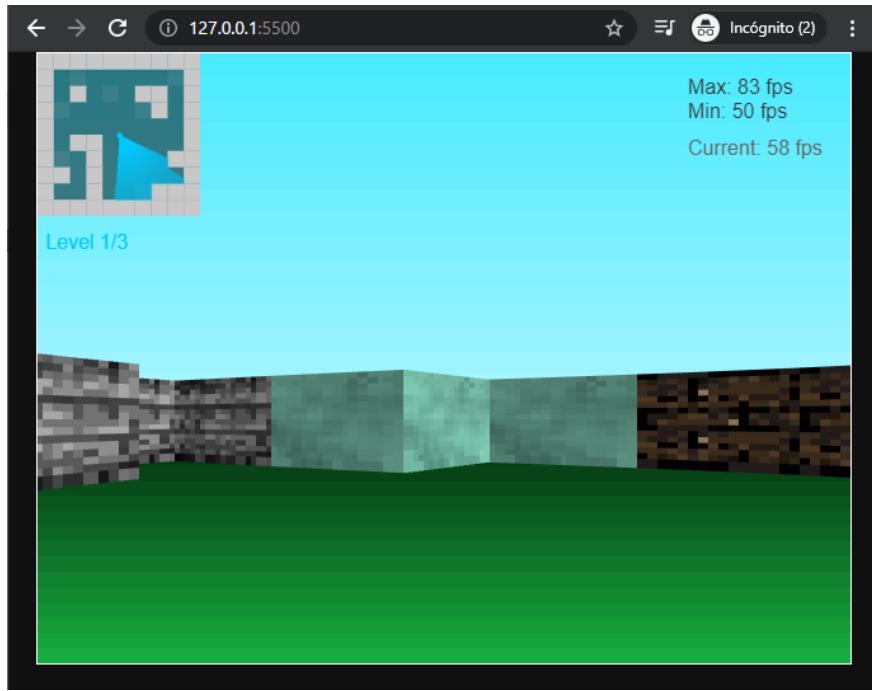


Figura 49. Aspecto visual de un renderizado del *Raycaster* para la muestra de parámetros (captura propia del *engine*).

Por último, para sobrepasar el límite de FPS configurado por defecto en Google Chrome, al principio de la función “setup” del archivo *main.js* se han añadido dos instrucciones adicionales.

```
noLoop();
setInterval(redraw, 0);
```

Éstas le “dicen” a la función *draw* que debe llamarse a un nuevo *frame* tras el período de tiempo indicado en el segundo parámetro de “setInterval”. En este caso se ha introducido un 0, forzando al sistema a mandar una nueva imagen al monitor nada más termine la ejecución de la anterior.

De no emplear este método, la velocidad de frames que *P5.js* establece para el navegador es de 60 fps, lo cual no sería una muestra real del rendimiento completo del sistema, sino que estaría limitada.

6.3 Parámetros del 3ngine (motor 3D)

Por su parte, los parámetros de configuración del 3ngine residen en las propiedades de la matriz que se utiliza como cámara (a recordar, la matriz de proyección, que simulaba las propiedades físicas de una lente para convertir los vértices de un espacio 3D a uno 2D).

Estos valores se encuentran en la función “getProjectionMatrix” de *UtilsMath*, que es la que devuelve dicha matriz (luego leída y almacenada en *CameraModel*), excepto el tamaño de la ventana del *engine*, que se encuentra en *EngineModel*. La ventana tiene unas dimensiones de 1280 píxeles de ancho por 720 píxeles de alto, mientras que los demás datos pueden obtenerse de la Figura 50:

```

public static float[][] getProjectionMatrix() {

    float focalLength = .15f;
    float sensorSizeX = EngineModel.dimX/10000f; //0.128f;
    float sensorSizeY = EngineModel.dimY/10000f; //0.072f;
    float vX = focalLength * (float)EngineModel.dimX / (2* sensorSizeX);
    float vY = focalLength * (float)EngineModel.dimY / (2* sensorSizeY);

    return new float[][] {
        {vX,      0.0f,      0.0f,      0.0f},
        {0.0f,     vY,      0.0f,      0.0f},
        {0.0f,     0.0f,     -1.0f,      0.0f},
        {0.0f,     0.0f,      0.0f,      1.0f}
    };
}

```

Figura 50. Datos de la matriz de proyección del 3ngine (captura propia del proyecto).

El modo de renderizado que se considera por defecto es el completo: *wireframe* y superficie, ya que incluye todos los subprocessos de cálculo que el motor debe realizar en su funcionamiento habitual, como la ordenación de triángulos, cálculo de los vectores normales, etc.

Además de ello, se ha modelado una malla que imita el mapa del raycaster. Ésta puede apreciarse, junto al aspecto final del renderizado, en la Figura 51.

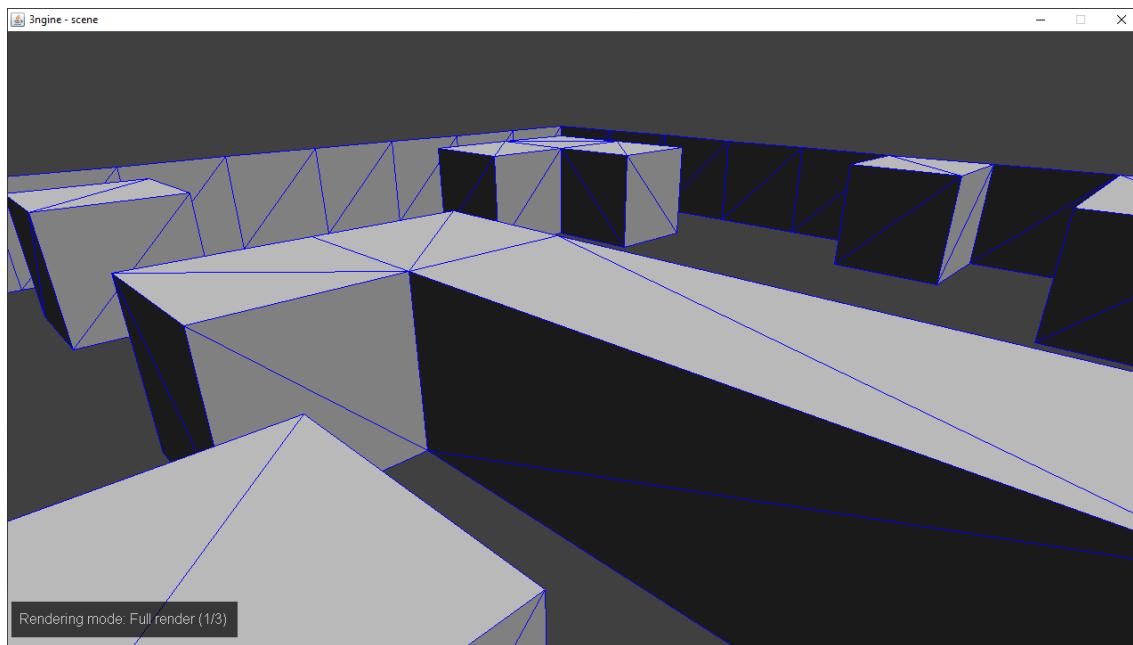


Figura 51. Resultado final del renderizado en el 3ngine. La malla consiste en una imitación del mapa del *Raycaster* (captura propia del proyecto).

