Información del Proyecto

Título:

Diseño y desarrollo de un juego en WebGL

Autor:

Jaime Crespo Amigo

Fecha:

13 de junio de 2012

Director:

Lluis Solano Albajes

Departamento del director:

Lenguajes y Sistemas Informáticos

Presidente:

Luis Pérez Vidal

Vocal:

Francisco Javier Heredia Cervera

Titulación:

Ingeniería Superior de Informática

Centro:

Facultad de Informática de Barcelona

Universidad:

Universidad Politécnica de Catalunya

Facultad de Informática de Barcelona

Proyecto Final de Carrera

Ingienería Superior de Informática

Diseño y desarrollo de un juego en WebGL

Autor:
Jaime Crespo Amigo

Director:
Lluis Solano Albajes

Un Proyecto realizado según los requerimientos

de la

Normativa Plan 2003

Universidad Politécnica de Catalunya

13 de junio de 2012

Declaración de Autoría

Yo, Jaime Crespo Amigo, declaro que el proyecto titulado *'Diseño y desarrollo de un juego en WebGL'* y el trabajo presentado son de mi autoría. Yo confirmo que:

- Este trabajo ha sido realizado principalmente con el objetivo de presentarlo como Proyecto Final de Carrera.
- Donde se ha consultado la publicación de otros está claramente especificado.
- Donde he citado el trabajo de otros, la fuente del código está presente. Con excepción de esas citaciones el Proyecto es enteramente mi trabajo.
- Donde el proyecto esta basado en trabajo compartido, está claramente especificado la aportación de cada uno.

Firmado:

Fecha: 13 de junio de 2012

"Como en cualquier lenguaje de programación, saber lo que haces marca la diferencia."

Douglas Crockford

Agradecimientos

Debo agredecer este proyecto a todas aquellas personas que sin ánimo de lucro escriben en foros y blogs explicando, ayudando y enriqueciendo las tecnologías en las que se basa este proyecto, tecnologías libres.

A todos ellos, gracias.

Índice general

Índice General IV

Lista de Imágenes VII

Lista de Código Fuente IX

Lista de Tablas X

Abreviaciones y Anglicismos XI

1. Introducción 1
- 1.1. Motivación y Contexto . . . 1
- 1.2. Objetivos . . . 3
  - 1.2.1. Objetivo Principal . . . 3
  - 1.2.2. Objetivos Secundarios . . . 3
- 1.3. Proyecto Compartido . . . 7

2. WebGL 8
- 2.1. ¿Qué es WebGL? . . . 8
  - 2.1.1. OpengGL ES 2.0 . . . 9
  - 2.1.2. Pipeline de renderizado de WebGL . . . 10
  - 2.1.3. HTML5 Elemento Canvas . . . 16
- 2.2. Navegadores . . . 18
  - 2.2.1. Historia . . . 18
  - 2.2.2. Funcionamiento Interno de WebGL . . . 18
  - 2.2.3. Soporte Mayo 2012 . . . 22
  - 2.2.4. Internet Explorer . . . 25
  - 2.2.5. Seguridad . . . 26
  - 2.2.6. WebGL en móviles y Tablets . . . 28
- 2.3. State of the art Web 3D . . . 29
  - 2.3.1. Historia 3D en la web . . . 29
  - 2.3.2. 3D surfing, Hype or real? . . . 31
  - 2.3.3. Competidores . . . 32
    - 2.3.3.1. ¿En qué podemos experimentar 3D en la Web? . . . 32
    - 2.3.3.2. ¿En qué merece la pena arriesgar? . . . 33

3. El Juego **35**
3.1. Visión Global . . . 35
3.2. Esquema General . . . 36
3.3. Capturas de Pantalla . . . 37
3.4. Análisis . . . 39
3.4.1. Requisitos Funcionales . . . 39
3.4.2. Requisitos No Funcionales . . . 40
3.4.3. Diagrama Entidad - Relación . . . 41
3.5. Especificación . . . 42
3.5.1. Metodología de Desarrollo . . . 42
3.5.2. Historias . . . 44
3.6. Arquitectura . . . 47
3.6.1. Control de Versiones . . . 47
3.6.2. Diseño . . . 48
3.6.3. Patrones de Diseño . . . 48
3.6.4. Componentes . . . 49
3.6.4.1. Render . . . 49
3.6.4.2. Jugador Principal . . . 52
3.6.4.3. Cámara . . . 52
3.6.4.4. Animaciones y Modelos MD5 . . . 53
3.6.4.5. Audio . . . 54
3.6.4.6. Física . . . 54
3.6.4.7. Shaders Factory . . . 56
3.6.4.8. Iluminación . . . 57
3.6.5. Sistema Global . . . 58
3.6.6. Diagramas de Secuencia de un Frame . . . 60
3.6.7. Ficheros de Configuración . . . 61
3.7. Implementación . . . 62
3.7.1. Javascript . . . 62
3.7.1.1. Partes Malas . . . 62
3.7.1.2. Partes Buenas . . . 63
3.7.2. Render Loop . . . 67
3.7.3. Instanciación . . . 68
3.7.4. Componentes . . . 70
3.7.4.1. Patrón Módulo . . . 70
3.7.4.2. Camera . . . 72
3.7.4.3. Render . . . 74
3.7.4.4. Modelos . . . 76
3.7.4.5. Iluminación . . . 77
3.7.4.6. Factoría de Shaders . . . 82
3.7.4.7. Shaders . . . 84
3.7.4.8. Jugador Principal . . . 89
3.7.4.9. Física . . . 92
3.7.4.10. Audio . . . 94

4. Técnicas, Optimizaciones y Rendimiento **96**
4.1. Javascript . . . 96

4.1.1. Arrays . . . 96
4.1.2. Objetos . . . 97
4.1.3. Optimizaciones internas de Javascript . . . 99
4.1.4. Garbage Collector . . . 99
4.1.5. Render Loop Memory . . . 101
4.2. Reglas Generales . . . 103
4.2.1. Llamadas a WebGL . . . 103
4.2.2. Buffering . . . 104
4.3. CPU vs GPU . . . 106
4.4. Rendimiento Pipeline . . . 107
4.5. Web Workers . . . 108
4.6. Optimizar Pintado . . . 110
4.7. Optimizar Geometría . . . 112
4.8. Optimizar Shaders . . . 114
4.9. Redimensionar Canvas CSS . . . 116
4.10. Flujo de Datos . . . 117

5. Planificación y Costes **119**
5.1. Planificación y Costes . . . 119

6. Conclusiones **125**
6.1. Evaluación de Objetivos . . . 125
6.2. Dificultades . . . 129
6.3. Conclusiones personales . . . 131

A. Modelos Wavefront **132**
A.1. Introducción . . . 132
A.2. Especificación . . . 133
A.3. Importación . . . 134

B. Modelos MD5 **136**
B.1. Introducción . . . 136
B.2. Especificación . . . 138
B.3. Importación . . . 144
B.4. Skinning GPU . . . 144

Bibliografía **151**

Lista de Imágenes

1.1. HTML5 Multi Plataforma . . . 2
1.2. Esquema Trabajo Compartido . . . 7

2.1. WebGL Logo . . . 8
2.2. OpenGL ES Logo . . . 9
2.3. OpenGL ES Pipeline . . . 10
2.4. Vertex Shader . . . 11
2.5. Rasterization . . . 12
2.6. Fragment Shader . . . 13
2.7. Per Fragment Operations . . . 14
2.8. WebGL Rendering Pipeline . . . 15
2.9. Chromium Logo . . . 18
2.10. Chromium Rendering Process . . . 19
2.11. Chromium GPU Process . . . 20
2.12. Soporte WebGL Mayo 2012 . . . 22
2.13. Webgl por Sistema Operativo . . . 22
2.14. Webgl por Navegador . . . 23
2.15. Tendencia navegadores Mayo 2012 . . . 24
2.16. Tendencia Sistemas Operativos Mayo 2012 . . . 24
2.17. Mecanismo de un ataque DoS en WebGL . . . 27
2.18. Historia 3D estándares abiertos . . . 29
2.19. Web3D logos . . . 33

3.1. Simulación de un borrador del jugador principal del juego DOOM3 . . . 35
3.2. Esquema General del Juego . . . 36
3.3. Captura de Pantalla del comienzo del juego . . . 37
3.4. Captura de Pantalla del juego en funcionamiento . . . 37
3.5. Captura de Pantalla del juego mostrando las Bounding Boxes . . . 38
3.6. Captura de Pantalla del final del juego . . . 38
3.7. Diagrama Entidad-Relación . . . 41
3.8. Modelo Iterativo Incremental . . . 43
3.9. Control de Versiones . . . 47
3.10. Modelo de Diseño . . . 48
3.11. Diseño Coponente Render UML . . . 50
3.12. Diagrama Secuencia Render Loop . . . 51
3.13. Diseño Componente Jugador Principal UML . . . 52
3.14. Diseño Componente Cámara UML. . . . 52
3.15. Diseño Componente Modelos MD5 UML. . . . 53

3.16. Diseño Componente Audio UML 54
3.17. Diseño Componente Físicas UML 55
3.18. Diseño Componente Shaders Factory UML 56
3.19. Diseño Componente Luces UML 57
3.20. Diseño Sistema UML 58
3.21. Diagrama de Secuencia de un frame 60
3.22. Secuencia de los Sistemas de coordenadas 72
3.23. Cámara en tercera Persona 73
3.24. Tipos de Modelos usados 76
3.25. Modelo Phons de Iluminación 77
3.26. Intensidad Luz Difusa 78
3.27. Intensidad Luz Especular 78
3.28. Físicas en un Web Worker 93

4.1. Etapa de un frame en WebGL 104
4.2. Buffering Input Usuario 104
4.3. Rendimiento Pipeline 107
4.4. Esquema Web Worker 108
4.5. Esquema Web Worker con la física 109
4.6. Pintado Lógico en Canvas 110
4.7. Pintado Lógico en WebGL 110
4.8. Orden de Pintado en WebGL 111
4.9. Triángulos sin índices 112
4.10. Triángulos con índices 112
4.11. Estructura de datos 113
4.12. Pirámide de número de llamadas 114
4.13. Redimensionar Canvas con CSS 116
4.14. Render Sync Flush 117
4.15. Render gl Flush 118

5.1. Diagrama Gantt de la Planificación 120
5.2. Diagrama de Rendimiento Personal 121

A.1. Modelos Wavefront 132

B.1. Modelos MD5 137
B.2. Representación de Skeleton y Skinning 139

Lista de Código Fuente

2.1. Crear un Canvas . . . 16
2.2. Preparar el contexto WebGL . . . 16
2.3. Inicializar WebGL . . . 17
3.1. Ejemplo Data Hiding Javascript . . . 64
3.2. Clases en Javascript . . . 65
3.3. Closures en Javascript . . . 65
3.4. Game Loop en HTML5 . . . 67
3.5. Pintado sin instanciación . . . 68
3.6. Pintado con instanciación . . . 69
3.7. Patrón Módulo en Javascript . . . 70
3.8. Matrices de Cámara . . . 73
3.9. Ejemplo de pintado de un modelo completo . . . 74
3.10. Entidad Luz . . . 79
3.11. Entidad Material . . . 79
3.12. Entidad Material . . . 80
3.13. Carga y representación de los Programas de Shading . . . 82
3.14. Vertex Shader . . . 84
3.15. Fragment Shader . . . 86
3.16. Fire Handler . . . 89
3.17. Movement Handler . . . 90
3.18. Implementación Componente Audio. . . . 94
4.1. Javascript Arrays Mala Idea . . . 96
4.2. Javascript Arrays Buena Idea . . . 97
4.3. Javascript Objects Mala Idea . . . 97
4.4. Javascript Objects Buena Idea . . . 98
4.5. Javascript Garbage Collector Mala Idea . . . 100
4.6. Javascript Garbage Collector Buena Idea . . . 100
4.7. Memoria Render Loop Mala Idea . . . 101
4.8. Memoria Render Loop Buena Idea . . . 102
4.9. Buffering Javascript . . . 105
4.10. Lectura del Buffer Javascript . . . 105
4.11. Dependencia de Datos GLSL . . . 115
A.1. Especificación Modelos Wavefront . . . 133
B.1. Skinning por CPU . . . 145
B.2. Skinning por GPU . . . 146

Índice de cuadros

2.1. Tabla tecnologías Web 3D - 1 32
2.2. Tabla tecnologías Web 3D - 2 32

5.1. Tabla de tareas y horas 122
5.2. Tabla de dedicación por perfil 123
5.3. Tabla de costes de trabajadores 124
5.4. Tabla de costes de recursos 124
5.5. Tabla de costes totales del proyecto 124

Abreviaciones y Anglicismos

HTML	**H**yper**T**ext **M**arkup **L**anguage
WebGL	**Web** **G**raphics **L**ibrary
OpenGL ES	**Open** **G**raphics **L**ibrary **E**mbedded **S**ystems
GPU	**G**raphics **P**rocessing **U**nit
CPU	**C**central **P**rocessing **U**nit
ANGLE	**A**lmost **N**ative **G**raphics **L**ayer **E**ngine
IPC	**I**nter **P**process **C**ommunication
API	**A**pplication **P**rogramming **I**nterface
JS	**J**ava**S**cript
FPS	**F**rames **P**er **S**econd

**Nota:* Se ha abusado de muchos anglicismos en palabras técnicas provinientes del inglés. Si los tradujéramos perderían rigor o precisión ya que su origen no es el castellano, lengua de este proyecto.

Capítulo 1

Introducción

1.1. Motivación y Contexto

A día de hoy, Mayo 2012, los juegos en la Web están empezando a crecer de forma muy interesante. Esto se debe principalmente a tres factores:

- El primer factor de todos es que la **tecnología** de hoy en día está preparada para albergar juegos en la Web. Hay alto ancho de banda y el Hardware de las plataformas Web es cada vez mejor. Un claro ejemplo son las típicas aplicaciones de escritorio (ofimática, programas multimedia, gestores de correo, juegos, etc) que están siendo absorbidas por el navegador como aplicaciones web, Web Apps.
- Hasta hace unos años los juegos eran territorio de aplicaciones de PC y de consolas. Gracias al primer punto y a los **dispositivos móviles** la web se ha introducido en todas las plataformas existentes. Ahora están tanto en televisiones, SmartPhones, PC's, Tablets, etc.
- Mucho de los navegadores hacen que esto sea posible gracias a los **estándares de la Web**. Estos estándares hacen que un mismo programa puede ser ejecutado en varias plataformas, bajo navegadores diferentes con idéntico resultado. Estos estándares son tales como HTML5 [1] y ECMA SCRIPT[2]

Por lo tanto, los navegores están en todos los lados y preparados. ¿Pero qué le falta para competir? Lo que siempre había dañado mucho a la Web es el rendimiento. Bajo rendimiento e inexistencia de aceleración 3D. Pero ya no es así, gracias a WebGL y HTML5 esto ya es posible.

[1] Es la quinta revisión del lenguage básico de la World Wide Web, HTML.
[2] Especificación de un lenguaje de programación en el que se basa Javascript.

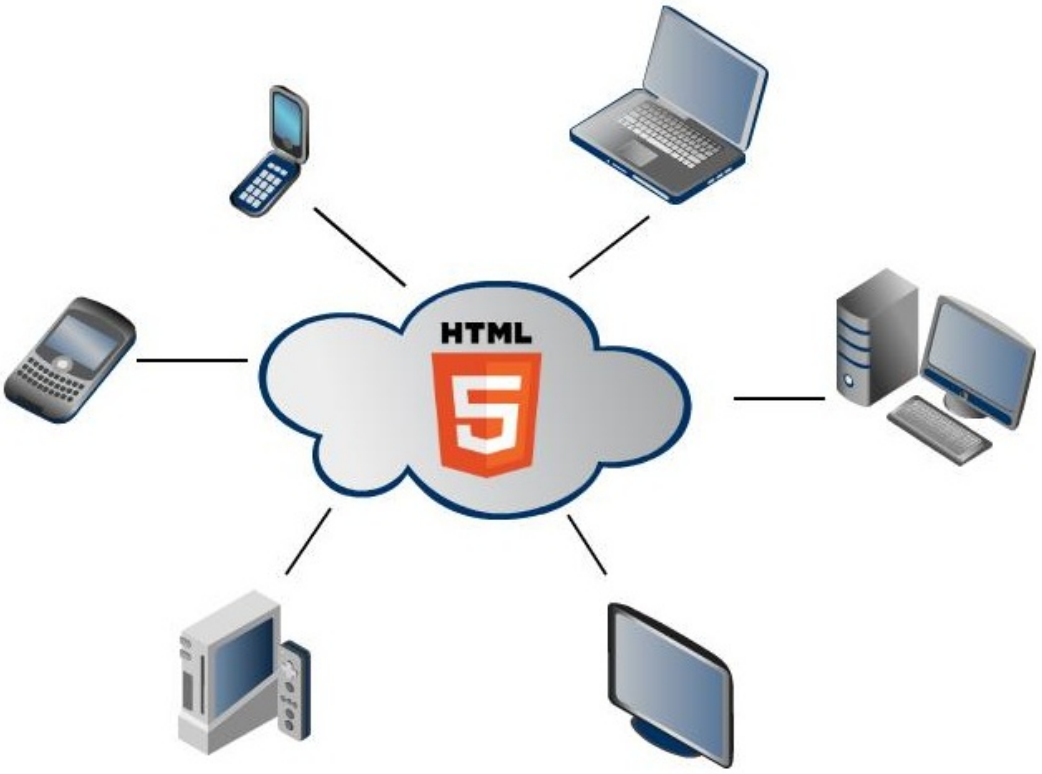


Imagen 1.1: HTML5 multi-plataforma: *"write once, run anywhere"*

Hasta hace poco tiempo los navegadores han tenido juegos pero ya sabemos todos de qué tipo. Juegos *casual*, de poco detalle gráfico e incomporable con el resto de juegos del mercado. No quiere decir que no triunfarán sino que a nivel técnico están muy por debajo.

