Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Centro de Computación Gráfica

Un motor de videojuego basado en WebGL

Trabajo Especial de Grado presentado ante la Ilustre
Universidad Central de Venezuela por el
Br. Andrés Alvarez Sordo
Br. Carlos Leopoldo Zapata Conforto
para optar al título de Licenciado en Computación

Tutor
Prof. Esmitt Ramírez

Caracas, Marzo del 2016

Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Centro de Computación Gráfica

ACTA DEL VEREDICTO

Quienes suscriben, Miembros del Jurado designado por el Consejo de la Escuela de Computación para examinar el Trabajo Especial de Grado, presentado por los Bachilleres Andrés Alvarez Sordo de C.I.: 20.185.976 y Carlos Leopoldo Zapata Conforto de C.I.: 20.654.154, con el título *Un motor de videojuego basado en WebGL*, a los fines de cumplir con el requisito legal para optar al título de Licenciado en Computación, dejan constancia de lo siguiente:

Leído el trabajo por cada uno de los Miembros del Jurado, se fijó el día 2 de Marzo de 2016, a las 03:00 PM, para que sus autores lo defendieran en forma pública, en el *Centro de Computación Gráfica*, lo cual se realizó mediante una exposición oral de su contenido, y luego respondieron satisfactoriamente a las preguntas que les fueron formuladas por el Jurado, todo ello conforme a lo dispuesto en la Ley de Universidades y demás normativas vigentes de la Universidad Central de Venezuela. Finalizada la defensa pública del Trabajo Especial de Grado, el jurado decidió aprobarlo.

En fe de lo cual se levanta la presente acta, en Caracas a los 2 días del mes de Marzo del año dos mil dieciséis, dejándose también constancia de que actuó como Coordinador del Jurado el Profesor Esmitt Ramírez.

_____
Prof. Esmitt Ramírez
(Tutor)

_____ _____
Prof. Rhadamés Carmona
(Jurado Principal)

Profa. Mercy Ospina
(Jurado Principal)

Resumen

En la actualidad, existen diversos motores de videojuegos que se basan o emplean WebGL con el fin de facilitar un conjunto de funcionalidades para el desarrollo de videojuegos web. Sin embargo, muchos de estos poseen una licencia privada y su curva de aprendizaje suele ser elevada por la misma cantidad de funcionalidades que proveen.

Por estas razones, el presente documento expone la investigación y el desarrollo de un motor de videojuego basado en WebGL de licencia libre, con una arquitectura y una documentación sencillas para tratar de reducir la curva de aprendizaje, que contempla las funcionalidades básicas de despliegue gráfico, manejo de eventos y de recursos.

Palabras clave: Motor de videojuego, WebGL, despliegue gráfico en tiempo real, navegación de escenas, iluminación, texturizado, sombras.

Agradecimientos

A Dios, por guiar nuestros pasos y por habernos brindado las capacidades que nos han permitido resolver los problemas que se han presentado a lo largo de nuestras vidas.

A la Universidad Central de Venezuela, a la Escuela de Computación y a todos y cada uno de los docentes y preparadores que contribuyeron en nuestra formación académica. En especial, a los docentes del Centro de Computación Gráfica Héctor Navarro, Walter Hernández, Francisco Moreno, Francisco Sans, Rhadamés Carmona y nuestro tutor Esmitt Ramírez, por brindarnos su apoyo y estar al tanto de este trabajo.

A nuestros queridos amigos de la universidad Alberto Cavadia, Amaro Duarte, Annerys Rivas, Elohina Guevara, Eric Bellet, Israel Rodríguez, Juan Gomez, Jesús Gomez, Juan Sanchez, Kimberly Mendoza, Pedro Valdivieso, Sebastian Ziegler, Viviana Gonzalez e Ysidro Alba por haber formado parte de nuestra formación y compartir con nosotros esta importante etapa de nuestras vidas.

Por parte de Andrés:

A mi abuela María del Pilar Sordo, por haberme cuidado con tanto amor y esfuerzo, por ayudarme a enfrentar cualquier adversidad que se me presentara sin importar lo compleja que fuera, por enseñarme a nunca rendirme y contribuir en mi formación con la mayor dedicación y cariño que me pudo haber brindado, sin ella no seria la persona que soy hoy en día.

A mis padres Francisco Alvarez y Nora Sordo por apoyarme en todo momento, acompañándome en las buenas y las malas de forma incondicional, por los valores que me han inculcado y haberme dado la oportunidad de tener una excelente educación. Sobre todo por ser unos magníficos ejemplos de vida a seguir.

A mi hermana Victoria Alvarez y a mi novia Valery Quintal por regalarme tantas sonrisas y momentos de alegría, entenderme cuando nadie más podía y ser mi principal voz de aliento en la realización de este trabajo, las amo.

A mi tía Ana Alvarez, a mi abuela Ana Martinez y a mi bisabuela María Rodrigo, por ayudarme en todo lo que podían y estar pendientes de mi a pesar de la distancia

que actualmente nos separa.

A mi amigo de la infancia Guillermo Andrade, por apoyarme siempre y ser mi compañero incondicional tanto en la vida real como en los videojuegos.

A mi amigo de la universidad Carlos Zapata, uno de mis grandes ejemplos de perseverancia, por contribuir en mi formación y haber aceptado emprender este último reto juntos.

A mis amichis Anais Pico, Aury Afonso, Karen Moncada, Nesmary Mendoza y Verónica Rodrigues por estar atentas de mi y de este trabajo.

Por parte de Carlos:

A mis padres Ana María Conforto y Nicola Loseto, por siempre cuidar de mi, apoyarme incondicionalmente en todo y ser mi ejemplo a seguir en la vida. También gracias por haberme inculcado los valiosos principios del estudio, la familia y sobre todo el amor como fuerza que todo lo puede.

A mi hermano Ricardo (Ricky) por ser mi ejemplo de lucha y perseverancia desde que tengo la capacidad de recordar. Gracias por siempre cuidarme, protegerme y enseñarme que nunca existe un muro lo suficientemente grande que no pueda ser atravesado trabajando arduamente.

A mi padre en el cielo José, gracias por siempre cuidarme y guiarme desde que tenía seis años, has sido un pilar de motivación y lucha en mi vida.

A mi nonna Catherina Guastaferro por siempre brindarme tu apoyo e inculcarme que en la unión familiar yace la fuerza así como de estar pendiente de mis avances constantemente, grazie nonna, ti voglio.

A Giovanni Figueroa y Carlos Gonzalez por ser los dos profesores que me enseñaron el cómo debía utilizar mis capacidades al entrar a la universidad. Gracias a ambos por siempre ponerme los retos más exigentes y siempre estar pendientes de mi aún después de haber terminado de ver clases con ustedes, gracias por todo mis amigos y maestros geómetras.

A Andrés Álvarez por todo el apoyo que me brindaste durante estos años de estudio, te doy las gracias por no solo por haber sido mi compañero durante estos años de universidad sino también por ser mi amigo, estar pendiente de mi, permitirme aprender muchas cosas de ti y por formar parte de esta última etapa del pregrado en la universidad.

A mi sis Bárbara La Grave, por todo el apoyo incondicional que me has brindado desde los 6 años de edad. Gracias por todos los momentos que he tenido la dicha de compartir contigo que me han enseñado a que también es importante la dispersión al momento de quedarme atascado en mi trabajo, I love you.

A mi compinche Manuela Arellano, por siempre ofrecerme tu hermosa amistad y estar presente en los buenos y malos momentos por los que he atravesado antes y después de la realización de este trabajo, así como por brindarme tu apoyo incondicional en todo, gracias por existir, te quiero mucho.

A mis partners del curso de inglés de la Universidad Central de Venezuela, Alessandra Bottini, Anderson Guzmán, Ariadna Lira, Claudia Sánchez, Daniela Linares, Estefanía Fontana, Heiker Guzmán, Jorge Méndez y Luis Chacón por estar presentes durante mis buenos y malos momentos, estar pendientes de mis avances, ayudarme a dispersarme y aconsejarme, I love you all my crew.

A todas las personas que de una u otra manera influyeron en la realización de este trabajo, gracias por todo su apoyo brindado.

Índice general

Resumen I

Agradecimientos II

Introducción XI

1. Problema de investigación **1**
1.1. Planteamiento del problema 1
1.2. Objetivos 2
1.2.1. Objetivo general 2
1.2.2. Objetivos específicos 2
1.3. Antecedentes 3
1.4. Alcance 4

2. Marco teórico **5**
2.1. Videojuego 5
2.2. Motor de videojuego 6
2.2.1. Géneros de videojuegos 8
2.2.2. Arquitectura general 8
2.2.2.1. Hardware, controladores de dispositivo y sistema operativo 9
2.2.2.2. SDKs y *middlewares* 10
2.2.2.3. Capa independiente de la plataforma 11
2.2.2.4. Subsistemas principales 11
2.2.2.5. Gestor de recursos 11
2.2.2.6. Motor de despliegue 12
2.2.2.7. Herramientas de depuración 13
2.2.2.8. Motor de física 14
2.2.2.9. Interfaces de usuario 14
2.2.2.10. Redes y multijugador 15
2.2.2.11. Subsistema de juego 15
2.2.2.12. Audio 15
2.2.2.13. Subsistemas específicos de juego 15

2.3. WebGL 16
2.3.1. Orígenes 16
2.3.2. Estructura de una aplicación 17

3. Solución propuesta **19**
3.1. Arquitectura 19
3.2. Técnica de despliegue 20
3.3. Iluminación y sombreado 21
3.3.1. Fuentes de luz 21
3.3.1.1. Luz puntual 21
3.3.1.2. Luz direccional 22
3.3.1.3. Luz reflector 23
3.3.2. Efectos creados por la interacción con la luz 23
3.3.2.1. Luz directa 24
3.3.2.2. Transparencia 24
3.3.2.3. Sombra 25
3.3.2.4. Refracción y reflexión 25
3.3.3. Tipos de superficies 25
3.3.4. Modelos para la reflexión difusa 26
3.3.4.1. Reflexión de Lambert 26
3.3.4.2. Reflexión de Oren-Nayar 26
3.3.5. Modelos para la reflexión especular 27
3.3.5.1. Reflexión de Phong 27
3.3.5.2. Reflexión de Blinn-Phong 27
3.3.5.3. Reflexión de Cook-Torrance 28
3.3.6. Técnicas de interpolación 28
3.3.6.1. Sombreado planar 28
3.3.6.2. Sombreado de Phong 29
3.4. Texturizado 29
3.4.1. Texturas 2D 30
3.4.2. Mapeado de ambiente 30
3.4.2.1. Efecto de reflexión 31
3.4.2.2. Efecto de refracción 31
3.4.3. Mapeado de normales 32
3.5. Sombras 34
3.5.1. Mapeado de sombras 34
3.6. Selección y descarte de geometría 36
3.6.1. Descarte de caras traseras 36
3.6.2. Descarte por pirámide de visualización 36
3.7. Grafo de escena 37

4. Implementación **39**
4.1. EZ3.js 39

4.2. Estilo de programación y documentación 40
4.3. Compilación 41
4.4. Metodología 44
4.5. Módulos 45
4.6. Sistema 45
4.7. Matemáticas 47
4.8. Entidad y escena 48
4.9. Cámaras 50
4.10. Texturas 51
4.11. *Framebuffers* 51
4.12. Luces 52
4.13. Geometría 53
4.14. Materiales 54
4.15. Malla 56
4.16. Entrada 57
4.17. Controles 58
4.18. Carga 58
4.19. Motor y manejador de pantallas 60
4.20. Despliegue 62

5. Pruebas y resultados **67**
5.1. Estudio del rendimiento 70
5.2. Estudio de memoria 72
5.3. Estudio de la cantidad de líneas de código 74

6. Conclusiones **75**
6.1. Trabajos futuros 76

Bibliografía **77**

Índice de figuras

2.1. Carátula del videojuego Doom. 7
2.2. Arquitectura general de un motor de videojuego [1]. 9
2.3. Arquitectura multicapa de un motor de despliegue [1]. 12
2.4. Relación entre OpenGL, OpenGL ES, WebGL y Vulkan. 17
2.5. Comparación de estructuras de aplicaciones web [2]. 18

3.1. Arquitectura propuesta. 20
3.2. Fuente de luz puntual. 22
3.3. Fuente de luz direccional. 22
3.4. Fuente de luz reflector. 23
3.5. Efecto de luz directa sobre una esfera. 24
3.6. Efecto de transparencia sobre un cubo. 24
3.7. Tipos de superficies [3]. 25
3.8. Comparación entre la reflexión difusa de Lambert y Oren-Nayar. . . . 27
3.9. Comparación entre la reflexión especular de Phong, Blinn-Phong y Cook-Torrance. 28
3.10. Comparación entre el sombreado planar y el sombreado de Phong. . . 29
3.11. Una textura 2D aplicada a un toro. 30
3.12. Representación de un *cubemap* [4]. 31
3.13. Comparación entre los efectos de reflexión y refracción. 32
3.14. Ejemplos de un mapa de normales y el espacio tangente. 33
3.15. Efecto generado por el mapeado de normales. 33
3.16. Primera pasada de la técnica de mapeado de sombras. 35
3.17. Principales problemas del mapeado de sombras. 35
3.18. Ejemplo de descarte por pirámide de visualización [5]. 37
3.19. Ejemplo de un grafo de escena. 38

4.1. Ciclo de desarrollo de EZ3.js. 44

5.1. Escena sencilla. 68
5.2. Palacio Sponza como escena intermedia. 68
5.3. Catedral de Sibenik como escena compleja. 69

5.4. Resultados obtenidos del estudio del rendimiento para el navegador Google Chrome. 70
5.5. Resultados obtenidos del estudio del rendimiento para el navegador Mozilla Firefox. 71
5.6. Resultados obtenidos del estudio del rendimiento para el navegador Microsoft Edge. 71
5.7. Resultados obtenidos del estudio de memoria para el navegador Google Chrome. 72
5.8. Resultados obtenidos del estudio de memoria para el navegador Mozilla Firefox. 73
5.9. Resultados obtenidos del estudio de memoria para el navegador Microsoft Edge. 73
5.10. Cantidad de líneas de código para cada escena. 74

Índice de códigos

1. Fracción del contenido del archivo .jshintrc.json de EZ3.js. 41
2. Fracción del contenido del archivo package.json de EZ3.js. 43
3. Función *checkAnimationFrame*. 46
4. Método *updateWorld* de la clase *Entity*. 49
5. Método *updateWorldTraverse* de la clase *Entity*. 50
6. Acumulación de las definiciones asociadas a cada bloque de instrucciones en una cadena de caracteres. 55
7. Bloques de instrucciones en el código fuente del sombreador de fragmentos para el cálculo de la reflexión difusa. 56
8. Método *add* de la clase *ScreenManager*. 62
9. Clasificación de las entidades según su naturaleza y actualización de la cámara. 63
10. Verificaciones y subclasificación sobre las mallas. 65
11. Ordenamiento por profundidad de las mallas transparentes. 65
12. Invocaciones a funciones para calcular los mapas de sombras y ejecutar el despliegue de las mallas. 65
13. Método *render* de la clase *Mesh*. 66

Introducción

Con el surgimiento de WebGL los navegadores web han sido capaces de emplear gráficos 3D acelerados por hardware, abriendo un rango de posibilidades para la creación de aplicaciones incluyendo videojuegos. Sin embargo, el proceso de elaboración de éstas comenzando desde cero, comúnmente resulta ser bastante complejo y tedioso, debido al conjunto de factores a considerar. En la actualidad, existe una gran variedad de aplicaciones y bibliotecas, conocidas como motores de videojuegos o *game engines* que se encargan de lidiar con dichos factores mediante diversas funcionalidades, disminuyendo la complejidad del proceso de elaboración.

Las funcionalidades presentes en los motores de videojuegos varían dependiendo de su propósito. No obstante, las funcionalidades típicas provistas por éstos, suelen ser despliegue gráfico, simulación de efectos físicos, manejo de eventos, de recursos y redes, entre otras.

Muchos de estos motores se encuentran en el mercado comercial y generalmente son de difícil acceso a toda la comunidad de desarrolladores y usuarios la cual es bastante extensa, por lo que encontrar ayuda por parte de esta es muy común. Adicionalmente, la documentación de la mayoría de los motores es extensa y completa. No obstante, aun así, la curva de aprendizaje suele ser alta por la gran cantidad de contenido y su complejidad.

Es por ello que el propósito general de esta investigación es abordar el diseño y la implementación de un motor de videojuego basado en WebGL de código abierto enfocado en poseer una baja curva de aprendizaje. Para ello, primero se plantea el problema de investigación, sus objetivos, los antecedes y el alcance de la misma. Consecuentemente, se describen un conjunto de conocimientos relacionados a los términos y técnicas empleadas. Luego, se expone la implementación hecha exhibiendo cada componente, como también, las tecnologías y convenciones empleadas. Posteriormente, se analizan las pruebas y resultados obtenidos, para así, establecer puntos de comparación con algunos de los motores de videojuego que empleen WebGL previamente existentes. Finalmente, se exponen las conclusiones de la presente investigación y propuestas de trabajos futuros.

1

Problema de investigación

1.1. Planteamiento del problema

Actualmente, existen diversos motores de videojuegos que se diferencian por las funcionalidades que proveen y sus mismas características. La comunidad que los emplea suele variar entre personas particulares, empresas pequeñas, medianas y grandes.

Los motores poseen una licencia para ser usados, ésta puede ser tanto libre como propietaria. Las licencias propietarias requieren de un pago, ya sea mensual, anual, por uso o por descarga, presentando una limitante. Por otro lado, las licencias libres son gratuitas y usualmente exponen el código fuente del motor; pero aún así, varias obligan al usuario a colocar en la aplicación final algún tipo de referencia hacia el motor usado o hacia sus creadores.

El hardware requerido para poder ejecutar los motores de videojuegos presenta otra limitante en el uso de los mismos, ya que gran parte de estos requieren un conjunto de especificaciones de hardware bastante exigentes.

La documentación de la mayoría de los motores es extensa y completa. No obstante, la curva de aprendizaje suele ser alta por la gran cantidad de contenido y la complejidad que estos usualmente tienen.

El rango de plataformas para las cuales se puede crear un videojuego varia dependiendo del propósito del motor. Muchos de estos se limitan a un grupo de pla-

taformas, mientras que otros no lo hacen o solo a las plataformas más relevantes (multiplataforma). Con la aparición de WebGL en los navegadores, los motores han tratado de darle soporte, para de este modo poder crear videojuegos web. Un aspecto relevante de WebGL es que es independiente de la plataforma, ya que se ejecuta en cualquier navegador que lo soporte.

De forma similar ocurre en el caso del o los géneros de videojuego para los cuales fue diseñado el motor. La mayorías de estos se especializan en uno en particular, mientras que pocos lo hacen en varios.

Los efectos gráficos que proveen los motores son elaborados a partir de un conjunto de técnicas, las cuales se suelen realizar en el hardware gráfico. Sin embargo, algunos presentan técnicas que se realizan parcialmente o completamente en CPU.

Por todo lo anteriormente expuesto, seria ideal tener un motor de videojuego basado en WebGL de código abierto, que no se encuentre restringido a un género de videojuego, con una baja curva de aprendizaje y con una documentación completa pero sencilla, tal que permita el desarrollo de videojuegos de alto desempeño gráfico empleando el hardware gráfico actual.

1.2. Objetivos

1.2.1. Objetivo general

Desarrollar un motor de videojuego basado en WebGL.

1.2.2. Objetivos específicos

- Desarrollar una arquitectura que contemple despliegue gráfico, manejo de eventos y recursos.
- Emplear diversos modelos de iluminación local y sombreado, considerando para los primeros, distintos tipos de fuentes de luz.
- Implementar diferentes efectos de texturizado.
- Generar sombras desde los distintos tipos de fuentes de luz.
- Permitir la selección y descarte de geometría.
- Agrupar los objetos a tratar en un grafo de escena.
- Realizar una documentación completa pero sencilla, donde se expongan todas las funciones desarrolladas.

- Realizar pruebas comparativas con otros motores que empleen WebGL.

1.3. Antecedentes

Previamente a la existencia de los motores de videojuegos, los videojuegos eran tratados como entidades singulares, es decir, debían ser diseñados y desarrollados desde cero para garantizar su ejecución óptima en el hardware gráfico. Para ese entonces, el desarrollo de videojuegos estaba orientado exclusivamente para máquinas recreativas o *arcades*. En algunos casos, éstas máquinas ofrecían más libertad de acción al momento de llevar a cabo el desarrollo de videojuegos que otras debido a su naturaleza, permitiendo a los desarrolladores obviar detalles a nivel de implementación como el alto detalle a nivel gráfico. Sin embargo, siempre existían otro tipo de restricciones que debían tomarse estrictamente en cuenta al momento del desarrollo, como por ejemplo, el uso de memoria [6].

Con la rápida evolución del hardware que se encontraba presente en las máquinas recreativas, la mayoría del código fuente de los videojuegos debía desecharse, ya que el desarrollo de los videojuegos debía reformarse totalmente de una generación a otra para aprovechar las nuevas capacidades que ofrecía el nuevo hardware.

Luego de la llamada era de oro de los videojuegos que se gestó a principios de los años setenta, surgieron los motores de videojuegos y con ellos se introdujo el concepto de reutilización de módulos, lo que trajo como principal ventaja que el desarrollo de videojuegos no necesariamente debía realizarse completamente desde cero. Algunos de los motores de videojuegos que surgieron fueron Unreal, Source, Unity, Frostbyte, id Tech y Anvil Next.

Particularmente en el ámbito web, con la evolución de los navegadores y el surgimiento de WebGL, se desarrollaron varios motores de videojuegos que se ejecutan en los navegadores, entre los cuales destacaron Babylon.js [7] y Three.js [8].

Three.js es un motor de videojuegos escrito en JavaScript de licencia MIT cuya primera versión fue publicada en GitHub en el año 2010. Actualmente posee una gran cantidad de funcionalidades, efectos disponibles y una comunidad de más de 390 desarrolladores a nivel mundial que lo actualizan constantemente [9].

Por otro lado, Babylon.js es un motor de videojuegos escrito en JavaScript de licencia Apache 2.0 resultante del desarrollo de un motor de videojuegos que lleva su mismo nombre que fue elaborado usando Silverlight 5 y XNA en el año 2012 por Microsoft. Actualmente, Babylon.js posee una amplia cantidad de funcionalidades y efectos para la creación de videojuegos y una gran comunidad de desarrolladores.

Adicionalmente, es importante destacar que, tanto Babylon.js como Three.js po-

seen módulos que se encuentran fuertemente acoplados, es decir, la lógica que les permite a estos motores ofrecer su amplio rango de funcionalidades se encuentra en la mayoría de los casos centralizada, dificultando su entendimiento.

1.4. Alcance

Los motores de videojuegos ofrecen un conjunto de funcionalidades las cuales varían dependiendo del o los géneros para los cuales fue desarrollado. Sin embargo, las funcionalidades típicas que son provistas por estos generalmente son despliegue gráfico, simulación de efectos físicos, manejo de eventos, de recursos y de redes, inteligencia artificial y audio.

El presente trabajo de investigación desarrollará un motor de videojuego cuya arquitectura contemplará las funcionalidades de despliegue gráfico, manejo de eventos y de recursos. Dicha arquitectura estará compuesta por un conjunto de módulos que llevarán a cabo las funcionalidades mencionadas anteriormente. Sin embargo, los módulos se encontrarán débilmente acoplados con la finalidad de fomentar la escalabilidad y legibilidad del código fuente del motor, para que de este modo se facilite el desarrollo de trabajos futuros.

Con respecto al despliegue gráfico, se proveerá iluminación local y sombreado, distintos tipos de fuentes de luz, efectos creados por la interacción con la luz y efectos de texturizado. Adicionalmente, se ofrecerán técnicas para optimizar el despliegue mediante técnicas de descarte y se agruparán los objetos de los escenarios virtuales usando un grafo de escena.