En cuanto a los *frames* por segundo, a la hora de programar el *loop* del *engine* ya se le asocia un valor de *fps* concreto, de modo que hacer el cálculo como en el caso del *Raycaster* no tiene sentido. En el 3ngine, el número de *fps* limitados equivale a los 70, y se mantiene estable durante el renderizado de la malla mostrada, independientemente del ángulo y la posición de la cámara o el número de vértices mostrado en el *canvas*.

Sin embargo, cabe tener en cuenta que el 3ngine no cuenta con texturizado de capas, a diferencia del *raycaster*, de modo que, aunque pueda resultar en un motor fiable, el consumo de recursos no está en sus máximas posibilidades. Además, ninguno de los dos sistemas cuenta con efectos visuales adicionales ni sistemas de partículas o “postprocesado”.

6.4 Rendimiento general

Medir ambos motores gráficos bajo las mismas condiciones no es del todo equivalente, dado que no cuentan con las mismas funcionalidades, pero sí puede hacerse una prueba de rendimiento pesado: ver cuántos fotogramas por segundo la máquina es capaz de ejecutar sin que se produzca una bajada global en el rendimiento de éstos.

En el caso del *Raycaster*, el proyecto sufre de un consumo suplementario por el entorno web en que se está ejecutando (en este aspecto, sería más adecuado montarlo como un ejecutable o aplicación de escritorio). Pese a ello, como se muestra en la Figura 52, se obtiene una sobresaliente media de 100 *fps* o más, con picos positivos de hasta 125. La tasa mínima alcanzada en el momento de la captura fue de 50 *fps*, durante la carga inicial y los primeros frames de la ejecución. De no ser por el límite de imágenes por segundo de Google Chrome, el *Raycaster* podría disfrutarse en su totalidad en pantallas de 120 Hz de tasa de refresco.

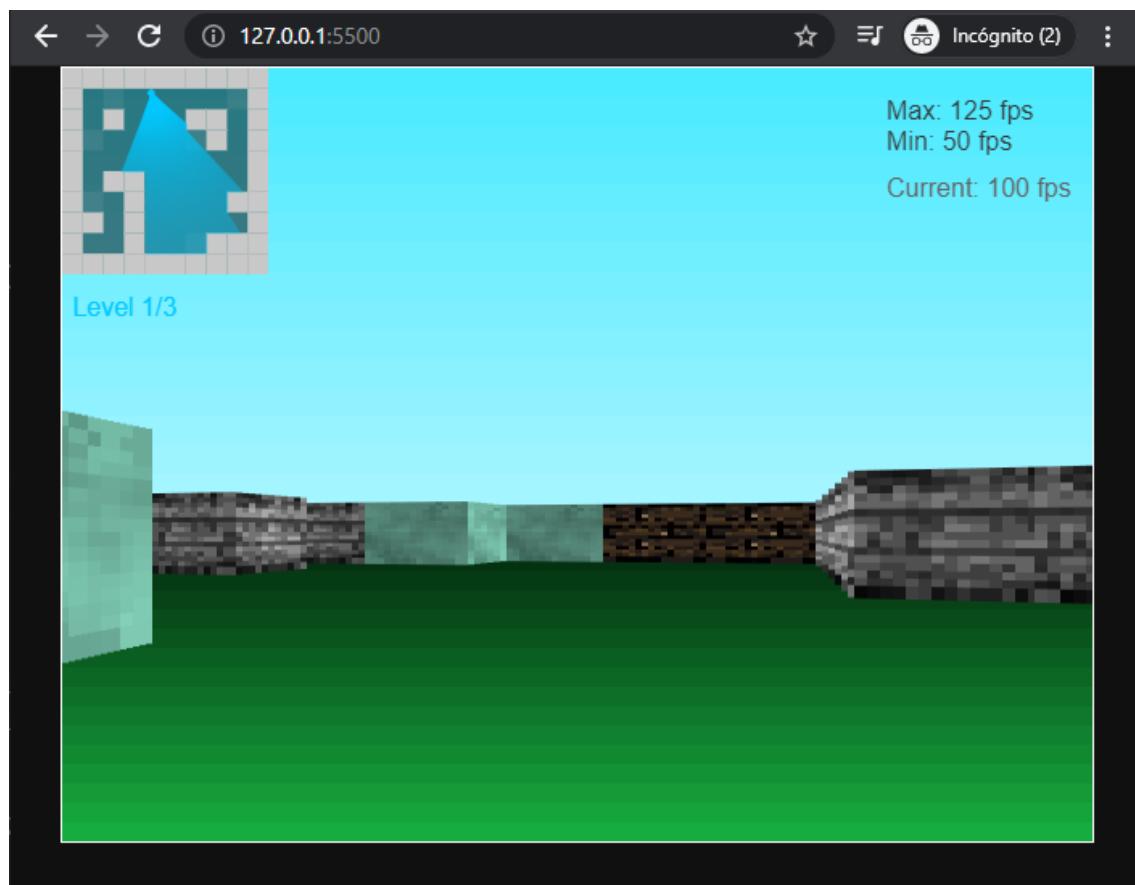


Figura 52. Captura de pantalla del *Raycaster*, durante prueba pesada sin tiempo límite de *frame rate*.

El resultado es muy satisfactorio, teniendo en consideración que el motor cuenta ya con texturas aplicadas, colisión de muros e incluso interacción por parte del usuario y funcionalidades adicionales como el nivel múltiple de la figura anterior.

Tras realizar la prueba del motor 2D, se puede ahora avanzar al 3ngine. Ésta es una aplicación Java de escritorio, y en ella no se han desarrollado ni la aplicación de texturas ni la colisión de la cámara (o jugador) con superficies, de modo que en primera instancia el rendimiento deberá ser mayor (pese a que los cálculos son varios más, y más pesados que los implementados en el *raycaster*).

Para la prueba, se han añadido algunos cálculos adicionales, listados por orden a continuación, junto a la Figura 53:

- **FPS** (arriba, a la derecha, en verde): el valor de *fps* establecido por el usuario y que el sistema utiliza como referencia para la velocidad del hilo de ejecución del motor. Como se ve en la imagen, el proyecto es capaz de cargar 500 *fps* (aunque, de éstos, al monitor sólo le pasan 60 o 120, según la pantalla). Si se incrementara aún más el número de *fps* a cargar, el 3ngine empieza a congelarse y se produce un poco de *flickering* en la imagen, ya que se estaría cargando un nuevo *frame* antes de que el monitor hubiera terminado de cargar el último.
- **Frame count**: indica el índice del *frame* que se está renderizando cada segundo (en la figura, se ha capturado el frame número 370). Al llegar a 500, el contador volvería a cero.
- **Elapsed time**: equivale al tiempo (en **segundos**) que discurre entre *frame* y *frame*. Es el inverso de los *fps*, como se muestra en la Ecuación 9. Por ello, si los *fps* son 500, el *elapsed time* será equivalente a 0.002 segundos, o 2 milisegundos.

$$fps = \frac{1}{elapsedTime}$$

Ecuación 9. Cálculo de los *fps* del sistema en función del *elapsed time* (o "tiempo de *frame*").