La principal motivación, como amante del desarrollo de videojuegos, es conocer y explorar esta nueva plataforma. La programación de videojuegos no es solo muy entretenida y gratificante sino que además es difícil porque siempre se quiere llevar la aplicación al máximo de recursos para proporcionar la máxima calidad de contenido. El gran reto de este proyecto, es intentar llevar esta expresión al máximo para ofrecer un juego digno en aceleración 3D en la Web que abra las puertas a otros proyectos y ofrezca soluciones a problemas tencológicos en este ámbito.

1.2. Objetivos

1.2.1. Objetivo Principal

El objetivo principal de este proyecto es dar una solución tecnológica al desarrollo de un juego completo, en todos sus ámbitos (diseño, implementación y experiencia del usuario dentro de la Web) y utilizando WebGL como tecnología Web 3D. Estamos hablando de un juego que contenga físicas, iluminación dinámica, 3D, modelos, Skinning, Audio y un renderizado digno de los juegos de escritorio. En otras palabras, sabemos como se hace un juego en PC, en los móviles, en las consolas, etc. **Pero con estas características ¿En la Web?**.

No se pretende conseguir un juego cerrado que se publique sino ofrecer unos patrones y técnicas para desarrollarlo en estas nuevas tecnologías: HTML5 y WebGL.

1.2.2. Objetivos Secundarios

El objetivo principal es muy ambicioso y extenso. Si analizamos todo el trabajo que hay que realizar es necesario especificar hasta donde queremos llegar subdividiendo el objetivo principal en otros secundarios más concretos:

Crear un juego innovador, accesible a la mayoría de plataformas sin instalador, sin plugins, sólo con un navegador apto y una tarjeta gráfica.

Casi la totalidad de todos los juegos en los navegadores requieren de un plugin para ser ejecutados, ya sean: Flash, Unity o Java. Gracias a WebGL podemos acceder a la GPU sin una pasarela privativa. Esto permite poder renderizar casi cualquier elemento 3D en la Web. WebGL surgió hace justamente un año y se han visto grandes propuestas pero no juegos que combinen todos los elementos de un videojuego típico.

Usar tecnologías libres durante todo el proyecto.

Las plataformas actuales están intentando cerrarse a un tecnología para enfocar el mercado a su antojo. Nuestro objetivo es usar en todo momento tecnologías libres para demostrar el potencial de ellas y no depender de los intereses de un software privativo. En este caso son: HTML5, Javascript DOM API, Javascript y WebGL como tecnologías de desarrollo. Usar HTML5 como base nos permite soñar con distribuir en todas las plataformas que lo apoyan.

Llegar a crear una escena realística en tiempo real

Para conseguir una escena realística entran muchos factores. Iluminación dinámica mediante Shaders. Desarrollar una física para que los elementos en la escena se comporten como en la realidad mediante fuerzas, empezando por la gravedad. Modelos estáticos y dinámicos texturizados y materializados para dar detalle a la escena. La Inteligencia Artificial nos ayudará a modelar comportamientos en los elementos controlados por el sistema. Y las técnicas de procesado nos servirán para crear efectos y detalles dentro del renderizado entre muchas otras.

Crear una sensación de inmersión digna de aplicaciones de escritorio.

Mediante un Gameplay acurado, un alto detalle gráfico, una escena realísitica y un viewport grande es posible desarrollar un juego altamente interactivo entre el jugador y el juego.

Conseguir una tasa de renderizado de 60FPS en el navegador.

La tasa perfecta de renderizado son 60 frames por segundo justo la frecuencia de actualizacion de la mayoría de las pantallas actuales. Esto quiere decir que cada frame se tiene que ejecutar en 17 milisegundos. Y en cada frame actualizar toda la escena, calcular posiciones mediante la física, leer inputs del usuario, calcular colisiones, preparar el renderizado y enviar todo a GPU. Todo este sistema en el navegador parece imposible pero no lo es si se usan las técnicas correctas.

Conocer WebGL en todo su potencial.

WebGL es una API DOM de acceso a la GPU basada en OPENGL ES 2.0. Es una api de muy bajo nivel y simple. La idea es que si WebGL es una API Html5 tiene que estar diseñada para ser accedida desde cualquier dispositivo: PC, smartphones,tablets...Los dispositivos móviles tienen un hardware muy limitado por eso WebGL simplifica el acceso y limita los recursos muchísimo. Hace falta conocer al detalle todo los elementos del Pipeline para aprovechar al máximo su potencial. WebGL funciona mediante un Pipeline programable requiriendo de Shaders (Vertex y Fragment Shaders). Por lo tanto no basta con conocer la API de WebGL sino profundizar en el conocimiento de los shaders para ejecutar la funcionalidad de la aplicación a la perfección y explotar el propóstio de la GPU , la paralelización de cálculos, que nos ofrece WebGL.

Analizar WebGL como futura posibilidad 3D y multi-plataforma.

El proyecto mostrará como está la Web 3D cada día. La idea es describir de qué es capaz esta tecnología y de qué son capaces las plataformas para analizar su futuro como herramienta principal del 3D en la web.

Estudiar estrategias y patrones para adaptarse a la web.

Javascript no es el entorno perfecto para desarrollar un videojuego porque el lenguaje en sí no se creó con ese fin. Por ello vamos a intentar adaptar los patrones de diseño típicos de lenguajes nativos en la Web. Javascript es muy lento comparado con C++ por lo que habrá que modificar el procesamiento para no vernos penalizados por este factor. Esto requiere que mucha funcionalidad típica de CPU tendrá que ser realizada en GPU para aprovechar al máximo los recursos que nos brinda WebGL. La Web en general exige ciertos estándares y requisitos que en un juego normal no existen. Como por ejemplo la carga de la aplicación. Es importante minimizar el impacto de las transferencias usando elementos comprimidos (texturas, modelos) y *catching*.

Explotar las posibilidades de HTML5

Gracias a los nuevos elementos de HTML es posible mostrar contenido multimedia en la Web sin requerir de plugins:

- $< audio >$: Módulo de audio del juego.
- $< video >$: Texturas dinámicas, cinemáticas, etc.
- $< canvas >$: Wrapper 3D y elementos GUI.

El objetivo es usarlos debidamente para que el juego contenga elementos multimedia que lo enriquezcan.

Diseño de alto nivel y escalable para que en el futuro poder añadir más elementos.

La idea es implementar el juego de una forma escalable y bien diseñada para reusar el "motor" en cualquier futuro desarrollo o para cambiar cualquier funcionalidad de forma adecuada.

Asentar los conocimientos de la carrera universitaria de informática.

Este proyecto nos brinda la oportunidad de entrar en todas las ramas de la carrera desde programación avanzada, ingeniería del Software, visualización avanzada, Inteligencia Artifical, algoritmia, estructuras de datos, programación consciente de la arquitectura, física,... Gracias a la ambición del proyecto se va a usar los multiples concocimientos asimilados en la carrera y plasmarlos en este proyecto.

1.3. Proyecto Compartido

Este proyecto ha sido desarrollado entre dos personas, Míquel de Arcayne y yo mismo, Jaime Crespo. El objetivo de hacerlo entre dos personas ha sido simple, unificar los conocimientos de ambos para llegar a profundizar más en la materia. Siendo conscientes de que el proyecto tiene un ámbito personal hemos diferenciado el trabajo en partes.

Este proyecto es un análisis profundo de todas las etapas del Software. Por lo tanto, la parte de especificación ha sido completamente común ya que ambos nos dedicamos a decidir cómo queríamos que fuera el juego. De ahí en adelante nos hemos separado esos requisitos para que cada uno trabajara en cosas diferentes y que cada uno pudiera sacar sus propias conclusiones del trabajo realizado.

Las partes realizadas por cada uno están claramente definidas en la planificación del proyecto, figuras: 5.1 y 5.2. Y todos los componentes de diseño e implementación en esta memoria son de mi propia autoría que en unión con el trabajo de mi compañero resulta en el juego final.

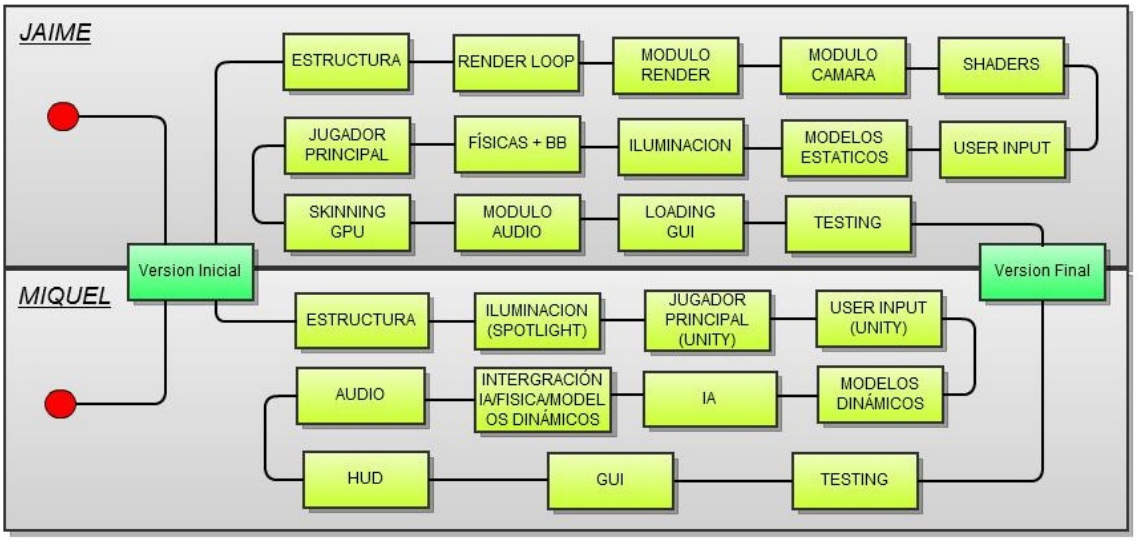


Imagen 1.2: Esquema Trabajo Compartido.

Capítulo 2

WebGL

2.1. ¿Qué es WebGL?

WebGL es una DOM API [1] para crear gráficos en el navegador. Al ser una API ofrece una serie de llamadas a una librería especializada en renderizado 3D. Esta librería está basada en OpenGL ES 2.0. WebGL trabaja bajo el elemento Canvas de HTML5 por lo tanto es accesible desde cualquier lenguaje compatible con DOM, Javascript básicamente.

En una rápida conclusión, para conocer al lenguaje de acceso a WebgGL habrá que referirse a Javascript y para explorar la API de WebGL habra que referirse a OpenGL ES 2.0.

Imagen 2.1

[1] Una DOM API es una interfaz de programación para aplicaciones relacionadas con el Document Object Model para acceder y actualizar contenido, estructura y estilo.

2.1.1. OpengGL ES 2.0

En todo lo relacionado al funcionamiento interno de WebGL tenemos que referenciarnos a OpenGL ES 2.0. La Open Graphics Library (OpenGL) es una librería usada para visualizar datos 2D y 3D , posee múltiples propósitos que soporta aplicaciones para la creación de contenido digital tales como diseño mecánico y arquitectónico, prototipos virtuales, simulaciones, videojuegos, etc,,,

Imagen 2.2

En el mundo de los ordenadores de sobremesa hay 2 APIs principales: DirectX y OpenGL. DirectX es la principal herramienta 3D para los sistemas que usan Microsoft Windows como sistema operativo y es la normalmente usada por la mayoría de juegos de esta plataforma. OpenGL es multi-plataforma usada mayoritariamente en sistemas Linux, Max OS X y Microsoft Windows.

OpenGL ES es una versión simplificada de la existente librería de OpenGL para sistemas más pequeños que contienen muchas más restricciones que la mayoría de plataformas. La version ES (Embedded Systems) está dirigida a dispositivos con limitada capacidad de proceso y memoria, poca banda ancha de memoria, sensibilidad al gasto de energía y falta de floating-point Hardware. Ha sido diseñada según estos criterios:

- OpenGL es una librería muy grande y compleja y se quería adaptar para dispositivos restringidos de hardware. Para conseguirlo se ha tenido que suprimir toda la redundancia existente en la librería. Un buen ejemplo es que la geometría en OpenGL se puede usar en Modo Immediato, Display Lists o Vertex Arrays. Sin embargo en el caso de OpenGL ES sólo se pueden usar Vertex Arrays.
- Eliminar la redundancia es importante pero mantener la compatibilidad con OpenGL también. En la medida de lo posible se ha intentando que la funcionalidad de OpenGL ES sea un subconjunto de las funcionalidad de OpenGL para la compatabilidad de las aplicaciones.

- Se han añadido algunas características nuevas para controlar el consumo de energía. Por ejemplo la precisión de los datos en los Shaders. La precisión de los datos puede incrementar o disminuir el rendimiento de los Shaders y por lo tanto el consumo energético.
- Se han preocupado de ofrecer un mínimo de caraterísticas para la calidad de las imágenes. Muchos dispositivos móviles tienen pantallas pequeñas por lo que requieren que la calidad de los píxeles dibujados sea la mejor posible.

La especificación de OpenGL ES 2.0 implementa un pipeline gráfico programable y deriva de la especificación 2.0 de OpenGL. La derivación especifica que subconjunto de la funcionalidad de la libería principal corresponde a la ES.

2.1.2. Pipeline de renderizado de WebGL

El pipeline de WebGL es exactamente el mismo que OpenGL ES 2.0. Éste implementa un pipeline gráfico programable mediante Shading y consiste en dos especificaciones:

- OpenGL ES 2.0 API specification
- OpenGL ES Shading Language Specification (OpenGL ES SL).

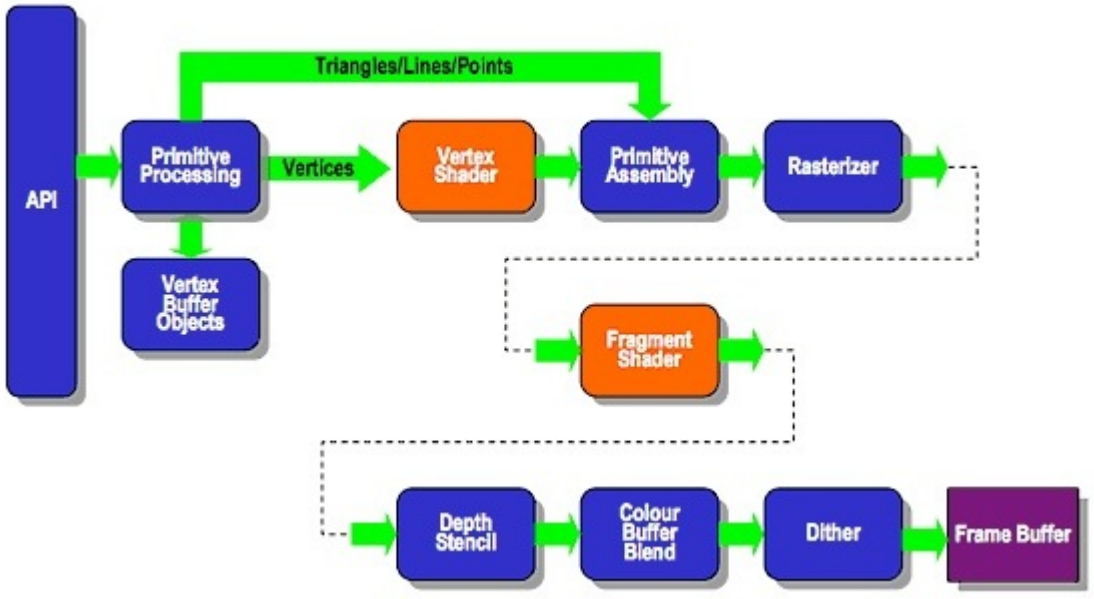


Imagen 2.3: OpenGL ES 2.0 Pipeline Programable

Vertex Shader

El Vertex Shader implementa métodos que operan con los vértices del Pipeline. Los Inputs del Vertex Shader son:

- Attributes - Datos por vértice en los Vertex Arrays.
- Uniforms - Datos constantes en el Vertex Shader.
- Samplers - Tipo específico que representa qué textura usar en el caso que se use.
- Shader Program - Código fuente o ejecutable que describe las operaciones que serán ejecutadas en cada vértice.
- Varyings - Datos de salida que pueden ser usados en etapas posteriores del pipeline.