2

Marco teórico

2.1. Videojuego

Un videojuego es una aplicación interactiva creada para el entretenimiento, en donde se plantean un conjunto de retos u objetivos a cumplir por el usuario en un mundo acotado. Dicho mundo, suele estar conformado por un conjunto de escenarios virtuales recreados en 2D o 3D donde conviven diversas entidades. Comúnmente, se le delega el control de una o varias entidades al usuario garantizando la interacción y la retroalimentación con éste.

Formalmente, la mayoría de videojuegos suponen un ejemplo representativo de aplicaciones gráficas de despliegue en tiempo real, donde los escenarios virtuales que componen su mundo son descritos matemáticamente para que puedan ser manipulados por un computador. Representando así, una simplificación de la realidad, incluso para los casos en que estos se basan en una ficticia, ya que resulta impráctico incluir todos los detalles que posee [10].

En la informática gráfica, el despliegue o *rendering* se define como un proceso de generación de una imagen que representa a una escena, bien sea 2D o 3D, en un dispositivo de salida. Generalmente, las escenas están compuestas por un conjunto de objetos, que se definen mediante recursos o por algoritmos que los generen.

Según [1], desde un punto de vista abstracto, una aplicación gráfica de despliegue en tiempo real se basa en un ciclo donde en cada iteración se realizan los siguientes pasos:

- El usuario visualiza una imagen resultante del proceso de despliegue.
- El usuario actúa en función de lo visualizado, interactuando directamente con la aplicación, por ejemplo mediante un teclado.
- En función de la acción realizada por el usuario, la aplicación gráfica genera una salida u otra, es decir, existe una retroalimentación que afecta a la propia aplicación.

En el caso de los videojuegos, este ciclo de visualización, actuación y despliegue ha de ejecutarse con una frecuencia lo suficientemente elevada como para que el usuario se sienta inmerso en el videojuego, y no lo perciba como una sucesión de imágenes estáticas. De esta forma, se debe desplegar al menos quince cuadros por segundo para que el ojo humano no note discontinuidades en la animación [11].

El rendimiento del despliegue es medido comúnmente a partir de la velocidad de despliegue de sus cuadros. La medida para expresar el rendimiento es la cantidad de cuadros por segundo o *frames per second* (fps).

El componente gráfico supone gran parte de la complejidad computacional de un videojuego. Sin embargo, no es el único que debe ser considerado, ya que como se mencionó anteriormente, un videojuego se caracteriza por su interacción. Razón por la cual, en cada ciclo de ejecución se debe tomar en cuenta la evolución del mundo donde se desarrolla, lidiando con todos los eventos que puedan ocurrir dentro del mismo. Adicionalmente, debe garantizar una tasa de fps adecuada para que la inmersión del usuario no se vea afectada [1].

2.2. Motor de videojuego

Según [10], el término motor de videojuego surgió a mediados de los años noventa con la aparición del famoso videojuego de acción en primera persona Doom (ver Figura 2.1), desarrollado por la compañía id Software [12]. Los videojuegos hasta ese momento se desarrollaban pensando únicamente en ellos, descartando su posible reutilización. Doom rompió este paradigma presentando una arquitectura orientada a la reutilización mediante una separación adecuada en distintos módulos de sus componentes fundamentales, como por ejemplo el sistema de despliegue gráfico, el sistema de detección de colisiones o el sistema de audio, y los elementos artísticos, por ejemplo los escenarios virtuales o las reglas que afectaban la experiencia del mismo.

Figura 2.1: Carátula del videojuego Doom.

El valor de dicha separación se hizo cada vez más evidente a medida que los desarrolladores empleaban videojuegos anteriormente elaborados para desarrollar nuevos del mismo tipo haciendo cambios leves en los componentes del motor, permitiéndoles centrar sus esfuerzos en la parte artística y en las reglas del mismo. Esto generó un gran incremento en la popularidad del concepto de motor de videojuego.

Este enfoque ha ido evolucionando y se ha expandido, desde la generación de *mods*[1] por desarrolladores independientes hasta la creación de una gran variedad de herramientas, bibliotecas e incluso lenguajes que facilitan el desarrollo de videojuegos. Actualmente, una gran parte de compañías de desarrollo de videojuego utilizan motores o herramientas pertenecientes a terceras partes, debido a que les resulta más rentable económicamente y obtienen, generalmente, resultados espectaculares [1]. No obstante, esta evolución también ha permitido que tanto empresas como personas involucradas en el desarrollo de videojuegos se planteen licenciar partes o completamente su propio motor de videojuego.

La separación entre motor de videojuego y videojuego nunca es total y, por una circunstancia u otra, siempre existen dependencias directas que no permiten la reusabilidad completa del motor para crear otro videojuego. La dependencia más evidente es el o los géneros a los que está vinculado el motor de videojuego [1]. Por ejemplo, un motor diseñado para elaborar videojuegos de estrategia, será difícilmente reutilizable para elaborar un videojuego de acción en primera persona.

El concepto de arquitectura dirigida por datos o *data-driven architecture* es lo que denota la diferencia entre el software que representa un videojuego y el motor de videojuego. Cuando un videojuego contiene parte de su lógica o funcionamiento en el

[1]Un *mod* es una extensión que modifica un videojuego original proporcionando nuevas funcionalidades.

propio código fuente (*hard-coded logic*), entonces no resulta práctico reutilizarla para otro videojuego, ya que implicaría modificar el código sustancialmente. Sin embargo, si dicha lógica no está definido a nivel de código, sino mediante entes externos, por ejemplo mediante una serie de reglas determinadas a través de un lenguaje de *scripting*, entonces la reutilización sí es posible y, por lo tanto, beneficiosa, ya que optimiza el tiempo de desarrollo [1]. El término motor de videojuego debería ser reservado para aquel software que puede ser utilizado como generador o base para diversos videojuegos sin modificaciones relevantes.

2.2.1. Géneros de videojuegos

Los motores de videojuegos suelen estar especializados para el desarrollo de un conjunto de tipos o géneros particulares de videojuegos. No obstante, existen ciertas características, sobre todo relativas al procesamiento de más bajo nivel, que son transversales a cualquier tipo de videojuego, es decir, que se pueden reutilizar en gran medida de manera independiente a los géneros que maneje el motor. Por ejemplo, todos los videojuegos 3D, pese a su genero, necesitan soportar el despliegue de mallas, el tratamiento de eventos de usuario donde se recoge y gestiona la interacción del usuario a través de dispositivos de entrada, etc.

A continuación, se listan los géneros más populares de la actualidad [1].

- Acción.
- Disparos.
- Estrategia.
- Simulación.
- Deporte.
- Carreras.
- Aventura.
- Rol.

2.2.2. Arquitectura general

Como se explica en [1, 10], en esta sección se propone una visión general de la arquitectura que posee un motor de videojuegos, siendo esta independiente del género y exhibiendo los componentes más importantes desde el punto de vista del desarrollo de videojuegos.

Como en todos los sistemas de software, los motores de videojuegos se construyen usando una arquitectura que se estructura en capas. Las capas de los niveles superiores dependen de las capas que se encuentran en niveles inferiores pero esto no ocurre de manera inversa. De esta forma, se pueden agregar capas progresivamente y, lo que es aún más importante, es que con esta estructura se puede modificar una capa en particular sin que las capas de niveles inferiores a esta se vean afectadas por los cambios.

A continuación, se describen los módulos que forman parte de la arquitectura que se expone en la Figura 2.2.

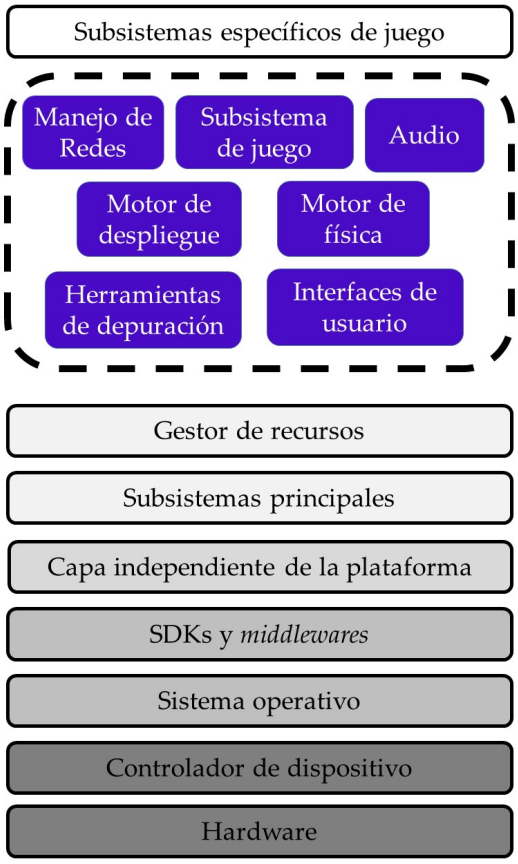


Figura 2.2: Arquitectura general de un motor de videojuego [1].

2.2.2.1. Hardware, controladores de dispositivo y sistema operativo

La capa asociada al hardware esta representada por la plataforma donde se ejecutará el motor de videojuego. Por ejemplo, un tipo de plataforma específica podría ser una consola de videojuegos. Muchos de los principios de diseño y desarrollo son comunes a cualquier videojuego, de manera independiente a la plataforma donde

se ejecutará el despliegue final. Sin embargo, en la práctica los desarrolladores de videojuegos siempre llevan a cabo optimizaciones en el motor de videojuegos para mejorar la eficiencia del mismo, considerando aquellas cuestiones que son específicas de una determinada plataforma.

Por otro lado, la capa de controlador de dispositivo o *driver* gestiona los componentes de software de bajo nivel. Normalmente, los *drivers* son provistos ya sea por el sistema operativo o por el fabricante del hardware. Su función, es controlar los recursos que ofrece el hardware y se encargan de abstraer al sistema operativo y a las capas de niveles superiores del motor de videojuegos de como llevar a cabo la comunicación con una gran cantidad de dispositivos de hardware que se encuentran actualmente disponibles, como la tarjeta de aceleración gráfica o las tarjetas de sonido.

Finalmente, la capa del sistema operativo representa la capa de comunicación entre los procesos que se ejecutan en el mismo y los recursos de hardware asociados a la plataforma.

2.2.2.2. SDKs y *middlewares*

Como ocurre en otros proyectos de software, el desarrollo de un motor de videojuegos se suele apoyar en SDKs[1] y *middlewares*[2] para ofrecer una determinada funcionalidad. No obstante, y aunque generalmente éste software se encuentra bastante optimizado, algunos desarrolladores prefieren personalizarlo para adaptarlo a sus necesidades particulares.

En el ámbito de los gráficos 3D, existe un gran número de bibliotecas de desarrollo que solventan determinados aspectos que son comunes a la mayoría de los videojuegos. Los ejemplos más representativos en este contexto son las APIs[3] gráficas OpenGL y Direct3D, mantenidas por el grupo Khronos y Microsoft, respectivamente [13, 14]. Este tipo de bibliotecas tienen como principal objetivo ocultar los diferentes aspectos de las tarjetas gráficas, presentando una interfaz común. Mientras OpenGL es multiplataforma, Direct3D está totalmente ligado a sistemas Windows.

Otro ejemplo representativo de SDKs vinculados al desarrollo de videojuegos son aquellos que dan soporte a la detección y tratamiento de colisiones y a la gestión de

[1] Un kit de desarrollo de software o *software development kit* (SDK) es un conjunto de herramientas de desarrollo de software que le permite al programador o desarrollador de software crear aplicaciones para un sistema concreto.

[2] Un *middleware* es un software que asiste a una aplicación para interactuar o comunicarse con otras aplicaciones, o paquetes de programas, redes, hardware y/o sistemas operativos.

[3] Una interfaz de programación de aplicaciones o *application programming interface* (API) se define como un conjunto de rutinas, protocolos y herramientas para construir aplicaciones a nivel de software.

la física de los distintas entidades que forman parte de un videojuego.

2.2.2.3. Capa independiente de la plataforma

La mayoría de los motores de videojuegos deben ser capaces de ejecutarse en más de una plataforma, ya que como estrategia de mercado, gran parte de los videojuegos se desarrollan teniendo en cuenta su potencial lanzamiento para una amplia variedad de plataformas. Por ejemplo, un título se puede desarrollar para diversas consolas de videojuegos y para computadores personales al mismo tiempo. En este contexto, es bastante común encontrar una capa software que aísle al resto de capas superiores de cualquier aspecto que sea dependiente de la plataforma. Dicha capa se suele denominar capa independiente de la plataforma.

2.2.2.4. Subsistemas principales

La capa de subsistemas principales está vinculada a todas aquellas utilidades o bibliotecas de utilidades que dan soporte al motor de videojuegos. Algunas de ellas son específicas del ámbito de los videojuegos pero otras son comunes a cualquier tipo de proyecto de software que tenga una complejidad significativa. A continuación se enumeran algunos de los subsistemas más relevantes.

- Biblioteca matemática: responsable de proporcionar al desarrollador diversas utilidades que faciliten el tratamiento de operaciones relativas a vectores, matrices, cuaterniones u operaciones vinculadas a líneas, rayos, esferas y otras figuras geométricas [15]. Las bibliotecas matemáticas son esenciales en el desarrollo de un motor de videojuegos, ya que éstos tienen una naturaleza inherentemente matemática.
- Estructuras de datos y algoritmos: responsable de proporcionar una implementación más personalizada y optimizada de diversas estructuras de datos.
- Gestión de memoria: responsable de garantizar la asignación y liberación de memoria de una manera eficiente.
- Depuración: responsable de proporcionar herramientas para facilitar la depuración y el volcado de bitácoras o *logs* para su posterior análisis.

2.2.2.5. Gestor de recursos

Esta capa es la responsable de ofrecer una interfaz unificada para acceder a las distintas entidades de software que conforman el motor de videojuegos, por ejemplo la escena o sus propios objetos.

2.2.2.6. Motor de despliegue

Dado que el componente gráfico es una parte clave de cualquier videojuego, junto con la necesidad de mejorarlo continuamente, el motor de despliegue es una de las partes más complejas de cualquier motor de videojuego.

Así como ocurre con la propia arquitectura de un motor de videojuegos, el enfoque más utilizado para diseñar el motor de despliegue consiste en elaborar una arquitectura multicapa, como se puede apreciar en la Figura 2.3. A continuación se describen los principales módulos que forman parte de cada una de las capas de este componente.

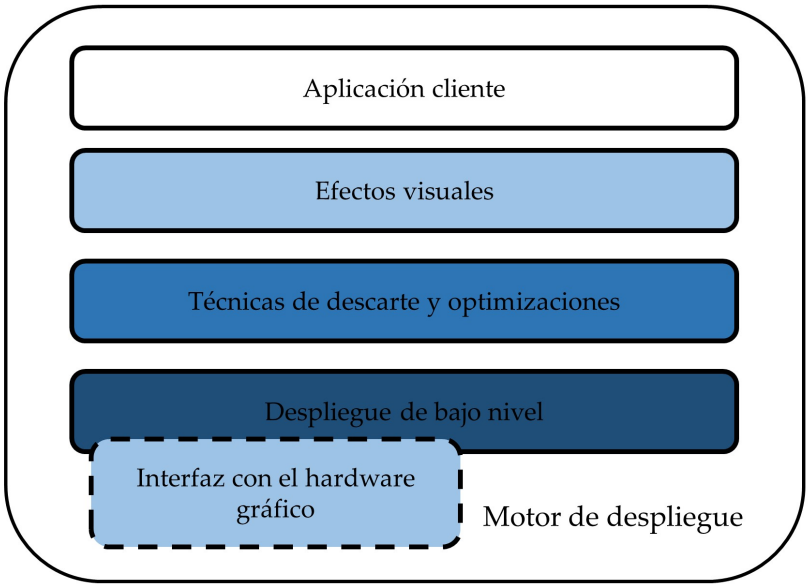


Figura 2.3: Arquitectura multicapa de un motor de despliegue [1].

La capa de despliegue de bajo nivel une las distintas utilidades de despliegue del motor de videojuego, como por ejemplo cámaras, primitivas de despliegue, materiales, texturas, etc. El objetivo principal de esta capa reside precisamente en desplegar las distintas primitivas geométricas tan rápido como sea posible.

Adicionalmente, esta capa también es responsable de gestionar la interacción con las APIs de programación gráficas, con la finalidad de tener acceso a los distintos dispositivos gráficos que estén disponibles. Típicamente, este módulo se denomina interfaz de dispositivo gráfico o *graphics device interface*.

También, en la capa de despliegue de bajo nivel existen otros componentes encargados de procesar el despliegue de distintas primitivas geométricas, así como de la gestión de la cámara y los diferentes modos de proyección. En otras palabras, esta

capa proporciona una serie de abstracciones para manejar tanto las primitivas geométricas como las cámaras virtuales y las propiedades vinculadas a las mismas.

Por otra parte, esta capa también gestiona el estado del hardware gráfico y los sombreadores[1] asociados. Básicamente, cada primitiva recibida por esta capa tiene asociado un material y se ve afectada por diversas fuentes de luz. Así mismo, el material describe la textura o texturas utilizadas por la primitiva y otros aspectos como por ejemplo qué sombreadores serán utilizados.

La capa superior a la de despliegue de bajo nivel se denomina descarte o *culling* y optimizaciones, desde un punto de vista general, es la responsable de seleccionar qué parte o partes de la escena se enviarán a la capa de despliegue. Esta selección, u optimización, permite incrementar el rendimiento del motor, debido a que se limita el número de primitivas geométricas enviadas a la capa de nivel inferior.

Sobre la capa de optimizaciones se sitúa la capa de efectos visuales, la cual ofrece soporte a distintos efectos que, posteriormente, se puedan integrar en los videojuegos desarrollados haciendo uso del motor.

Finalmente, la capa de *front-end* suele estar vinculada a una funcionalidad relativa a la superposición de contenido 2D sobre el escenario 3D. Por ejemplo, es bastante común utilizar algún tipo de módulo que permita visualizar el menú de un videojuego o la interfaz gráfica que permite conocer el estado del personaje principal del videojuego (inventario, armas, herramientas, etc). En esta capa, también se incluyen componentes para reproducir vídeos previamente grabados y para integrar secuencias cinemáticas, a veces interactivas, en el propio videojuego.

2.2.2.7. Herramientas de depuración

Debido a la naturaleza intrínseca de un videojuego, vinculada a las aplicaciones gráficas en tiempo real, resulta esencial contar con buenas herramientas que permitan depurar y optimizar el propio motor de videojuegos para obtener el mejor rendimiento posible. En este contexto, existe un gran número de herramientas de este tipo. Algunas de ellas son herramientas de propósito general que se pueden utilizar de manera externa al motor de videojuegos. Sin embargo, la práctica más habitual consiste en elaborar herramientas, vinculadas al análisis del rendimiento, o depuración que estén asociadas al propio motor.

[1] Un sombreador o *shader* es un programa que permite la generación de efectos visuales complejos. Dicho programa, se elabora utilizando un lenguaje de programación que por lo general tiene una gran semejanza con el lenguaje de programación C.

2.2.2.8. Motor de física

La detección de colisiones en un videojuego y su posterior tratamiento resultan esenciales para dotar de realismo al mismo. Sin un mecanismo de detección de colisiones, los objetos se traspasarían unos a otros y no sería posible interactuar con ellos. Un ejemplo típico de colisión está representado en los videojuegos de conducción por el choque entre dos o más vehículos. Desde un punto de vista general, el sistema de detección de colisiones es responsable de detectar, determinar y generar la respuesta a una colisión [11].

Debido a las restricciones impuestas por la naturaleza de tiempo real de un videojuego, los mecanismos de gestión de colisiones se suelen aproximar para simplificar la complejidad de los mismos y no reducir el rendimiento del motor. Por ejemplo, en algunas ocasiones los objetos 3D se aproximan con una serie de líneas, utilizando técnicas de intersección de líneas para determinar la existencia o no de una colisión.

Por otra parte, algunos videojuegos incluyen sistemas realistas o semirealistas de simulación dinámica. En el ámbito de la industria del videojuego, estos sistemas se suelen denominar sistema de física y están directamente ligados al sistema de gestión de colisiones.

2.2.2.9. Interfaces de usuario

En todo videojuego, es necesario desarrollar un módulo que ofrezca una abstracción respecto a la interacción del usuario, es decir, un módulo que principalmente sea responsable de procesar los eventos de entrada del usuario. Típicamente, dichos eventos estarán asociados a la pulsación de una tecla, al movimiento del ratón, entre otros.

Desde un punto de vista más general, el módulo de interfaces de usuario también es responsable del tratamiento de los eventos de salida, es decir, aquellos eventos que proporcionan una retroalimentación al usuario. Esta interacción puede estar representada, por ejemplo, por el sistema de vibración del mando de una consola o por la fuerza ejercida por un volante que está siendo utilizado en un videojuego de conducción. Debido a que este módulo gestiona los eventos de entrada y de salida, se suele denominar comúnmente componente de entrada/salida o *I/O component*.

El módulo de interfaces de usuario actúa como un puente entre los detalles de bajo nivel del hardware utilizado para interactuar con el videojuego y el resto de controles de más alto nivel. Este módulo también es responsable de otras tareas importantes, como la asociación de acciones o funciones lógicas al sistema de control del videojuego, es decir, permite asociar eventos de entrada a acciones lógicas de alto nivel.

2.2.2.10. Redes y multijugador

El módulo de redes o *networking* es el responsable de informar de la evolución del videojuego a los distintos actores o usuarios involucrados en el mismo mediante el envío de paquetes de información. Con el objetivo de reducir la latencia del modo multijugador, especialmente a través de la Internet. Particularmente, sólo se envía/recibe información relevante para el correcto funcionamiento de un videojuego.

2.2.2.11. Subsistema de juego

El subsistema de juego o *gameplay*, integra todos aquellos módulos relativos al funcionamiento interno del videojuego, es decir, une tanto las propiedades del mundo virtual como la de los distintos personajes. Adicionalmente, permite la definición de las reglas que gobiernan el mundo virtual en el que se desarrolla el videojuego, como por ejemplo la necesidad de derrotar a un enemigo antes de enfrentarse a otro de mayor nivel.

Este subsistema sirve también como capa de aislamiento entre las capas de más bajo nivel, como por ejemplo la de despliegue, y el propio funcionamiento del videojuego. Es decir, uno de los principales objetivos de diseño que se persiguen consiste en independizar la lógica del videojuego de la implementación subyacente. Por ello, en esta capa es bastante común encontrar algún tipo de sistema de *scripting* o lenguaje de alto nivel para definir, por ejemplo, el comportamiento de los personajes que participan en el videojuego.

2.2.2.12. Audio

Comúnmente, el mundo del desarrollo de videojuegos siempre ha prestado más atención al componente gráfico. Sin embargo, el apartado sonoro también tiene una gran importancia para conseguir una inmersión total del usuario en el videojuego. Por ello, el motor de audio ha ido cobrando un rol importante.

A raíz de la aparición de nuevos formatos de audio de alta definición y la popularidad de los sistemas de cine en casa, se han hecho contribuciones cada vez más relevantes a la evolución del apartado sonoro.

2.2.2.13. Subsistemas específicos de juego

Por encima de la capa de subsistema de juego y otros componentes de más bajo nivel se sitúa la capa de subsistemas específicos de juego, donde se integran aque-

llos módulos responsables de ofrecer las características propias del videojuego. En función del tipo de videojuego a desarrollar.

Idealmente, la línea que separa el motor de videojuego y el propio videojuego en cuestión estaría entre la capa de subsistema de juego y la capa de subsistemas específicos de juego.

2.3. WebGL

Tradicionalmente, para utilizar gráficos 3D era necesario crear una aplicación autónoma[1] usando un lenguaje de programación como C o C++ y una API gráfica especializada, restringiendo su uso a sistemas muy específicos como computadoras con una gran capacidad de cómputo o a consolas de videojuegos. Con el gran avance que han experimentado los navegadores, junto con su amplia gama de tecnologías de desarrollo se volvió una necesidad el uso de gráficos 3D en la web.