Por tanto, indica también cada cuánto tiempo se empezará a calcular una nueva imagen.

- **Delta time (operation loop)**: equivale al **tiempo** que le toma al 3ngine **realizar el cálculo de los vértices** y vectores del proceso. Es decir, empieza a contar al llamar a la "loopFunction" de *EngineController*, y termina de contar justo antes de llamar al *repaint* para renderizar la escena por pantalla.
- **Framing time (drawing loop)**: equivale al **tiempo que le toma** al 3ngine **renderizar la escena**. Empieza a contar al principio de la función "repaint" del *EngineView*, y termina al acabar de dibujar el último triángulo y mostrar la información por pantalla.
- **Safe-limit time (full loop)**: equivale al tiempo que tarda el 3ngine en realizar la **carga completa de un *frame***, desde que empieza el cálculo de los vértices hasta que termina de renderizar la escena. Es la suma del *delta time* y el *framing time*.

En la captura de pantalla (Figura 53) puede verse que los tiempos de carga son increíblemente rápidos, sumando un tiempo total de *safe-limit time* de 1 sólo milisecondo (por ello, dado que el *elapsed time*, o tiempo entre *frames*, es de 2 segundos, el sistema tiene tiempo suficiente como para ir cargando una nueva imagen antes de que empiece el cálculo de la siguiente. Los

problemas de carga de imagen se producen por estos tiempos. Los efectos más comunes son dos: el *flickering* (las imágenes se congelan brevemente y con cierta frecuencia), que es meramente visual y se produce cuando al 3ngine no le da tiempo a renderizar el *frame* al completo antes de llamar al siguiente; y el *freezing*, que es cuando el sistema se congela durante algunos *frames* porque el 3ngine tarda más tiempo en realizar los cálculos (y, por tanto, tampoco puede renderizar los gráficos) que el *elapsed time* en pasar al siguiente *frame*.

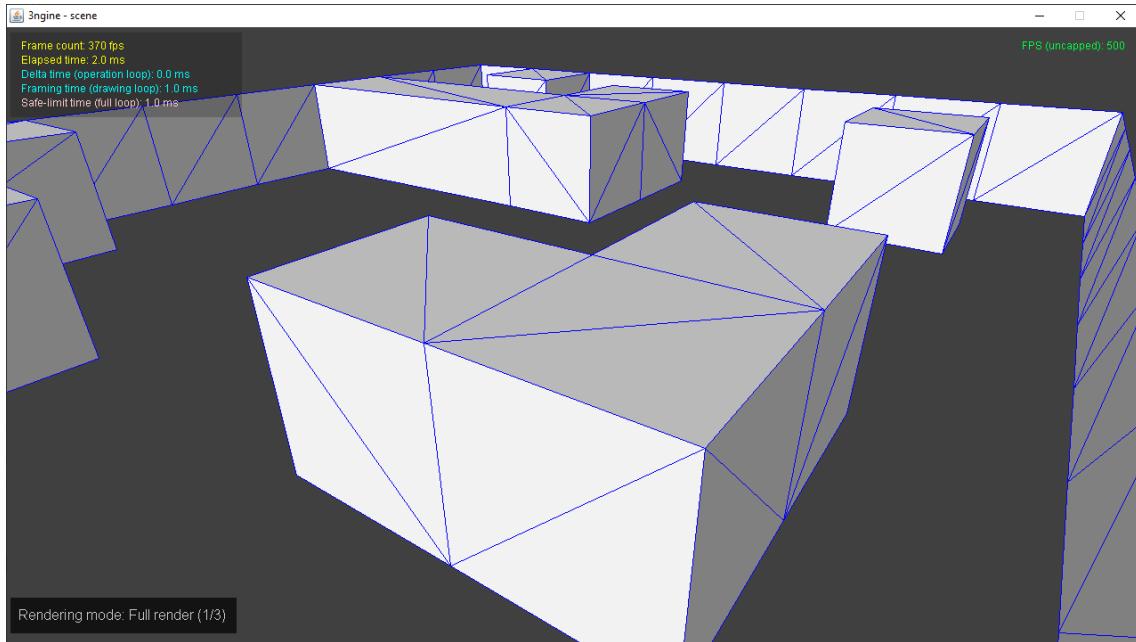


Figura 53. Captura de pantalla del 3ngine, durante prueba pesada (tiempo límite de *frame-rate*: 500 imágenes por segundo).

Aunque la potencia depende únicamente de la máquina en ejecutar la aplicación, y el modelo a renderizar, es lógico tener una gran velocidad en modelos con pocos vértices y sin aplicar texturas.

6.5 Optimización y limitaciones técnicas

El *Raycaster* cuenta con un método de renderizado extremadamente óptimo, ya que la proyección de los muros va de la mano únicamente con la colisión de cada rayo a lo largo de la matriz (mapa) en X e Y. Además de esto, la posibilidad de elegir el número concreto de rayos para el renderizado más adecuado ofrece un grado de configuración a nivel de usuario que el motor 3D no permite. Por ello es un sistema rápido incluso en sistemas con poca potencia (el cálculo que más sobrecarga la máquina es la carga de texturas, y ya se ha visto que incluso esa propiedad es igualmente modificable).

No es sorprendente saber entonces que, el cálculo de la proyección de muros para simular un entorno 3D es más eficaz en un emisor de rayos que en un sistema real 3D, donde únicamente se podrán obviar algunas de las operaciones matemáticas necesarias en función de los objetos que haya en la escena.

De este modo, para cargar un modelo con algún tipo de movimiento por parte de los vértices, además de una cámara no estática, como en un FPS o un videojuego de carreras, deberían irse sumando los 11 cálculos que ya se mostraron previamente, en el apartado 5.3 (página 95). En el

Diagrama 56 se muestran de nuevo los pasos del proceso completo de transformación de vértices.

Transformaciones de los vértices

Fase 1: Transformación del objeto

- 1** Rotación del objeto en el eje Y (*objRotY*)
- 2** Rotación del objeto en el eje Z (*objRotZ*)
- 3** Rotación del objeto en el eje X (*objRotX*)
- 4** Desplazamiento del objeto (*objTraslation*)

Fase 2: Transformación de la cámara

- 5** Desplazamiento de la cámara (*camTraslation*)
- 6** Rotación en el eje Z de la cámara (*camRotZ*)
- 7** Rotación en el eje X de la cámara (*camRotX*)
- 8** Rotación en el eje Y de la cámara (*camRotY*)

Fase 3: Proyección (paso de 3D a 2D)

- 9** Proyección ortogonal de la cámara (*camProj*)
- 10** Paso de proyección ortogonal a perspectiva
- 11** Escalar a la ventana (*scaleView*)

Diagrama 56. Cálculos a realizar para la transformación de los vértices (3ngine).

Diez de estas operaciones concatenan una multiplicación de matrices, que a su vez es un cálculo de múltiples valores (cabe recordar que el paso 10 eran únicamente dos divisiones).