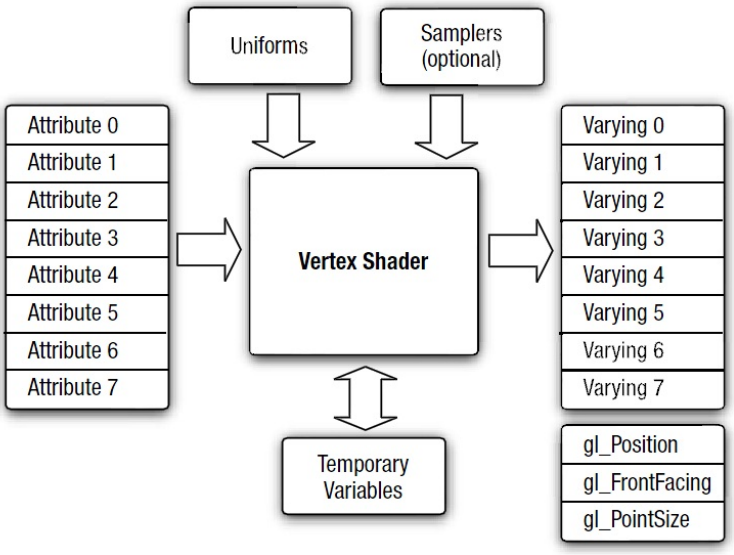


Imagen 2.4: Representacíon Gráfica del Vertex Shader

El objetivo del Vertex Shader es depositar el valor de la posición transformada en la variable predefinida *gl_Position*.

Primitive Assembly

Después del procesamiento del Vertex Shader, la siguiente etapa del pipeline es Primitive Assembly. Una primitiva es un objeto geométrico que puede ser representado gráficamente por OpenGL ES. Así que después del Vertex Shader los vértices son ensamblados en formas geométricas individuales tales como triángulos, líneas o puntos-Sprite con sus respectivas posiciones. En esta etapa también se guarda el estado de si la forma geométrica está dentro o no del frustum de visión de la pantalla. Si la primitiva esta enteramente fuera del frustum éste es descartado y no se tratará en etapas posteriores. Si la primitiva está parcialmente fuera del frustum tendrá que ser recortada para ajustarse al frustum. A este proceso de elección de que está dentro y no se le llama *clipping*. Si se activa la opción de *culling* se ejecutará un procesamiento de caras que descartará aquellas que no estén orientadas con la normal hacia el punto de visión. Despué del clipping y del culling la primitiva está preparada para la siguiente etapa del pipeline, Rasterization.

Rasterization

Es etapa es el proceso que convierte la primitiva (línea, triángulo o point-sprite) en un conjunto 2D de fragmentos que serán tratados posteriormente por el Fragment Shader.

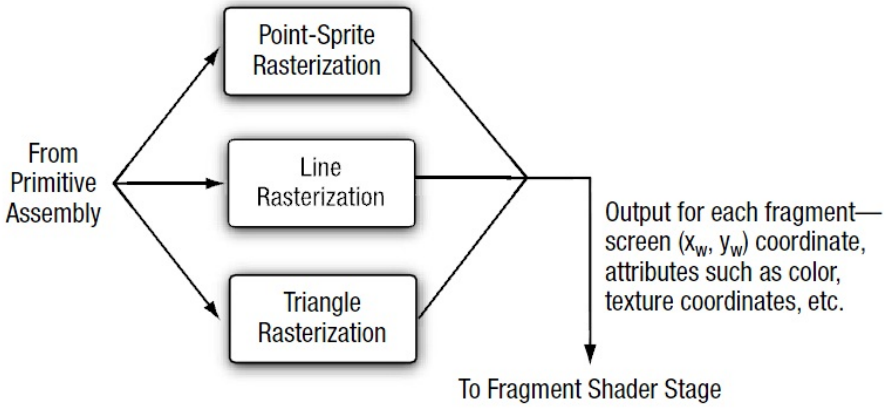


Imagen 2.5: Representacíon Gráfica de la Rasterization

Fragment Shader

El Fragment Shader opera sobre los fragmentos generados en la etapa anterior y trabaja con los siguientes inputs:

- Varyings - Datos de salida del Vertex Shader que ahora son los de entrada de este Shader.
- Uniforms - Datos constantes en el Vertex Shader.
- Samplers - Tipo específico que representa qué textura usar en caso de que se use.
- Shader Program - Código fuente o ejecutable que describe las operaciones que serán ejecutadas en cada fragmento.

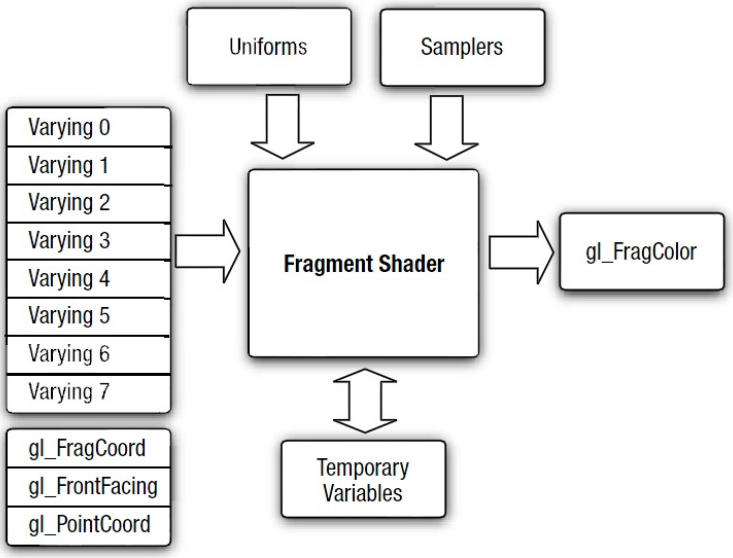


Imagen 2.6: Representacíon Gráfica del Fragment Shader

Los principales objetivos del Fragment Shader son descartar el fragmento o generar un color específico para ese fragmento en la variable predefinida *gl_FragColor*.

Per-Fragment Operations

Una vez tenemos el Fragmento definido con su color y su posición en pantalla (píxel) hay un conjunto de operaciones que aún se tienen que ejecutar antes de enviar el Frame Buffer [1] al dispositivo de salida. El conjunto de operaciones, por fragmento, que se realizan son los siguientes:

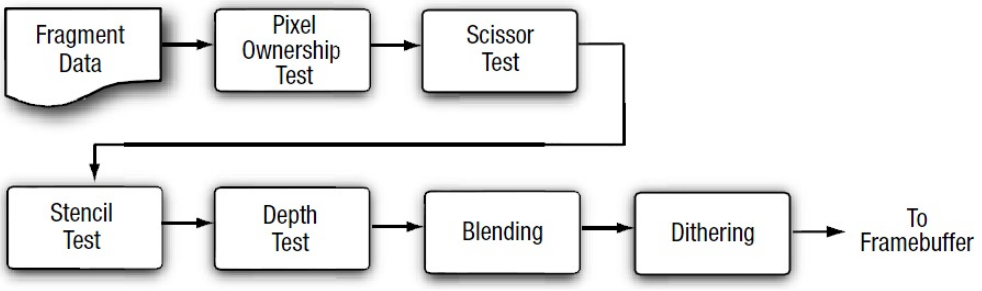


Imagen 2.7: Per Fragment Operations

- Pixel Ownership Test - Es un test que determina si el píxel actual es propiedad de OpenGL ES. Este test permite que el sistema de ventanas descarte o no el píxel. Por ejemplo, podemos tener una pantalla del Sistema Operativo que tape parte del contexto de OpenGL ES y por lo tanto no se tenga que pintar, ya que no es visible.

- Scissor Test - En OpenGL ES se puede definir áreas donde no hay que pintar, las llamadas *Scissors Regions*. Este test se encarga de determinar si el fragmento está dentro o fuera para descartarlo.

- Stencil y Depth Test - Según el valor de los buffers de profundidad (z test) y stencil se determina si el fragmento es rechazado o no.

- Blending - Esta operación combina el nuevo píxel generado con el valor ya guardado en el Frame Buffer en la misma posición. Hay casos en que por ejemplo una transparencia se puede realizar usando esta técnica y combinar el color del elemento transparente con el fondo.

- Dithering - En sistemas que poseen poca resolución de colores esta técinca puede mejorar diche resolución tramando el color de la imagen espacialmente.

Al final de este proceso, si se han pasado todas las etapas, se escribe en el Frame Buffer el resultado correspondiente de dicho fragmento. Ese resultado es un color, una profundidad y un valor Stencil.

[1] El Frame Buffer es un Array bidimensional de píxeles que representa lo que se va a ver en pantalla.

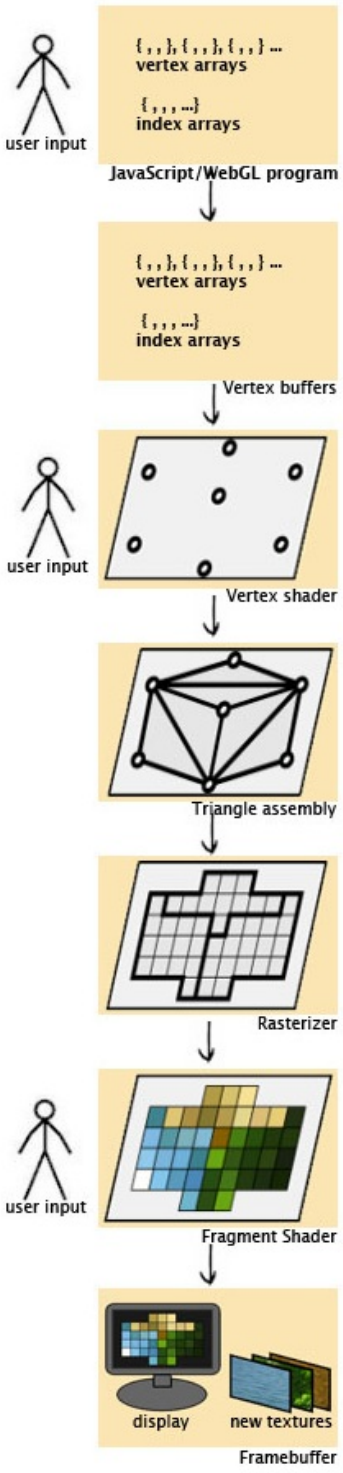


Imagen 2.8: WebGL Rendering Pipeline

Por lo tanto WebGL nos ofrece la posibilidad de usar todo el potencial de OpenGL ES 2.0 desde el navegador mediante Javascript. Si nos fijamos en la actual especificación de WebGL:

Especificación WebGL Mayo 2012

Esta especificación contiene todas esas llamadas que somos capaces de hacer desde el entorno DOM a la máquina de estados de OpenGL ES 2.0. Tal como se muestra en la Figura 2.8 el desarrollador tendrá que especificar mediante los Arrays de vértices, un Vertex Shader y un Fragment shader lo que quiera mostrar en pantalla. Y el proceso interno de WebGL sigue exactamente el mismo procedimiento que OpenGL ES 2.0 con la salvedad de que las llamadas nativas a la unidad gráfica no se harán directamente desde el código Javascript sino que primero serán interpretadas por el propio intérprete de Javascript de cada navegador y este según el estado de ese momento procederá a la ejecución de estas mismas intentando aproximarse a una aplicación nativa.

Esta pequeña diferencia, en cómo internamente WebGL es procesado por los navegadores, creará un pequeño descenso del rendimiento que para pequeñas aplicaciones gráficas no supondrá ningún problema, pero para aplicaciones que vayan al límite, se tendrá que ir con más cuidado para no verse penalizado. Mediante diferentes técnicas y patrones, que se comentarán más adelante, es posible minimizar este impacto y poder llegar a generar escenas realmente complejas, uno de los objetivos de este proyecto.

2.1.3. HTML5 Elemento Canvas

WebGL usa el elemento *Canvas* introducido en el estándard *HTML5* para renderizar. El elemento Canvas se introdujo como herramienta para poder pintar gráficos de forma Scriptable, mayoritariamente con Javascript. Aparte también ha servido para el alojamiento de WebGL.

Por ejemplo para crear un elementos Canvas de 640x480 píxeles es tan sencillo como se define a continuación:

```
<body onload="start()">
  <canvas id="glcanvas" width="640" height="480">
    Your browser doesn't appear to support the HTML5 <code>&lt;canvas&gt;</code>
    element.
  </canvas>
</body>
```

Código Fuente 2.1: Crear un Canvas

Una vez tenemos un elemento Canvas dentro del entorno DOM hace falta recoger un contexto WebGL e inicializarlo. Cuando recibimos un contexto WebGL es necesario inicializarlo tal y como se especifica aquí:

```
function start() {
  var canvas = document.getElementById("glcanvas");
  // Inicializar el Contexto WebGL
  initWebGL(canvas);
  // Solo continuar si Webgl está activo
  if (gl) {
    // Setear a negro el clear color
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    // Activar el test de profundidad
    gl.enable(gl.DEPTH_TEST);
    // Oscurecer partes lejanas
    gl.depthFunc(gl.LEQUAL);
    // Limpiar el buffer color y el depth buffer con clear color.
    gl.clear(gl.COLOR_BUFFER_BIT|gl.DEPTH_BUFFER_BIT);
  }
}
```

Código Fuente 2.2: Preparar el contexto WebGL

A continuación se muestra el código necesario para crear un contexto WebGL:

```
1  function initWebGL(canvas) {
2    // Inicializar la variable gl a null.
3    gl = null;
4    try {
5      // Coger el contexto webgl estándard. Si falla, intentarlo con el
       experimental.
6      gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");
7    }
8    catch(e) {}
9    // Si no se ha podido crear avisar al usuario.
10   if (!gl) {
11     alert("No se ha podido inicializar WebGL. Posiblemente stu navegador no lo
       soporte");
12   }
13 }
```

Código Fuente 2.3: Inicializar WebGL

Con estas tres tiras de código ya tenemos un Canvas WebGL listo para empezar a recibir llamadas.

2.2. Navegadores

Los navegadores son los encargados de recoger la especificación de WebGL e implementarla si quieren dar soporte al contexto WebGL. Para publicar que un navegador soporta WebGL es necesario que dicha implementación pase unos test de conformidad:

Tests de Conformidad para implementaciones de WebGL

2.2.1. Historia

WebGL nació como un experimento que comenzó en Mozilla por *Vladimir Vukicevic* en 2006. En 2009 el consorcio Khronos Group [1] empezó el WebGL Working Group con la participación inicial de *Apple, Google, Mozilla y Opera* entre otros. La primera especificación 1.0 salió en Marzo de 2011.

2.2.2. Funcionamiento Interno de WebGL

Este apartado está basado en la información extraída sobre el desarrollo de *Chromium*[2]. El resto de navegadores que implementan WebGL no adoptan esta misma estructura sobretodo Firefox que no usa *WebKit* [3] como sistema de renderizado. Aún habiendo estas diferencias es importante entender cómo un navegador tan importante como Google Chrome trabaja internamente para ofrecer WebGL. En aplicaciones intensas bajo WebGL, como el caso de este proyecto, el navegador va a tener un papel muy importante en el rendimiento de la aplicación sobre todo en momentos de mucha carga gráfica y procesado. Conocer el funcionamiento interno nos ayudará a adoptar medidas para adaptar nuestra aplicación al sistema local.

Imagen 2.9: Chromium Logo

[1] El Khronos Group es una asociación sin ánimo de lucro centrada en crear estándares abiertos para la reproducción de contenido multimedia. Llevan proyectos tales como OpenGL y OpenGL ES.

[2] Chromium es un proyecto de navegador web de código abierto, a partir del cual se basa el código fuente de Google Chrome. Es de participación comunitaria bajo el ámbito de Google Code.

[3] WebKit es una plataforma para aplicaciones como base para el navegador. Uno de los componentes más famoso es el WebCore para diseñar, renderizar y librería DOM para HTML y SVG.

Proceso de Renderizado en Chromium

Dentro de WebKit existen varios procesos encargados de controlar diferentes aspectos del navegador. Los más importantes son los controladores de CSS, HTML y Javascript. Cuando cualquiera de estos tres controladores quiere renderizar algo en la página enviará los comandos al RPC Buffer [1] que se aloja en un proceso llamado Renderer Process. El navegador se encargará de ir recogiendo esos comandos depositados en el Buffer y transfieriéndolos a un proceso propio del navegador llamado GPU Process. Es importante observar como esta estructura es compartida entre todas las ejecuciones del navegador que quieran renderizar algo.

El GPU Process se encargará de comunicarse con la unidad de proceso gráfico mediante los drivers del propio Hardware. Pero hay un par de piezas fundamentales en esta comunicación. Una de ellas es ANGLE [2] que en Windows permite a los usuarios que no tengan los drivers adecuados de OpenGL ES se traduzcan a DirectX. También existe otra traducción llamada SwiftShader para máquinas que no tengan drivers de Hardware adecuados se hará Rasterization vía Software.