En un principio, se tomó un subconjunto de instrucciones de OpenGL y surgió una versión de dicha API para sistemas embebidos o *embedded systems* llamada OpenGL ES, donde el sufijo "ES" es un acrónimo que se usa para hacer referencia a dichos sistemas. OpenGL ES estaba diseñado para ser utilizado exclusivamente en dispositivos móviles. Posteriormente, éste se adaptó para que pudiera funcionar en los navegadores, dando como resultado la creación de WebGL.

Por lo tanto, se puede definir a WebGL como una API que permite el despliegue e interacción de gráficos 3D dentro de los navegadores. WebGL es capaz de combinarse con tecnologías como HTML5[2] y JavaScript[3] dándole así a los desarrolladores web un fácil acceso a los gráficos 3D. Adicionalmente, juega un rol importante en el desarrollo de interfaces de usuario minimalistas e intuitivas así como contenido web interactivo [2].

2.3.1. Orígenes

OpenGL fue desarrollado originalmente por Silicon Graphics y publicado como un estándar abierto en el año 1992. Desde entonces, ha evolucionado a lo largo de varias versiones y ha tenido un profundo impacto en el desarrollo de gráficos 3D, el desarrollo de productos de software e incluso en la producción cinematográfica

[1]Una aplicación autónoma o *stand-alone* se refiere a que es capaz de funcionar sin ningún tipo de conexión a la Internet.

[2]HTML5 es la ultima versión del estándar del Lenguaje de Marcado de Hipertexto o *HyperText Markup Language* (HTML) empleado para la elaboración de páginas web.

[3]JavaScript es un lenguaje de *scripting*, es pequeño, posee soporte orientado a objetos, es multiplataforma y es utilizado ampliamente en la web.

a lo largo de los años. Como se explicó anteriormente WebGL tiene sus raíces en OpenGL ES el cual fue desarrollado entre los años 2003 y 2004, posteriormente fue actualizado en el año 2007 a su versión 2.0 y finalmente se actualizó a la versión 3.0 en el año 2012, WebGL particularmente adopta la versión 2.0 de OpenGL ES.

En los últimos años, el número de dispositivos que soportan su especificación se ha incrementado rápidamente y gracias a este exitoso soporte se han agregado a OpenGL ES nuevas características y muchas otras se han eliminado ya que han sido innecesarias u obsoletas, dando como resultado una especificación bastante pequeña que provee suficiente poder para llevar a cabo la producción de gráficos 3D.

Como muestra la Figura 2.4, al pasar a la versión 2.0 de OpenGL se introdujo la posibilidad de programar sombreadores, cualidad que es un punto vital en la especificación 1.0 de WebGL. Es importante destacar que para el caso particular de WebGL, el lenguaje de programación que se utiliza para elaborar los sombreadores se llama *OpenGL ES Shading Language* o GLSL ES.

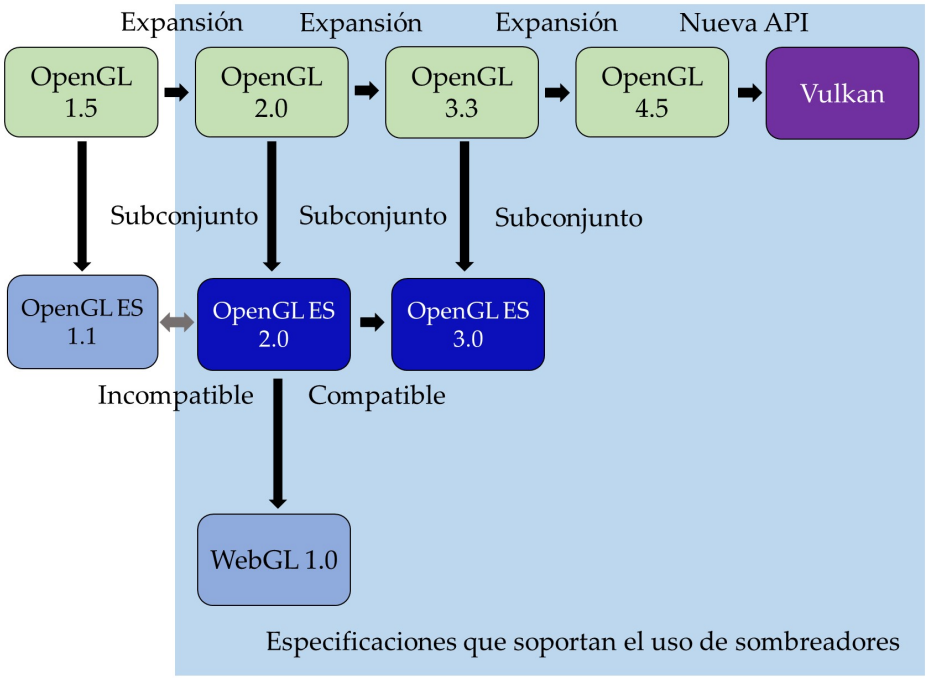


Figura 2.4: Relación entre OpenGL, OpenGL ES, WebGL y Vulkan.

2.3.2. Estructura de una aplicación

En el mundo web, las aplicaciones dinámicas pueden ser creadas combinando HTML y JavaScript. Sin embargo, con la introducción de WebGL, GLSL ES se agrega a la combinación, trayendo como consecuencia que las aplicaciones que usan WebGL

son creadas a partir del uso de tres lenguajes, HTML5 como lenguaje de marcado, JavaScript como lenguaje de scripting y GLSL ES como lenguaje de sombreado. La Figura 2.5 muestra la estructura de las aplicaciones dinámicas tradicionales en el lado izquierdo y las que emplean WebGL en el lado derecho.

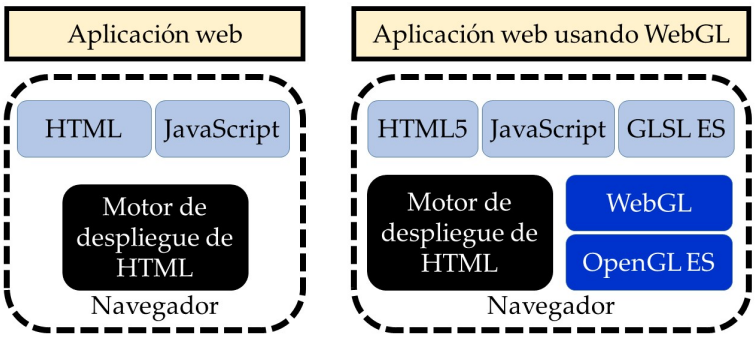


Figura 2.5: Comparación de estructuras de aplicaciones web [2].

Aunque la capacidad técnica de WebGL es impresionante, es tal vez la facilidad de uso y accesibilidad que lo diferencian de otras tecnologías, ya que permite:

- Desarrollar aplicaciones con contenido gráfico 3D empleando únicamente un editor de código y un navegador.
- Obtener todos los beneficios por ser una misma aplicación web como la fácil publicación, manejo de eventos, comunicación con servidores, entre otros.

Sin embargo, a pesar de todas las grandes ventajas que posee, también se pueden resaltar ciertas desventajas:

- No es soportado de manera estable en todos los navegadores que se encuentran disponibles actualmente en el mercado.
- Su rendimiento se ve afectado ya que JavaScript es un lenguaje interpretado.
- Debido a que esta formado por un conjunto reducido de instrucciones, no permite el aprovechamiento total de las capacidades que ofrece el hardware gráfico.

3

Solución propuesta

3.1. Arquitectura

Se propone una arquitectura estructurada en capas, adaptada a un entorno web y que no se encuentre restringida a un género de videojuego (ver Figura 3.1). Sin embargo, que se implemente de forma débilmente acoplada, es decir, en vez de conservar toda la lógica en cada capa o módulo, que se encuentre dispersa entre todas las entidades que se relacionan con éstas.

Dado a que el motor de videojuego se ejecuta dentro de un entorno web, las capas de hardware, controladores y sistema operativo son abstraídas por el navegador. Adicionalmente, el único SDK empleado es WebGL. Por lo tanto, el motor de juego se podrá ejecutar en cualquier navegador que lo soporte. Los subsistemas principales están compuestos únicamente por la biblioteca de matemáticas, estructuras de datos y algoritmos y gestión de memoria.

El módulo de interfaces de usuario interactúa con el navegador para manejar los eventos de entrada del ratón, teclado y pulsaciones táctiles. Aunque los módulos de motor de física, audio, subsistema de juego y redes no se encuentren, no significa que los efectos provistos por estos no puedan estar presentes. Pueden estarlo siempre y cuando estos sean implementados por el usuario o por alguna biblioteca o SDK existente.

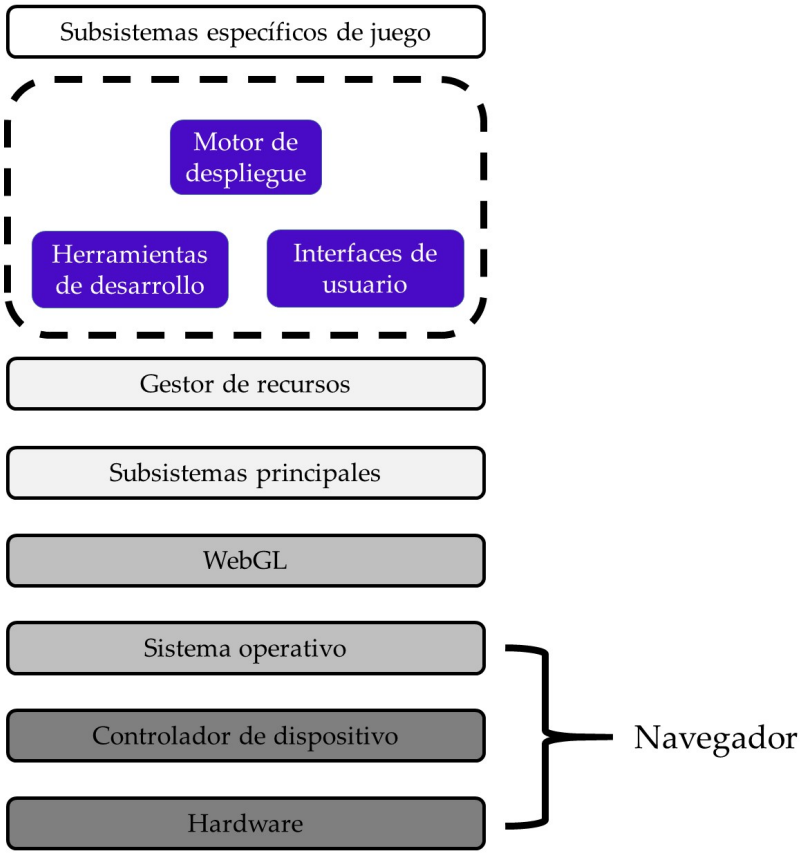


Figura 3.1: Arquitectura propuesta.

3.2. Técnica de despliegue

El resultado del proceso de despliegue es una imagen 2D, donde se determina el valor de cada píxel de la imagen; dicho valor es influenciado por las características naturales que se quieran representar. La manera intuitiva de realizar un despliegue es ir línea a línea de la imagen final determinando el valor de cada píxel, trabajo que suele ser lento e impráctico.

Nuestra propuesta usa la rasterización implementada por el hardware gráfico como técnica de despliegue, donde su finalidad es aproximar la imagen final primitiva a primitiva modificando sólo los píxeles relacionados aprovechando el principio de localidad espacial, ya que los píxeles que ocupa una geometría tienden a ser contiguos, haciendo que el despliegue finalice en corto tiempo.

3.3. Iluminación y sombreado

En nuestra propuesta, la iluminación se lleva a cabo usando fuentes de luz que generan efectos cuando un objeto se encuentra expuesto como mínimo a una de ellas. La idea es obtener una imagen resultado cuya calidad sea visualmente aceptable sin comprometer drásticamente el rendimiento que tiene el videojuego, para ello, se consideró únicamente el uso de la iluminación local que se implementa mediante un conjunto de modelos físicos y matemáticos que solo toman en cuenta la información local de una superficie obviando todas las interacciones causadas por las reflexiones difusas y especulares entre los objetos de una escena por su gran costo computacional.

3.3.1. Fuentes de luz

Una parte crucial de la implementación de la iluminación en la informática gráfica es la posibilidad de modelar fuentes de luz y su correspondiente distribución de energía. De esta manera, los efectos visuales resultantes, serán obtenidos de la interacción de la energía proveniente de una fuente de luz sobre una geometría. El calculo de la distribución de la energía desde cualquier fuente de luz es imposible de lograr computacionalmente en tiempo real en estos momentos. Sin embargo, utilizando ciertas fuentes de luces aproximadas, podemos lograr despliegues con una iluminación de alto nivel de realismo.

En nuestra implementación los tipos de luces que se considerados son: puntual, direccional y reflector. Cada fuente de luz tiene diferentes características de rendimiento y un número reducido de ellas pueden ser usadas al mismo tiempo en una escena. Por ello, es importante conocer sus características, de tal manera que sea posible colocar la cantidad mínima de luces en la escena, logrando la mejor iluminación posible y manteniendo el rendimiento al máximo.

3.3.1.1. Luz puntual

Este tipo de fuente de luz está definida por un punto desde el cual la energía es emitida en todas direcciones [13], como puede verse en la Figura 3.2. Los cálculos de una luz puntual son un poco más complicados que los utilizados en otros tipos de luces más simples como la luz direccional. Por ejemplo, la dirección de la luz con respecto a un vértice va a cambiar con respecto a la posición del vértice para hacer un cálculo individualizado.

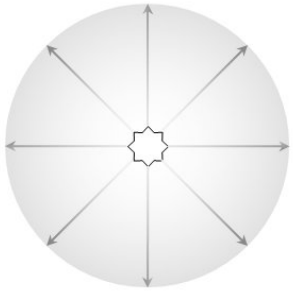


Figura 3.2: Fuente de luz puntual.

3.3.1.2. Luz direccional

Este tipo de fuente de luz es el más sencillo tanto en teoría como en la práctica. Sin embargo, este tipo de luces no existen en el mundo real [13]. Está definida por una dirección, en la cual todos los rayos luminosos viajan de manera paralela. Esto no ocurre en la realidad, ya que estos rayos parten de alguna fuente luminosa. Sin embargo, este tipo de luz asume que la fuente de luz está totalmente alejada en el infinito, lo que crea que los rayos estén muy cercanos a ser paralelos. Con ella se imitan efectos luminosos como el Sol, el cual se encuentra a una distancia considerable de la Tierra, un ejemplo de ello se ve en la Figura 3.3.

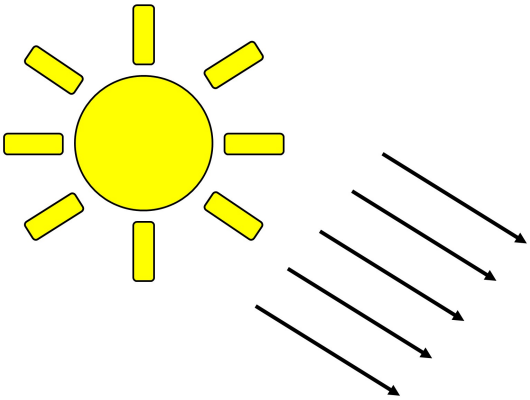


Figura 3.3: Fuente de luz direccional.

3.3.1.3. Luz reflector

Esta categoría de luces está definida por un punto que representa la fuente de luz, una dirección y un ángulo en la cual la energía se encuentra proyectada [13]. Un caso de la vida real de este tipo de luz es la linterna, en la cual la luz está encerrada en un contenedor que solo permite que ilumine en una dirección específica, como puede observarse en la Figura 3.4.

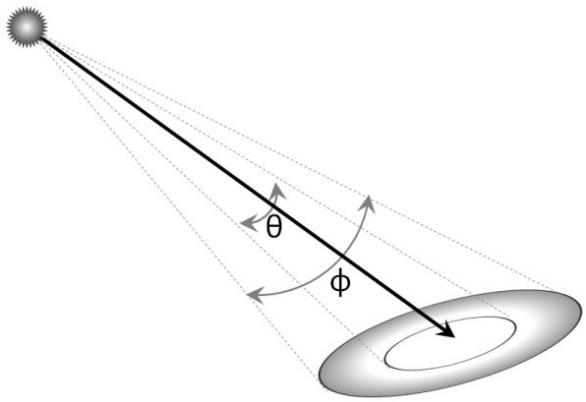


Figura 3.4: Fuente de luz reflector.

Si denominamos L como el vector que va desde la fuente de luz hasta un vértice de un objeto y L_{dir} como la dirección de la fuente de luz tenemos que:

$$\frac{L \cdot L_{dir}}{|L||L_{dir}|} = \cos(\theta) \tag{3.1}$$

Donde θ es la distancia angular entre el objeto con respecto a la dirección de la fuente de luz. Por lo tanto, si $\cos(\theta) \leqslant \cos(\phi)$ entonces ese punto del objeto se encontrará iluminado. En caso contrario, el objeto se encontrará fuera del cono del foco y no será iluminado por esta fuente de luz.

3.3.2. Efectos creados por la interacción con la luz

Dado que la iluminación tiene como objetivo calcular la distribución de energía de la luz en una escena, es importante aclarar los diversos efectos que se crean por la interacción de la luz con las superficies presentes en una escena. En nuestra propuesta se implementan un conjunto básico de efectos como son la luz directa, transparencia, sombras, reflexión, refracción.

3.3.2.1. Luz directa

Es el rayo de luz que viaja directamente desde una fuente de luz hasta una superficie. No se consideran los diferentes caminos que podría tomar la luz para incidir sobre la superficie [3].

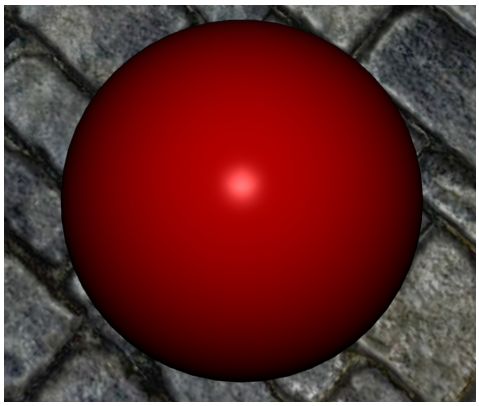


Figura 3.5: Efecto de luz directa sobre una esfera.

3.3.2.2. Transparencia

La transparencia es una propiedad óptica de la materia que tiene diversos grados y propiedades, la cual describe la transmisión de la luz a través de objetos sólidos, haciendo posible que se pueda ver a través de ellos [3].

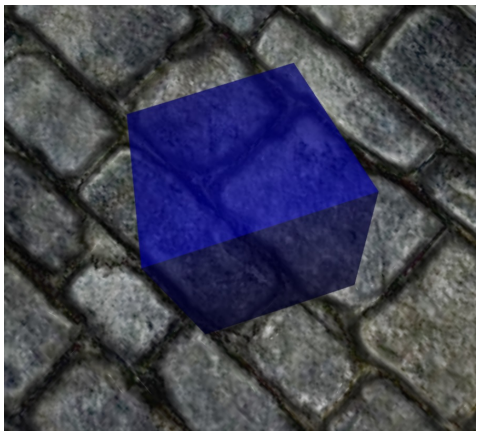


Figura 3.6: Efecto de transparencia sobre un cubo.

3.3.2.3. Sombra

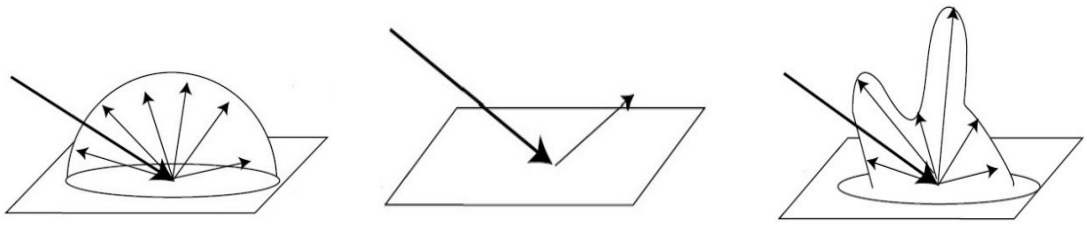
Es el efecto ocasionado por un objeto que obstaculiza el paso normal de la luz, resultado será el obscurecimiento de los objetos ocultos. En la informática gráfica se utilizan principalmente dos tipos de sombras: duras y suaves [3]. El efecto de sombra será explicado con mayor detalle en la sección 3.5.

3.3.2.4. Refracción y reflexión

La refracción es el fenómeno por el cual un rayo de luz incidente sobre una superficie cambia su dirección. Sólo se produce sobre una superficie que separa dos medios que poseen índices refractarios diferentes. Por otra parte, la reflexión representa un cambio de dirección del rayo [3]. Los efectos de reflexión y refracción serán explicados con mayor detalle en la sección 3.4.2.

3.3.3. Tipos de superficies

A raíz del uso de la iluminación local dentro de los escenarios virtuales y de las limitaciones a nivel de realismo, en la informática gráfica se han hecho esfuerzos para poder llevar a cabo la simulación de superficies que se asemejen de una manera aceptable a como se verían en la realidad. Dentro de los esfuerzos llevados a cabo se plantean las superficies difusas que son las que reflejan la luz de manera uniforme en todas las direcciones posibles. Por otro lado, se encuentran las superficies especulares perfectas que son aquellas que reflejan la luz en una sola dirección específica. Sin embargo, como generalmente las superficies en la naturaleza no son puramente especulares ni puramente difusas, sino que exhiben una combinación de los dos comportamientos, lo ideal es llevar a cabo la simulación de una superficie donde se mezcle ambos comportamientos, éstas superficies se denominan superficies brillosas [3].



(a) Difusas perfectas. (b) Especulares perfectas. (c) Brillosas.

Figura 3.7: Tipos de superficies [3].

En nuestra propuesta, se lleva a cabo dicha simulación combinando los modelos para el cálculo de la reflexión difusa con los modelos para el cálculo de la reflexión especular.

3.3.4. Modelos para la reflexión difusa

La reflexión difusa es el fenómeno que ocurre cuando un rayo de luz es reflejado en varias direcciones con distintos ángulos al impactar una superficie. Nuestra propuesta considera dentro de su implementación los modelos básicos para llevar a cabo el cálculo de la reflexión difusa, ellos son la reflexión de Lambert y la reflexión de Oren-Nayar.

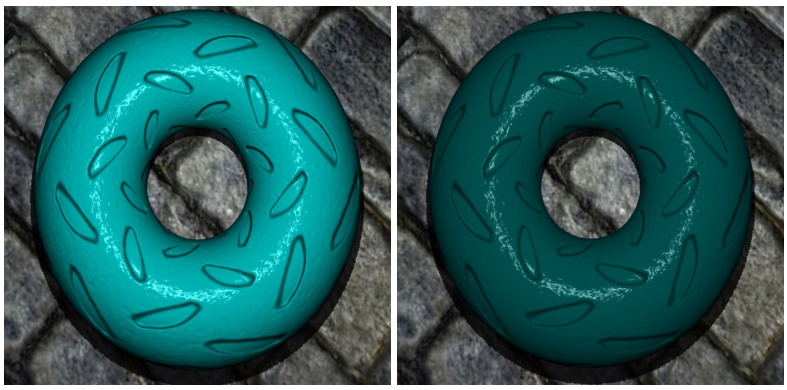
3.3.4.1. Reflexión de Lambert

La reflexión lambertiana es un modelo matemático propuesto por Johann Heinrich Lambert en el año 1760 en su libro *Photometria* para el cálculo de la reflexión difusa. Dicho modelo, plantea la existencia de superficies ideales que él nombró "lambertianas", cuya principal característica es que son capaces de reflejar un rayo de luz incidente desde una dirección de igual manera en todas las direcciones, en consecuencia, el brillo que genera la superficie al estar expuesta a una fuente de luz no se verá afectado desde el punto de vista en el que se encuentre el observador [13, 16].

3.3.4.2. Reflexión de Oren-Nayar

La reflexión de Oren-Nayar es un modelo matemático que plantea una solución al cálculo de la reflexión difusa. Fue desarrollado por Michael Oren y Shree K. Nayar en el año 1994 [17]. El modelo Oren-Nayar se utiliza para el cálculo de la iluminación de superficies ásperas como el concreto, la arena y el yeso. Por otro lado, éste modelo toma en cuenta fenómenos físicos complejos como son el oscurecimiento mediante sombreado y las reflexiones internas entre las facetas[1] de una superficie.