Sin embargo, esto no significa que el 3ngine sea un motor gráfico menos eficaz. Simplemente el usuario debe tener en cuenta qué necesita del motor que vaya a desarrollar. Aunque tampoco cuente con texturas, a diferencia del *engine 2D*, éste permite una renderización real de mallas con distintos tipos de renderizado, por lo que puede utilizarse como *software* de testeo para comprobar la optimización de un modelo 3D o incluso como prueba para detectar incongruencias en los vectores normales de un modelo.

Aquí radica la principal diferencia entre ambos: sacrificar optimización por realismo. La optimización se vería en el caso del *raycaster*, que es capaz de cargar escenas (que pueden llegar a ser considerablemente grandes, y más si se mejora el código a nivel de *depth values*) de múltiples niveles, además de texturas y colisiones —2 funcionalidades que no se han aplicado en el 3ngine, dado que simplemente se ha ofrecido un motor para transformar los vértices, el más básico—. Pese a la gran utilidad de las implementaciones que contiene este motor, se sigue contando con las limitaciones técnicas que se comentaron al principio de la fase de implementación del *Raycaster*:

- Todos los muros tienen la misma altura.
- Todas las “caras” de los muros enfocarán siempre a Norte, Sur, Este y Oeste (es decir, modo cuadrícula).
- La cámara no puede subir ni bajar (ni la lente inclinarse arriba o abajo).

Si el sistema que el usuario quiere desarrollar puede sobrellevar estas limitaciones o evadirlas de algún modo —por ejemplo, si el mapa es un tablero—, entonces quizás puede ser más útil

emplear este sistema de simulación 3D, que consume menos recursos y cuenta con las funcionalidades principales dictadas previamente.

6.6 Escalabilidad general

Se considerará “escalabilidad general” la **libertad** del usuario **de desarrollar cualquier tipo de software** en uno de los dos motores implementados. Este punto va de la mano con el apartado de “limitaciones técnicas”, ya que éstas constituyen la barrera principal de motor a servir como base de algún tipo de *software* específico.

Como ya se ha comentado, las tres principales barreras del *Raycaster* son el muro principal que impide que dicho sistema sea una simulación perfecta de un entorno 3D. Por lo tanto, este motor queda limitado únicamente al uso de lo que podría denominarse “juego de mapa tablero”. Los ejemplos que se dieron al principio, durante la parte teórica de este sistema daban los dos tipos de juego más ambiciosos que podrían llegar a desarrollarse mediante esta base: “Wolfenstein 3D” (un *first person shooter*, o juego de disparos en primera persona), y “Alternate Reality: The City” (un juego de rol). Aunque ha habido otros, como “The Elder Scrolls: Arena” —que también era de rol y acción—, la esencia del mapa cuadrado y el sistema de cámara es una constante en estos videojuegos. Y es que, saliendo de la familia de los videojuegos, este sistema no tiene otra utilidad.

Es decir, pese al gran potencial de compartirlo públicamente en un entorno web, y la facilidad de su implementación, es **muy poco escalable**.

Sin embargo, al considerar el motor 3D como base para un proyecto puede verse que, no sólo permite el desarrollo de un videojuego, sino que además puede utilizarse para implementar un *software* de testeo de mallas (el comentado anteriormente), o la carga de animaciones (ya que puede leer archivos de formato .OBJ), o incluso un programa que detectara los vértices e hiciera transformaciones y modificaciones de las mallas en tiempo real: un software de modelado 3D, como 3DS Max o Malla (ambos desarrollados por “Autodesk”). Además, a diferencia del caso del *raycaster*, éste no cuenta con ninguna ley o norma de limitación técnica (si bien es cierto que podría optimizarse la proyección de los objetos).

Aquí se encuentra la primera distinción: el motor gráfico **3ngine sí es escalable**.

6.7 Experimentación de gráficos y efectos visuales

En este apartado, el *raycaster* cuenta con un grado de apoyo adicional, ya que están implementadas las texturas para los muros. Es decir, de por sí ya hace uso del color, a diferencia del *3ngine*, que es enteramente en blanco y negro, aunque a la luz se le podría dar una tonalidad, así como a los muros, sin remarcada dificultad.

Ambos sistemas pueden ofrecer una especie de “filtro” de color para una mayor variedad visual, pero a nivel de efectos visuales el problema se acrecienta.

Si bien es cierto que el *raycaster* puede soportar y muestra color (con un cierto impacto en el rendimiento, obviamente, pero no ridículo), efectos visuales como un destello, o simplemente un sistema de partículas como fuego o líquido de modo convencional queda fuera de la ecuación. El usuario debería, del mismo modo que el efecto 3D es una simulación, crear también

una simulación de las partículas (ya que el 3D no está implementado, ni las operaciones de dicho tipo que pudieran ser necesarias). Además, el entorno es enteramente estático, de modo que, para animar una secuencia de partículas (que serían dibujar píxeles sobre la cámara en cierta posición del jugador), también se necesitaría un hijo de ejecución aparte, o cargar todos los sistemas de todas las partículas con la escena, y a la vez. Incluso así tiene limitaciones la experimentación de gráficos.

Es decir, el concepto es el mismo que el empleado al sombrear los muros que horizontales del mapeado: no es que haya una luz, es una simulación, y es un efecto falso para “imitar” el comportamiento de la luz y dar algo de distinción 3D a la escena (además de la proyección de muros).

Por otra parte, en el **3ngine** se cuenta con una variable de tipo luz vectorial (o “rayo” de luz), que calcula su impacto sobre los triángulos de la malla y los sombraea en función de dicho ángulo de incidencia. Este efecto, pese a ser del mismo modo una simulación, no deja de imitar el comportamiento real de la luz —no es una mera multiplicación por un factor arbitrario, como en el caso del *raycaster*— y, el usuario podría, en vez de crear un mero vector y hacer dicho cálculo, crear una clase “Luz” y asignarle un color y las propiedades necesarias para ofrecer a la escena un mayor grado de realismo mediante iluminación.

Por si fuera poco, además, al ser 3D, el sistema de partículas o efectos no queda descartado en absoluto, pudiendo imitar el comportamiento del fuego si se le aplicara un grado de tonalidad a dicho motor, además del tono blanco de los muros.

Por ello, podemos ver que (pese a no haber profundizado tanto en esta faceta del motor), el **3ngine** cuenta también con la posibilidad de un **sistema de partículas más realista**; dado que el *raycaster* debería buscar en sí mismo **otro tipo de simulación** para dicho efecto.

7 Conclusiones

En este último apartado de contenidos del trabajo se hará un análisis de los resultados obtenidos tras el desarrollo de las aplicaciones, a nivel de *software* y los motores implementados, así como en cuanto a la experiencia obtenida y posibles vías de evolución de dichos proyectos.

7.1 Resultados obtenidos y conclusiones generales

Este extenso documento ha sido una recopilación de ensayos y errores más allá de una mera investigación inicial y un desarrollo predefinido. Es cierto que no ha sido necesaria la intervención ni el ofrecimiento de pruebas a clientes o usuarios (de ello se hablará también en el próximo apartado 7.5 —líneas de continuación—) para llevar a cabo la implementación de la parte práctica, pero sí se ha ofrecido una posible solución al problema que se planteó inicialmente.