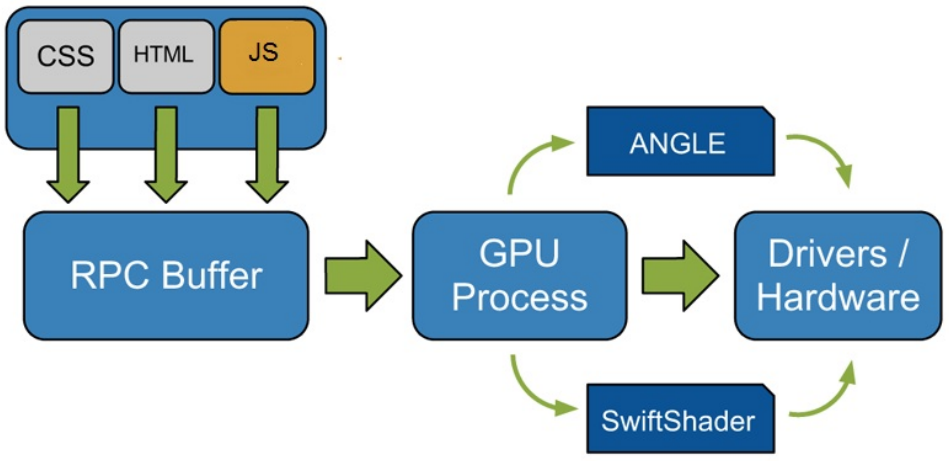


Imagen 2.10: Chromium Rendering Process

[1] Se le llama RPC porque las llamadas al motor gráfico son asíncronas y trabajan bajo el modelo cliente-servidor que más adelante de explicará con detalle y responde a las siglas Remote Procedure Call.

[2] ANGLE : Almost Native Graphics Layer Engine. Es una capa de proceso que traduce comandos OpenGL ES 2.0 a DirectX 9

RPC Buffer

Este Buffer se encarga de almacenar todas las llamadas del entorno Web que tienen que ir a la unidad de proceso gráfico. Este Buffer tiene un tamaño fijo y todos las recursos (texturas, Buffers, comandos, etc) que van a ser subidos a la GPU se depositarán ahí. Si se sobrepasa este Buffer el proceso de renderizado tendrá que vaciarlo automáticamente para liberar recursos y atender a las siguientes peticiones. A este evento de limpieza del RPC Buffer se le llama Sync Flush. Es algo que se tiene que evitar porque para la ejecución de un Sync Flush hasta que se ejecuta todo lo que ya estaba depositado en ese RPC Buffer con el objetivo de poder limpiarlo. En conclusión, hay que evitar llenar el RPC Buffer y que el proceso de renderizado fluya a medida que se ejecuta y no estresar al navegador porque nos penalizará con ello. En los apartados de optimizaciones se hablará de cómo evitar estos Sync Flush.

GPU Process

El GPU Process se ejecuta exclusivamente en un SandBox [1] para que el Renderer Process no tenga acceso directo a las llamadas 3D suministradas por el Sistema Operativo. Este proceso trabaja en modo cliente-servidor con la aplicación:

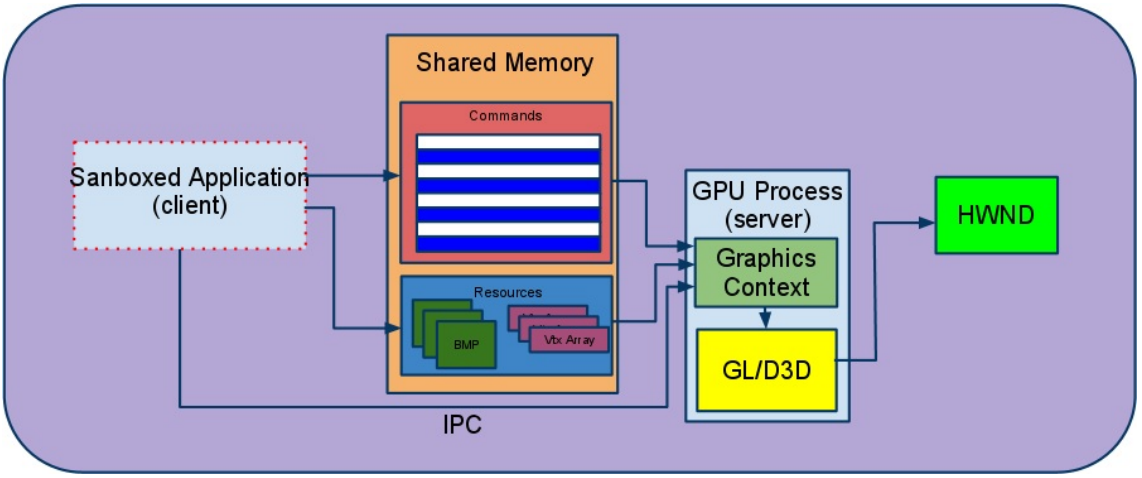


Imagen 2.11: Chromium GPU Process

[1] Aislamiento de proceso, mediante el cual, se pueden ejecutar distintos programas con seguridad y de manera separada. A menudo se utiliza para ejecutar código nuevo, o software de dudosa confiabilidad, con objeto de evitar la corrupción de datos del sistema en donde estos se ejecutan.

- Cliente - El Renderer Process es el cliente y en vez de ejecutar directamente las instrucciones en la unidad de proceso gráfico, las serializa y las deposita en un Buffer de comandos en un trozo de memoria compartido entre él y el proceso servidor.
- Servidor - El GPU Process trabaja aislado y recoge los datos serializados, los parsea y los ejecuta apropiadamente en la unidad de proceso gráfico.

Como vemos en el gráfico, la Shared Memory es el RPC Buffer donde residen todos los recursos de nuestra aplicación en cada llamada. Como casi todas las llamadas de OpenGL no tienen valores de retorno el cliente y el servidor pueden trabajar asíncronamente para mantener un buen rendimiento y no tener que validar ningún retorno. En algunos modos de debug o llamadas muy específicas es necesario controlar retornos para recoger ciertos valores del servidor, para estos casos se usa un mecanismo IPC [1]. Los beneficios de que el GPU Process se ejecute aislado son:

- Seguridad - La lógica de renderizado trabaja aislada no compartiendo recursos básicos.
- Robusteza - Si el GPU Process se interrumpe o deja de funcionar no interfiere en el resto de procesos y no para la ejecución del navegador.
- Uniformidad - Estandarizando OpenGL ES 2.0 como la API de renderizado permite manter un único código para las diferentes distribuciones de Chromium.

Por último vemos como el contexto gráfico accede directamente a las librerias nativas del Sistema Operativo GL, Direct3D , o Direct3D mediante ANGLE.

Drivers

Los *Drivers* de las tarjetas gráficas no han sido diseñados con la seguridad en mente y es relativamente fácil que en *Drivers* un poco antiguos haya agujeros de seguridad importantes. Ahora que la GPU va a estar expuesta al mundo Web, los navegadores están controlando muy de cerca qué tarjetas gráficas son aptas para WebGL. En el siguiente link está la lista de tarjetas aceptadas por navegador:

http://www.khronos.org/webgl/wiki/BlacklistsAndWhitelists

[1] IPC: Inter-process communication es un conjunto de métodos para intercambiar datos entre hilos o procesos de ejecución.

2.2.3. Soporte Mayo 2012

Tal y como se muestra en la Figura 2.12 los navegadores principales que soportan WebGL son Firefox, Chrome, Safari y Opera en sus últimas versiones. Hace falta destacar que Firefox lo soporta completamente pero está marcado como parcialmente soportado, ya que para aquellas tarjetas gráficas no permitidas no hay solución posible y se niega el acceso. En caso de Chrome han usado Swift Shader para hacer renderizado vía Software.

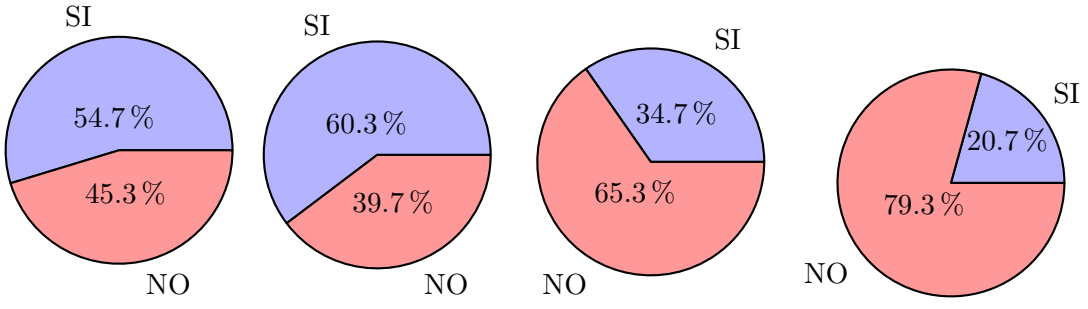
Soportado | No Soportado | Parcialmente Soportado | Soporte Desconocido

IE	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Opera Mobile	Android Browser
	3.6						10.0	2.1
6.0	9.0				3.2		11.0	2.2
7.0	10.0				4.0-4.1		11.1	2.3
8.0	11.0	17.0	5.0		4.2-4.3		11.5	3.0
9.0	12.0	18.0	5.1	11.6	5.0	5.0-6.0	12.0	4.0
10.0	13.0	19.0	5.2	12.0				
	14.0	20.0						

IMAGEN 2.12: Soporte de WebGL en los navegegadores Mayo 2012 Información extraida de CanIUse.com.

En las siguientes imágenes muestro porcentualmente qué usuarios soportan WebGL según el Sistema Operativo y a continuación por Navegador en sistemas operativos. Hay que entender que los porcentajes son dependientes de la cantidad de usuarios que usa ese Sistema Operativo, por ejemplo, el 100 % de los usuarios de Windows son alrededor del 80 % del total por eso la influencia de Windows es grande en la habilitación de WebGL.

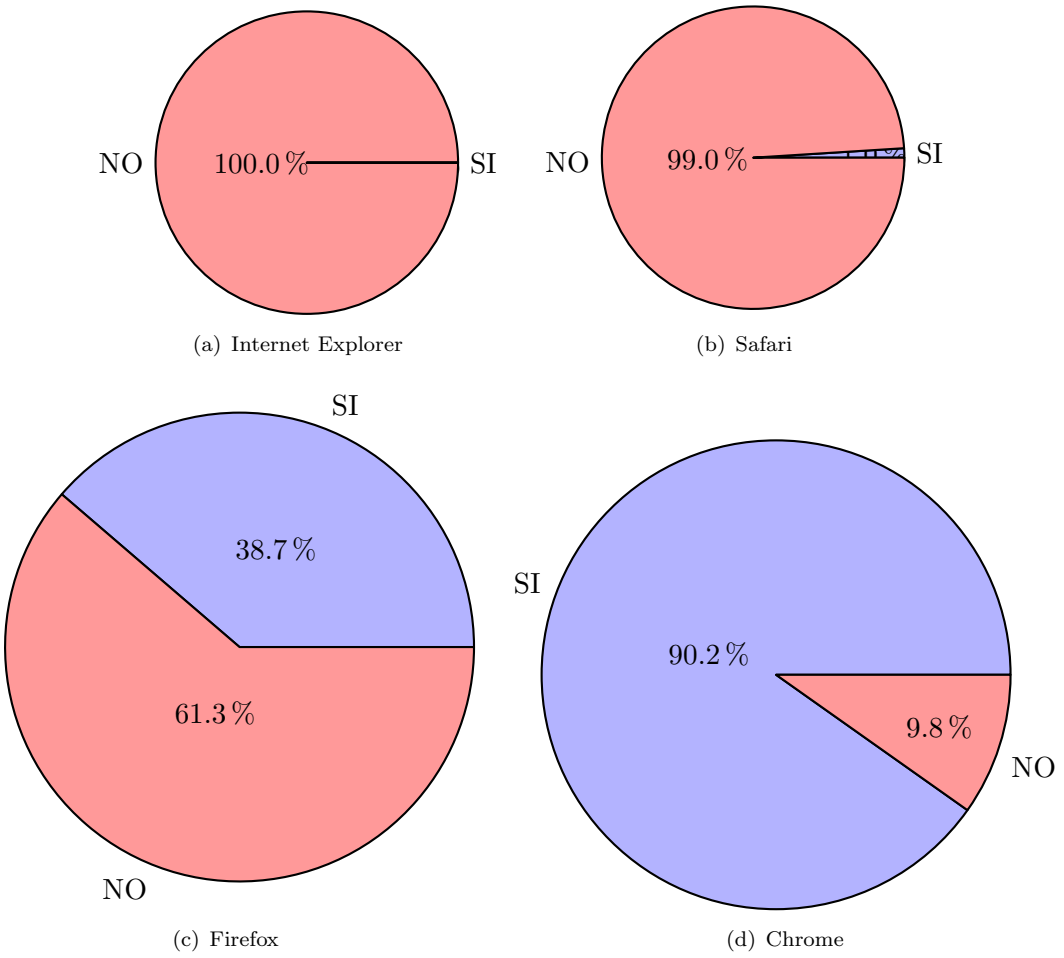
WebGL por Sistema Operativo



(a) Todos los Sistemas Operativos (b) Windows (c) Linux (d) MacOS

IMAGEN 2.13: Porcentajes de aceptación de WebGL por Sistema Operativo. Información extraída de webglstats.com.

WebGL por Navegador



(a) Internet Explorer

(b) Safari

(c) Firefox

(d) Chrome

Imagen 2.14: Porcentajes de aceptación de WebGL por Navegador. Información extraída de webglstats.com

.

Tendencia Cuota de uso del Navegador

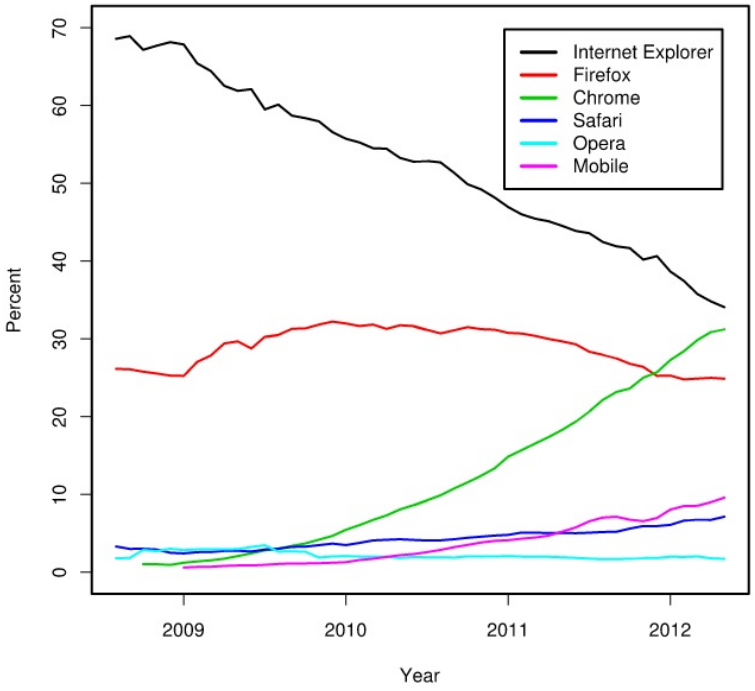


Imagen 2.15: Tendencia cuota de uso de los navegadores. Información extraida de StatCounter.com.

Cuota de uso del Sistema Operativo

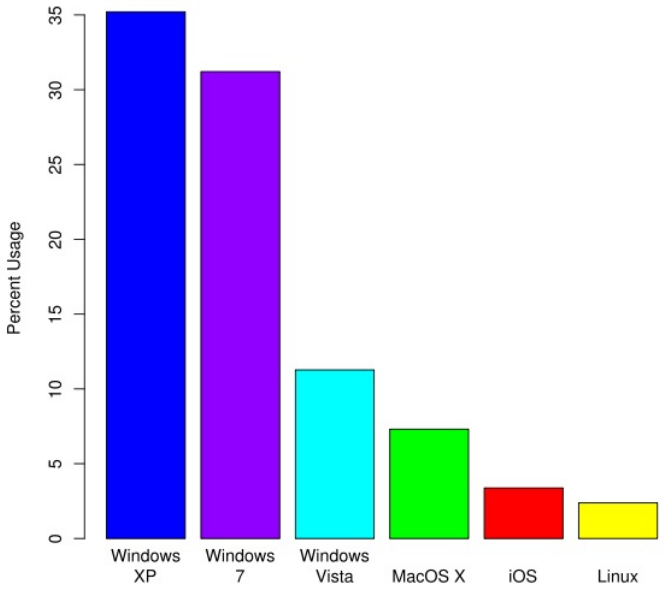


Imagen 2.16: Cuota de uso de los sistemas operativos. Información extraida de StatCounter.com.

Viendo estos datos parece que el futuro de WebGL está asegurado. Si tenemos en cuenta que el Sistema Operativo que mejor lo soporta es el que más se usa (Windows) y lo mismo para el navegador (Chrome) que su uso está creciendo cada día más y es cuestión de semanas para que se proclame el más usado, se puede decir con cierta seguridad que el soporte de WebGL sólo va a seguir creciendo.