[1]Una faceta se define como un lado o una cara que posee una superficie compuesta por polígonos. Por ejemplo, un cubo es una superficie que tiene seis caras o facetas.



(a) Reflexión de Lambert. (b) Reflexión de Oren-Nayar.

Figura 3.8: Comparación entre la reflexión difusa de Lambert y Oren-Nayar.

3.3.5. Modelos para la reflexión especular

La reflexión especular es el fenómeno que ocurre cuando un rayo de luz es reflejado en una única dirección con un determinado ángulo al impactar una superficie. En nuestra propuesta se implementaron los modelos básicos para la reflexión especular, los cuales son la reflexión de Phong, la reflexión de Blinn-Phong y la reflexión de Cook-Torrance.

3.3.5.1. Reflexión de Phong

La reflexión de Phong es un modelo matemático empírico de puntos sobre una superficie. Fue desarrollado por Bui Tuong Phong en el año 1975 y publicado junto con la técnica de interpolación que lleva su mismo nombre. Phong describe la forma en que una superficie refleja la luz como la combinación de la reflexión difusa de superficies ásperas con la reflexión especular de superficies brillosas [18].

3.3.5.2. Reflexión de Blinn-Phong

Dos años más tarde en 1977, James Blinn propone una modificación que hace que el modelo de reflexión planteado inicialmente por Phong sea mucho más eficiente. En dicha modificación, Blinn propone el uso del vector de camino medio o *halfway vector* en vez del vector de reflexión planteado por Phong inicialmente para el cálculo del coeficiente especular. Buscando así, que el ángulo que se genera entre el vector de camino medio y la normal sea proporcional al ángulo que se genera entre el vector de reflexión y la normal con un menor costo computacional [19].

3.3.5.3. Reflexión de Cook-Torrance

La reflexión de Cook-Torrance es un modelo matemático que calcula la reflexión especular de una superficie que fue desarrollado por Robert L. Cook y Kenneth E. Torrance en el año 1981. El modelo de Cook-Torrance se acerca más a la realidad física que los modelos Phong y Blinn-Phong ya que plantea que una superficie está compuesta por muchas microfacetas que reflejan la luz [20].

(a) Reflexión de Phong. (b) Reflexión de Blinn-Phong. (c) Reflexión de Cook-Torrance.

Figura 3.9: Comparación entre la reflexión especular de Phong, Blinn-Phong y Cook-Torrance.

3.3.6. Técnicas de interpolación

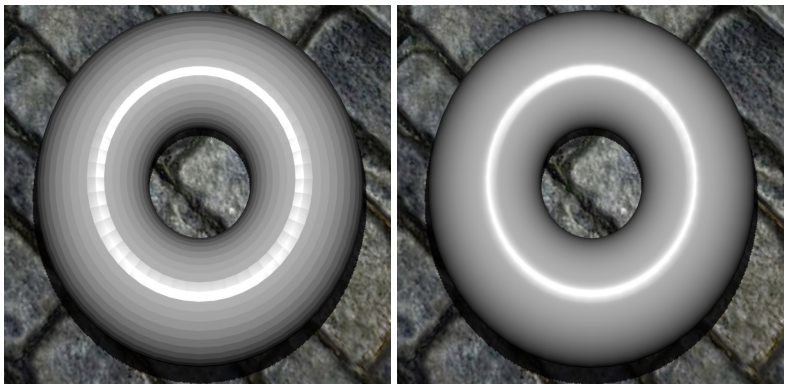
Las técnicas de interpolación son un conjunto de algoritmos para el despliegue de los objetos utilizando el hardware gráfico para que así sea posible obtener una imagen resultado cuya calidad sea visualmente aceptable [16]. En nuestra implementación, se consideraron las técnicas de sombreado planar y sombreado de Phong.

3.3.6.1. Sombreado planar

El sombreado planar o *flat shading* es una técnica de interpolación para dibujar cada polígono de un objeto, basándose en el ángulo que adopta la normal de una faceta y la dirección de la luz. Típicamente, se usa para dibujar rápidamente una superficie ya que otras técnicas de interpolación son muy costosas a nivel computacional. El resultado del sombreado planar es una superficie cuyos puntos de cada polígono estarán coloreados del mismo color, observándose la diferencia entre facetas adyacentes. Un ejemplo del dibujado haciendo uso del sombreado planar se puede apreciar en la Figura 3.10(a).

3.3.6.2. Sombreado de Phong

El sombreado de Phong o *phong shading* es una técnica de interpolación propuesta por Bui Tuong Phong en el año 1975. El sombreado de Phong posee una mejor calidad visual pero es más costosa que el sombreado planar ya que la interpolación de los colores de una faceta se lleva a cabo por píxel [18]. El resultado del sombreado de Phong se puede ver en la Figura 3.10(b) previamente expuesta en la reflexión especular.



(a) Sombreado planar. (b) Sombreado de Phong.

Figura 3.10: Comparación entre el sombreado planar y el sombreado de Phong.

3.4. Texturizado

En informática gráfica una textura es una imagen que es capturada por una cámara, dibujada por un artista o generada mediante algoritmos. Las texturas están compuestas por *texels*[1], los cuales frecuentemente contienen valores de color. Sin embargo, es de mucha utilidad considerar meramente a una textura como una tabla de valores que puede ser consultada por el hardware gráfico y ser usada para cualquier propósito que se desee. En nuestra propuesta, se implementan varias técnicas de texturizado las cuales son: texturas 2D, mapeado de ambiente y mapeado de normales.

[1]Interpretando una textura como una tabla de valores, un *texel* se define como una entrada en dicha tabla, donde su contenido es un valor de color.

3.4.1. Texturas 2D

La técnica de texturas 2D consiste en tener un objeto 3D que posea una textura 2D asociada a él y un conjunto de coordenadas de textura, donde cada una le corresponde a un vértice de dicho objeto. La idea consiste en muestrear la textura usando dichas coordenadas correspondientes a cada vértice en relación uno a uno o *mapping* para así obtener los valores de color que tienen los *texels* y asignárselos al píxel que percibe el vértice [13, 16]. Un ejemplo de las texturas 2D se puede apreciar en la Figura 3.11.

Figura 3.11: Una textura 2D aplicada a un toro.

3.4.2. Mapeado de ambiente

El mapeado de ambiente o *environment mapping* es una técnica que se encarga de tomar el ambiente y almacenarlo en una representación intermedia, donde dicha representación puede ser una esfera, una elipse o un cubo. En nuestra propuesta, se usa el enfoque del cubo para para la representación previamente mencionada usando *cubemaps*[1]. En esta técnica, se asume que el observador se encuentra en el centro del cubo y desde esta perspectiva, observa el ambiente que lo rodea, hecho que se puede apreciar en la Figura 3.12, con el mapeado de ambiente se logran los efectos de reflexión y refracción [16, 21].

[1]Un *cubemap* es un conjunto de seis texturas que hacen analogía a un cubo, las cuales poseen una perspectiva del ambiente proyectado en cada una de ellas.

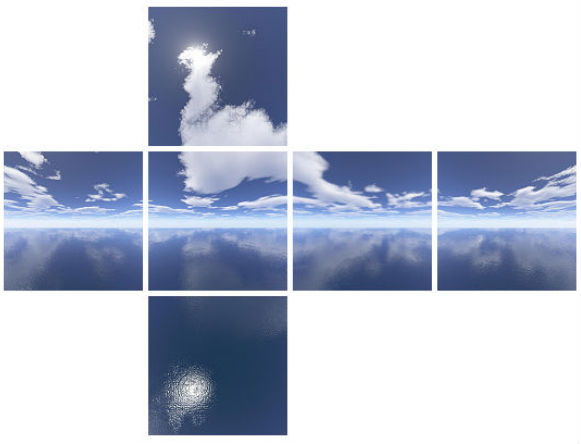


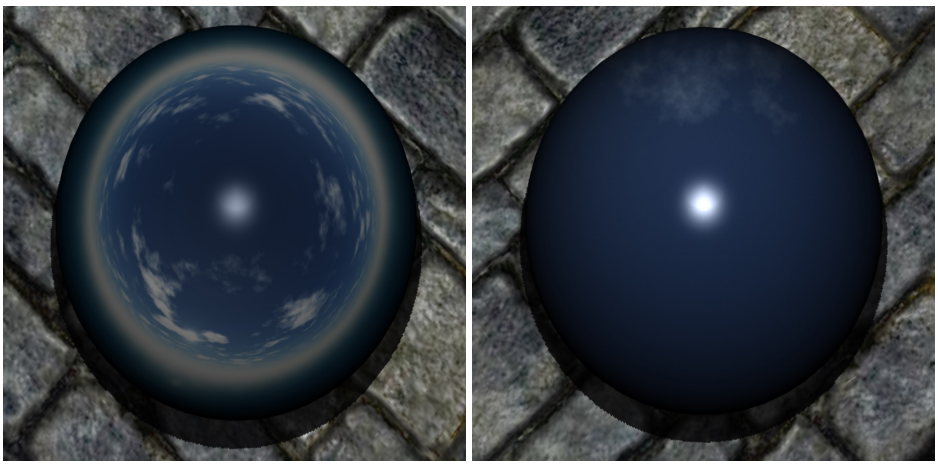
Figura 3.12: Representación de un *cubemap* [4].

3.4.2.1. Efecto de reflexión

Consiste en simular un objeto con la propiedad de reflejar el ambiente que lo rodea. Es importante destacar que en la vida real no existen objetos que posean una propiedad perfectamente reflectiva, sin embargo, para el caso donde existe un modelo que considera una reflexión difusa o especular perfecta, por cada vértice que yace sobre una superficie se debe calcular un vector de reflexión, hecho que se puede apreciar en la Figura 3.13(a).

3.4.2.2. Efecto de refracción

Consiste en simular un objeto con la propiedad de refractar el ambiente que lo rodea. Se presenta cuando la luz pasa de un medio que posee un índice refractario a otro que posee uno diferente, por ejemplo, del aire al vidrio. La idea es trazar rayos desde el ojo, haciéndolos pasar a través de una superficie para finalmente tomar la intersección del rayo con el ambiente como el color final, el resultado se puede ver en la Figura 3.13(b).



(a) Efecto de reflexión. (b) Efecto de refracción.

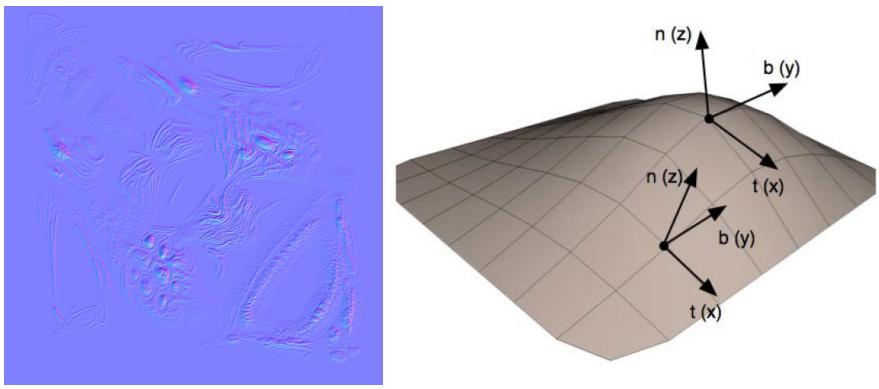
Figura 3.13: Comparación entre los efectos de reflexión y refracción.

3.4.3. Mapeado de normales

El mapeado de normales o *normal mapping* es una técnica que se encarga de producir perturbaciones ficticias sobre una superficie. Esta técnica es útil para producir efectos como abolladuras, asperezas o arrugas sin tener directamente algún tipo de información adicional acerca de la nueva ubicación que deben tener los vértices que yacen sobre una superficie para generar este tipo de perturbaciones. Para ello, se debe llevar a cabo una alteración sobre los vectores normales de la superficie usando un mapa de normales [16].

Un mapa de normales o *normal map* es una textura cuyos *texels* contienen información que se debe interpretar como vectores normales en vez de colores. Dichos vectores se codifican haciendo uso de los tres canales de color rojo, verde y azul o RGB, donde cada canal de color hace analogía a las componentes de cada vector normal. Un ejemplo de un mapa de normales se puede apreciar en la Figura 3.14(a).

Adicionalmente, un mapa de normales es un conjunto de vectores que pertenecen al espacio tangente o *tangent space*. En dicho espacio, cada vértice de la superficie representa el origen de un sistema de referencia particular asociado a cada uno de ellos. En este sistema de referencia por cada vértice, la normal representa el eje Z, el vector tangente representa el eje X y finalmente el vector binormal representa el eje Y. Un ejemplo gráfico del espacio tangente se puede ver en la Figura 3.14(b).



(a) Mapa de normales [16]. (b) Espacio tangente [16].

Figura 3.14: Ejemplos de un mapa de normales y el espacio tangente.

Cuando se lleva a cabo el proceso de dibujado de un objeto donde se aplica la técnica del mapeado de normales, su superficie real es totalmente lisa, pero tiene un aspecto áspero gracias a las alteraciones de los vectores normales. En nuestra propuesta se implementó el mapeado de normales como solución al problema de simular superficies con rugosidad, un ejemplo del resultado lo podemos ver en la Figura 3.15.

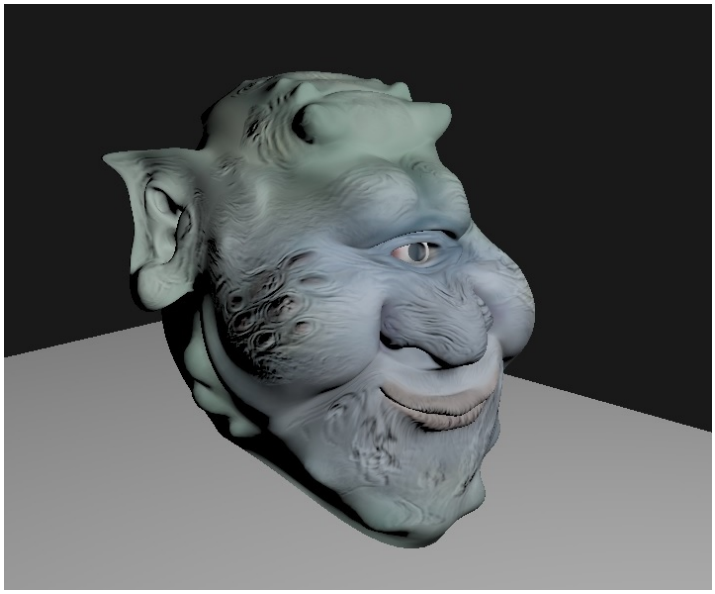


Figura 3.15: Efecto generado por el mapeado de normales.

3.5. Sombras

Como se explicó en la sección 3.3.2.3, las sombras representan un área de oscuridad que se hace presente cuando un objeto es alcanzado por la luz y este ocluye a otros, para así aportar información que ayuda a comprender la relación espacial entre los objetos. Adicionalmente, las sombras no son totalmente uniformes en oscuridad, hay porciones de área en las cuales la sombra es más oscura denominada *umbra* y donde la sombra se hace mas clara por la transición hacia una zona iluminada que se conoce como *penumbra* [3, 22].

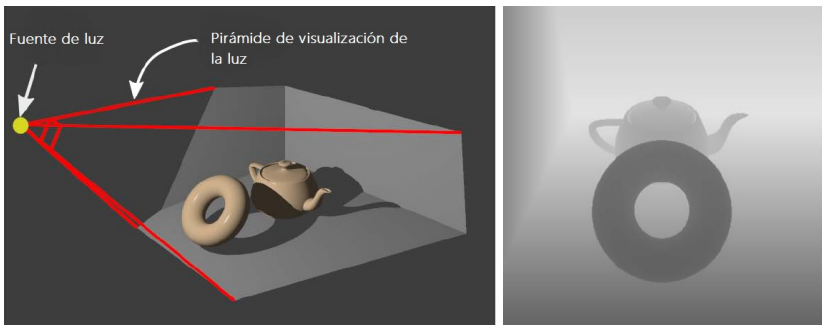
En nuestra propuesta para lograr obtener el efecto de sombra, se implementaron sombras duras utilizando la técnica del mapeado de sombras. Adicionalmente, se estableció una clasificación entre los objetos que se encuentran expuestos a una fuente de luz, esto, para darle la potestad al usuario de tomar la decisión acerca de los objetos que pueden dar sombra y cuales recibirla. Dicha clasificación permitió dividir los objetos en dos conjuntos, uno de ellos son los receptores o *receivers* que son los objetos que se encuentran en sombra. Por otro lado, se encuentran los objetos emisores o *casters* que son aquellos que bloquean la luz de los receptores.

3.5.1. Mapeado de sombras

Es una técnica para la producción de sombras que fue propuesta por Lance Williams en el año 1978. El mapeado de sombras posee dos pasadas y usa texturas que almacenan la profundidad desde un punto de vista específico [23].

En la primera pasada, la escena se dibuja desde el punto de vista de la fuente de luz, consecuentemente, la información de profundidad es almacenada en una textura denominada mapa de sombra o *shadow map*, que va a proveer de información acerca de la visibilidad de los objetos desde la perspectiva previamente mencionada. Así, para cada punto de la escena que tenga valores de profundidad más cercanos a la fuente de luz de los que tiene almacenados un mapa de sombras estarán iluminados, en caso contrario se encontrarán en sombra. Un ejemplo de la primera pasada se puede apreciar en las Figuras 3.16(a) y 3.16(b).

En la segunda pasada, la escena es dibujada normalmente, pero la profundidad de cada fragmento debe ser primero comparada contra el mapa de sombra para determinar si está o no en sombra. El fragmento entonces será coloreado de una forma diferente dependiendo del resultado de la prueba.



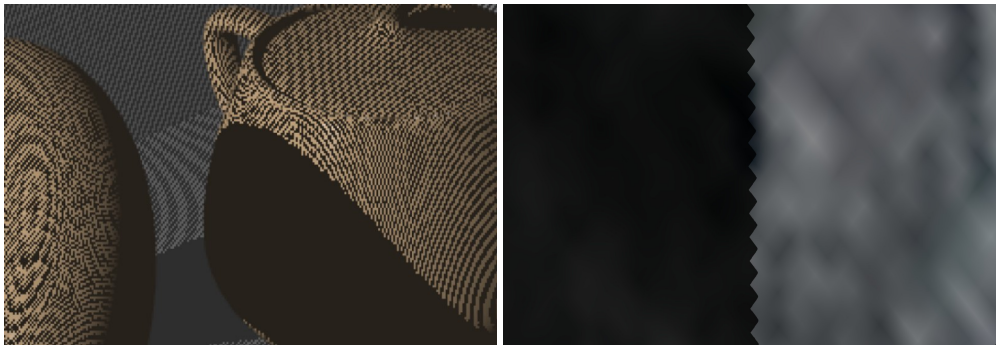
(a) Punto de vista de la fuente de luz [16]. (b) Mapa de sombra [16].

Figura 3.16: Primera pasada de la técnica de mapeado de sombras.

Es importante destacar que el mapeado de sombras posee dos enfoques, donde ambos son implementados en nuestra propuesta. Ellos son:

- Direccional: En este enfoque solo se consideran luces direccionales y reflector. En éste, se debe generar un mapa de sombras desde la perspectiva de cada fuente de luz, dicho mapa de sombras será una textura 2D.
- Omnidireccional: Este es el enfoque en el que consideran solamente luces puntuales. En éste, se debe generar dinámicamente por cada fuente de luz puntual un *cubemap* de sombra, para así garantizar una perspectiva completa del ambiente que la rodea.

Los principales problemas que tiene el mapeado de sombras son el acné de sombra o *shadow acne* que representa una fluctuación entre los píxeles que se encuentran iluminados y los que se encuentran en sombra por falta de precisión de la aritmética punto flotante en los valores de profundidad almacenados en el mapa de sombra y el *aliasing* que hace que la sombra no tenga bordes lisos sino escalonados. Ambos problemas se pueden apreciar en las Figuras 3.17(a) y 3.17(b).



(a) Acné de sombra [16]. (b) Aliasing.

Figura 3.17: Principales problemas del mapeado de sombras.

3.6. Selección y descarte de geometría

Como se mencionó en el capítulo anterior, la selección y descarte de geometría limita el número de primitivas geométricas a ser desplegadas. La geometría que puede ser obviada es aquella que no es visible en la escena que se desplegará [24]. En nuestra propuesta se utilizan las siguientes técnicas de descarte.

3.6.1. Descarte de caras traseras

El descarte de caras traseras o *backface culling* es una técnica de descarte basada en la selección de aquellos polígonos que están de frente al ojo, el resto de los polígonos son descartados y por lo tanto no son desplegados. El descarte de caras traseras no puede ser aplicado a cualquier tipo de objeto, ya que en muchas ocasiones existen polígonos que no están en frente al ojo y que en realidad son visibles. Es por esto que para obtener buenos resultados es necesario que los objetos sean cerrados y preferiblemente convexos [25]. Para determinar cuáles polígonos están de frente al ojo se emplean las normales de cada polígono del objeto.

3.6.2. Descarte por pirámide de visualización

Una de las maneras más sencillas de hacer descarte es aplicando la técnica de descarte por pirámide de visualización o *frustum culling*. El criterio para decidir si un objeto se debe descartar es simple. Aquellas geometrías que estén completamente fuera de la pirámide de visualización son obviadas (ver Figura 3.18). Las geometrías que están completamente dentro de la pirámide deber ser desplegadas (al menos que sean descartadas por otra técnica de descarte) y las geometrías que están parcialmente adentro de la pirámide generalmente se coloca en la lista de los objetos que están completamente adentro y se le designa al hardware la tarea de cortar la sección que no puede ser vista. Para determinar si un objeto está completamente dentro de la pirámide de visualización es utilizado un volumen delimitador o *bounding volume*, el cual usualmente es una caja o una esfera [11].

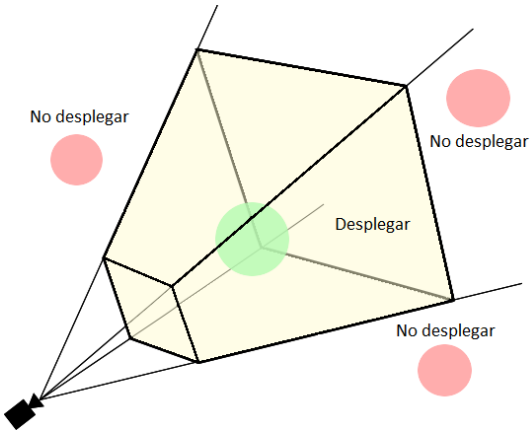


Figura 3.18: Ejemplo de descarte por pirámide de visualización [5].

El hardware también puede descartar secciones que no serán vistas en la pantalla, el problema yace en procesar cada vértice de la geometría de un objeto y en muchos casos ésta tiene miles de vértices que luego serán descartados. Esto ocurre en el caso que no se use el descarte por pirámide de visualización el cual procesa solo los límites del objeto para descartarlo.

3.7. Grafo de escena

Un grafo de escena o *scene graph* es un grafo dirigido acíclico. Los arcos del grafo definen dependencias del nodo hijo respecto al padre, de modo que la aplicación de una transformación en un nodo padre hace que se aplique a todos sus nodos hijos [25].

La raíz del grafo de escena es comúnmente representada con un nodo abstracto que proporciona un punto de partida conocido para acceder a la escena. Gracias al grafo, es posible especificar fácilmente transformaciones de escenas complejas de forma relativa a los elementos padre de la jerarquía. En la Figura 3.19, las transformaciones que se apliquen al nodo Edificio se aplicarán a su nodo hijo Piso y a su vez a sus nodos hijos Pasillo, Apartamento y Ascensor. Sin embargo, las transformaciones que se apliquen sobre el nodo Cielo solo lo afectaran a él ya que no posee nodos hijos.