Si se permite la metáfora, éste era tal que así: echar una persona al mar en una barca, decirle que no hay peligro porque la barca flota, y darle los remos, pero no enseñarle a remar. Sin duda, hoy en día los desarrolladores cuentan con remos más que suficientes, pero no todos son marineros cuando terminan los estudios.

Gracias al desarrollo de estos dos motores gráficos mediante esta guía, completamente ilustrada y exhaustiva en las explicaciones sobre el cálculo matemático y conceptual de los procesos, cualquier interesado con mínimos conocimientos de trigonometría, matrices y programación puede ahora comprender la tecnología y los *scripts* que permiten el avance tecnológico de las aplicaciones interactivas que hoy en día inundan la red.

Desde “simular” un mero efecto 3D gracias a la magnitud de un rayo hasta concatenar operaciones matriciales y emular el funcionamiento de una cámara mediante conversión de vértices de 3D a 2D, se ha realizado un documento que aclara los pasos y su orden de ejecución para comprender cómo funciona la luz que permite ver el mundo en una dirección y a la inversa.

Además de ello, se han realizado en lenguajes de programación y entornos de desarrollo distintos, además de dos estructuras diferentes para cada proyecto (eso sí, manteniendo como eje principal la programación orientada a objetos).

Por último, para dar un enfoque más global, a continuación, se lista un orden esquemático con los resultados generales y también las conclusiones a nivel propio, como experiencia. Gracias a este trabajo-guía, se han cumplido los siguientes objetivos:

- Investigación exhaustiva pre-implementación y análisis de código y sistemas de renderizado antiguos (*Wolfenstein 3D, Doom, The Elder Scrolls: Arena, Alternate Reality: The City...*), así como motores gráficos utilizados por empresas actuales (From Software, ID Software, Unreal Engine, Unity...).
- Periodo de pruebas para testeo de motores y definición de las funcionalidades principales a implementar de éstos. Evolución del trabajo a nivel de parte práctica (decisión de realizar 2 motores gráficos, con distintos sistemas y entornos).

- Implementación de los dos motores gráficos que constituyen el marco práctico del trabajo.
- Buscar documentación sobre archivos .OBJ, y estudiar su estructura para poder abrirlos como formato para el 3ngine.
- Desarrollo de la guía a nivel visual: creación de todos los diagramas y *assets* para el documento.
- Desarrollo completo definitivo de la guía (ambas, para el *raycaster* y el 3ngine), y comparación entre ambos proyectos a nivel de rendimiento, limitaciones técnicas y escalabilidad general.
- También se ha obtenido una gran experiencia al trabajar desde cero, sumergiéndose en el mundo de los motores gráficos y sistemas antiguos, fortaleciendo los conocimientos adquiridos durante el grado, extrapolándolos a un nuevo nivel de desarrollo, y descubriendo dos modos distintos de aplicarlos mediante la creación de herramientas y la implementación de las funcionalidades de ambos.
- Salir de la caja para atreverse a variar en determinados procesos de trabajo o algoritmos a la hora de programar, innovando para obtener nuevos resultados, especialmente en el ámbito de la optimización de los *engine*.

No cabe duda que se han aprendido y aprehendido muchos métodos de investigación y desarrollo a lo largo del trabajo, y completar la guía para poder ofrecerla a los futuros estudiantes o desarrolladores sin duda contentará a todos los interesados en el mundo de los entornos virtuales interactivos, videojuegos e incluso películas de animación o emuladores de sistemas 3D antiguos.

7.2 Coste en horas del trabajo

Para organizar el coste en horas del trabajo, se han agrupado las tareas según el bloque y el contenido que ofrecen al documento. Por ello, de cada motor gráfico se han separado las líneas de investigación (el proceso teórico y de explicación de éste, su implementación, y el trabajo de optimización posterior de dicho proyecto). Además de ello, también se han indicado el tiempo dedicado a la generación de “assets” para la guía, como los diagramas propios, y, obviamente, la redacción de la memoria.

Tabla 7. Lista de horas dedicadas por tarea al trabajo.

Tarea	Horas dedicadas
Definición del proyecto	40
Investigación del <i>Raycaster</i>	40
Implementación del <i>Raycaster</i>	50
Optimización del <i>Raycaster</i>	20
Investigación del 3ngine	30
Implementación del 3ngine	200
Optimización del 3ngine	50
Producción de diagramas	80
Redacción de la memoria	140
Total	650 horas

7.3 Puntos débiles y limitaciones del trabajo

Sin duda lo que se ha ofrecido es un mero documento y dos proyectos con ciertas funcionalidades, pero éstos podrían haber optado a más. Las limitaciones técnicas de ambos proyectos, así como un tiempo insuficiente para una muestra completa de potencial en ambos, han supuesto un recorte a la hora de definir el resultado final a mediados de la investigación.

Pese a mostrar dotes de optimización suficientemente eficaces a nivel de rendimiento general, no es trivial recordar que el *raycaster* es un sistema antiguo, empleado cuando las máquinas no tenían suficiente potencia como para operar en tres dimensiones y, aunque poder desarrollar este mismo algoritmo de forma tan completa (incluso con texturas y colisiones) en un entorno mundial como es internet es una gran base para comenzar en el mundo de la programación de entornos interactivos, éste sistema sigue ligado a las tres leyes básicas que desde un principio han marcado su esencia: “mapa-cuadrícula”, muros de misma altura, cámara sin *tilt*.

Por su parte, aunque el 3ngine permite una base suficiente para seguir evolucionando como motor gráfico (de hecho, consta de las transformaciones de mallas a aplicar a nivel de vértices), no se ha implementado la funcionalidad de las texturas, a diferencia del *raycaster*. Además, el método de ordenación de triángulos es extremadamente simple y no es exacto al superponer dos superficies en la misma profundidad (no corta los triángulos entre mallas, como ocurre con el *clipping* de la cámara).

7.4 Puntos fuertes

La gran mayoría de puntos fuertes han podido ser listados en el apartado de conclusiones generales, pero, a nivel técnico, sin duda destacan los siguientes:

- De ahora en adelante, hay otro archivo útil para la documentación de los estudiantes a la hora de desarrollar videojuegos o motores gráficos, lo cual es siempre un punto positivo de cara a los interesados.
- Se ha profundizado curiosa y agradablemente en las simulaciones de los sistemas antiguos para generar una ilusión de tres dimensiones en un sistema de mapeado matricial.
- El *raycaster* está implementado en un entorno web, y cuenta con las principales funcionalidades de una base sólida para prácticamente cualquier videojuego, de modo que está al alcance de todos el poder realizar otro proyecto similar a éste y compartirlo con el mundo (de forma gratuita o no) gracias a internet.
- El 3ngine ofrece herramientas ideales para el análisis de mallas u objetos, así como los tiempos de cálculo y *framing*, entre otros, y varios sistemas de renderizado que permiten una clara visión de la estructura de archivos .OBJ.
- La guía ofrecida está completamente ilustrada y las operaciones se muestran tal y como son en los proyectos, mediante ecuaciones, capturas de pantalla de los resultados, y diagramas exactos con todas las variaciones de los distintos cálculos o comportamientos y hasta las clases programadas de cada proyecto (como el UML del 3ngine).
- La comparativa tras el marco más “práctico” del trabajo ofrece un análisis de los resultados de forma mucho más rigurosa en cuanto a las posibles utilidades de cada sistema, su rendimiento general e incluso desde el punto de vista de una posible

evolución de los sistemas en su escalabilidad general y las funcionalidades implementadas.