Hay que remarcar que Firefox tiene menos soporte que Chrome un 38 % vs el 90 % de Chrome ya que Chrome usa una alternativa para los ordenadores con gráficas no permitidas, SwiftShader. Safari soporta WebGL pero viene desactivado por defecto por lo tanto su porcentaje es casi nulo. Con el tiempo, todos los navegadores que tengan WebGL tendrán el 100 % de soporte, ya que las nuevas gráficas vendrán con drivers más seguros y ajustados a las necesidades actuales. Así que el futuro de esta tecnología está asegurado en cuanto a soporte. Sin embargo hace falta analizar qué otras tecncologías existentes hay en el mundo Web. El único punto negro es Internet Explorer.

2.2.4. Internet Explorer

Microsoft está en decadencia en cuanto a tecnologías Web y cuota de navegador pero aún es el navegador más usado por lo tanto es un lastre que los desarrolladores web no puedan disfrutar de un soporte total.

En un Blog de Microsoft se justifica porqué no se implementa WebGL en sus navegadores. En resumen decían:

«*Creemos que WebGL probablemente se convertirá en una fuente continua de vulnerabilidades difíciles de solucionar. En su forma actual, WebGL no es una tecnología que Microsoft puede admitir desde una perspectiva de seguridad.*»

En el siguiente apartado nos centraremos en la seguridad. Conociendo la filosofía de Microsoft en cuanto a tecnologías abiertas podemos decir que WebGL compite directamente con la API DirectX de Microsoft, que es más popular en Windows que OpenGL, y no hay ningún interés por parte de Microsoft de popularizar esta tecnología. El futuro de WebGL en Internet Explorer es bastante incierto. Microsoft lo rechaza por temas de seguridad pero se sospecha que tiene más interés en proporcionar su propio framework 3D basado en DirectX.

Dicho esto, Google, Mozilla, Opera y Apple admiten WebGL y tienen el soporte de los dos grandes empresas de Hardware gráfico: Nvidia y AMD. Así que WebGL tendrá un gran apoyo pero no total y viendo la tendencia de uso, cada vez más.

2.2.5. Seguridad

WebGL es una tecnología muy nueva y accede profundamente al Software y al Hardware del ordenador. Accede a los controladores de la tarjeta de vídeo y éstos no están implementados con la seguridad en mente. Se han detectado vulnerabilidades pero no tanto por parte de WebGL sino más por parte de los controladores que acceden a los recursos hardware, problema que tendrá Microsoft si algún día se dedica a hacer algo similar. Así fue de contundente la respuesta de Mozilla al artículo de Microsoft. Les invitaban a ayudar y enriquecer la Web en lugar de rechazar y privar.

WebGL no es un plugin, como Flash o ActiveX, sino un *built-in* [1] del navegador y no proporciona el acceso nativo extremo que por ejemplo tiene ActiveX, no se puede escribir en disco, no se puede acceder a la memoria principal y no se puede ejectuar código CPU fuera del SandBox de Javascript. Sus problemas de seguridad están relacionados sólo con el Hardware gráfico.

¿De qué vulnerabilidades estamos hablando?

En Mayo de 2011, dos meses después de la primera versión de WebGL, la firma Context Information Security publicó dos vulnerabilidades presentes en Google Chrome y en Mozilla Firefox. Hubo un gran revuelo ya que la especificación no tenía ni 3 meses. La dos vulnerabilidades de las que estamos hablando son:

- Robo de Imágenes Cross-Domain.
- Denegación de servicio.

Robo de Imágenes Cross-Domain.

Uno de los puntos fundamentales de la seguridad en la especificación DOM son las fronteras de los dominios. Estas fronteras previenen acceder a contenido autenticado y confiado de un dominio a otro. WebGL permitió el uso de textura Cross-Domain. Es totalmente aceptable poder cargar una imagen externa, fuera de tu dominio, en tu DOM porque no tienes forma de acceder al contenido y descifrar que estás mostrando. Pero WebGL mediante un Shader malicioso puede llegar a interpretar esa imagen, convertida en textura y actuar en consecuencia leyendo los píxeles correspondientes.La respuesta de los navegadores fue muy rápida y contundente: negar texturas Cross-Domain y activar la política del mismo origen. Esta política es una medida de seguridad para Scripts en la parte de cliente y previene que un documento o Script cargado en un origen pueda cargarse o modificar propiedades de un documento desde un origen diferente.

[1] Programa interno, parte original del navegador.

Denegación de servicio.

Un ataque de denegación de servicio, también llamado ataque DoS [1], es un ataque a un sistema de computadoras o red que causa que un servicio o recurso sea inaccesible a los usuarios legítimos. Normalmente provoca la pérdida de la conectividad de la red por el consumo del ancho de banda de la red de la víctima o sobrecarga de los recursos computacionales del sistema de la víctima.

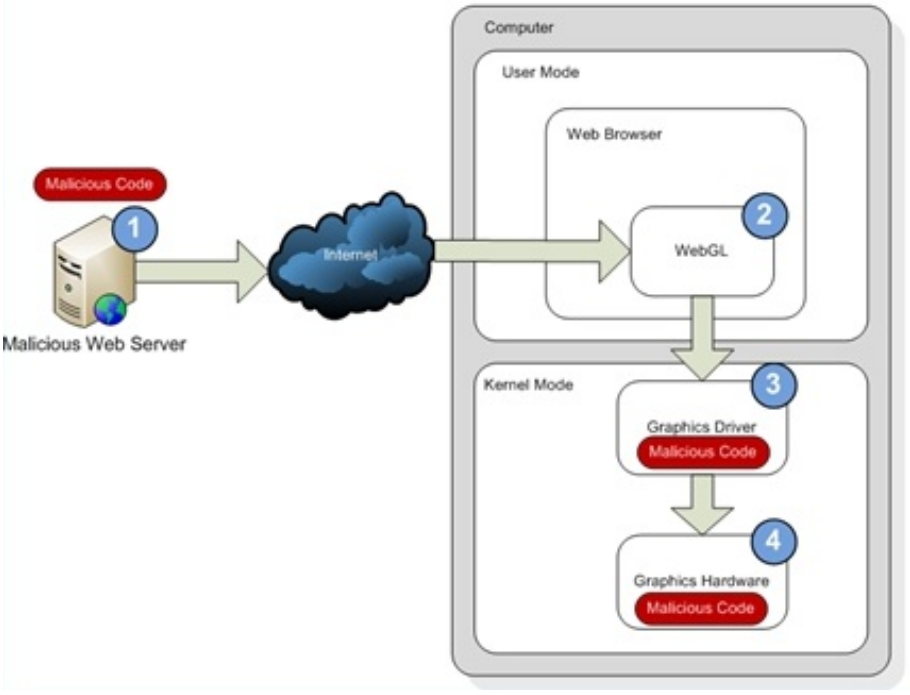


IMAGEN 2.17: Mecanismo de un ataque DoS en WebGL

El procedimiento de este ataque es el siguiente:

- 1 - Un usuario visita un sitio donde reside un script WebGL malicioso.
- 2 - El componente WebGL sube cierta geometría y cierto código en forma de Shader a la tarjeta gráfica del usuario.
- 3 - El código o la geometría se aprovecha de Bugs o *Exploits* [2] en los drivers de la tarjeta gráfica.
- 4 - El Hardware gráfico puede ser atacado causando que todo el sistema se paralice.

[1] DoS: de las siglas en inglés Denial of Service

[2] Exploit: del inglés aprovechar, secuencia de comandos con el fin de causar un error o un fallo en alguna aplicación, a fin de causar un comportamiento no deseado

La solución adoptada por los navegadores que sufrían esta vulnerablidad ha sido restringir el acceso a todas aquellas tarjetas gráficas con drivers vulnerables (ver página 21 para consular la lista). En ningún caso el Exploit puede ofrecer control de la máquina o robo de información.

A día de hoy, Mayo 2012, no hay ninguna entrada relacionada con WebGL en la National Vulnerability Database que referencie a las versiones actuales de producción de los navegadores.

2.2.6. WebGL en móviles y Tablets

Sólo hay un navegador que soporte WebGL en SmartPhones y Tablets: OperaMobile. Los navegadores son una aplicación en los móviles muy costosa, podríamos decir que la más costosa. Todos los móviles quieren ofrecer funcionalidades nuevas que funcionen bien. No tiene sentido que por ejemplo, Google soporte WebGL en el Android Browser si no es capaz de reproducir sus propios ejemplos de WebGL. WebGL tiene un coste grande, no sólo por Javascript, sino por el compositor Web. Un coste muy alto que, por ahora, no están preparados para asumir en aplicaciones grandes.

Existe el Hardware necesario, WebGL está basado en OpenGL ES 2.0 y es la misma librería que usan los móviles de última generación para renderizar 3D tanto en Android como en iOS (iPhone e iPad). Pero tienen que mejorar el tratamiento de WebGL internamente.

También cabe decir que no le están poniendo mucho énfasis en solucionar esto porque se cree que hay intereses detrás. Tanto Google como Apple no quieren generar una nueva plataforma de videojuegos como WebGL en HTML5 si ya tienen sus propios mercados: Android Market y el App Store. Un claro ejemplo es una implementación que hizo Sony Ericsson para Android 2.3 para el móvil Xperia pero no ha recibido ninguna respuesta por parte del equipo de Google. Otro claro ejemplo es que Apple está soportando WebGL pero sólo para su plataforma de publicidad iAds. En cambio para el resto de aplicaciones ¿Por qué no?

Hay una cierta hipocresía en las grandes empresas cuando hablan de tecnologías libres ya que para navegadores de escritorio están muy interesados en proporcionar WebGL para luchar contra otras tecnologías privativas como Adobe Flash o Unity. Pero cuando ellos son los que dominan el mercado, no les interesa promocionarlo. Por eso el único que ha generado algo interesante, sólo para sumar más adeptos, es Opera Mobile. Cuando uno de los dos mercados se imponga el otro querrá innovar y esperamos que se abra la lata de WebGL.

2.3. State of the art Web 3D

En este capítulo vamos a analizar el estado de la Web 3D y que aportaciones tecnológicas ofrece WebGL respecto a sus competidores.

2.3.1. Historia 3D en la web

En la rama de superior del gráfico 2.18 están representadas las tecnologías abiertas y estándares que han surgido entorno a la Web. En la rama inferior la tecnología principal, OpenGL, a lo largo de la historia. Y cómo éstas han confluido en WebGL bajo el dominio de HTML5 y Javascript en la Web.

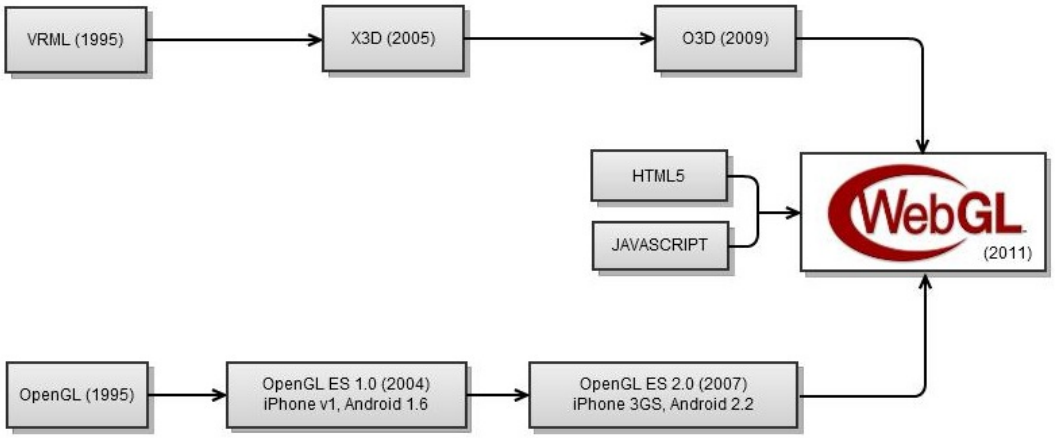


IMAGEN 2.18: Historia 3D estándares abiertos.

Web3D es un término usado para describir contenido 3D incluido en una página HTML visible desde un navegador Web. También el término Web3D se usa para referirse a la experiencia de inmersión en el espacio 3D. Para entender y poder analizar lo que nos proponemos es importante echar un vistazo atrás y ver como ha ido evolucionando. La Web3D empezó en 1994 cuando en una conferencia de World Wide Web en donde se empezó a pensar sobre un lenguaje capaz de describir escenarios e hyperlinks 3D. Es así como apareció la primera versión de VRML (Virtual Reality Modeling Language), un lenguaje de programación que prometía dar a Internet navegación en 3D. Después de un gran interés y desarrollo, las compañías involucradas, decidieron dar un paso atrás en el proyecto. Los motivos fueron la poca participación por parte de las compañías de software y el factor tecnológico; ya que el entorno 3D requiere de características hardware inviables para aquella fecha. Y el ancho de banda de los módems de la época no daba suficiente para conseguir una experiencia realmente buena. Se pensó que el proyecto VRML fue un completo fracaso, pero no fue así. El proyecto no llegó a morir sino se perdió el interés de ir más allá y se dejó de lado. Hasta que en 2004, gracias a la posibilidad de acceder a

un Hardware específico en aceleración de gráficos surgió X3D, el nuevo VRML. X3D es un estándar XML para representar objetos 3D. Pero Microsoft, el grande del momento, consideró que los clientes Web no estarían dispuestos a asumir requirimientos extras para desarrollarlo por eso Internet Explorer nunca implementó VRML or X3D. Si el mayor navegador del momento no incluía esa característica no había ningún interés comercial.

En paralelo, fuera de los estándares Web, grupos privados desarrollaban sus propios productos. Intel, Sillicon Graphics, Apple y Sun hacia 1996 dicidieron hacer una versión Java. Así que colaboraron en ello. El proyecto fue Java3D y fue acabado en 1997 y la primera release en 1998. Y desde 2003 el desarrollo ha sido intermitente hasta que en 2004 se sacó como proyecto de código abierto. Java 3D nunca ha brillado en el mundo Web. Aparte de necesitar un entorno Java en el navegador a modo de plugin su uso se ha visto muy limitado a pequeños ejemplos por culpas de sus deficiencias.

En cuanto a Adobe y Macromedia, antes de que la comprara Adobe, nunca habían trabajado con renderizado 3D real, sino han jugado con sus reproductores de Flash y Shockwave para dar perspectiva a escenas 2D, simulando 3D. A partir de Shockwave 5 se incluyo una API 3D en Shockwave pero sin nada de éxito. Es ahora, en la actualidad, cuando han empezado a desarrollar una API 3D real de bajo nivel, Stage3D, nombre en clave, Molehill.

En 2005, las compañías de videojuegos, Sony entre las más importantes, estaban interesadas en usar algún formato intermedio entre aplicaciones digitales. De ahí surgió COLLADA, COLLAborative Design Activity. Gracias a estas dos nuevas tecnologías, el efecto Web 2.0 y el avance tecnológico han dado paso a nuevas tecnologías para conseguir un verdadero efecto 3D en el navegador, como Webgl, Unity, O3D o Molehill, entre las más importantes. Actualmente la Web3D está un poco inmadura pero las posibilidad de mostrar contenido 3D en tiempo real es totalmente posible. Sólo hay que echar un vistazo a los últimos ejemplos creados por el grupo Mozilla o los famosos experimentos del Google Chrome, como Google Body.

2.3.2. 3D surfing, Hype or real?

"3D surfing, Hype or real?" se refiere a si la experiencia 3D en la web es una realidad o es el *"boom"* de una tecnología nueva que crea expectación.

Cabe decir que aunque Adobe no haya trabajado comercialmente con 3D en la Web de forma nativa, sí que ha proporcionado Shockave y Flash, una forma de reproducir contenido multimedia y gráfico en la Web. Durante 5 años han acaparado casi todo el contenido multimedia en la Web y de forma exitosa. Gracias, en parte a ellos, hay cada vez más necesidad de crear aplicaciones gráficas más complejas, entre ellas las de 3D.

Las causas que anteriormente hicieron que el 3D no inundara la web están ahora cubiertas, principalmente las tecnológicas. La más importante la aceleración 3D que entraremos en detalle adelante. Si a esto le sumamos que los navegadores han pasado a ser la aplicación más usada de un ordenador casual y que cada vez más, las aplicaciones de escritorio están siendo embedidas por el navegador podemos creer que es el momento de la transición 3D. Si echamos un vistazo atrás, 2004, cuando el término Web 2.0 estaba de moda podemos apreciar como ha ido creciendo el número de aplicaciones Webs que antes eran propiamente de escritorio. Desde clientes de correo (Gmail), gestores de fotos (Flicker), Google Maps, Wikipedia, etc. Las lista es interminable, hasta ChromeOS está orientado a la navegación Web.

Por lo tanto, es el momento del 3D en la Web. Entre las aplicaciones 3D, las más jugosas, los videojuegos ya están aquí y este proyecto quiere corroborarlo. Que la Web3D esté ya aquí no quiere decir que todo Internet se convierta en un mundo virtual 3D, sino que las Webs comunes se podrán enriquecer de un contenido mucho más realista y dinámico. El texto seguirá siendo texto, las imágenes seguirán siendo imágenes y nada cambiará radicalmente. Pero por ejemplo, nada impide que sin necesidad de instalaciones previas inicies una Web y estés decorando tu propia casa en 3D con objetos con un detalle muy real y puntos de luz propios de renders ilustrativos.