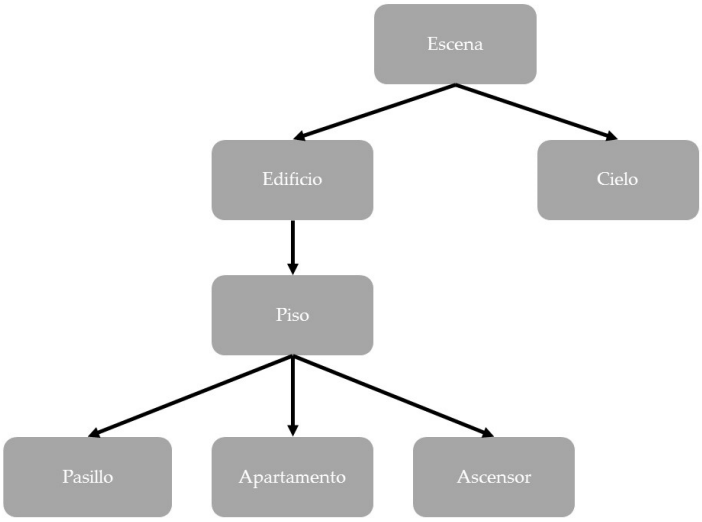


Figura 3.19: Ejemplo de un grafo de escena.

Nuestra propuesta emplea el grafo de escena como estructura de datos para la representación de relaciones jerárquicas en el despliegue de los objetos que conforman la escena.

4

Implementación

4.1. EZ3.js

El motor de juego desarrollado fue pensado primordialmente para ser de fácil uso y aprendizaje. En el idioma inglés fácil o *easy* se suele abreviar coloquialmente como *ez*. Adicionalmente, el motor se enfoca en el despliegue 3D y el código fuente está escrito en su totalidad en JavaScript, el cual es abreviado como js. La unión de estas características en dicho orden dio origen a su nombre, EZ3.js.

De esta forma, EZ3.js está orientado a los programadores que desean un motor de fácil uso y con una buena documentación, proveyéndoles una infraestructura sencilla para elaborar videojuegos en un entorno web y abstrayéndolos de la complejidad de bajo nivel de interactuar directamente con WebGL. El código fuente se encuentra actualmente publicado en un repositorio en GitHub como de código abierto bajo la licencia MIT [26].

La licencia MIT es una de tantas licencias de software que ha empleado el Instituto Tecnológico de Massachusetts o *Massachusetts Institute of Technology* a lo largo de su historia, la cual es completamente permisiva y sin protección heredada, lo que favorece enormemente la retroalimentación y el aporte de la comunidad.

Para el manejo de versiones de dicho repositorio se empleó Git. Git es un software de control de versiones, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente [27].

4.2. Estilo de programación y documentación

EZ3.js emplea tanto un estilo de programación como una documentación de su API orientada a objetos. El estilo de programación de dicho motor se encuentra basado en la guía de estilo de JavaScript Airbnb, donde se definen un conjunto de convenciones que cubren de forma general el nombramiento y declaración de variables y funciones, el uso de espacios en blanco, identación y comentarios, practicas y principios de programación [28]. Adicionalmente, se definieron las convenciones descritas a continuación.

- Tanto los nombres de variables y funciones, como la descripción de notificaciones, comentarios y documentación deben estar en inglés.
- Toda definición de variable o función se debe encontrar dentro del espacio de nombres o *namespace* EZ3, el cual es representado como un objeto.
- Los nombres de los miembros o métodos de las clases que se consideren de acceso privado, deben comenzar con un guión bajo.
- Las lineas de código no deben poseer más de ciento veinte caracteres.

Las convenciones impuestas tienen como objetivo mejorar sustancialmente la lectura del código fuente y por ende su mantenimiento y escalabilidad. Sin embargo, para asegurar que la mayoría de éstas se cumplan, se empleó un programa denominado JSHint, específicamente un *plug-in* de Atom llamado atom-jshint.

Atom es el editor de texto que se usó a lo largo de toda la realización del código fuente del motor, debido a su gran versatilidad, control de versiones y manejo de *plug-ins* [29]. JSHint es una herramienta impulsada por la comunidad de desarrolladores para detectar errores y problemas potenciales en un código JavaScript, los cuales pueden ir desde un error de sintaxis hasta una falta a una convención o regla [30]. Esta herramienta posee un conjunto de reglas especificas la cuales tienen unos valores asignados por defecto que pueden ser sustituidos.

La ejecución de JSHint es delagada al *plug-in* de Atom, entonces la forma de realizar la sustitución es mediante un archivo de configuración en formato JSON[1], el cual se debe denominar .jshintrc.json, éste es detectado y vinculado automáticamente por el editor. Es importante destacar que el alcance de las reglas impuestas por dicho archivo, se aplican recursivamente sobre todos los archivos JavaScript en el mismo directorio y subdirectorios a partir de donde se encuentre. En el caso que se encuentren varios archivos de configuración en diferentes directorios, se tomará en cuenta el más cercano al archivo a editar. Opcionalmente, en cada archivo se pueden colocar valores de reglas particulares para el mismo, sustituyendo cualquier otro. No obstante, en el motor sólo se encuentra un archivo de configuración ubicado en la raíz

[1]JSON o *JavaScript Object Notation* es un subconjunto de la notación literal de objetos de JavaScript.

del proyecto, una fracción de su contenido se puede observar en el Código 1.

```
{
  "browser": true,
  "devel": true,
  "indent": 2,
  "maxlen": 120,
  "camelcase": true,
  "undef": true,
  "unused": true,
  "quotmark": "single"
}
```

Código 1: Fracción del contenido del archivo .jshintrc.json de EZ3.js.

La documentación de la API del motor es creada empleando YUI Doc, el cual es una aplicación que a partir de comentarios en el código fuente usando una sintaxis especializada genera dicha documentación en un formato web (HTML y CSS) [31]. La generación de la documentación es realizada en el proceso de compilación del proyecto, expuesto en la siguiente sección.

4.3. Compilación

En EZ3.js, la compilación es un proceso comprendido por el empaquetado, oscurecimiento y generación de documentación del código fuente del motor. Realizar este proceso manualmente resulta ser bastante tedioso por lo que fue automatizado mediante el uso de un manejador de tareas denominado Grunt [32]. Éste es instalado y controlado mediante el manejador de paquetes de Node.js (*Node.js package manager* o npm).

Node.js es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor (pero no limitándose a ello) basado en el lenguaje de programación JavaScript, asíncrono, con entrada y salida de datos en una arquitectura orientada a eventos que emplea el motor V8 de Google [33]. Node.js incorpora varios módulos básicos compilados en el propio binario (por ejemplo el módulo de red, que proporciona una capa para programación de red asíncrona y otros módulos fundamentales). Es posible utilizar módulos desarrollados por terceros, ya sea como archivos precompilados, o como archivos en JavaScript plano. Dichos módulos se implementan siguiendo la especificación CommonJS para módulos, utilizando una variable de exportación para dar a estos *scripts* acceso a funciones y variables implementadas por los mismos [34].

Los módulos de terceros pueden extender Node.js o añadir un nivel de abstracción, implementando varias utilidades *middleware* para utilizar en aplicaciones web. Pese a que los módulos pueden instalarse como archivos simples, normalmente se instalan utilizando npm, facilitando su compilación, instalación y actualización así como la gestión de sus dependencias.

Todos los módulos de npm contienen un archivo, usualmente en la raíz del proyecto, llamado package.json, el cual contiene diversos metadatos relevantes al proyecto. Este archivo es usado para darle a npm la información necesaria para identificar el proyecto como para manejar sus dependencias. Adicionalmente, puede contener otros metadatos como su descripción, información de su licencia, etc.

Por lo tanto, Node.js es un requisito privativo para el uso de Grunt. Y éste último es un módulo de terceros del mismo. Grunt se caracteriza particularmente por tener una gran cantidad de *plug-ins* o submódulos elaborados por sus mismos creadores o por terceros para la automatización de diversas tareas, los cuales son instalados y controlados también por npm. Las tareas a realizar y su configuración, sean propias o de terceros, se encuentran especificadas en un archivo denominado mandatoriamente gruntfile.js, en donde estas tareas actúan comúnmente sobre un conjunto de archivos y se pueden ejecutar empleando la interfaz de comando de Grunt o *Grunt's command line interface* (Grunt CLI).

En la raíz del proyecto del motor se encuentra el archivo package.json (ver Código 2) donde se especifican todas las características y dependencias de desarrollo de éste, las cuales son Grunt y los diferentes *plug-ins* empleados por el mismo. Particularmente, se desarrollaron y publicaron los módulos denominados grunt-depsconcat [35] y grunt-shdrsconcat [36], ya que no existían similares.

```
{
  "name": "ez3",
  "version": "1.0.0",
  "description": "JavaScript 3D library",
  "authors": [
    "Andres Alvarez",
    "Carlos Zapata"
  ],
  "license": "MIT",
  "devDependencies": {
    "grunt": "^0.4.5",
    "grunt-contrib-uglify": "^0.9.1",
    "grunt-contrib-watch": "^0.6.1",
    "grunt-depsconcat": "^0.1.0",
    "grunt-shdrsconcat": "^0.1.3",
    "jit-grunt": "^0.9.1",
    "load-grunt-config": "^0.17.2",
```

```
    "grunt-contrib-yuidoc": "^0.10.0"
  }
}
```

Código 2: Fracción del contenido del archivo package.json de EZ3.js.

El proceso de compilación inicia con el empaquetado que une todos los archivos JavaScript en uno denominado ez3.js, tomando en cuenta las dependencias que puedan existir entre ellos, por ejemplo por herencia. Para esto se utilizó el módulo grunt-depsconcat el cual concatena y analiza el contenido de los archivos en cuestión para determinar dependencias a través de una expresión regular configurable. También, se agregan todos los códigos fuente de los sombreadores en un objeto denominado *ShaderLibrary*, donde se distinguen por el nombre y tipo de sombreador, tomados de los archivos donde yacen los códigos, para llevar a cabo esta tarea se empleó el módulo grunt-shdrconcat. Para su posterior oscurecimiento se utilizó el módulo grunt-contrib-uglify, el cual recibe como entrada el archivo resultante del proceso de empaquetado, generando el archivo ez3.min.js. Finalmente, la documentación es generada con un *plug-in* de Grunt que emplea YUI Doc denominado grunt-contrib-yuidoc.

El módulo jit-grunt es empleado para la inclusión automática de los *plug-ins* de Grunt, haciendo que no se tengan que incluir en el gruntfile.js.

Las dos tareas principales elaboradas en el gruntfile.js son la depuración y el lanzamiento. La primera se encarga de hacer una ejecución parcial del proceso de compilación donde se realiza el empaquetado cada vez que alguno de los archivos que conforman el código fuente se ve alterado, para lo cual se uso grunt-contrib-watch y la segunda realiza el proceso de compilación completo.

En resumen, para poder ejecutar las tareas de depuración o lanzamiento es necesario seguir los siguientes pasos:

- Clonar el repositorio de GitHub de EZ3.js.
- Instalar Node.js.
- Instalar Grunt CLI.
- Instalar las dependencias de desarrollo.
- Ejecutar vía Grunt CLI la tarea *debug* en caso de depuración o *release* en caso de lanzamiento.

4.4. Metodología

Para llegar a la solución de un problema primero se debe de seguir un proceso, en este contexto, está el desarrollo de software como el proceso y el producto de software como la solución. El proceso para el desarrollo de software es un conjunto de actividades en una forma deliberada, estructurada y metódica, requiriendo cada etapa del desarrollo desde la formación de la idea hasta la entrega del producto final. El proceso de creación de software puede llegar a ser muy complejo, por esta razón, se debe utilizar una metodología que implica identificar los requerimientos, planificar, diseñar, documentar, implementar y probar el software.

Específicamente, para este trabajo se decidió utilizar una metodología Ad-Hoc que define una serie de iteraciones basadas en las fases de la metodología ágil Programación Extrema (XP) [37] y donde cada una de ellas representa un incremento o modificación en las clases de EZ3.js.

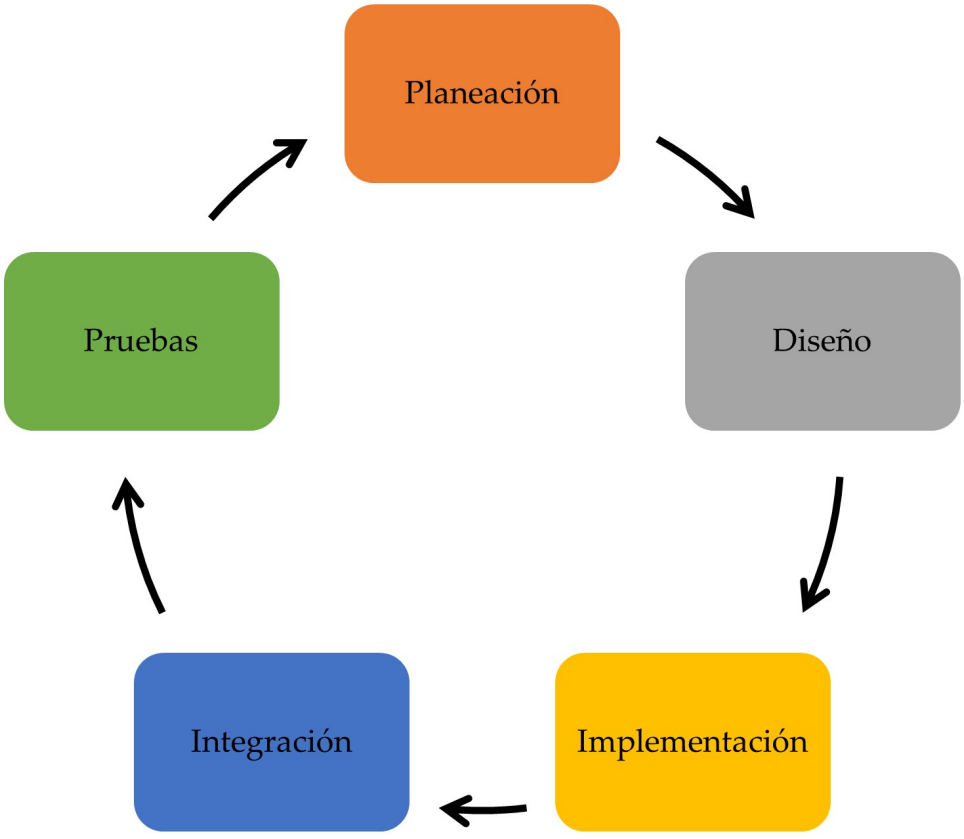


Figura 4.1: Ciclo de desarrollo de EZ3.js.

En la Figura 4.1, se presenta el ciclo de desarrollo de EZ3.js. Este ciclo esta compuesto de varias fases que se describen a continuación:

- Para la planeación se investigó y se definió cual sería la clase a agregar y por qué debía ser agregada.
- Con el diseño se modelaron los atributos y métodos que iba a poseer la clase.
- En la implementación se llevó a cabo el desarrollo de la clase.
- Para la integración se crearon las instancias de la clase que fueran necesarias.
- Finalmente para las pruebas, se realizaron sobre las instancias pruebas de integración.

4.5. Módulos

Las clases de EZ3.js se agruparon en módulos en base a su descripción y funcionamiento fundamentados en la arquitectura propuesta. Los módulos que se contemplaron son listados a continuación y serán explicados en las secciones subsiguientes.

- Sistema
- Matemáticas
- Entidad y escena
- Cámaras
- Texturas
- *Framebuffers*
- Luces
- Geometría
- Materiales
- Malla
- Entrada
- Controles
- Carga
- Motor y manejador de pantallas
- Despliegue

4.6. Sistema

Tener conocimiento del sistema donde se ejecuta un videojuego es fundamental, ya que permite realizar acciones en basa a su descripción.

El objeto *Device* es el encargado de detectar cuando el dispositivo se encuentra listo para ser usado. Cuando sea así, se ejecutan un conjunto de funciones que se encargan de almacenar las características del dispositivo como el sistema operativo y el navegador. Igualmente se encarga de estandarizar los nombres de los eventos y funciones del navegador empleados por el motor, ya que suelen variar dependiendo del mismo.

En el Código 3 se puede apreciar la función encargada de estandarizar los nombres de las funciones del navegador que permiten ejecutar y cancelar un ciclo de animación denominadas *requestAnimationFrame* y *cancelAnimationFrame* respectivamente.

```
function checkAnimationFrame() {
    if (window.requestAnimationFrame) {
      that.requestAnimationFrame = 'requestAnimationFrame';
      that.cancelAnimationFrame = 'cancelAnimationFrame';
    } else if (window.webkitRequestAnimationFrame) {
      that.requestAnimationFrame = 'webkitRequestAnimationFrame';
      that.cancelAnimationFrame = 'webkitCancelAnimationFrame';
    } else if (window.mozRequestAnimationFrame) {
      that.requestAnimationFrame = 'mozRequestAnimationFrame';
      that.cancelAnimationFrame = 'mozCancelAnimationFrame';
    } else if (window.msRequestAnimationFrame) {
      that.requestAnimationFrame = 'msRequestAnimationFrame';
      that.cancelAnimationFrame = 'msCancelAnimationFrame';
    } else if (window.oRequestAnimationFrake) {
      that.requestAnimationFrame = 'oRequestAnimationFrame';
      that.cancelAnimationFrame = 'oCancelAnimationFrame';
    }
}
```

Código 3: Función *checkAnimationFrame*.

Es importante destacar que si algún evento no es soportado por el navegador donde se ejecuta el videojuego, por ejemplo los eventos de pantalla táctil en un dispositivo que no cuente con una, serán almacenados como nulos denotando su inexistencia.

La mayoría de los navegadores HTML5 han implementado las nuevas funciones de animación mencionadas anteriormente, siendo *requestAnimationFrame* una mejor alternativa que las funciones *setTimeout* y *setInterval* para realizar animación y despliegue, ya que fue diseñada específicamente para eso. Dicha función se ejecuta únicamente cuando el navegador posee el foco, ahorrando recursos cuando no es así. Empleándola se puede obtener el mejor rendimiento que el navegador puede otorgar para dichas tareas.

La clase *AnimationFrame* se encarga de proporcionar un ciclo de animación y despliegue usando la función *requestAnimationFrame* por defecto. En caso de ésta no estar disponible o no querer usarla, es sustituida por *setTimeout*, la cual es soportada por más navegadores que la anterior función.

Conocer el tiempo en que tarda el ciclo de animación y despliegue en cada iteración o *elapsed time* es de suma importancia, ya que es empleado para que las animaciones sean consistentes en diferentes dispositivos. La actualización de dicho tiempo, como el trascurrido desde el inicio del ciclo hasta la iteración actual y el anterior es provisto por la clase *Time*.

4.7. Matemáticas

En un motor de juego es necesario que exista una biblioteca de matemáticas debido a la naturaleza del mismo. EZ3.js provee un conjunto de clases que representan diferentes entidades matemáticas y un objeto llamado *Math* que ofrece diversas funciones y constantes. Las clases que provee son:

- *Euler*, representa los ángulos de Euler de tipo real.
- *Matrix3*, representa una matriz 3x3 de tipo real.
- *Matrix4*, representa una matriz 4x4 de tipo real.
- *Quaternion*, representa un cuaternión de tipo real.
- *Vector2*, representa un vector 2D de tipo real.
- *Vector3*, representa un vector 3D de tipo real.
- *Vector4*, representa un vector 4D de tipo real.
- *Box*, representa una caja.
- *Sphere*, representa una esfera.
- *Plane*, representa un plano.
- *Frustum*, representa una pirámide de visualización.

Las clases mencionadas previamente poseen métodos para llevar a cabo sus operaciones matemáticas correspondientes como la suma, resta, multiplicación, asignación de valores, verificación de igualdad, etc. También, poseen métodos de conversión que se encargan de llevar el tipo de una instancia de una clase a otro, por ejemplo, *toVector3* que transforma el tipo de una instancia de *Vector4* a uno de *Vector3*, *toString* a una cadena de caracteres y *toArray* a un arreglo.

4.8. Entidad y escena

Una entidad en EZ3.js es un nodo de un grafo escena representado por la clase *Entity*. Esta clase posee un conjunto de atributos que constituyen las transformaciones a aplicar en una instancia de la misma, las cuales son traslación, rotación y escalamiento.

Los atributos que representan la traslación y escalamiento son de tipo *Vector3* donde sus componentes simbolizan dichas transformaciones en cada uno de ellos respectivamente. Pese a todos los beneficios que proveen los cuaterniones con respecto a los ángulos de Euler como por ejemplo, que no ocurra *gimbal lock*[1], éstos suelen ser mucho más difíciles de comprender y manipular. Por ende, la rotación es parametrizada empleando ambos, mediante los atributos de tipo *Euler* y *Quaternion*. Debido a su relación de dualidad, si uno de éstos atributos varia, el otro se debe actualizar haciendo la conversión respectiva provista por estos mismos.

A partir de las transformaciones descritas, se construye un atributo de tipo *Matrix4* que lleva las coordenadas de los vértices de su propio sistema de referencia al de la escena, conocida comúnmente como matriz de mundo o *world matrix*. No obstante, para el cálculo final de dicha matriz se considera la matriz de mundo del padre, a la cual se accede mediante la referencia hacia el nodo padre.

En el Código 4 se muestra el método usado para la actualización de la matriz de mundo de un nodo, para ello primero se verifica si alguno de sus atributos de transformación ha sido modificado, de ser así, se procede al cálculo de la matriz de modelo o *model matrix*, denotada por un atributo de tipo *Matrix4*, que almacena únicamente las transformaciones locales al nodo. Para ello se emplea el método *compose* de éste, el cual recibe como parámetros sus atributos de transformación excluyendo los ángulos de Euler, por lo que se evitan los problemas relacionados a los mismos. Posteriormente, se verifica si la matriz de mundo de su padre cambió y en ese caso, se premultiplica con la matriz de modelo. En caso contrario, a la matriz de mundo se le asigna la de modelo.

```
EZ3.Entity.prototype.updateWorld = function() {
  var scaleDirty = false;
  var modelDirty = false;
  var positionDirty = false;
  var quaternionDirty = false;
  var parentWorldDirty = false;

  if (this.position.isDiff(this._cache.position)) {
    this._cache.position = this.position.clone();
```

[1]El *gimbal lock* es un fenómeno que ocurre en las rotaciones 3D cuando se pierde un grado de libertad de rotación.

```
    positionDirty = true;
  }

  if (this.quaternion.isDiff(this._cache.quaternion)) {
    this._cache.quaternion = this.quaternion.clone();
    quaternionDirty = true;
  }

  if (this.scale.isDiff(this._cache.scale)) {
    this._cache.scale = this.scale.clone();
    scaleDirty = true;
  }

  if (positionDirty || quaternionDirty || scaleDirty)
    this.model.compose(this.position, this.quaternion, this.scale);

  if (!this.parent) {
    modelDirty = this.model.isDiff(this._cache.model);

    if (modelDirty) {
      this.world = this.model.clone();
      this._cache.model = this.model.clone();
    }
  } else {
    modelDirty = this.model.isDiff(this._cache.model);
    parentWorldDirty = this.parent.world.isDiff(this._cache.parentWorld);

    if (parentWorldDirty || modelDirty) {
      if (modelDirty)
        this._cache.model = this.model.clone();

      if (parentWorldDirty)
        this._cache.parentWorld = this.parent.world.clone();

      this.world.mul(this.parent.world, this.model);
    }
  }
};
```

Código 4: Método *updateWorld* de la clase *Entity*.

Para recorrer el grafo de escena a partir de un nodo se emplea el método *trasverse*.

En el Código 5 se muestra el método *updateWorldTraverse*, donde se puede apreciar un ejemplo de su funcionamiento para actualizar las matrices de mundo.

```
EZ3.Entity.prototype.updateWorldTraverse = function() {
  this.traverse(function(entity) {
    entity.updateWorld();
  });
};
```

Código 5: Método *updateWorldTraverse* de la clase *Entity*.