7.5 Líneas de continuación

Aunque las posibles vías de evolución de este proyecto podrían dividirse en dos bloques, ya que ambos motores gráficos implementados son, en concepto y parte de desarrollo, bastante distintos; se verá de forma conjunta excepto cuando en el ítem se marque dicha distinción:

- Como se ha insistido en repetidas ocasiones, el mayor muro a superar en el *raycaster* son sus limitaciones básicas y casi inquebrantables para el sistema. De modo que, según una visión bastante clara del motor, la principal línea de continuación de éste consistiría en desarrollar los métodos necesarios para que el mapeado no consistiera necesariamente de formas cuadradas o rectangulares (es decir, la estructura matricial de la escena), distintas alturas para los muros —sin que ello afecte a las texturas aplicadas— y un posible efecto de *tilt* de la cámara (que, de nuevo, también sería una mera ilusión, ya que el sistema es claramente bidimensional, no 3D). El 3ngine no sufre de estas limitaciones técnicas, pero sí podría utilizar una nueva base de optimización.
- Según el proyecto que el interesado quisiera llevar a la práctica, simplemente sería necesaria la implementación de la IA necesaria, en forma de scripts de JavaScript o clase de Java, y la lógica del videojuego a incluir en el proyecto.
- También hay varios métodos de optimización para ambos proyectos, como un posible efecto de *postprocessing* para la proyección de los muros sin la necesidad de emitir tantos rayos por parte de la cámara en el *raycaster* (hacer una media de la magnitud de los rayos, por ejemplo, o suavizar los bordes). En el caso del 3ngine, la detección automática de qué transformaciones son necesarias y cuáles no para cada *game object* ahorraría al sistema una gran carga de cálculo y parte del método de ordenación de los triángulos, que es el que consume más recursos en el rendimiento.
- Por último, y a nivel más estético, (además de aplicar texturas al 3ngine) también se puede aplicar un filtro de color o efectos visuales parecidos a las partículas para dar una mayor sensación de realismo a los sistemas e incluso suavizar dichas texturas cuando la cámara está particularmente lejos de la superficie indicada (además de un Z-buffer para gestionar la profundidad de los elementos del mapeado).

8 Índices de referencias

8.1 Referencias a tablas

Tabla 1. Videojuegos más exitosos de 2020, y sus motores.	17
Tabla 2. Videojuegos varios por motor y género.	17
Tabla 3. Motores principales de las desarrolladoras más exitosas.	18
Tabla 4. Lista de operaciones de cada motor a implementar.	20
Tabla 5. Proceso de corrección de ángulos (por columna), función de <i>utils.js</i>	42
Tabla 6. Controles de teclado del 3ngine.	105
Tabla 7. Lista de horas dedicadas por tarea al trabajo.	138

8.2 Referencias a ecuaciones

Ecuación 1. Conversión de grados sexagesimales a radianes.	41
Ecuación 2. Media aritmética de las coordenadas X de los tres vectores de un triángulo.	110
Ecuación 3. Producto escalar entre dos vectores normalizados.	115
Ecuación 4. Cálculo de la intersección de un vector en el plano de la cámara.	116
Ecuación 5. Relación entre el vector del punto B a P y el vector de B a A.	116
Ecuación 6. Sistema de ecuaciones a resolver para la colisión de un vector en un plano.	116
Ecuación 7. Cálculo del factor <i>t</i> de escalado de vértices entre el vector BP y BA.	116
Ecuación 8. Producto escalar entre el vector L (luz) y el vector VN (normal del triángulo).	119
Ecuación 9. Cálculo de los fps del sistema en función del elapsed time (tiempo de <i>frame</i>).	132

8.3 Referencias a diagramas

Diagrama 1. Proceso de creación de nivel desde una matriz numérica.	30
Diagrama 2. Documentos invariantes del proyecto.	33
Diagrama 3. Esquema con los ficheros del proyecto, sus clases y funciones.	34
Diagrama 4. Estructura de ficheros del proyecto con ícono y formato.	35
Diagrama 5. Cálculo del tamaño del <i>canvas</i> según la constante <i>TILE_SIZE</i>	38
Diagrama 6. Generación del array de Tiles a partir de la <i>grid</i>	39
Diagrama 7. Aspecto del Player con el ángulo hacia el que mira y su ángulo de renderizado.	43
Diagrama 8. Ángulo de visión del jugador (en radianes), con su orientación y tres ejemplos.	43
Diagrama 9. Cálculo trigonométrico del ángulo de visión del jugador.	44
Diagrama 10. Distinción entre colisiones en "X" e "Y" o ambas.	47
Diagrama 11. Obtener filas y columnas de la <i>grid</i> desde la posición en píxeles del <i>canvas</i>	48
Diagrama 12. Proceso ordenado de la detección de colisiones horizontales del <i>raycaster</i>	53
Diagrama 13. Proceso ordenado de la detección de colisiones verticales del <i>raycaster</i>	56
Diagrama 14. Distribución de los archivos del proyecto (ahora con imágenes).	63
Diagrama 15. Proceso de lectura de imágenes, en 5 pasos y ejemplos de nomenclatura.	64
Diagrama 16. Proceso para cargar una escena multinivel mediante imágenes.	66
Diagrama 17. Conversión a matriz numérica desde un mapa a color (en escena multinivel).	67
Diagrama 18. Representación de las texturas a aplicar en función del muro de la <i>grid</i>	69