2.3.3. Competidores

En toda carrera hay competidores y más hoy en día. La tecnología que se lleve el pastel podrá tener el futuro asegurado hasta aquí unos años. Hay mucho interés, sobre todo por las compañías de videojuegos en ver cual es la mejor opción de 3D en la Web. Es importante mencionar que muchas de las tecnologías Web que triunfaron en su momento no eran las de mejor calidad entre sus competidoras. Puede ser que una multinacional la use y se expanda su uso en cadena. Puede ser que su proceso de desarrollo sea mucho más rápido que otras. Puede ser que su coste sea menor. O puede ser que haya pluraridad de tecnologías para el mismo propósito. Es momento de experimentar y arriesgar. En este proyecto se ha decidido por WebGL.

2.3.3.1. ¿En qué podemos experimentar 3D en la Web?

En esta tabla podemos observar las tecnologías más importantes que lidian con 3D actualmente. Se ha añadido DirectX y OpenGl, que aunque no estén basadas en el navegador, son usadas por el resto a modo de Low-level API.

	DirectX	OpenGL	Java3D	FlashMolehill
Para Navegador	No	No	Sí	Sí
Aceleración HW	Sí	Sí	Sí	Sí
Plugin	-	-	Sí	Sí
Calidad Gráfica	Excelente	Excelente	Buena	Regular
Adopción 3D	Alta	Alta	Baja	Baja
Estabilidad	Estable	Estable	Estable	Estable

Tabla 2.1: Tecnologías Web 3D - 1

	Silverlight	Unity	Canvas	WebGL
Para Navegador	Sí	Sí	Sí	Sí
Aceleración HW	No	Sí	No	Sí
Plugin	Sí	Sí	No	No
Calidad Gráfica	Mala	Buena	Mala	Buena
Adopción 3D	Baja	Buena	Mala	Regular
Estabilidad	Estable	Estable	Estable	Estable

Tabla 2.2: Tecnologías Web 3D - 2

2.3.3.2. ¿En qué merece la pena arriesgar?

DirectX y OpenGL podemos descartalas porque están orientadas directamente a lenguajes de escritorio que tengan acceso directo al sistema mediante low-level API's que no interfieren al rendimiento. No son API's orientadas al navegador. Solo han sido mostradas de forma descriptiva para su comparativa.

IMAGEN 2.19: Web3D logos: Adobe Flash, WebGL, Unity, Silvelight, HTML5 y Java3D

Empezando por Java. Java3D lleva en la Web desde hace más de una década y nunca ha sido explotado para fines comerciales, ni para una inmersión en la Web3D. Sufre de agujeros de seguridad, necesidad de un plugin de gran tamaño, como la máquina virtual de Java; más el plugin Java3D. Es molesto con los cuadros de diálogos que aparecen cada vez que se quiere ejecutar algo en él. A pesar de sus ventajas, no está preparado para albergar lo que esperamos de una Web rápida y dinámica.

Flash, el plugin de Adobe, hasta hace poco no soportaba aceleración 3D. Adobe sacó hace poco, la versión 11.0 con stage3D, o Molehill, una API que usa OpenGL o DirectX para el renderizado. El incoveniente, bajo mi punto de vista, es que no sólo tienes que integrar un plugin privativo en la Web sino además conocer Action Script típico y añadir un framework para la adaptación a 3D. En muchos de los foros de desarrolladores de Flash comentan lo difícil que es lidiar directamente con la API 3D por eso el uso de un Framework. Son demasiadas cosas que están fuera de control de un desarrollador que tiene que conocer el estado de muchos sistemas cambiantes. Es importante comentar que la nueva especificación de HTML, HTML5, ha intentando absorber toda el modelaje y animación de formas para evitar que se use Flash en todo a lo que multimedia se refiera. Por lo tanto Flash irá perdiendo peso poco a poco a medida que HTML5 se incorpore más y posiblemente Molehill se vea afectada. Pero es sólo una suposición muy discutible, es difícil prever que puede ocurrir con Flash. Sigue en la cima de contenido multimedia en la Web.

Opción Silverlight. Ha sido la respuesta de Microsoft al gran acaparamiento de Flash. Silverlight sólo trabaja bien en Windows. No es nada popular porque no integra ninguna funcionalidad nueva a las existentes y no soporta aceleración hardware. Y si le quitamos la portabilidad de plataformas se queda en una opción nula. No ha sido desarrollado con el objetivo de albergar contenido 3D pero tampoco puede. Se espera un respuesta de Microsoft viendo el gran avance del resto pero aún no se sabe nada.

HTML5 introduce grandes avances y opciones, una de ellas es el elemento Canvas. El elemento Canvas de HTML5 permite generar gráficos dentro de él. Al no tener aceleración 3D se considera un elemento 2D i/o 2.5D ya que la generación de elementos 3D dinámicamente sin ayuda de hardware específico es inviable. Por lo tanto es una gran opción para desarrollos orientados al 2D. Para 3D real, no es una opción.

Considero que faltan los dos más importantes, Unity y WebGL (no olvidemos Molehill o Stage3D). Unity es una herramienta para la creación de juegos 3D, visualizaciones arquitectónicas o animaciones 3D. Soporta aceleración completa de Hardware y requiere de un plugin para su reproducción. Su entorno de reproducción funciona en Windows, Max OS X, Xbox360, PlayStation3D, Wii, iPad, Iphone y Android. No en Linux. Es impresionante la portatibilidad que han conseguido. Su forma de desarrollo es parecida a la de Flash, orientada a alto nivel y diseño, más que a programación de bajo nivel. Esto puede ser una ventaja y una desventaja a la vez. Permite que todos los desarrolladores de Flash puedan trabajar en Unity pero desconcierta a los reales programadores de videojuegos, acostumbrados a lidiar con cuestiones de bajo nivel. Unity es una versión estable y tiene ya sus adeptos en producción. Varios juegos ya han sido distribuidos a varias plataformas con éxito sobre todo en los móviles. En el navegador aún no han conseguido asentarse como opción real pero lo será por su opción a exportar en Flash. Es un producto privativo, desarrollo de pago y fuera de los estándares HTML5. A día de hoy es la opcion más segura si tu perfil es el correcto para el desarrollo de videojuegos tal y como Unity propone.

WebGL ha sido la respuesta de los navegadores libres, que viendo que han hecho un gran esfuerzo por embellecer HTML5 que no soporta aceleración 3D directa, no quieran ver su navegador tapado por un plugin privativo por lo que han decidido implementar una API directa a OpenGL desde el entorno DOM. Su rendimiento es muy bueno. Se han visto ejemplos realmente complejos corriendo a 60FPS, la frecuencia ideal de renderizado. Su portabilidad también es muy buena está atada a HTML5 pero no es una especificación de él. Pero hay dos grandes desventajas. El mayor navegador del mundo no lo soporta, Internet Explorer. Pero la suma en porcentajes del resto supera la mayoría de IE, así que hay cierta esperanza por la comunidad de que lo acaben implantando. Y la otra desventaja es que el desarrollo de aplicaciones 3D en el entorno DOM y de bajo nivel como es WebGL puede llegar a enloquecer. DOM no es el entorno más correcto ni Javascript el lenguaje ideal para este tipo de desarrollos. Puede asustar y por eso aún no se han visto grandes aplicaciones por parte de grandes empresas 3D, pero sí por parte de particulares y Start-Ups independientes. Pero la viabilidad está ahí, con una mejora del entorno y un pequeño empujón es un gran candidato a ser la plataforma correcta para el desarrollo de 3D en la Web. Y este proyecto quiere ofrecer esas soluciones, entre otras, a estos problemas.

Capítulo 3

El Juego

3.1. Visión Global

El juego es un *Resistence*[1] *Shoot'em up*[2] en perspectiva *Top-Down*[3] en 3D. El objetivo principal del juego es eliminar el máximo número de enemigos sin que te eliminen a tí. Se tiene que conseguir una sensación de sentirse acosado y confundido por la escena y por la multitud de enemigos intentando generar tensión y rápidez de movimiento al usuario. Se proporcionara un escenario cerrado para generar una sensación de acorralamiento y fustración, enemigos de aspecto monstruoso, tales como zombies. Se jugará con la iluminación para generar oscuridad y cierto terror en la escena por no saber controlar la situación. El juego se tiene que ir acelerando y crear más tensión a medida que el tiempo pasa ya que es una cuenta atrás.

Imagen 3.1: Simulación de un borrador del jugador principal del juego DOOM3.

[1]Dinámica de juegos en el que el objetivo es sobrevivir a ataques.

[2]Shoot'em up: género de videojuegos en los que que el jugador controla un personaje y dispara contra hordas de enemigos que van apareciendo en pantalla

[3]Perspectiva de visión del jugador principal en tercera persona y desde arriba hacia abajo, viendo como interactúa el jugador principal con la escena y los enemigos

3.2. Esquema General

Viendo la definición podemos generar un esquema principal de nuestra aplicación que nos de una idea de los componentes más importantes involucrados. La aplicación tiene que tener tres sistemas básicos: aplicación, lógica y vista. La aplicación va a ser la base donde todo se va a desarrollar. La lógica va a ser un conjunto de subsistemas encargados de hacer que la aplicación se comporte como debe. Si quieres hacer un juego con la definición anterior va a hacer falta que la lógica tenga Inteligencia Artificial, físicas, eventos tales como disparos y ataques. Y la vista va a ser la encargada de mostrar todo lo necesario al usuario mediante modelos, audio y un renderizado más un tratamiento de los eventos de usuario para manejar la posición del jugador principal.

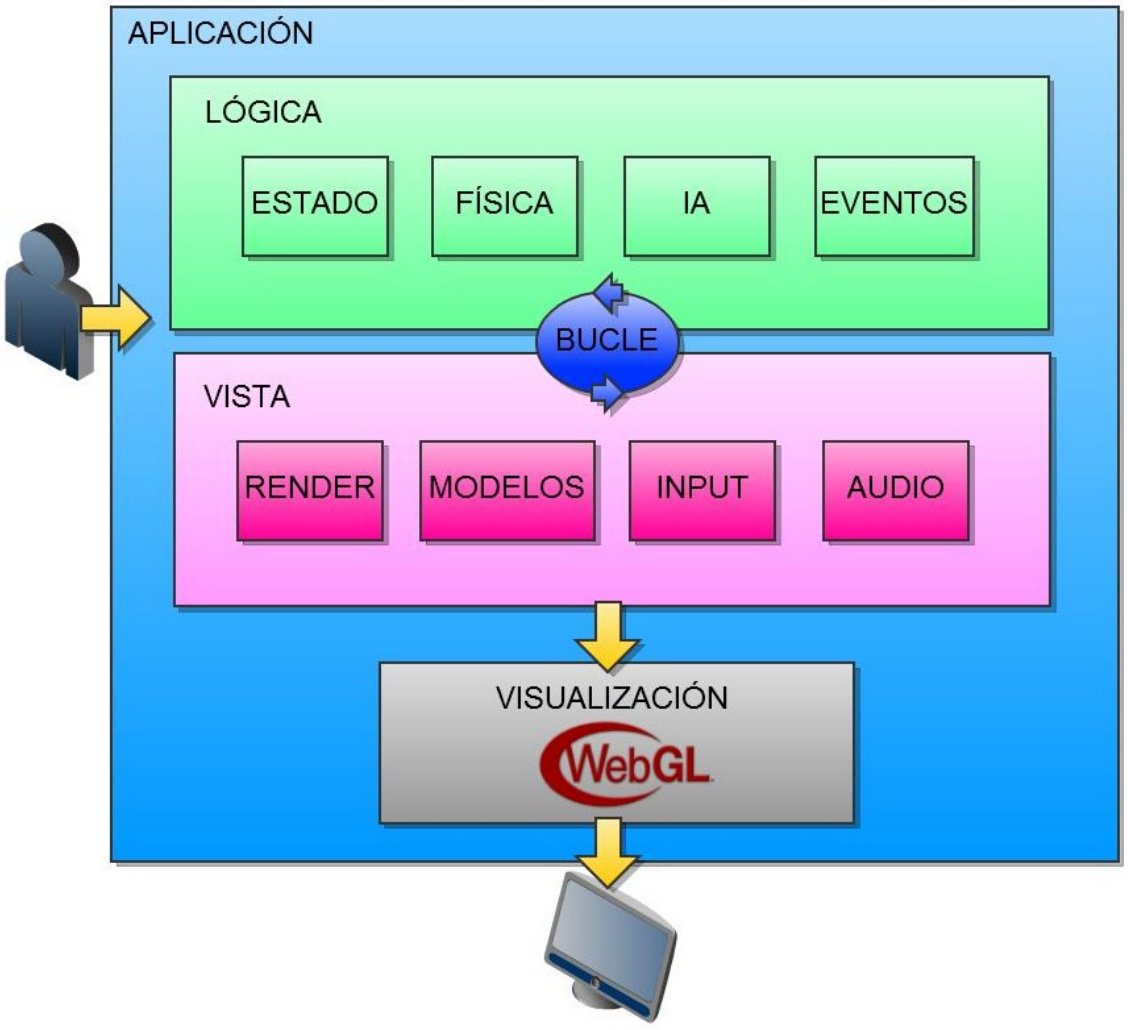


Imagen 3.2: Esquema General del Juego.

3.3. Capturas de Pantalla

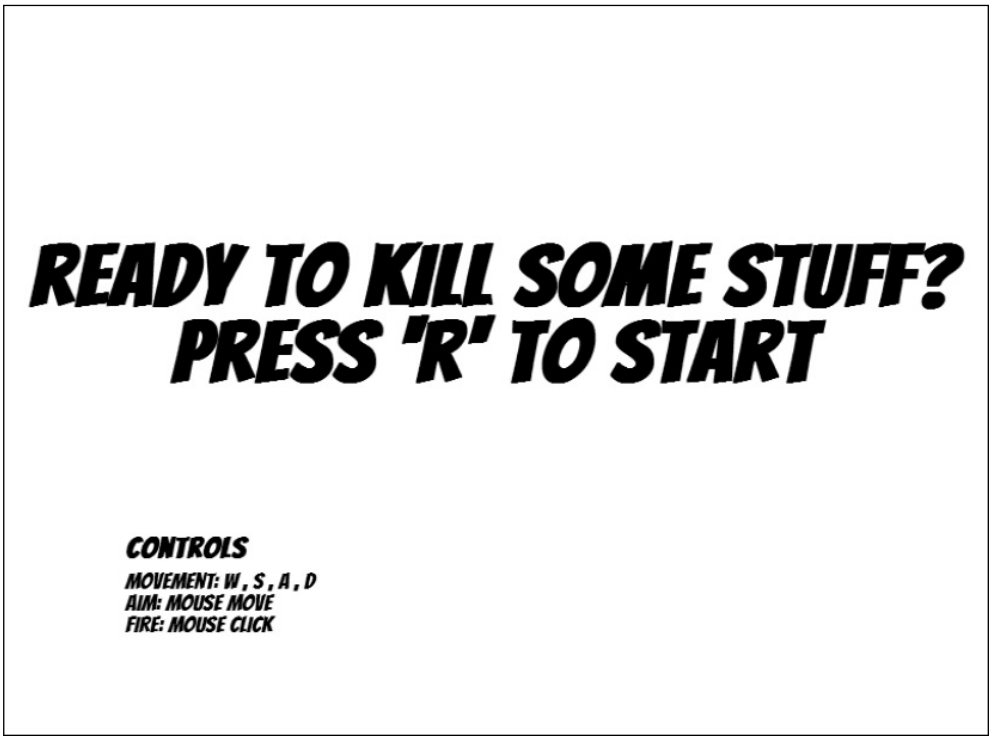


Imagen 3.3: Captura de Pantalla del comienzo del juego.

Imagen 3.4: Captura de Pantalla del juego en funcionamiento.

Imagen 3.5: Captura de Pantalla del juego mostrando las Bounding Boxes.

Imagen 3.6: Captura de Pantalla del final del juego.

3.4. Análisis

El objetivo del análisis de cualquier Software es entender el problema. Normalmente esta etapa es de las más importantes porque es la primera definición de lo qué se quiere hacer y a partir de aquí desenvocará el resto.

No entender el problema puede generar diseños erróneos y pérdida de tiempo. Como primera parte del análisis tenemos que saber las necesidades y condiciones a satisfacer por el Software a realizar mediante una descripción completa del comportamiento del sistema.

Separaremos los requisitos en funcionales y no funcionales.

3.4.1. Requisitos Funcionales

Los requisitos funcionales expresan una capacidad de acción, funcionalidad y declaración en forma verbal de qué tiene que hacer el sistema.