La clase *Entity* es de naturaleza no desplegable. Sin embargo, por medio de la herencia se definen un conjunto de clases con fines específicos, que pueden o no ser desplegables. Estas clases determinan el tipo de nodo, siendo los de tipo *Mesh* los únicos desplegables. La clase *Scene* no posee lógica adicional a la que hereda de la clase *Entity*, siendo los nodos de su tipo los empleados comúnmente como raíces del grafo.

4.9. Cámaras

En informática gráfica una cámara es el punto desde donde se observa una escena acotado por un volumen de visualización. La clase *Camera* se encarga de la representación y actualización de dos matrices fundamentales para el proceso de despliegue las cuales son la matriz de vista y la de proyección, así como también del mismo volumen de visualización simbolizado con un objeto del tipo *Frustum* y calculado a partir de dichas matrices.

La matriz de vista define la posición y orientación de la cámara respecto a la escena. Como la clase *Camera* hereda de la clase *Entity* posee por lo tanto una matriz de mundo asociada. La matriz de vista es calculada a partir de la matriz de mundo. Por otra parte, la matriz de proyección define el volumen de visualización, el cual depende del modelo de proyección a emplear.

Existen diferentes modelos de proyección de los objetos sobre el plano de visualización. Uno de ellos es la proyección de los objetos empleando líneas paralelas sobre el plano de proyección, mediante la denominada proyección paralela. En este modo de proyección se conservan las proporciones relativas entre objetos, independientemente de su distancia.

Mediante la proyección perspectiva se proyectan los puntos hasta el plano de visualización empleando trayectorias convergentes en un punto. Esto hace que los objetos situados más distantes del plano de visualización aparezcan más pequeños en

la imagen resultante. Las escenas generadas utilizando este modelo de proyección son más realistas, ya que así es como el ojo humano y las cámaras físicas forman imágenes.

Las clases *OrthographicCamera* y *PerspectiveCamera* heredan de *Camera* y se distinguen por el uso de diferentes modelos de proyección. La primera clase emplea un modelo de proyección paralelo u ortogonal y la segunda uno perspectivo.

Ambas clases poseen métodos para la actualización de las matrices de vista y proyección, donde son calculadas únicamente cuando alguno de los valores que las describen es modificado. La función de actualización de la matriz de proyección depende del tipo de cámara, y la de vista es común para ambos tipos.

4.10. Texturas

Una textura en EZ3.js es toda instancia cuya clase hereda de *Texture*. La clase *Texture* se encarga de gestionar los atributos que poseen en general todos los distintos tipos de textura que provee el motor, tales como un identificador único por instancia que es generado por WebGL, filtros para la magnificación y minificación, generación de *mipmaps*[1] y patrones de muestreo. Por esto, se llevó a cabo la implementación de cuatro clases diferentes llamadas *Texture2D*, *Cubemap*, *TargetTexture2D* y *TargetCubemap* que heredan de *Texture* y cumplen funciones específicas.

Las clases *Texture2D* y *Cubemap* representan una textura 2D y un *cubemap* respectivamente y con ellas se logran las técnicas de texturas 2D, mapeado de ambiente y mapeado de normales. Por otro lado, las clases *TargetTexture2D* y *TargetCubemap* son *render targets*[2] que heredan de *Texture2D* y *Cubemap*, ellas son utilizadas para llevar a cabo la técnica del mapeado de sombras utilizando *framebuffers*.

4.11. *Framebuffers*

Un *framebuffer* en EZ3.js es una clase que se encarga de gestionar el almacenamiento y manipulación de un cuadro intermedio resultante del proceso de despliegue. Dicho cuadro se genera fuera de pantalla (*offscreen*) y posteriormente su información de color es almacenada en una región de memoria. Específicamente, está compuesto por un identificador único por instancia que es generado y utilizado por WebGL para

[1] Los *mipmaps* de una textura es una secuencia de texturas que van desde una resolución alta hasta una baja de la misma.

[2] Un *render target* es una región de memoria donde se lleva a cabo el almacenamiento de un cuadro intermedio.

llevar a cabo las operaciones de vinculación y actualización. Además, posee una textura que representa un *render target* y una resolución que determina las dimensiones del mismo.

En EZ3.js se utilizan los *framebuffers* para llevar a cabo la técnica del mapeado de sombras mencionada en la sección 3.5.1 en sus dos enfoques, para ello, se poseen dos clases que heredan de la clase *framebuffer* las cuales son:

- *DepthFramebuffer*: almacena el mapa de sombra desde la perspectiva de una fuente de luz direccional o reflector usando una instancia de la clase *TargetTexture2D* para el caso del mapeado de sombras direccional.
- *DepthCubeFramebuffer*: almacena los mapas de sombra desde la perspectiva de una fuente de luz puntual, utilizando una instancia de la clase *TargetCubemap* para el caso del mapeado de sombras omnidireccional.

4.12. Luces

EZ3.js provee tres clases que implementan los distintos tipos de luz que son *PointLight*, *DirectionalLight* y *SpotLight* que representan una luz puntual, direccional y reflector respectivamente. Sin embargo, a pesar de ser distintas, poseen atributos en común que adquieren por herencia de la clase *Light*. La clase *Light* posee como atributos un color difuso y especular que aportan por canal de color la contribución de reflexión difusa y especular respectivamente y dos factores numéricos, donde uno controla el grado de oscuridad que van a tener las sombras de los objetos y el otro ayuda a evitar el problema de *aliasing* que posee la técnica del mapeado de sombras.

Como se requiere llevar a cabo un proceso de despliegue desde la perspectiva de una fuente de luz para producir sombras, una luz debe poseer un *framebuffer*, una matriz de vista y una matriz de proyección. En consecuencia, las clases asociadas a cada tipo de luz van a heredar de *OrthographicCamera* o de *PerspectiveCamera* dependiendo del comportamiento de cada una de ellas.

Las clases *PointLight* y *SpotLight* heredan de la clase *PerspectiveCamera* porque ambas emiten luz desde un punto asemejandose a lo que sería el ojo humano. La luz puntual debe observar todo el ambiente que la rodea, por ello, se utiliza un campo de visión o *field of view* de noventa grados para abarcar una visualización completa. Por otro lado, la luz reflector no necesita visualizar todo el ambiente que la rodea porque ésta posee como atributo una dirección de visualización, por ende, utiliza un campo de visión de sesenta grados. Finalmente, la clase *DirectionalLight* hereda de la clase *OrthographicCamera*, porque al encontrarse en el infinito los rayos de luz emitidos viajan de forma paralela hacia los objetos.

4.13. Geometría

La clase *Geometry* es la encargada de gestionar la información geométrica de un objeto, dicha información está compuesta por datos como vértices, normales, coordenadas de textura, índices que describen la triangulación de los vértices, etc. Para ello, *Geometry* utiliza la clase *ArrayBuffer* que representa un VAO[1] que almacena un estado que está compuesto por el formato de los datos que componen la información geométrica y la información en sí. Para almacenar dicha información geométrica se hace uso de la clase *Buffer*.

Buffer representa un *Buffer Object*[2] cuyos datos geométricos pueden ser actualizados de forma total o parcial según se desee, dicha actualización, se logra indicando mediante intervalos las porciones de datos a actualizar. De la clase *Buffer* heredan las clases *VertexBuffer* e *IndexBuffer* que representan un *Vertex Buffer Object* o VBO y un *Index Buffer Object* o IBO respectivamente. Tanto un VBO como un IBO son *Buffer Objects* que almacenan datos gráficos pero su diferencia yace en el hecho que un VBO almacena datos del tipo real, mientras que los datos que se almacenan en un IBO son del tipo entero.

La clase *VertexBuffer* implementa el concepto de *buffers* intercalados o *interleaved buffers* cuyo significado es que dentro de éste se pueden almacenar distintos datos. En consecuencia, la clase *VertexBufferAttribute* se encarga de establecer como debe ser interpretado el contenido de la clase *VertexBuffer*. Por otro lado, la clase *IndexBuffer* se utiliza para almacenar los índices que describen como conectar los vértices de la geometría.

La clase *PrimitiveGeometry* que representa una primitiva geométrica hereda de la clase *Geometry*, ofreciendo un conjunto de geometrías por defecto en EZ3.js. Estas geometrías tienen una naturaleza matemática y se describen mediante ecuaciones paramétricas que son evaluadas por algoritmos que se implementaron en las clases que heredan de *PrimitiveGeometry*, ellas son:

- *AstroidalEllipsoidGeometry*, representa la geometría de un elipsoide astroidal.
- *BoxGeometry*, representa la geometría de un paralelepípedo.
- *EllipsoidGeometry*, representa la geometría de un elipsoide.
- *PlaneGeometry*, representa la geometría de un plano.
- *SphereGeometry*, representa la geometría de una esfera.

[1] Un *Vertex Array Object* o VAO es un objeto que provee WebGL para almacenar un estado que describe el como se envía la información geométrica asociada a un objeto al hardware gráfico para su posterior despliegue.

[2] Un *Buffer Object* es un objeto que provee WebGL cuya representación es un arreglo que almacena datos geométricos en la memoria del hardware gráfico.

- *TorusGeometry*, representa la geometría de un toro

Adicionalmente, la clase *Geometry* presenta métodos para el cálculo y actualización de sus normales, índices de alambrado o *wireframe* y volúmenes delimitadores. Los volúmenes delimitadores son una caja y una esfera, representados por las clases *Box* y *Sphere* respectivamente.

4.14. Materiales

Los materiales describen las propiedades físicas de los objetos relativas a cómo reflejan la luz incidente. Por razones obvias, el aspecto final obtenido será dependiente tanto de las propiedades del material, como de la propia definición de las fuentes de luz. De este modo, materiales e iluminación están íntimamente relacionados. En EZ3.js, los materiales permiten configurar la apariencia que va a tener un objeto dentro de la escena, para ello, se utilizan los atributos que estos poseen para lograr los efectos que se deseen simular, particularmente, todo material hereda de la clase *Material*.

En la clase *Material*, se encuentran una serie de atributos que sirven para configurar el descarte de caras, la visibilidad, el tipo de relleno, la transparencia y el tipo de prueba de profundidad o *depth test*. Adicionalmente, cada material posee una forma de notificarle al hardware gráfico las configuraciones que este tiene para que se lleve a cabo el despliegue apropiado, esto se logra mediante un atributo de tipo *GLSLProgram* que representa un programa de GLSL ES. Dicho programa sirve de interfaz entre los atributos y las variables que yacen en las instrucciones dictadas a priori por este en el código fuente de los sombreadores que son ejecutados por el hardware gráfico. En EZ3.js, se implementaron dos clases llamadas *MeshMaterial* y *ShaderMaterial* que son las encargadas de proveer el aspecto que tienen los objetos en una escena.

La clase *MeshMaterial* representa un material que ofrece un conjunto de atributos que dependiendo de sus valores configuran su programa. Dicha configuración se realiza mediante bloques de instrucciones que yacen en el código fuente de los sombreadores definidos en *ShaderLibrary*. Estos bloques, se encuentran encerrados dentro de preprocesadores que el compilador del lenguaje GLSL ES va analizar para decidir si agregarlos o descartarlos para generar el programa final. El método *updateProgram* analiza los valores de los atributos almacenando en una cadena de caracteres las definiciones asociadas a cada bloque de instrucciones según los valores de los mismos. Posteriormente, esta cadena es concatenada al código fuente de los sombreadores y finalmente, el compilador analiza el código y genera el programa. Como ejemplo de esto, se puede apreciar en el Código 6 el proceso de construcción de la cadena de caracteres que define los bloques de instrucciones que se encuentran en el Código 7 asociados al cálculo de la reflexión difusa.

```
EZ3.MeshMaterial.prototype.updateProgram =
function(gl, state, lights, shadowReceiver) {
  var id = 'MESH.';
  var defines = [];
  var prefix = '#define ';

  // ...

  if(this.diffuseReflection === EZ3.MeshMaterial.OREN_NAYAR)
    defines.push('OREN_NAYAR');
  else
    defines.push('LAMBERT');

  // ...

  id += defines.join('.');
  prefix += defines.join('\n ' + prefix) + '\n';

  if (this._id !== id) {
    this._id = id;
    this.program = state.createProgram(
        id,
        EZ3.ShaderLibrary.mesh.vertex,
        EZ3.ShaderLibrary.mesh.fragment,
        prefix
    );
  }
};
```

Código 6: Acumulación de las definiciones asociadas a cada bloque de instrucciones en una cadena de caracteres.

```
#ifdef LAMBERT
float lambert(in vec3 s, in vec3 n) {
  return max(dot(s, n), 0.0);
}
#endif

#ifdef OREN_NAYAR
uniform float uAlbedo;
uniform float uDiffuseRoughness;

float orenNayar(in vec3 v, in vec3 s, in vec3 n) {
```

```
  float PI = acos(-1.0);

  float SdotV = dot(s, v);
  float SdotN = dot(s, n);
  float NdotV = dot(n, v);

  float o = SdotV - SdotN * NdotV;
  float t = mix(1.0, max(SdotN, NdotV), step(0.0, o));
  float sigma2 = uDiffuseRoughness * uDiffuseRoughness;

  float A = 1.0 + sigma2 * (uAlbedo / (sigma2 + 0.13) + 0.5 / (sigma2 + 0.33));
  float B = 0.45 * sigma2 / (sigma2 + 0.09);

  return uAlbedo * max(0.0, SdotN) * (A + B * o / t) / PI;
}
#endif
```

Código 7: Bloques de instrucciones en el código fuente del sombreador de fragmentos para el cálculo de la reflexión difusa.

Por otro lado, la clase *ShaderMaterial* ofrece un material cuya principal característica es otorgar la potestad de suministrar los sombreadores a emplear por este. Exclusivamente, para crear una instancia de *ShaderMaterial* se deben suministrar dos cadenas de caracteres que almacenan el código fuente de los sombreadores de vértices y fragmentos respectivamente. Esto, ya que no tiene sentido tener atributos adicionales que sean configurables porque estos dependen de la lógica que sea plasmada en el código fuente de los sombreadores que se suministran. Sin embargo, esta clase posee métodos para que se puedan actualizar los valores de las variables que se conocen a priori en el código fuente de los sombreadores, estos métodos utilizan los nombres de las variables para actualizar números enteros y reales, vectores, matrices y texturas.

4.15. Malla

En EZ3.js los objetos dentro de las escenas se representan mediante mallas que son instancias de la clase *Mesh* que a su vez hereda de la clase *Entity*. La clase *Mesh* es el resultado de la unión del componente de geometría con el componente de aspecto, es decir, ésta clase se compone por una instancia de tipo *Geometry* y por una instancia de tipo *Material*. En consecuencia, la malla es la única entidad que es desplegable en el grafo de escena, porque en su interior es donde se encuentran las invocaciones

correspondientes a WebGL para llevar a cabo su despliegue. Adicionalmente, como se explicó en la sección 3.5, existe una forma para clasificar los objetos que reciben y dan sombra dentro de las escenas, la clase *Mesh* provee atributos para asignarle a una malla la propiedad de emisión o recepción de sombras.

4.16. Entrada

EZ3.js ofrece una abstracción respecto a la interacción del usuario agrupando un conjunto de objetos como atributos en la clase *InputManager* que se encargan de procesar los eventos de entrada. Dichos objetos son de tipo *Keyboard*, *Mouse* y *Touch*, los cuales representan el soporte a los dispositivos de entrada, teclado, ratón y pantalla táctil respectivamente. Para ello, emplean los nombres de los eventos estandarizados que provee el objeto *Device*.

La clase *Keyboard* almacena el estado de las teclas mediante objetos de tipo *Switch*. La clase *Switch* actúa como un interruptor, de este modo, brinda información de si éste se encuentra pulsado o no. Adicionalmente, dicha clase posee un identificador, al cual, en el caso de las teclas, se le asigna su respectivo código ASCII.

Los elementos que suelen interactuar con la pantalla táctil son los dedos y los lápices ópticos. Estos son asociados a punteros en la pantalla, que como el del ratón poseen una posición en la misma. Razón por la cual, las clases *TouchPointer* y *MousePointer* que simbolizan dichos punteros correspondientemente heredan de *Pointer*.

Los botones y la rueda del ratón son representados en su puntero. Estos botones y los punteros táctiles, como las teclas, poseen características provistas por la clase *Switch*, por lo que *TouchPointer* también hereda de *Switch* y los botones son objetos de tipo *Switch*.

La clase *Mouse* almacena el estado del puntero del ratón, mientras que la clase *Touch* almacena el de hasta un máximo de diez punteros táctiles (P_{max}). El identificador de un puntero táctil es un entero comprendido entre uno y P_{max} que denota la cantidad de elementos táctiles que deben estar pulsados al mismo tiempo para que el puntero respectivo se considere pulsado.

El estado de los elementos almacenados en las clases anteriormente mencionadas, puede ser obtenido mediante métodos dados por las mismas, proporcionando un identificador o una función de notificación.

4.17. Controles

Para proporcionar diversos movimientos definidos sobre una o varias entidades, EZ3.js emplea un clase denominada *Control*, la cual en su inicialización requiere de forma mandatoria una entidad. Adicionalmente, las clases que hereden de ésta, se deben encargar de aplicar transformaciones sobre dicha entidad.

La clase *CameraControl* hereda de *Control* e implementa ciertos componentes lógicos para el manejo de transformaciones que comparten las clases *TargetControl* y *FreeControl* heredando de ésta. El objetivo principal de estas dos últimas clases es permitir la navegación de una escena mediante el uso de una cámara, por lo que comúnmente son inicializadas con una entidad de tipo *Camera*. Sin embargo, podrían ser inicializadas con cualquier otro tipo de entidad.

La diferencia entre la clases *TargetControl* y *FreeControl* yace en que las transformaciones de la primera se realizan en base a un punto, el cual siempre es el objetivo a observar, mientras que las transformaciones de la segunda se realizan en base a la misma posición de la entidad otorgando así, libertad de variar dicho objetivo. Ambas clases poseen diferentes tipos de movimientos que generalmente se suelen vincular a eventos de entrada.

4.18. Carga

En un entorno web, los elementos a introducir al videojuego o *game assets* no pueden ser accedidos localmente, razón por la cual, primero deben ser descargados. EZ3.js provee para esto solicitudes o *requests* representadas por diversas clases.

Dichas clases heredan de la clase *Request* y definen un conjunto de atributos en común para toda solicitud como el URL, el elemento asociado a la misma, entre otros. Éstas clases deben implementar el método *send*, que se encarga de descargar e interpretar uno o varios archivos de formatos específicos para convertirlos en elementos manejables por el motor. Cada tipo de solicitud se distingue por el formato de los archivos a tratar.

Los elementos manejables por el motor son archivos, imágenes y entidades. Los archivos e imágenes son representados por las clases *File* e *Image* respectivamente. La primera solo almacena el contenido del archivo en una cadena de caracteres, mientras que la segunda almacena el tamaño, el formato de color y el contenido de la imagen, razón por la cual hereda de la primera. Adicionalmente, la clase *Image* provee métodos para su escalado y conversión.

La clase *RequestManager* representa un manejador de solicitudes. Estas son agregadas mediante tres métodos vinculados al tipo de elemento que se desee tratar, en

ellos se detecta el formato a partir del URL que debe ser suministrado como parámetro para instanciar la solicitud pertinente. Las solicitudes son realizadas invocando al método *send* de cada una cuando el método *send* de la misma es invocado. Para mantener seguimiento de las solicitudes, la clase *RequestManager* emplea los parámetros del método *send* de éstas, los cuales son una función de notificación de suceso y otra error. De este modo, dicha clase puede notificar la culminación satisfactoria o errónea de una o todas las solicitudes agregadas.

Para la descarga de los archivos en el método *send* de las solicitudes se emplea AJAX. JavaScript asíncrono y XML o *asynchronous JavaScript and XML* (AJAX) es una técnica de desarrollo web para el intercambio de información entre el servidor y el cliente (navegadores) sin la necesidad de recargar la página. Esta técnica pese a su nombre, no se encuentra limitada al formato XML, se puede aplicar con cualquier formato. En este caso, se emplea para descargar el archivo referente al URL de la solicitud.

Las solicitudes en la clase *RequestManager* que son completadas satisfactoriamente cuyo elemento vinculado sea de tipo *File* o *Image*, son albergadas en un objeto llamado *Cache*. Este objeto es consultado por los métodos de agregación de estos dos tipos, con la finalidad de mejorar los tiempos de descarga sucesivos.

En el Cuadro 4.1 se puede apreciar la relación de los formatos a tratar y el tipo de elemento resultante con los diferentes tipos de solicitudes.

Tipo de solicitud	Formatos	Tipo de elemento resultante
FileRequest	Todos	*File*
ImageRequest	PNG, JPG y BMP	*Image*
TGARequest	TGA	*Image*
OFFRequest	OFF	*Mesh*
OBJRequest	OBJ	*Entity*
MDLRequest	MDL	*Mesh*

Cuadro 4.1: Comparación entre los diferentes tipos de solicitudes.

Los formatos PNG, JPG, BMP y TGA son empleados para representar imágenes, mientras que los formatos OFF, OBJ y MDL son empleados para representar mallas. HTML5 interpreta las imágenes de formato PNG, JGP y BMP por medio de su clase *Image*. Sin embargo, no le da soporte a las de formato TGA. Por esta razón, se elaboraron intérpretes para el contenido de los archivos de formatos TGA, OFF, OBJ y MDL en sus respectivas clases.

A diferencia de los archivos OFF y MDL, los OBJ pueden poseer varias mallas. Por esto, se utiliza un nodo de tipo *Entity*, al cual se le asocian todas sus mallas.

4.19. Motor y manejador de pantallas

La clase *Engine* representa el punto de partida del motor. Para su inicialización requiere tres parámetros, de los cuales, solo el primero es obligatorio. Dichos parámetros en orden son, el elemento HTML5 *canvas* que simboliza un lienzo donde se mostrará el resultado del proceso de despliegue, un objeto que posee todas las configuraciones a aplicar sobre dicho elemento y la representación de un objeto o función de tipo *Screen*.

El método de inicialización comienza cuando el dispositivo se encuentra listo para ser usado, para esto se emplea el objeto *Device*. En este método, se inicializan un conjunto de atributos, los cuales son objetos de tipo *AnimationFrame*, *Renderer*, *Time*, *Input* y *ScreenManager*.

Los atributos de tipo *Time* y *ScreenManager* son actualizados en un método conocido como bucle principal o *main loop* realizado por el atributo de tipo *AnimationFrame*.

La clase *ScreenManager* representa un manejador de pantallas y requiere en su inicialización el *canvas* y los objetos de tipo *Renderer*, *Time* e *Input* obligatoriamente, los cuales son almacenados como atributos en la misma. También requiere pero de forma opcional la representación de *Screen*. En caso de poseer dicha representación, se invoca el método *add* con un identificador por defecto y la misma.

En el Código 8 se puede apreciar el método *add* que realiza la inserción de una pantalla en el manejador. Para esto primero se verifica si la representación de *Screen* que simboliza una pantalla, no es una instancia de esa clase, de ser así, se comprueba si la representación es una función. En ese caso, se crea un objeto a partir de la misma y en caso de que sea un objeto, se deja así, para vincularle un conjunto de atributos que toda pantalla debe poseer, los cuales son, un identificador, una referencia al mismo manejador de pantallas, el tamaño, la posición, el manejador de solicitudes, la escena y la cámara. El tamaño y la posición son vinculados con valores por defecto únicamente si no se encuentran definidos.

Toda pantalla debe definir obligatoriamente el método *create*. Opcionalmente puede definir los métodos *preload* y *update* y un conjunto de métodos de entrada.