Diagrama 19. Obtención de columna de píxeles de la textura.....	69
Diagrama 20. Descripción visual de los atributos de la clase <i>VLine</i>	70
Diagrama 21. Distinción de variables para el cálculo del <i>Texel</i> de la colisión (paso 1).....	71
Diagrama 22. Proceso de mapeado del <i>Texel</i> mediante la función <i>map</i> (librería <i>P5.js</i>)	72
Diagrama 23. Diagrama UML, package <i>ModelGeometry</i>	80
Diagrama 24. Diagrama UML, package <i>Model</i>	81
Diagrama 25. Diagrama UML, package <i>Controller</i>	82
Diagrama 26. Diagrama UML, package <i>View</i> y <i>Utils</i>	83
Diagrama 27. Diagrama relacional del 3ngine.	84
Diagrama 28. Modelo MVC básico inicial para el 3ngine.....	87
Diagrama 29. Loop principal del 3ngine, agrupado por clases y llamadas a funciones.....	88
Diagrama 30. Estructura básica de una escena del EngineController.	92
Diagrama 31. Estructura del cubo (vértices, coordenadas y ejes).....	93
Diagrama 32. Triángulos orientados del cubo, para definir sus vectores normales.....	94
Diagrama 33. Creación de triángulos de una cara del cubo de testeo.	94
Diagrama 34. Transformaciones a aplicar a los vértices de cada objeto de la escena.	95
Diagrama 35. Primer paso de la proyección de vértices de los objetos de la escena (paso 9)...	96
Diagrama 36. Pasos 2 y 3 de la proyección de los vértices (10 y 11 del proceso completo).	97
Diagrama 37. Matrices de rotación para los vértices del objeto.	99
Diagrama 38. Matriz de traslación para el desplazamiento de vértices.	100
Diagrama 39. Ejemplo de traslación de los vértices de un cubo.	100
Diagrama 40. Cálculo de la matriz genérica para la transformación de los vértices.	101
Diagrama 41. Multiplicación de los vértices por la matriz genérica de transformación.	101
Diagrama 42. Matrices de transformación de los vértices respecto a la cámara.	103
Diagrama 43. Cálculo de la matriz genérica para la transformación de la cámara.....	104
Diagrama 44. Proceso de cálculo del vector de dirección de la cámara (<i>vDir</i>).	106
Diagrama 45. Desplazamiento de la cámara a lo largo del eje X.	107
Diagrama 46. Cálculo del vector “ <i>vRight</i> ”, para <i>CameraModel</i>	107
Diagrama 47. Método de ordenación de los triángulos por selección.	112
Diagrama 48. Procedimiento de “triangle clipping”.	114
Diagrama 49. Representación de un vector que atraviesa el plano de la cámara.....	115
Diagrama 50. Distintos casos que pueden darse en el <i>triangle clipping</i>	117
Diagrama 51. Formando un <i>quad</i> a partir de dos triángulos.	118
Diagrama 52. Representación gráfica del rayo de luz incidiendo sobre un triángulo.	119
Diagrama 53. Cálculo del vector normal de los triángulos en función de dos de los lados.....	119
Diagrama 54. Distinción de los triángulos que enfocan a la cámara.	123
Diagrama 55. Efecto al activar <i>checkIfFacingCamera</i>	123
Diagrama 56. Cálculos a realizar para la transformación de los vértices (3ngine).	134

8.4 Referencias a figuras

Figura 1. Representación gráfica del procedimiento de renderizado por <i>raycasting</i>	23
Figura 2. Tipos de control que se pueden implementar en un shooter.....	24
Figura 3. Distinción entre pad direccional (o “cruceta”), y pads analógicos.....	25
Figura 4. Captura de pantalla de Alternate Reality: The City (desde emulador).	26

Figura 5. Captura de pantalla de Wolf3D	27
Figura 6. Falso efecto de iluminación en el renderizado de <i>vLines</i>	27
Figura 7. Falso degradado de iluminación en los muros proyectados.....	28
Figura 8. Versión final del raycaster a implementar en este proyecto, nivel 1.....	29
Figura 9. <i>Raycaster</i> con lectura de varios niveles y colisión de rayos implementada.	30
Figura 10. <i>Raycaster</i> con la proyección de muros terminada.....	31
Figura 11. Resultado final de la fase 3 de la implementación del <i>raycaster</i>	32
Figura 12. 4 códigos de color para el entorno del <i>raycaster</i> (tonalidades grises).	36
Figura 13. Resultado de la generación de un nivel mediante matriz numérica.....	40
Figura 14. Incremento del eje de abscisas y de ordenadas en un canvas de JavaScript.....	43
Figura 15. Renderizado del <i>canvas</i> tras implementar la clase <i>Player</i> y <i>FOV</i>	45
Figura 16. Clase <i>Player</i> con controles de movimiento y giro de cámara.	47
Figura 17. Representación comparativa del jugador con varios FOVs.....	49
Figura 18. Captura del <i>raycasting</i> implementado.....	50
Figura 19. Resultado del <i>raycaster</i> sin colisiones.....	51
Figura 20. Ilustración de un posible renderizado de la escena.....	52
Figura 21. Aspecto del <i>raycaster</i> con la colisión de rayos sólo en "X".....	54
Figura 22. Posible renderizado de la escena (prioridad de colisiones en "Y").....	55
Figura 23. Aspecto del <i>raycaster</i> con la colisión de rayos sólo en "Y".....	57
Figura 24. Implementación de las colisiones del <i>raycaster</i> en "X" e "Y".	58
Figura 25. Comparación de dos capturas de la misma escena.	59
Figura 26. Resultado de la proyección de muros y mini mapa.	62
Figura 27. Resultado de la generación de niveles por imagen.....	68
Figura 28. Resultado de ejemplo de texturizar los muros proyectados.	73
Figura 29. Resultado final de la implementación del <i>raycaster engine</i>	75
Figura 30. Renderizado en modo <i>wireframe</i> de un cubo de testeo.	79
Figura 31. Muestra de ventana con figuras sencillas, utilizando java.awt.Graphics.	90
Figura 32. Resultado de la proyección del cubo..	98
Figura 33. Resultado del <i>engine</i> tras añadir rotación y desplazamiento inicial al cubo.	101
Figura 34. Representación visual del efecto falso de movimiento de cámara.	102
Figura 35. Resultado de aplicar transformaciones de cámara al cubo de testeo.	104
Figura 36. Captura de pantalla del renderizado de triángulos del cubo (preordenación).....	110
Figura 37. Render del cubo de testeo con los triángulos ordenados.....	113
Figura 38. Captura del error de proyección usando el cubo de testeo.....	113
Figura 39. Captura del cubo de testeo con el <i>clipping</i> de los triángulos implementado.....	118
Figura 40. Renderizado del cubo con la luz direccional de la escena (0, -0.5, -1).....	120
Figura 41. Estructura de un <i>.obj</i> de ejemplo.	121
Figura 42. Captura resultante de la lectura del triángulo de ejemplo.	122
Figura 43. Renderizado de la Tetera de Utah, en modo <i>full render</i> (captura propia).....	124
Figura 44. Renderizado de la Tetera de Utah, en modo <i>surface</i> (captura propia).	125
Figura 45. Renderizado de la Tetera de Utah, en modo <i>wireframe</i> (captura propia).	125
Figura 46. Propiedades del sistema (captura del menú de "Configuración" de Windows)....	127
Figura 47. Tarjeta gráfica (captura del "Administrador de Dispositivos" de Windows).	127
Figura 48. Captura de información de Google Chrome.	128
Figura 49. Renderizado del <i>Raycaster</i> para la muestra de parámetro.	129

Figura 50. Datos de la matriz de proyección del 3ngine.	130
Figura 51. Resultado final del renderizado en el 3ngine.....	130
Figura 52. Captura del <i>Raycaster</i> (prueba pesada sin límite de <i>frame rate</i>).	131
Figura 53. Captura del 3ngine (prueba pesada, límite de <i>frame-rate</i> : 500 fps).	133