- El juego ha de contener una personaje principal que responda a las acciones del usuario:
  - Moverse
  - Apuntar
  - Disparar
- Las acciones del usario deben estar representadas con sus respectivas animaciones.
- Las acciones del usario deben estar representadas con sus respectivos sonidos.
- El juego tiene que estar desarrollado en una escena realista y cerrada:
  - Física: Los elementos de la escena han de comportarse como en la realidad.
  - Luces: Los elementos de la escena tienen que estar preparados para ser iluminados según la naturaleza de su composición.
- El juego tiene que ser capaz de generar enemigos de una forma lógica.
- Estos enemigos tienen que tener un comportamiento inteligente de:
  - Movimiento
  - Ataque

- El juego tiene que controlar las acciones de ataque, restando la vida tanto del jugador principal como la de los enemigos.
- Si el jugador principal se queda sin vida, se acaba el juego.
- El juego durará un tiempo fijo y el objetivo es eliminar tantos enemigos como se pueda.
- El juego tiene que ser rápido y dinámico de gran acción y tensión.
- Minimizar el acceso al juego simplificando los pasos.
- Proporcionar una pantalla de carga que mantenga informado al usuario.
- Proporcionar una pantalla que informe al usuario que está preparado para empezar.
- Proporcionar una pantalla de final de juego que explique los resultados.
- El juego ha de guardar las estadísticas de las acciones del jugador para que cuando acabe se muestre la puntuación total.
- El juego ha de proporcionar una inmersión dentro de la escena mediante luces y sonidos.
- El juego se tiene que adaptar a todos los posibles tamaños de pantalla.

3.4.2. Requisitos No Funcionales

Los requisitos no funcionales son tales como los recursos del sistema o restricciones generales del sistema no de su comportamiento.

- Estabilidad: Al ser un juego en 3D se ha de garantizar que el cliente es capaz de jugar y que la aplicación es robusta.
- Portabilidad: El juego tiene que garantizar que puede ser ejecutado en aquellas plataformas que soporten WebGL, siempre que el usuario contenga los mínimos requisitos de Hardware conformes con el juego.
- Compatibilidad: No usar extensiones, ni librerías, ni técnicas fuera de las especificaciones básicas de WebGL ni HTML5.
- Usabilidad: Tienes que ser sencillo de usar sin generar confusión contextual.
- Coste: El juego ha desarrollarse bajo licencias sin coste y con tecnologías abiertas y estándares.

3.4.3. Diagrama Entidad - Relación

El objetivo del siguiente diagrama es identificar las principales entidades (rectángulos verdes), como se asocian entre ellas (rombos azules) y que propiedades principales tienen (globos rojos). Nos da una idea de qué se compone el juego. En este caso, nos muestra la sencillez del juego y que las entidades que lo representan no son complejas. No se quiere ni especificar agregaciones, ni herencias ni atributos propios de diseño, sólo una proyección inicial de los datos de la aplicación extraído de los requisitos funcionales.

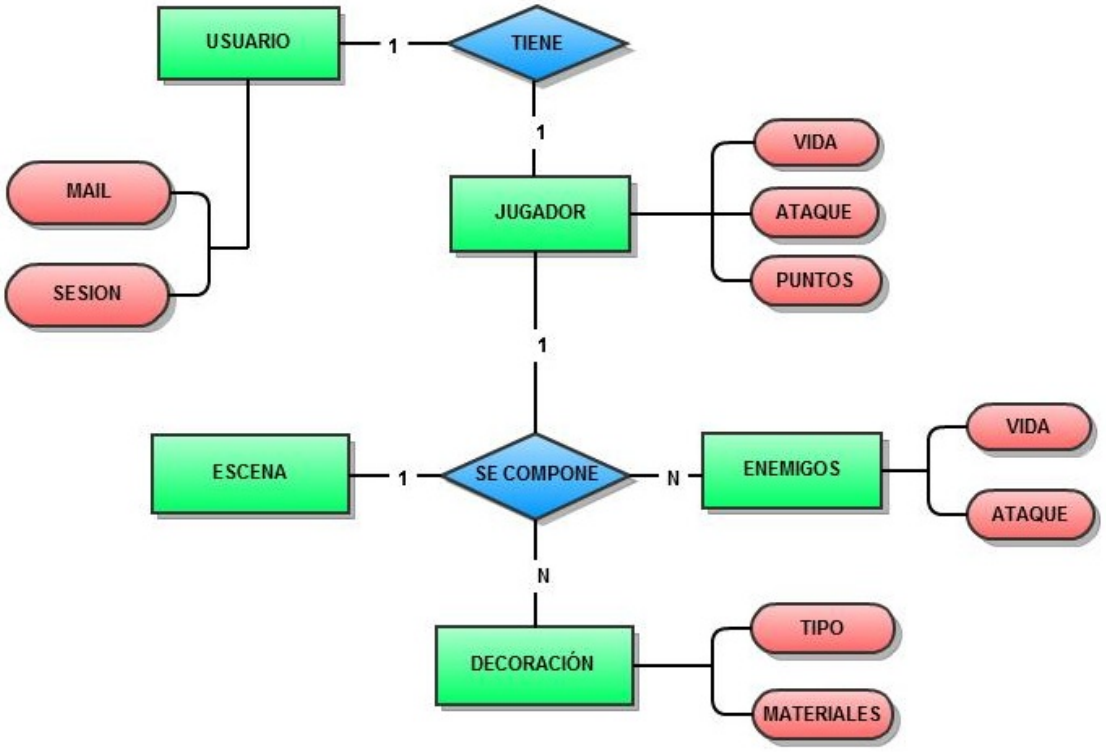


Imagen 3.7: Diagrama Entidad-Relación.

Se ha querido simplificar mucho el juego para centrarnos básicamente en los aspectos más importates como Gameplay, detalle gráfico y rendimiento, propias de etapas más posteriores, con el objetivo de extraer lo máximo de esta tecnología, no de ofrecer un juego complejo y fuera del ámbito del proyecto.

Todo el resto de requisitos funcionales que no se plasman aquí van a ser que ser atendidos en la etapa de diseño.

3.5. Especificación

En esta etapa del desarrollo nos centraremos en describir qué tiene que hacer el sistema. Desde un inicio se sabía que los requerimientos funcionales serían cambiantes debido al desconocimiento de la tecnología de implementación y a la flexibilidad de ellos mismos por eso se ha preferido trabajar bajo un modelo de desarrollo de Software más informal, más ágil e iterativo que facilite el cambio de un requirimiento sin grandes impactos.

3.5.1. Metodología de Desarrollo

Se usarán historias para definir las funciones del sistema. A diferencia de los casos de uso, las historias intentan facilitar el manejo de estos requerimientos. La idea de las historias es capturar el "quién", "qué" y "por qué" de una forma concisa, casi en una línea. Es una forma de manejar requerimientos sin mucha formalidad y documentos que relajen el trabajo de mantenerlos. Su objetivo principal es responder rápido a cambios de requisitos gracias a que:

- Las historias son pequeñas, pueden ser implementadas en días o semanas.
- Permiten la discusión del requerimiento durante su proceso.
- Tienen poco mantenimiento.
- Permiten dividir el proyecto en pequeños incrementos.
- Encajan con Software donde los requerimientos son volátiles.
- Encajan con Software que se define mediante iteraciones.
- Son fáciles de asignar a roles específicos.
- Son fáciles de testear.

Muchas metodologías de moda en desarrollo de Software ágil como Extreme Programming o Scrum trabajan con este modelo de especificación. En el caso de este proyecto, no se tenía 100 % especificado como se quería que fuera el producto final. Más bien se dejaron cosas sin especificar que serían adaptadas según las capacidades de implementación. Por eso no podíamos referirnos a casos de uso completos.

El proceso de desarrollo ha seguido un modelo iterativo incremental que responda a la naturaleza del proyecto y de las historias. Un proceso iterativo incremental es un

modelo de desarrollo de Software que permite desarrollar versiones cada vez más complejas y completas, hasta llegar al objetivo final. Es decir, a medida que cada incremento definido llega a su etapa final se incorporan nuevas historias que fueron analizadas como necesarias.

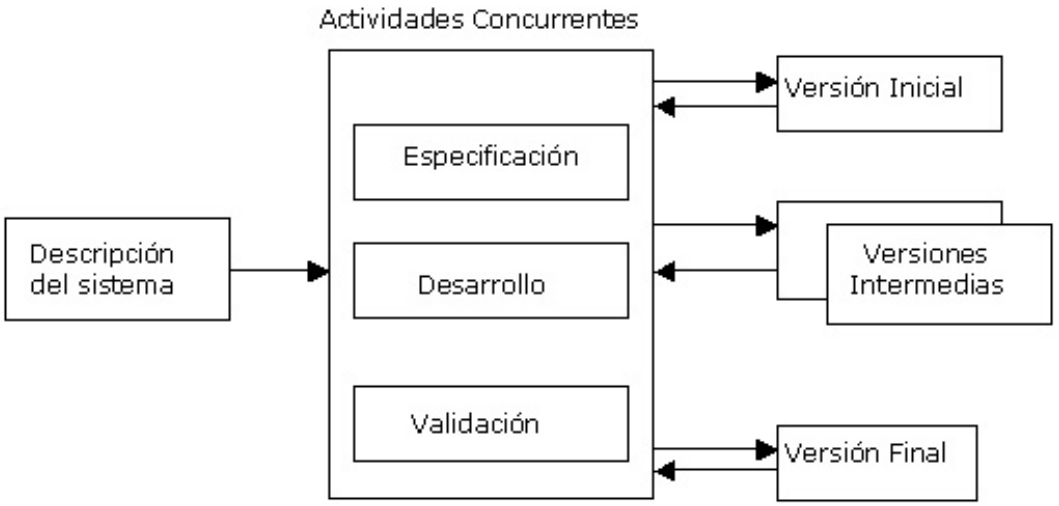


Imagen 3.8: Representación del Modelo Iterativo Incremental.

Tal y como muestra la figura anterior, este proceso permite que constantemente se esté redifiniendo, desarrollando y validando los nuevos objetivos. Cada incremento de Software vendrá dado por las historias especificadas. Estas nuevas historias generarán versiones que incluirán las historias anteriores produciendo nuevas versiones más completas.

3.5.2. Historias

Normalmente las historias vienen dadas por los clientes de la aplicación o por el equipo de negocio. En el caso de este proyecto los propios desarrolladores han sido los generadores de estas historias por eso están más cerca del lado propiamente informático que de negocio, agilizando aún más el proceso. Se intenta definir un rol , un algo y un beneficio.

Interfaz de Usuario

1. El jugador quiere ver una portada del juego para poder sentirse introducido.
2. El jugador quiere ver cómo carga el juego para no creer que el juego ha parado.
3. El jugador necesita conocer las instrucciones para disfrutar del juego.
4. El juego tiene que escalarse al tamaño de la pantalla para visualizar todo el contenido correctamente.
5. El juego tiene que poderse visualizar en pantalla completa para mejorar la inmersión del jugador.
6. El jugador tiene que conocer la vida del jugador principal durante el juego para actuar en consecuencia y sobrevivir.
7. El jugador tiene que conocer cuanto tiempo queda de juego para actuar en consecuencia.
8. El juego tiene que presentar una pantalla de fin cuando el juego acaba para informar al usuario de que ha acabado, mostrando la puntuación.

Cámara

1. El jugador tiene que ver al personaje principal en tercera persona.
2. El jugador tiene que ser capaz de ver la escena para actuar en consecuencia y adelantarse a la acción.

Escena

1. La escena tiene que tener una iluminación dinámica.
2. Cada elemento de la escena tiene que tener definido un material que representa la contribución de luz ambiente, difusa y especular para generar realismo.

3. La escena tiene que estar decorada según el contexto del juego.

Renderizado

1. El renderizado ha de ser constante y al mayor ritmo posible tendiendo a los 60 Frames por segundo para proporcionar suavidad entre la transición de los frames.
2. Se tiene que proporcionar un efecto de disparo que muestre la dirección de disparo para que el usuario pueda apuntar con eficacia.
3. Se tiene que proporcionar un efecto de muerte de los enemigos para que el usuario se de cuenta que ese enemigo está muerto.

Personajes

1. El personaje principal tiene que disparar y moverse para representar las acciones del usuario.
2. Los enemigos tienen que moverse y atacar según un patrón lógico.
3. Todos los personajes tienen que poseer atributos de ataque y vida.

Lógica

1. La lógica de los enemigos es acercarse al personaje principal y cuando estén cerca atacar.
2. La lógica del juego es durante un tiempo finito ir creando más enemigos en posiciones aleatoria.
3. Si el tiempo acaba o el jugador principal muere, el juego acaba.
4. El juego ha de capturar los eventos del navegador para que el jugador pueda interactuar.

Física

1. Todos los elementos de la escena tienen que tener un comportamiento físico real.
2. El juego tiene que intersectar el disparo con la escena y actuar en consecuencia. Si el disparo da a un enemigo, el enemigo tendrá que restarse un porcentaje de vida según los atributos del persnaje principal. Lo mismo para el jugador principal.

Audio

1. El juego tiene que tener una melodía de fondo que ambiente la escena.
2. Se deben proporcionar sonidos para todas las acciones del juego para que el usuario note la respuesta del sistema a sus acciones.

3.6. Arquitectura

El objetivo de la arquitectura es dar soluciones tecnológicas a los problemas previamente expuestos desde una perspectiva de diseño. Es la etapa previa al propio desarrollo.

3.6.1. Control de Versiones

Se ha usado el control de versiones Subversion que proporciona las soluciones necesarias para los requisitos de este proyecto. Se usará la típica configuración de *Trunk,Branches y Tags* ya que se ajusta perfectamente a la metodolgía de desarrollo ya explicada.

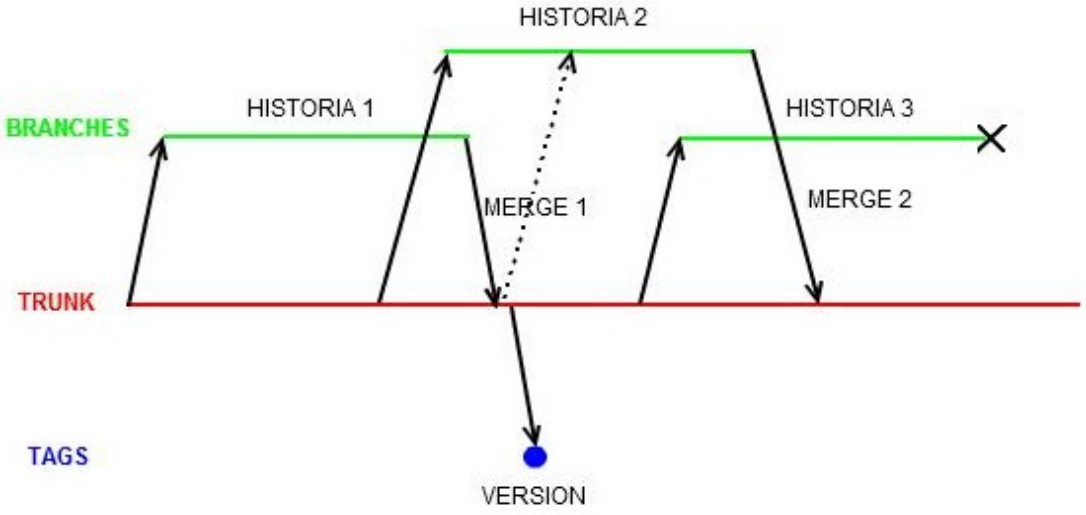


Imagen 3.9: Control de Versiones.

Como se muestra en la figura anterior el Trunk será el cuerpo de proyecto donde se residirá las etapas ya completas. Cuando se incorpora una nueva historia al desarrollo principal, se creará un Branch nuevo donde se aplicarán las nuevas funcionalidades y hasta que ésta no este acabada y testeada no se incorporará al Trunk. El proceso de incoporporar una rama al Trunk se llama Merge. Durante el desarrollo principal del Trunk hay momentos en los que se quiere preservar ciertas versiones. Las razones pueden ser varias, como una release, una beta o cualquier propósito que lo requiera. La forma de preservar una copia es haciendo un Tag. En el Trunk no se ha desarrollado nueva funcionaldiad, sólo arreglos o Fixes para mejorar la integración de historias.

En conclusión, aparte de ofrecernos un mecanismo de almacenamiento, posibilidad de realizar cambios dinámicos y un registro histórico nos proporciona un soporte físico a las historias.

3.6.2. Diseño

Se ha decantado por un modelo en componentes y orientado a objetos. Cada componente será responsable de unas tareas concretas. La idea es que cada componente sea una caja cerrada de lógica con una fachada de acceso al resto. De esta forma cada componente es independiente, escalable y substituible independientemente del resto.