Después de vincular dichos atributos en el objeto, si éste define el método *preload* se invoca. En este método se emplea el manejador de solicitudes para obtener acceso a los elementos a introducir al videojuego, cuando éste culmine, se procede a invocar el método *create* donde se agregan las entidades a la escena y se asigna la cámara a emplear, por medio de los atributos de la misma. Finalmente, se invoca al método *addScreenEventListeners* que vincula los métodos de entrada del objeto con funciones de notificación por medio del atributo de tipo *Input* y luego se apila en un atributo denominado *screens*.

```
EZ3.ScreenManager.prototype.add = function(id, screen) {
  var Screen;
  var onComplete;

  if(!(screen instanceof EZ3.Screen)) {
    if (typeof screen === 'function') {
      Screen = screen;
      screen = new Screen();
    } else if (typeof screen !== 'object')
      return;

    screen.load = new EZ3.RequestManager();
    screen.scene = new EZ3.Scene();
    screen.camera = null;

    if (!screen.position)
      screen.position = new EZ3.Vector2();

    if (!screen.size)
      screen.size = new EZ3.Vector2(this.canvas.width, this.canvas.height);
  }

  screen.id = id;
  screen.manager = this;

  if (screen.preload) {
    screen.preload();

    if (screen.onLoadProgress)
      screen.load.onProgress.add(screen.onLoadProgress, screen);

    onComplete = function(assets, failed, loaded) {
      if (screen.onLoadProgress)
        screen.load.onProgress.remove(screen.onLoadProgress, screen);

      screen.load.onComplete.remove(onComplete, this);

      screen.create.call(screen, assets, failed, loaded);

      this._addScreenEventListeners(screen);
      this._screens.unshift(screen);
    };
```

```
    screen.load.onComplete.add(onComplete, this);

    screen.load.start();
  } else {
    screen.create();
    this._addScreenEventListeners(screen);
    this._screens.unshift(screen);
  }

  return screen;
};
```

Código 8: Método *add* de la clase *ScreenManager*.

El método de actualización de esta clase, se encarga de recorrer la pila *screens*, desplegando usando el atributo de tipo *Renderer* y actualizando cada objeto. Para realizar la actualización se invoca el método *update* del objeto.

Además de los métodos mencionados anteriormente, la clase *ScreenManager* también provee métodos para obtener y remover una pantalla en especifico y entrar y salir del modo pantalla completa empleando el atributo *canvas*.

4.20. Despliegue

La clase *Renderer* a partir del *canvas* y sus opciones procede a inicializar, el contexto de WebGL, un objeto de tipo *RendererState* que se encarga de gestionar el estado de WebGL, un objeto de tipo *RendererExtensions* que administra las extensiones necesarias para el funcionamiento del mismo y finalmente un objeto de tipo *RendererCapabilites* que representa las capacidades que posee el hardware gráfico en el que se llevará a cabo el despliegue. Comúnmente, tanto el contexto como alguno de éstos tres objetos, serán propagados a lo largo de las invocaciones a funciones que son internas en el *Renderer* con la finalidad de suministrarle a cada instancia la información necesaria para crearse o para cumplir sus funciones.

El proceso de despliegue inicia con la invocación del método *render* que recibe como parámetros un grafo de escena y una cámara provenientes de la pantalla a desplegar. En este método, se recorre el grafo de escena donde se actualizan las matrices de mundo que posee cada nodo según lo expuesto en el Código 4. Consecuentemente, se clasifican las entidades según su naturaleza almacenándolas en arreglos y se actualizan las matrices de mundo, vista y proyección de la cámara, hecho que se puede apreciar en el Código 9.

```
scene.traverse(function(entity) {
  if (entity instanceof EZ3.Light) {
    lights.empty = false;

    entity.updateProjection();

    if (entity instanceof EZ3.PointLight)
      lights.point.push(entity);
    else if (entity instanceof EZ3.DirectionalLight) {
      entity.updateView();
      lights.directional.push(entity);
    } else if (entity instanceof EZ3.SpotLight) {
      entity.updateView();
      lights.spot.push(entity);
    }
  } else if (entity instanceof EZ3.Mesh && entity.material.visible)
    meshes.common.push(entity);
});

if (!camera.parent)
  camera.updateWorld();

camera.updateView();
camera.updateProjection();

viewProjection = new EZ3.Matrix4().mul(camera.projection, camera.view);
```

Código 9: Clasificación de las entidades según su naturaleza y actualización de la cámara.

Una vez clasificadas las entidades, se lleva a cabo el procesamiento de las mallas. En este procesamiento, lo primero que se realiza es un conjunto de verificaciones sobre estos para evitar calcular datos que puedan ser innecesarios para el despliegue.

La primera verificación consiste en preguntar si la geometría que posee una malla es del tipo *PrimitiveGeometry*, esto con la finalidad de generarla o actualizarla. Adicionalmente, aquellas mallas que generan sombra al estar en presencia de fuentes de luz son almacenados en un arreglo de emisores o *casters*. La segunda verificación realiza el descarte por pirámide de visualización relacionado a la cámara donde se emplea la esfera delimitadora de la geometría. En la tercera verificación, se pregunta si el tipo de relleno configurado en el material es de tipo alambrado, en caso de serlo, se debe calcular un arreglo de índices adicional que describen las líneas que lo

componen. La cuarta verificación consiste en preguntar por la cantidad de fuentes de luz que se recolectaron al recorrer el grafo y hacer el cálculo de las normales de la geometría y crear o actualizar la matriz normal. Finalmente en la última verificación, se pregunta si el programa asociado al material ha sido creado o si necesita actualizarse. Consecuentemente, se procede a clasificar las mallas en dos arreglos, uno que representa los opacos y otro que representa los transparentes. Este proceso se lleva a cabo en el Código 10.

```
for (i = 0; i < meshes.common.length; i++) {
  mesh = meshes.common[i];
  position = new EZ3.Vector3().setPositionFromWorldMatrix(mesh.world);
  depth = position.setFromViewProjectionMatrix(viewProjection).z;

  if (mesh.geometry instanceof EZ3.PrimitiveGeometry)
    mesh.geometry.updateData();

  if (mesh.shadowCaster)
    meshes.shadowCasters.push(mesh);

  if (!camera.frustum.intersectsMesh(mesh))
    continue;

  mesh.updateLines();

  if (!lights.empty) {
    mesh.geometry.updateNormals();
    mesh.updateNormal();
  }

  mesh.updateProgram(gl, this.state, lights);

  if (mesh.material.transparent)
    meshes.transparent.push({
      mesh: mesh,
      depth: depth
    });
  else
    meshes.opaque.push({
      mesh: mesh,
      depth: depth
    });
}
```

Código 10: Verificaciones y subclasificación sobre las mallas.

Luego, se aplica un ordenamiento por profundidad o *depth sort* sobre las mallas transparentes con la finalidad de tener una percepción adecuada del efecto de transparencia.

```
meshes.transparent.sort(function(a, b) {
  if (a.depth !== b.depth)
    return b.depth - a.depth;
});
```

Código 11: Ordenamiento por profundidad de las mallas transparentes.

Consecuentemente, se aplica el despliegue de las mallas, para ello, se deben calcular los mapas de sombras desde la perspectiva de las fuentes de luz para que estén disponibles cuando se calcule la iluminación y luego, se despliegan primero los opacos y luego los transparentes enlazando su programa.

```
if (meshes.shadowCasters.length && !lights.empty) {
  this._renderDepth(meshes.shadowCasters, lights);
  this.state.viewport(position, size);
}

for (i = 0; i < meshes.opaque.length; i++)
  this._renderMesh(meshes.opaque[i].mesh, camera, lights);

for (i = 0; i < meshes.transparent.length; i++)
  this._renderMesh(meshes.transparent[i].mesh, camera, lights);
```

Código 12: Invocaciones a funciones para calcular los mapas de sombras y ejecutar el despliegue de las mallas.

De forma subsiguiente, en el interior del método que despliega una malla (ver Código 13) se pregunta por el tipo de relleno que éste posee. Luego, se enlaza el VAO que se encuentra en la instancia de la clase de tipo *Geometry* para crearlo y actualizarlo, por transitividad se enlazan los VBOs e IBOs correspondientes para crearse y actualizase. Finalmente, se procede a invocar a WebGL para ejecutar el despliegue.

```
EZ3.Mesh.prototype.render = function(gl, attributes, state, extensions) {
  var mode;
  var index;
```

```
  var buffer;
  var indexType;

  if (this.material.fill === EZ3.Material.WIREFRAME) {
    index = EZ3.IndexBuffer.LINEAR;
    buffer = this.geometry.buffers.getLinearBuffer();
    mode = gl.LINES;
  } else if (this.material.fill === EZ3.Material.POINTS) {
    buffer = this.geometry.buffers.getPositionBuffer();
    mode = gl.POINTS;
  } else {
    index = EZ3.IndexBuffer.TRIANGULAR;
    buffer = this.geometry.buffers.getTriangularBuffer();
    mode = gl.TRIANGLES;
  }

  if (index) {
    indexType = buffer.getGLType(gl, extensions);
    this.geometry.buffers.bind(gl, attributes, state, extensions, index);
    gl.drawElements(mode, buffer.data.length, indexType, 0);
  } else {
    this.geometry.buffers.bind(gl, attributes, state, extensions);
    gl.drawArrays(mode, 0, buffer.data.length / 3);
  }
};
```

Código 13: Método *render* de la clase *Mesh*.

5

Pruebas y resultados

Todos los experimentos realizados en este trabajo de investigación fueron ejecutados en un computador portátil cuyas características son las siguientes:

- Procesador Intel Core i7-4700MQ 3.40 GHz.
- 8 GB de memoria RAM DDR3.
- Disco duro de 500 GB.
- Tarjeta gráfica NVIDIA GeForce GT 740m 2GB DDR5.
- Sistema operativo Windows 10.

Para las pruebas se evaluaron únicamente características cuantitativas de EZ3.js, para ello, se utilizaron los navegadores Google Chrome, Mozilla Firefox y Microsoft Edge. Como casos de prueba, se emplearon tres escenas donde su complejidad es incremental según lo expresado en el Cuadro 5.1, iniciando con una escena sencilla (ver Figura 5.1), luego con una intermedia (ver Figura 5.2) y finalmente con una compleja (ver Figura 5.3). Adicionalmente para la navegación, se utilizó una cámara de proyección perspectiva y se le adjudicó un control que permitió el desplazamiento de ésta dentro de las escenas.

Escena	Cantidad de triángulos	Cantidad de mallas
Sencilla	13.180	7
Intermedia	275.222	399
Compleja	88.239	1.093

Cuadro 5.1: Cantidad total de triángulos y mallas por escena.

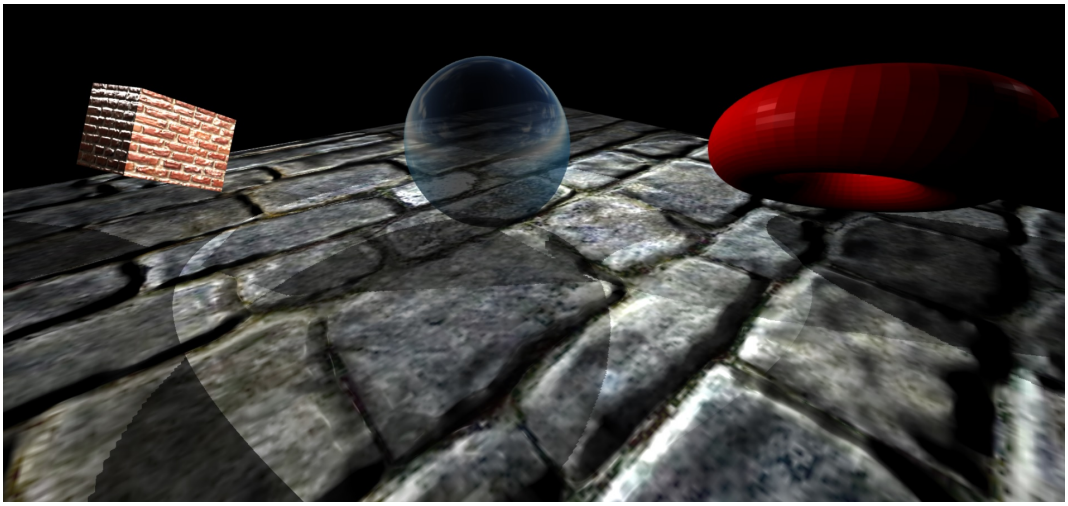


Figura 5.1: Escena sencilla.

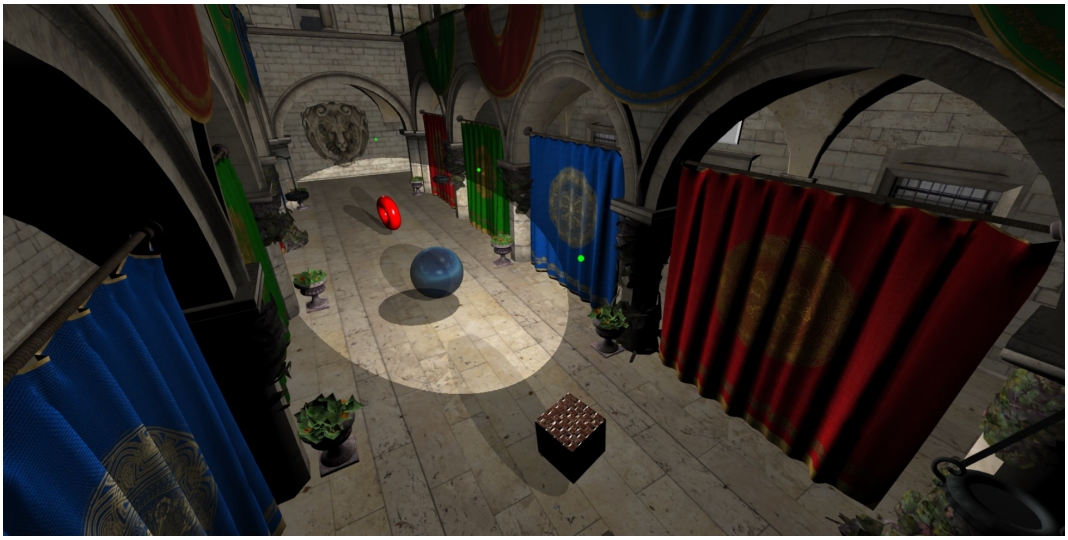


Figura 5.2: Palacio Sponza como escena intermedia.

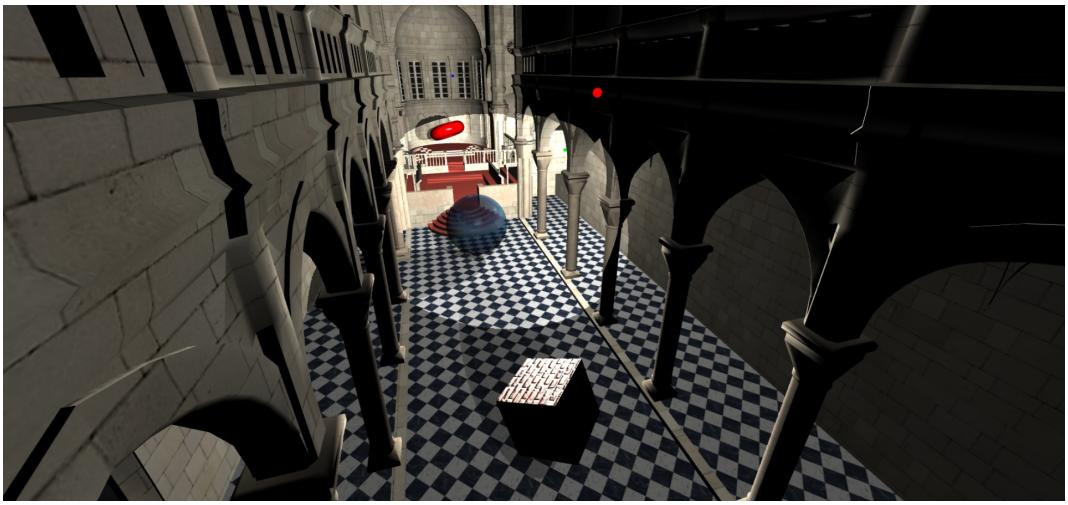


Figura 5.3: Catedral de Sibenik como escena compleja.

El criterio que se utilizó para determinar la complejidad de dichas escenas fue la cantidad de mallas que éstas poseen. Esto, porque a medida que una escena tiene una mayor cantidad de mallas, más laborioso será el proceso de despliegue por lo explicado en la sección 4.20.

La escena sencilla consta de un plano que posee mapeado de normales y una textura difusa, una esfera con los efectos de transparencia, reflexión y refracción, un cubo con mapeado de normales y una textura difusa, un toro con interpolación planar y tres luces cada una de distinto tipo (puntual, direccional y reflector) y generando sombras. Por otro lado, las escenas intermedia y compleja están representadas por mallas descritas en archivos de formato OBJ. Estas dos últimas, también incluyen las mallas, luces y sombras de la escena sencilla excluyendo al plano.

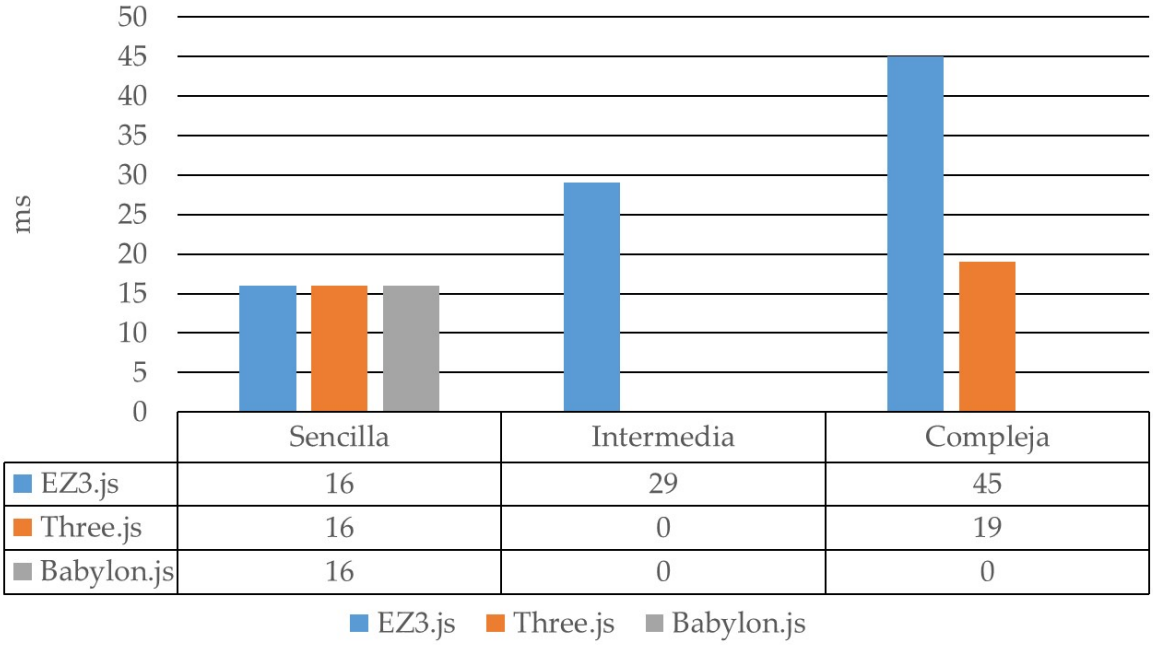
En las pruebas, por cada escena se estudió:

- El rendimiento.
- La cantidad de memoria utilizada.
- La cantidad de líneas de código para describirla.

Todos los resultados obtenidos para EZ3.js fueron comparados con los resultados que generaron los motores de videojuegos que usan WebGL mencionados previamente en la sección 1.3. Adicionalmente durante las pruebas, en algunos casos ciertas escenas no pudieron desplegarse y en consecuencia no se pudo tomar las medidas necesarias para su estudio, la representación que se utilizó para notificar este hecho es un cero en la celda del cuadro asociado a la prueba.

5.1. Estudio del rendimiento

Para estudiar el rendimiento, por cada navegador, se tomó la cantidad de tiempo en milisegundos (ms) que necesitó cada motor para desplegar un cuadro asociado a cada escena desde un mismo punto de vista. La herramienta que se utilizó para esta prueba fue un supervisor de rendimiento llamado Stats.js que permite observar mediante una caja de información los milisegundos necesarios para desplegar un cuadro [38].



	Sencilla	Intermedia	Compleja
EZ3.js	16	29	45
Three.js	16	0	19
Babylon.js	16	0	0

Figura 5.4: Resultados obtenidos del estudio del rendimiento para el navegador Google Chrome.

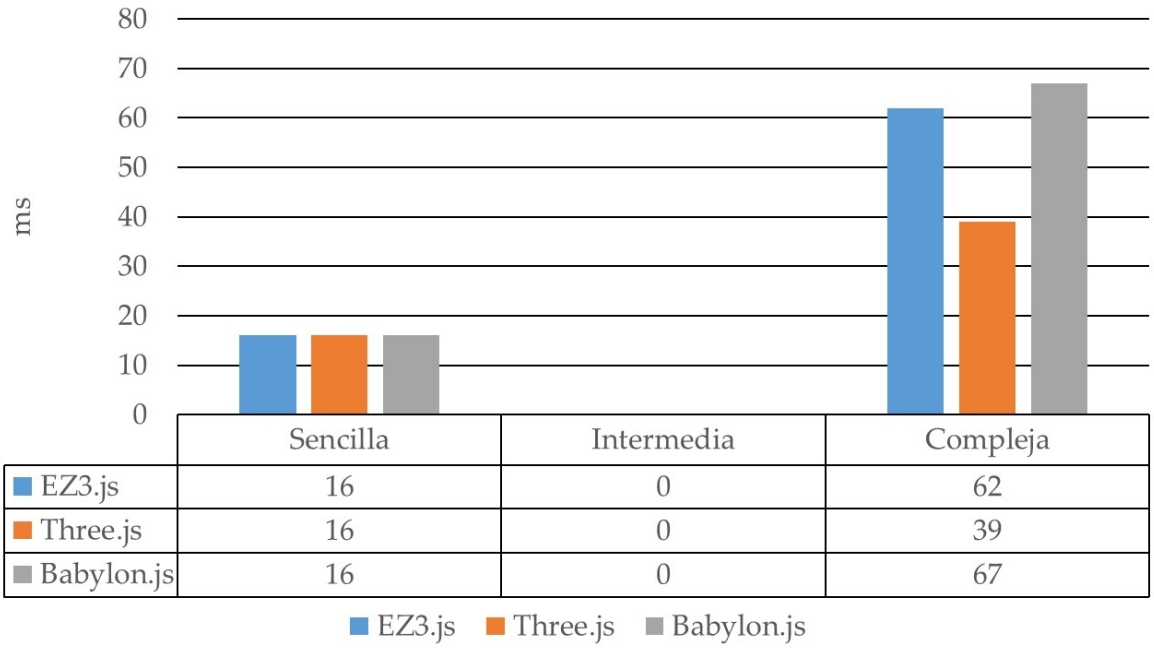


Figura 5.5: Resultados obtenidos del estudio del rendimiento para el navegador Mozilla Firefox.

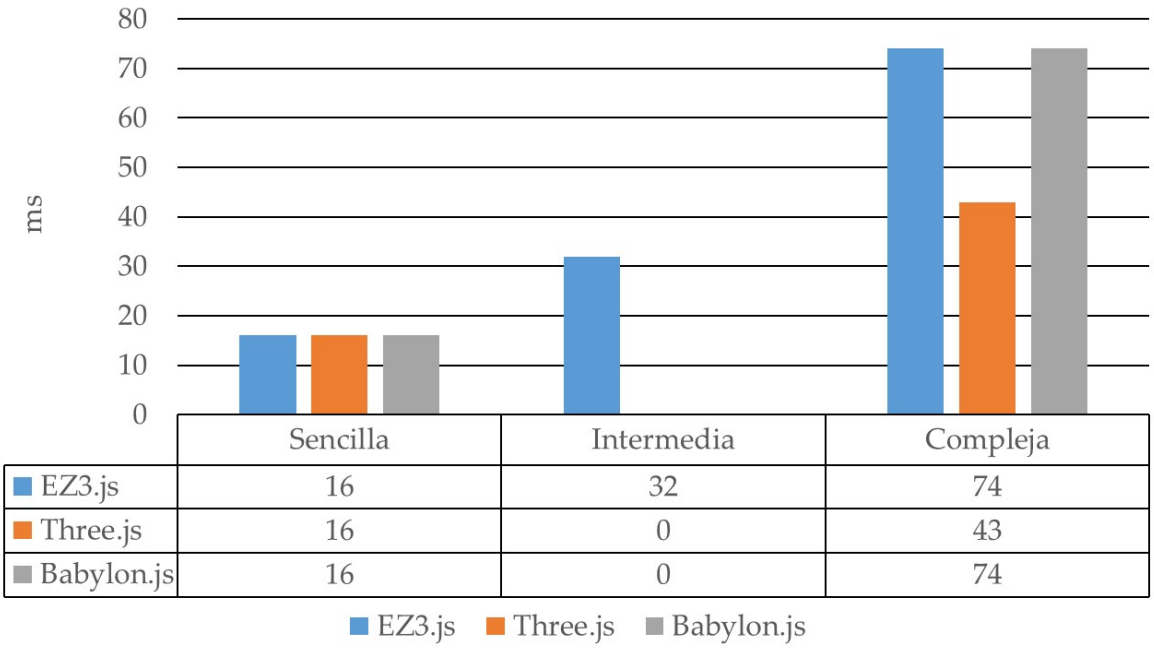


Figura 5.6: Resultados obtenidos del estudio del rendimiento para el navegador Microsoft Edge.