9 Bibliografía

- [1] A-Frame. (7 de 2 de 2021). *Raycaster*. Recuperado el 5 de 2021, de A-Frame: <https://aframe.io/docs/1.2.0/components/raycaster.html>
- [2] Anónimo (187.160.160.57), & Miguel4523. (21 de 7 de 2021). *DmC: Devil May Cry*. Recuperado el 3 de 2021, de Wikipedia: https://es.wikipedia.org/wiki/DmC:_Devil_May_Cry
- [3] Anónimo (189.129.0.87), ShaThricks, A. C., & Rosario, C. A. (23 de 7 de 2021). *Asphalt 9: Legends*. Recuperado el 4 de 2021, de Wikipedia: https://es.wikipedia.org/wiki/Asphalt_9:_Legends
- [4] Anónimo (62.43.32.133), Danfc88, Elitecombine666, ZebaX 2010, & Anónimo (189.216.220.239). (23 de 8 de 2021). *Dark Souls*. Recuperado el 4 de 2021, de Wikipedia: https://es.wikipedia.org/wiki/Dark_Souls
- [5] Asioasa, & Overviews. (8 de 5 de 2021). *Doom Eternal*. Recuperado el 3 de 2021, de Wikipedia: https://es.wikipedia.org/wiki/Doom_Eternal
- [6] Baratheon King, Overviews, & Igg-foxs. (1 de 7 de 2021). *Tekken 7*. Recuperado el 3 de 2021, de Wikipedia: https://es.wikipedia.org/wiki/Tekken_7
- [7] Barreto Cegarra, J., & Overviews. (13 de 6 de 2021). *Devil May Cry 5*. Recuperado el 3 de 2021, de Wikipedia: https://es.wikipedia.org/wiki/Devil_May_Cry_5
- [8] campusMVP. (8 de 2 de 2019). *Los 5 lenguajes de programación más fáciles de aprender*. Recuperado el 4 de 2021, de Campus MVP: <https://www.campusmvp.es/recursos/post/los-5-lenguajes-de-programacion-mas-faciles-de-aprender.aspx>
- [9] Cerón Soria, M. (7 de 4 de 2020). *Lenguajes de programación más demandados en el 2020*. Recuperado el 4 de 2021, de Hack A Boss: <https://www.hackaboss.com/blog/lenguajes-programacion-demandados-2020>
- [10] Darklex, & CarlitosNN. (26 de 8 de 2021). *Fall Guys: Ultimate Knockout*. Recuperado el 3 de 2021, de Wikipedia: https://es.wikipedia.org/wiki/Fall_Guys:_Ultimate_Knockout
- [11] Darklex, Alejan98, & Dimartz. (26 de 8 de 2021). *Among Us (videojuego)*. Recuperado el 3 de 2021, de Wikipedia: https://es.wikipedia.org/wiki/Among_Us
- [12] Eliamsweet, & Bustamente, J. (26 de 8 de 2021). *Call of Duty: Warzone (videojuego)*. Recuperado el 3 de 2021, de Wikipedia: https://es.wikipedia.org/wiki/Call_of_Duty:_Warzone
- [13] Frankie MB. (9 de 12 de 2020). *Los 19 juegos que lo han petado en 2020*. Recuperado el 3 de 2021, de Vida Extra: <https://www.vidaextra.com/listas/juegos-que-han-petado-2020>
- [14] Herrero, P. (26 de 12 de 2020). *Las cifras de los videojuegos en 2020*. Recuperado el 3 de 2021, de MeriStation: https://as.com/meristation/2020/12/26/noticias/1608992024_963325.html
- [15] Ivanretro, ZebaX2010, Athos777, & Overviews. (29 de 8 de 2021). *Cyberpunk 2077 (videojuego)*. Recuperado el 3 de 2021, de Wikipedia: https://es.wikipedia.org/wiki/Cyberpunk_2077
- [16] JalenFolf, & ChickenHatMan. (21 de 7 de 2021). *Grip: Combat Racing*. Recuperado el 4 de 2021, de Wikipedia: https://en.wikipedia.org/wiki/Grip:_Combat_Racing
- [17] Johnson, L. (15 de 5 de 2016). *1993's 'Doom' Wasn't the 3D Game We Think It Was*. Recuperado el 4 de 2021, de Vice: <https://www.vice.com/en/article/4xaagg/doom-wasnt-3d>

- [18] Kanfor. (21 de 9 de 2011). *Doom, las 3D que eran 2D*. Recuperado el 4 de 2021, de Pixfans: <https://www.pixfans.com/doom-las-3d-que-eran-2d/>
- [19] Kimaster1989. (12 de 8 de 2021). *Assassin's Creed: Valhalla*. Recuperado el 3 de 2021, de Wikipedia: https://es.wikipedia.org/wiki/Assassin%27s_Creed:_Valhalla
- [20] Manuelmb97, & Tiberioclaudio99. (12 de 12 de 2020). *Tetera de Utah*. Recuperado el 7 de 2021, de Wikipedia: https://es.wikipedia.org/wiki/Tetera_de_Utah
- [21] Marcello, J., Kalinine, Gicombat, Borkowski, B., & Josif12. (19 de 8 de 2021). *Game Engine*. Recuperado el 4 de 2021, de Call of Duty Wiki: https://callofduty.fandom.com/wiki/Game_Engine
- [22] Mechemik, Anónimo (122.3.5.17), & Evilquoll. (7 de 8 de 2019). *Doom rendering engine*. Recuperado el 4 de 2021, de Doom Fandom: https://doom.fandom.com/wiki/Doom_rendering_engine
- [23] Osh33m, Redsparta, & FlyingKangeroo. (28 de 8 de 2021). *Demon's Souls (2020 video game)*. Obtenido de Wikipedia: [https://en.wikipedia.org/wiki/Demon%27s_Souls_\(2020_video_game\)](https://en.wikipedia.org/wiki/Demon%27s_Souls_(2020_video_game))
- [24] Overviews, Cheposo, Valeriano Julca, K. A., & Geom. (21 de 8 de 2021). *Cuphead*. Recuperado el 4 de 2021, de Wikipedia: <https://es.wikipedia.org/wiki/Cuphead>
- [25] Overviews, Kirangel, & Zettasaurio. (24 de 8 de 2021). *Demon's Souls*. Recuperado el 3 de 2021, de Wikipedia: https://es.wikipedia.org/wiki/Demon%27s_Souls
- [26] p5.js. (13 de 1 de 2017). Recuperado el 4 de 2021, de P5.js: <https://p5js.org/es/>
- [27] Sirgermanl, Oegerman, & Josec92. (15 de 3 de 2021). *Crysis*. Recuperado el 4 de 2021, de Wikipedia: <https://es.wikipedia.org/wiki/Crysis>
- [28] Steam. (2 de 5 de 2010). *Wolfenstein 3D*. Recuperado el 3 de 2021, de Steam (store): https://store.steampowered.com/app/2270/Wolfenstein_3D/?l=spanish
- [29] Uncle Ape, G. B. (12 de 8 de 2017). *Discussion: The ultimate DOOM*. Recuperado el 4 de 2021, de Steam (community): <https://steamcommunity.com/app/2280/discussions/0/1471967615877502834/>
- [30] Zardoz84, & nsxwolf. (3 de 6 de 2015). *Doom's engine user discussion*. Recuperado el 4 de 2021, de Hacker News: <https://news.ycombinator.com/item?id=9652869>