Como se puede ver en los resultados que se exponen en las Figuras 5.4, 5.5 y 5.6 los tres motores tienen un rendimiento idéntico para la escena sencilla en los tres navegadores. Por otro lado, se observa que Three.js posee un rendimiento superior con respecto a Babylon.js y EZ3.js en el caso de la escena compleja. Adicionalmente, EZ3.js demostró tener igual y mejor rendimiento que Babylon.js en los navegadores Microsoft Edge y Mozilla Firefox respectivamente para dicha escena.

EZ3.js posee un rendimiento bastante aceptable comparado con el que poseen Babylon.js y Three.js. En algunos casos se observó que EZ3.js igualó y mejoró de forma ligera el rendimiento que posee Babylon.js. Por otro lado, hubieron casos en los que EZ3.js igualó el rendimiento de Three.js y otros en los que dicho motor obtuvo un mejor rendimiento.

5.2. Estudio de memoria

Para estudiar la cantidad de memoria utilizada se captó la cantidad de mega-bytes (MB) que consumía la pestaña del navegador donde se desplegaba cada escena. Las herramientas que se utilizaron para este estudio fueron el administrador de tareas del navegador Google Chrome, *Tab Memory* que es un complemento para el navegador Mozilla Firefox que reporta por pestaña la cantidad de memoria utilizada [39] y las herramientas de desarrollador en Microsoft Edge.

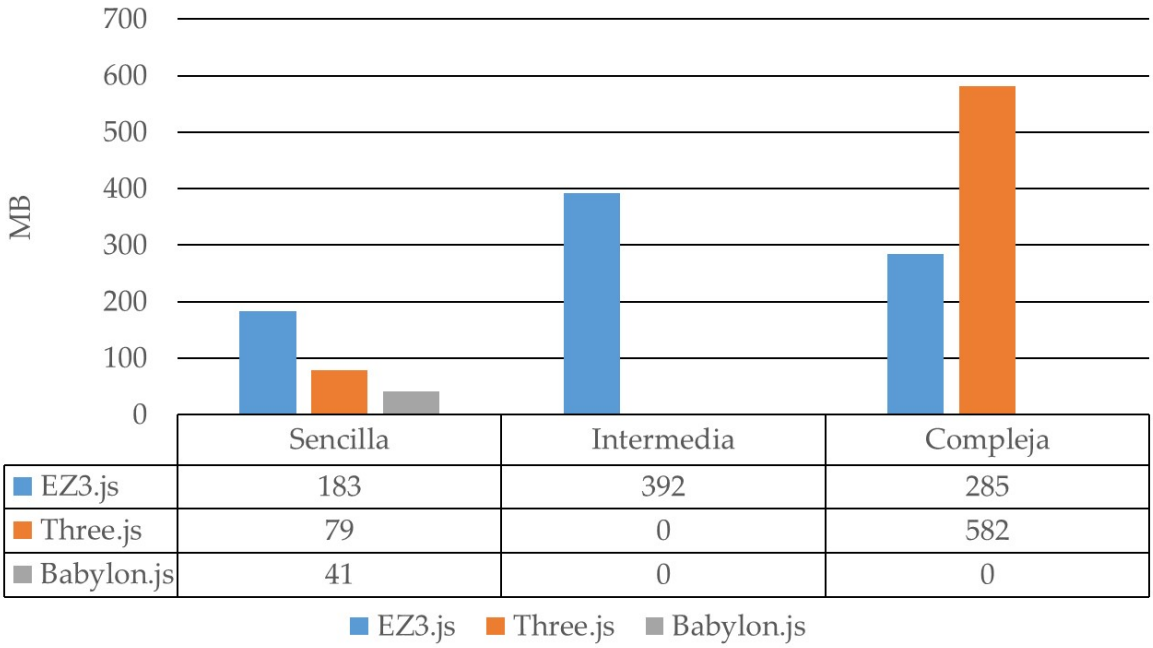


Figura 5.7: Resultados obtenidos del estudio de memoria para el navegador Google Chrome.

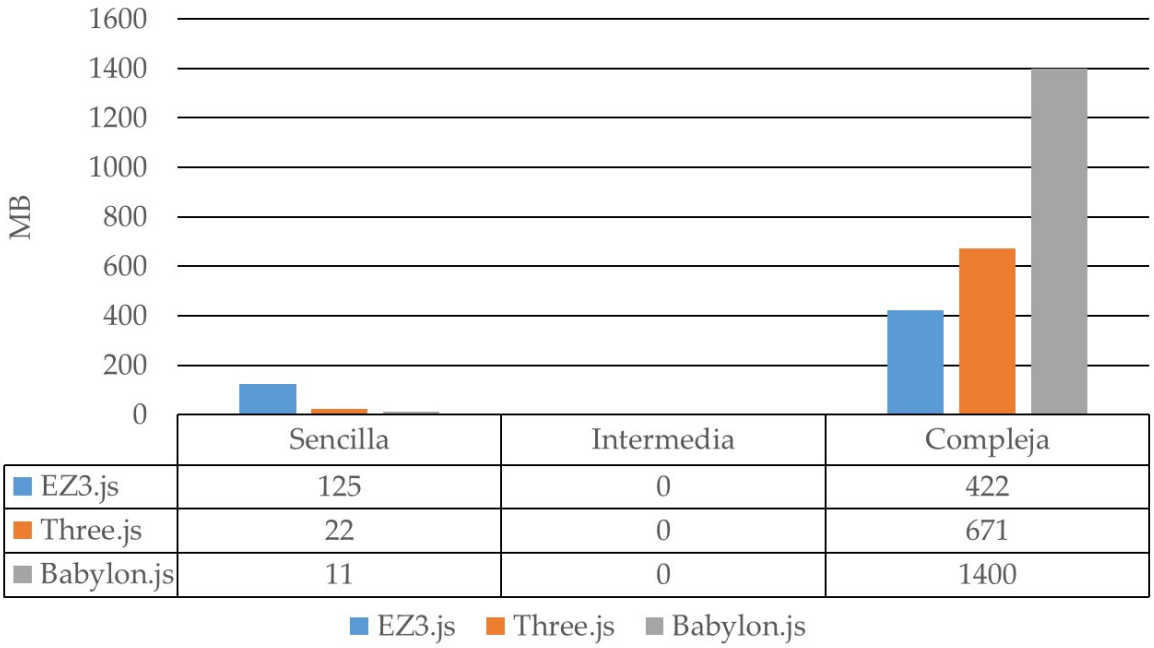


Figura 5.8: Resultados obtenidos del estudio de memoria para el navegador Mozilla Firefox.

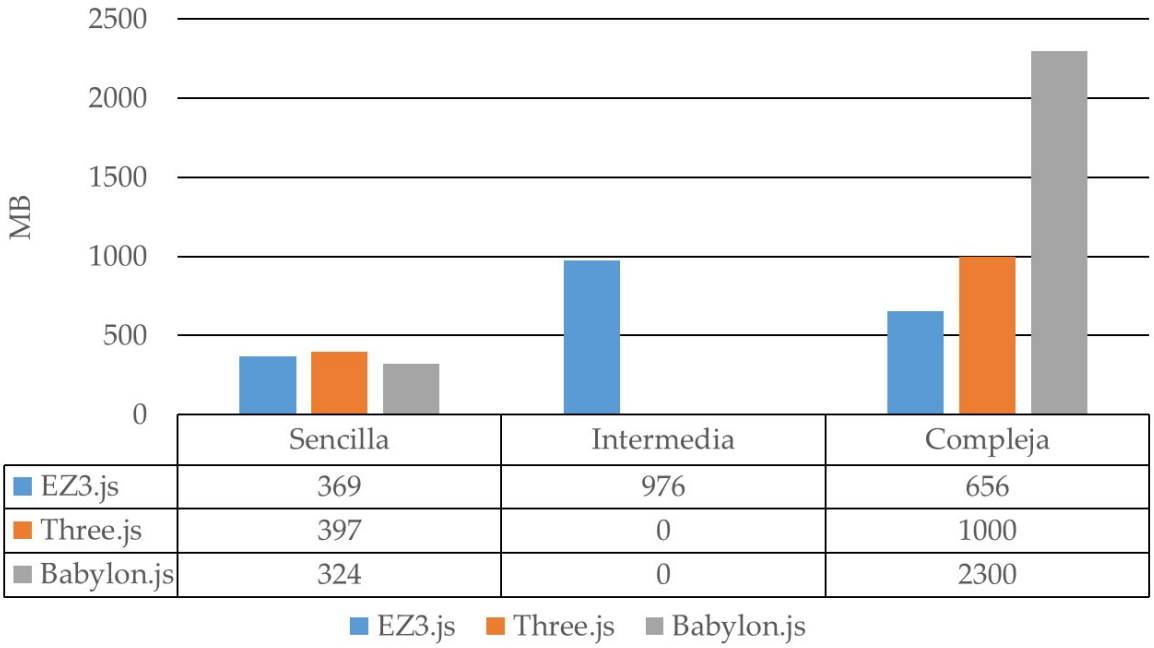
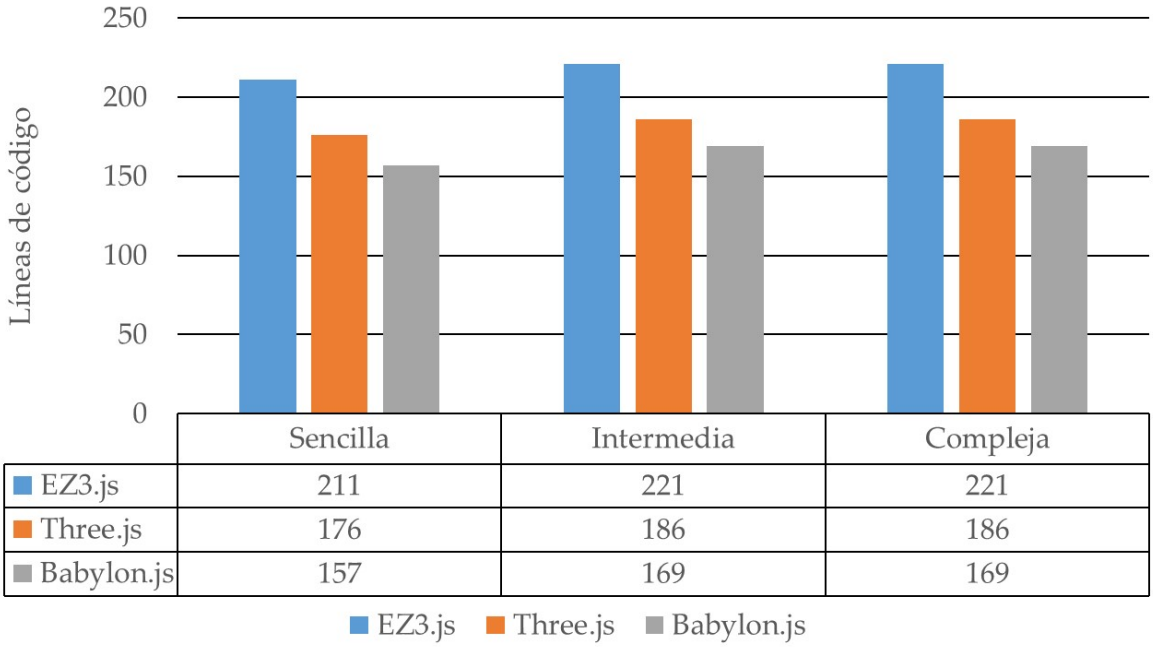


Figura 5.9: Resultados obtenidos del estudio de memoria para el navegador Microsoft Edge.

Tal como muestran los resultados que se exponen en las Figuras 5.7, 5.8 y 5.9 se puede observar que para el caso de la escena sencilla en Microsoft Edge, EZ3.js toma ventaja sobre Three.js y en la escena compleja usa menos memoria que los otros motores en los tres navegadores.

5.3. Estudio de la cantidad de líneas de código

Para determinar la cantidad de líneas de código utilizadas para describir las escenas, se contaron por cada archivo JavaScript involucrado, para de este modo, poder establecer una comparación entre la cantidad de líneas que se necesitan para describirlas por motor.



	Sencilla	Intermedia	Compleja
EZ3.js	211	221	221
Three.js	176	186	186
Babylon.js	157	169	169

Figura 5.10: Cantidad de líneas de código para cada escena.

Como el gráfico de la Figura 5.10 expone, los archivos en los que se describieron las escenas usando EZ3.js poseen más líneas de código que los archivos que las describen usando Babylon.js y Three.js, esto, porque EZ3.js obliga al usuario a especificar que hacer cuando se invoca a una función de notificación relacionada con los eventos de entrada del control de navegación otorgando más libertad de configuración que los otros dos motores. Por otro lado, Babylon.js y Three.js poseen una configuración definida a priori lo que permite reducir la cantidad de lineas de código que describen las escenas pero limitan la libertad de configuración.

6
Conclusiones

En esta investigación se estudió e implementó un motor de videojuegos basado en WebGL con una arquitectura débilmente acoplada en diversos módulos compuestos por un conjunto de clases. Suministrando de esta forma, una infraestructura sencilla para elaborar videojuegos en un entorno web y abstrayendo la complejidad de bajo nivel de interactuar directamente con WebGL.

Las funcionalidades que brinda el motor son despliegue gráfico, manejo de eventos y recursos. Las entidades a tratar son agrupadas en un grafo de escena donde únicamente las visibles son consideradas para su despliegue. A través de estas entidades se pueden configurar diferentes efectos de iluminación, texturizado y sombras.

La licencia del motor es de código abierto haciendo que sea accesible a toda la comunidad, favoreciendo enormemente la retroalimentación y el aporte de la misma. Adicionalmente, el motor presenta una documentación simple pero completa donde se describen todas sus funcionalidades.

Pruebas comparativas con respecto a dos de los motores que emplean WebGL más populares (Three.js y Babylon.js) fueron realizadas en diversos navegadores (Google Chrome, Mozilla Firefox y Microsoft Edge). Las pruebas consistieron en el despliegue y navegación de tres escenas diferentes con complejidad incremental (sencilla, intermedia y compleja) dada por el número mallas que las componen. Para cada prueba se realizaron estudios de rendimiento, memoria y lineas de código usadas para su descripción.

En el estudio del rendimiento se pudo apreciar que EZ3.js tuvo uno similar frente a los otros, pese que Three.js lo supera en la escena compleja. En el estudio de memoria se observó que para la escena sencilla no obtuvo el mejor manejo de memoria. Sin embargo, en la escena compleja si logró obtenerlo. En el estudio de la cantidad de líneas de código, con los otros motores se implementaron las escenas con una menor cantidad de líneas ya que encapsulan en sus controles de navegación código de manejo de eventos predefinido que reduce la cantidad de líneas de los archivos pero limita la libertad de configuración. Finalmente, es importante resaltar que EZ3.js fue el único capaz de desplegar la escena intermedia en Google Chrome y Microsoft Edge.

Las pruebas muestran que EZ3.js es una alternativa emergente a Three.js y Babylon.js. EZ3.js posee un largo camino por recorrer ya que su primera versión fue publicada recientemente en su respectivo repositorio. Tanto Three.js como Babylon.js poseen una mayor cantidad de tiempo en desarrollo que éste. Sin embargo, diversos trabajos futuros pueden ser realizados para agregarle nuevas funcionalidades y optimizaciones.

6.1. Trabajos futuros

La arquitectura débilmente acoplada y modular del motor permite que sea altamente escalable. Existen diversos módulos de gran importancia que pueden ser agregados extendiendo las funcionalidades ofrecidas, como el motor de física, audio y redes. Estos módulos no necesariamente tienen que ser implementados desde cero, una opción bastante factible sería emplear alguna biblioteca o SDK existente y elaborar un *plug-in* para el motor que lo encapsule. Adicionalmente, se podrían agregar varios efectos gráficos que no fueron contemplados, técnicas para la reducción del *aliasing* presente en las sombras, soporte a más formatos de imágenes y mallas, controles que faciliten la interacción con el usuario como la selección y arrastre de objetos, animación, sistemas de partículas, etc.

Conjuntamente con la documentación, seria relevante tener diferentes ejemplos ordenados por complejidad que ilustren todas las características del motor, un generador de plantillas de proyecto que permita crear una a partir de una configuración dada y un editor de código web con el fin de fomentar su uso y reducir la curva de aprendizaje del mismo.

Bibliografía

[1] D. V. Fernández y C. M. Angelina, *Desarrollo de Videojuegos: Arquitectura del Motor*, 2da. ed. Bubok, 2013.

[2] K. Matsuda y R. Lea, *WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL*, 1ra. ed. Addinson-Wesley, 2013.

[3] E. Ramirez, "Estado del arte en iluminación global de escenas," 2012. [En línea]. Disponible en: http://ccg.ciens.ucv.ve/~esmitt/publications/2012/RT-2012-03.pdf

[4] F. Group. Realizzare foto cubiche e ambient maps. [En línea]. Disponible en: http://www.fastvr.com/web/it/tech_CubeMaps.htm [Accedido: Feb. 22, 2016].

[5] K. Szántó. An environment simulator for mobile context-aware system design. [En línea]. Disponible en: http://karolyszanto.ro/MastersThesis/presentation/pres.html#/ [Accedido: Feb. 22, 2016].

[6] S. L. Kent, *The Ultimate History of Video Games*, 1ra. ed. Three Rivers Press, 2001.

[7] D. Catuhe y D. Rousset. Babylon.js - 3d engine based on webgl/web audio and javascript. [En línea]. Disponible en: http://www.babylonjs.com/ [Accedido: Feb. 14, 2016].

[8] R. Cabello. Javascript 3d library. [En línea]. Disponible en: http://threejs.org/ [Accedido: Feb. 14, 2016].

[9] pnaylor. White paper for three.js. [En línea]. Disponible en: https://github.com/mrdoob/three.js/issues/1960 [Accedido: Feb. 14, 2016].

[10] J. Gregory, *Game Engine Architecture*. AK Peters, 2009.

[11] T. Akenine-Moller, E. Haines, y N. Hoffman, *Real-Time Rendering*, 3ra. ed. AK Peters, 2008.

[12] id Software. Doom. [En línea]. Disponible en: http://doom.com/en-us/ [Accedido: Ene. 12, 2016].

[13] D. Shereiner, G. Sellers, J. Kessenich, y B. Licea-Kane, *OpenGL Programming Guide*, 8th ed. Addison-Wesley, 2013.

[14] F. Luna, *Introduction to 3D Game Programming with DirectX 11*, 1ra. ed. Mercury Learning & Information, 2012.

[15] E. Lengyel, *Mathematics for 3D Game Programming and Computer Graphics*, 3ra. ed. Cengage Learning PTR, 2011.

[16] D. Wolff, *OpenGL 4 Shading Language Cookbook*, 2da. ed. Pack-Publishing, 2013.

[17] M. Oren y S. K. Nayar, "Generalization of lambert's reflectance model," *SIGGRAPH*, pp. 239–246, 1994.

[18] B. T. Phong, "Illumination for computer generated pictures," *Communitacions of the ACM*, vol. 18, pp. 311–317, 1975.

[19] J. F. Blinn, "Models of light reflection for computer synthesized pictures," *Proc. 4th annual conference on computer graphics and interactive techniques*, pp. 192–198, 1977.

[20] R. L. Cook y K. E. Torrance, "A reflectance model for computer graphics," *Computer Graphics (SIGGRAPH '81 Proceedings)*, vol. 15, pp. 301–316, 1981.

[21] NVIDIA, "Cubemap opengl tutorial." [En línea]. Disponible en: http://www.nvidia.com/object/cube_map_ogl_tutorial.html

[22] E. Hainess y T. Möller, "Real-time shadows," 2001. [En línea]. Disponible en: http://www.gamasutra.com/features/gdcarchive/2001/haines.pdf

[23] L. Williams, "Casting curved shadows on curved surfaces," 1978. [En línea]. Disponible en: http://www.cs.berkeley.edu/~ravir/6160/papers/p270-williams.pdf

[24] H. Barad, "Aggressive performance optimization for 3d graphics," *Intel Corporation*, 2000.

[25] K. Cok, "Developing efficient graphics software," *Silicon Graphics*, 2000.

[26] A. Alvarez y C. Zapata. Ez3.js. [En línea]. Disponible en: https://github.com/andresz1/ez3.js [Accedido: Feb. 14, 2016].

[27] Git. [En línea]. Disponible en: https://git-scm.com/ [Accedido: Feb. 15, 2016].

[28] J. Harband. Airbnb javascript style guide. [En línea]. Disponible en: https://github.com/airbnb/javascript [Accedido: Feb. 14, 2016].

[29] GitHub Inc. Atom a hackable text editor. [En línea]. Disponible en: https://atom.io/ [Accedido: Feb. 14, 2016].

[30] N. Ribaudo. Jshint. [En línea]. Disponible en: http://jshint.com/ [Accedido: Feb. 14, 2016].

[31] Yahoo! Yui doc. [En línea]. Disponible en: http://yui.github.io/yuidoc/ [Accedido: Feb. 14, 2016].

[32] Grunt. Grunt the javascript task runner. [En línea]. Disponible en: http://gruntjs.com/ [Accedido: Feb. 14, 2016].

[33] Node.js. [En línea]. Disponible en: https://nodejs.org/en/ [Accedido: Feb. 14, 2016].

[34] Commonjs notes. [En línea]. Disponible en: http://requirejs.org/docs/commonjs.html [Accedido: Feb. 14, 2016].

[35] A. Alvarez y C. Zapata. grunt-depsconcat. [En línea]. Disponible en: https://github.com/andresz1/grunt-depsconcat [Accedido: Feb. 14, 2016].

[36] A. Alvarez y C. Zapata. grunt-shdrsconcat. [En línea]. Disponible en: https://github.com/andresz1/grunt-shdrsconcat [Accedido: Feb. 14, 2016].

[37] Desarrollo ágil de software - caso programación extrema - xp. [En línea]. Disponible en: http://ldc.usb.ve/~abianc/materias/ci4713/metodologiasagiles.pdf [Accedido: Feb. 27, 2016].

[38] R. Cabello. Javascript performance monitor. [En línea]. Disponible en: https://github.com/mrdoob/stats.js/ [Accedido: Feb. 14, 2016].

[39] J. Schomery. Tab memory usage. [En línea]. Disponible en: https://addons.mozilla.org/en-US/firefox/addon/tab-memory-usage/ [Accedido: Feb. 14, 2016].

[40] M. Parr y F. Randima, *GPU Gems*, 2da. ed. NVIDIA, 2013.

[41] E. Lengyel, *Game Engine Architecture*, 3ra. ed. Cengage Learning PTR, 2011.

[42] C. González, J. Albusac, C. Mora, y S. Fernández, *Desarrollo de Videojuegos: Programación Gráfica*, 2da. ed. Bubok, 2013.

[43] T. Whitted, "An improved illumination model for shaded display," *Communications of the ACM*, vol. 23, pp. 343–349. [En línea]. Disponible en: http://doi.acm.org/10.1145/358876.358882