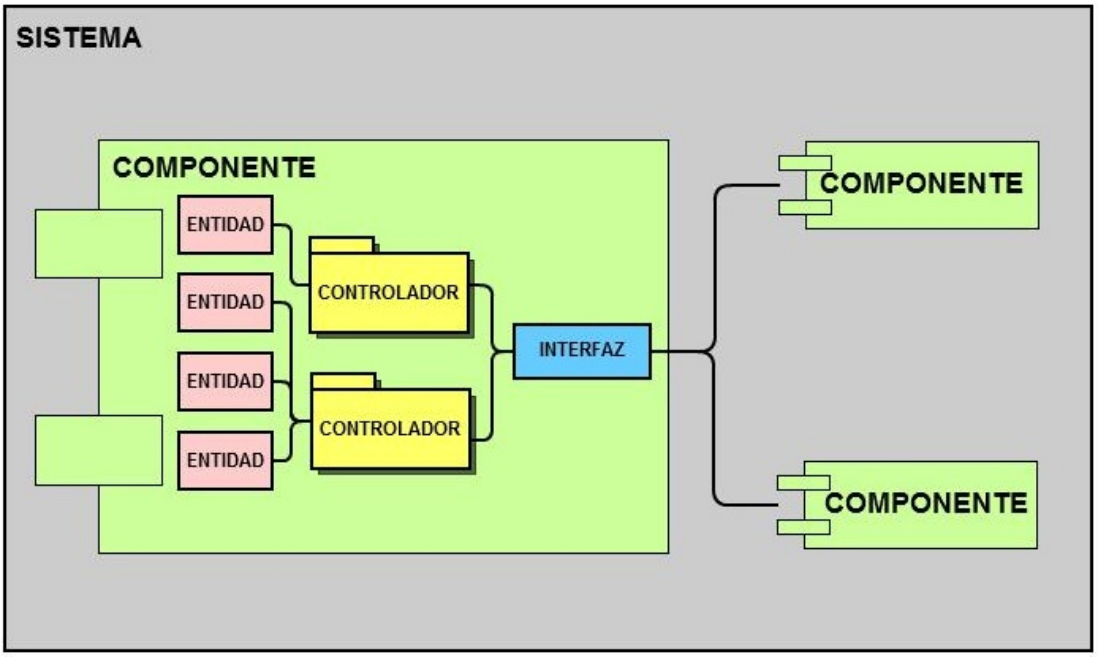


Imagen 3.10: Modelo de Diseño.

Como se muestra en la figura el sistema está dividio en componentes que internamente están construdios por controladores y entidades. Estos componentes exponen una interfaz de acceso para que el sistema y otros componentes tengan acceso a la funcionalidad que éste implementa. Dentro del sistema hay porciones de lógica que no han podido ser encapsuladas en componentes y que residen en controladores y entidades ligadas directamente al sistema.

3.6.3. Patrones de Diseño

El sistema va a estar partido en componentes o módulos, por lo tanto claramente hay un patrón de modularidad incluido en el diseño. El **patrón módulo** es una variante del patrón singleton con un propósito más específico para organizar nuestro código en diferente componentes.

Dentro de los componentes se le asignará la responsabilidad de cierta tarea a las clases que contengan la información necesaria. Conocido como **patrón experto**. Dentro

de un componente pueden haber muchos expertos que colaborarán para ofrecer, en conjunto, la funcionalidad del componente mediante una interfaz.

Dentro de los componentes podemos indentificar dos tipos de objetos: los controladores y las entidades. Ambos tipos de clases son expertos en sus respectivas responsabilidades pero la diferencia es que los controladores tienen la responsabilidad de crear las entidades, **patrón creador**, de eliminarlas y de gestionarlas. Las entidades son contenedores de información. La entidad no tiene responsabilidad para ejecutar nada fuera de su ámbito. El que las maneja es el controlador. Lo que se quiere de esta forma es separar la lógica de creación, destrucción y manejo de la la lógica de datos propios de una clase. De esta forma se consigue que dentro de los componentes haya una alta cohesión entre sus miembros pero muy encapsulada para que visto desde fuera se trate como un subsistema externo, como si trabajaramos con librerías. Los controladores serán clases **Singleton** porque no tiene sentido instanciar dos clases con las mismas responsabilidades y estado único. Se han usado **Factorías** para aquellos componentes que su objetivo es crear ciertas clases, prepararlas y ofrecerlas al sistema.

3.6.4. Componentes

En este apartado se van a definir los diseños de los componentes más importantes del juego. No van a ser definidos con todos los atributos reales de la aplicación, sólo con aquellos que hacen posible realizar las historias.

3.6.4.1. Render

Este componente es el encargado de cargar los modelos del juego y de renderizarlos. Como se aprecia en el diagrama 3.11 hay un responsable de crear todo el componente que se llama Game Renderer. Game Renderer creará tantos controladores como sea necesario. Cada controlador será encargado de un tipo específico de Modelo. Ha sido necesario separarlo en diferentes controladores porque la lógica de carga de un modelo difere mucho de uno a otro. Cada controlador creará, guardará y eliminará las instancias de Modelos de las que es responsable. Cada controlador tendrá un programa de pintado específico para sus modelos. Por otro lado está el árbol de entidades. Model es la super clase que contiene todos los atributos comunes de Renderizado. Cada Modelo require de un Material para responder a la iluminación de la escena. Y a partir de aquí desciende una jerarquía de especialización necesaria para representar las necesidades de cada tipo de Modelo, la potencia de la orientación a objetos, el polimorfismo.

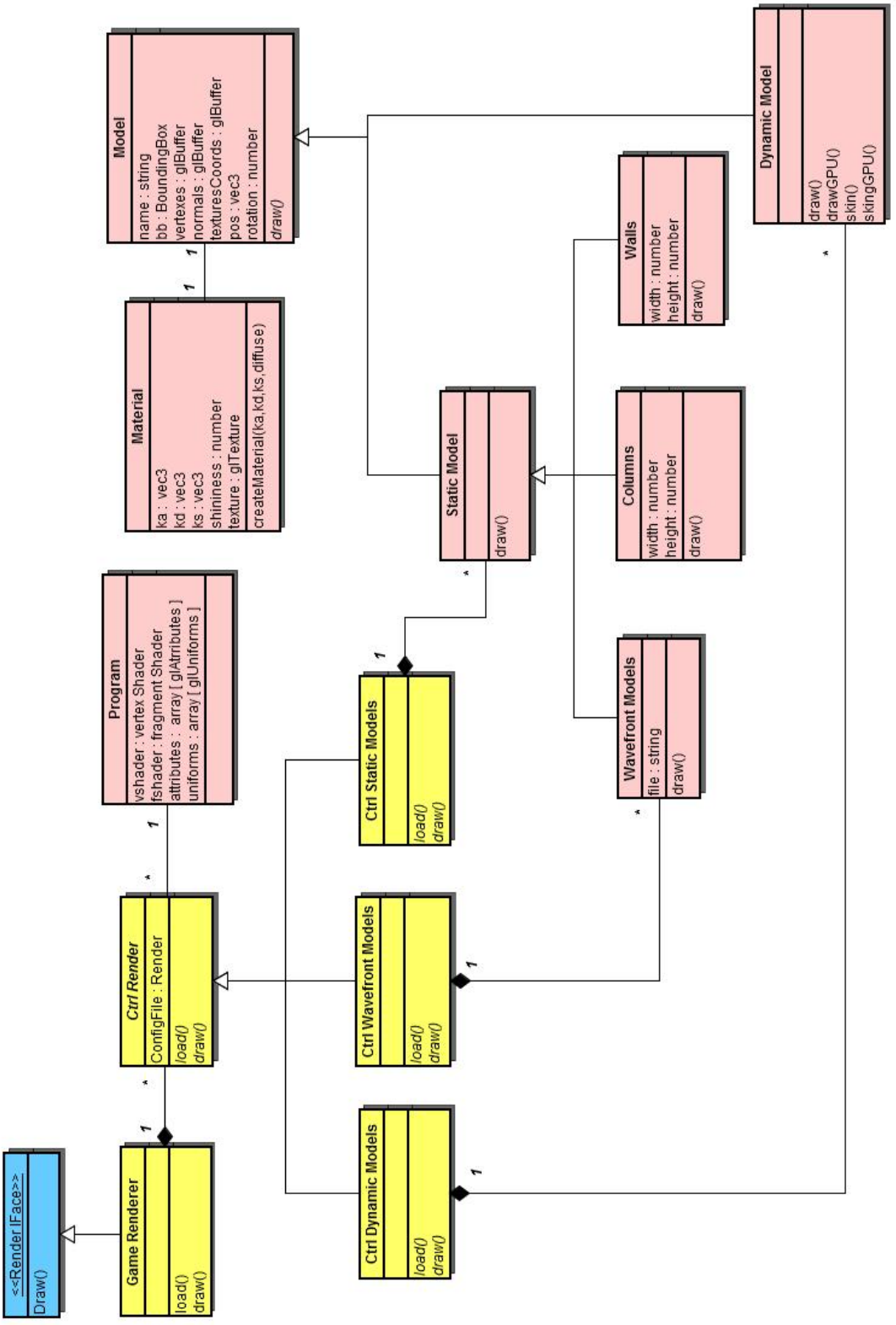


Imagen 3.11: Diseño Coponente Render UML.

La gran responsabilidad de este componente es el ejecutar el Render Loop. A continuación se muestra de forma simple como funcionará el Render Loop del juego mediante un diagrama de secuencia:

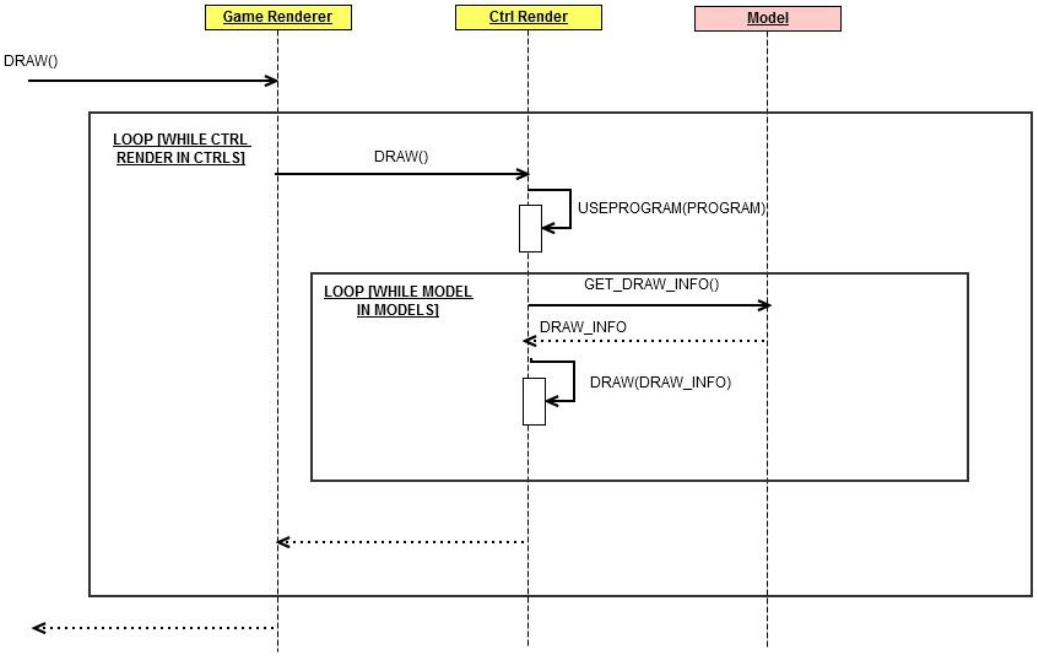


Imagen 3.12: Diagrama Secuencia Render Loop.

El Game Renderer le dirá a cada controlador que renderize, cada controlador usará su programa de renderizado, extraerá la información de las entidades y pintará lo necesario. Por lo tanto en cada frame se tiene que ejecutar esto para pintar la escena, entre otras cosas.

3.6.4.2. Jugador Principal

Este componente es el encargado de atender las acciones del usuario y aplicarlas al jugador principal.

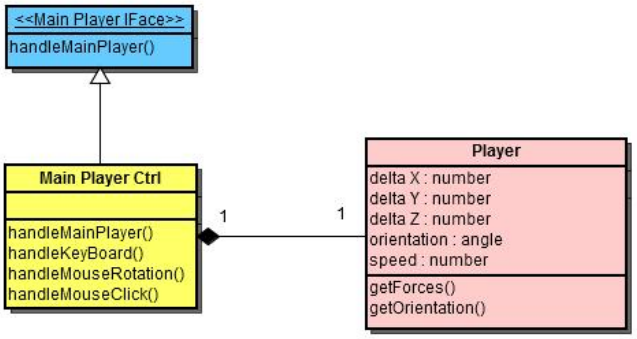


Imagen 3.13: Diseño Componente Jugador Principal UML.

El controlador de este componente estará escuchando los eventos de teclado y ratón. Según el Game Play y las propiedades del jugador principal se calcularán unos incrementos de movimiento y una rotación del jugador principal, esa información se depositará en la entidad Jugador Principal. Este componente deberá de transferir toda esta información al componente de la Física para que este aplique las fuerzas correspondientes.

3.6.4.3. Cámara

Este componente aloja la cámara y sus propiedades. La cámara ha sido definida en tercera persona y Top Down, por lo tanto tendrá que abastecerse de la posición del jugador para posicionarla acordemente. La cámara tendrá que definir la matriz de proyección según las medidas del Viewport [1] y la matriz de Vista.

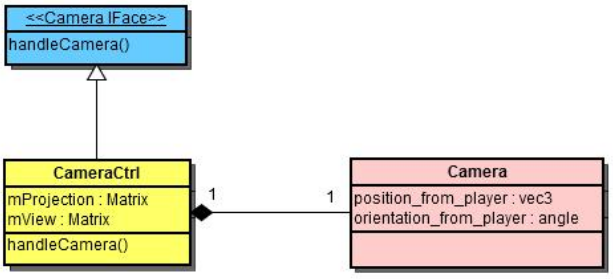


Imagen 3.14: Diseño Componente Cámara UML.

[1] En gráficos 3D el Viewport es la rectángulo 2D donde se proyecta la escena 3D según un cámara virtual

En cada frame habrá que posicionar la cámara en la posicion adecuada, según la posición del jugador principal y una cierta lejanía definida por entidad Camara para que sea en tercera persona. El controlador se encargará de dejar las matrices correctas en cada frame.

3.6.4.4. Animaciones y Modelos MD5

Para los modelos dinámicos que tienen animaciones se va a usar los modelos MD5, apéndice B. Este componente nos ofrecerá la opción de aplicar propiedades de elementos animados a nuestra escena.

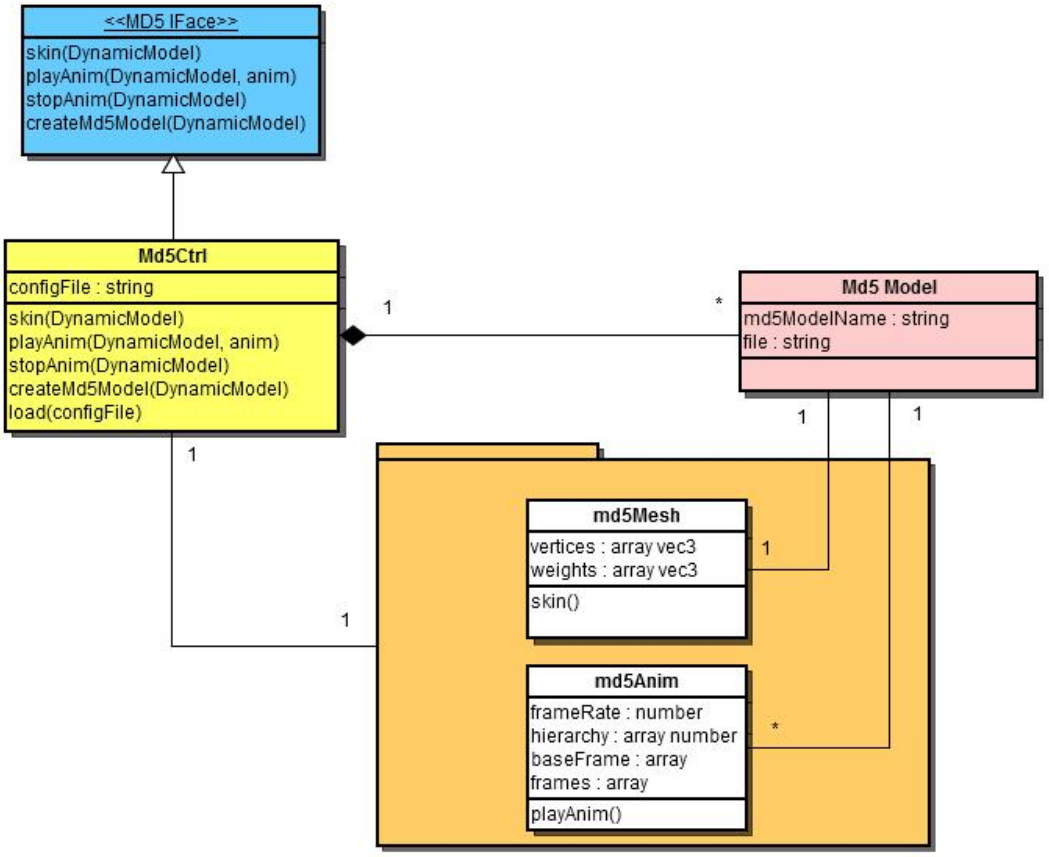


Imagen 3.15: Diseño Componente Modelos MD5 UML.

Este componente se nutre de una librería externa, pintada en color naranja, que nos ofrece básicamente dos clases, md5Mesh y md5Anim. La primera representa la malla de vértices del modelo MD5 y la segunda la animación. Durante el renderizado habrá que llamar a este componente para que haga Skinnning [1] de los modelos dinámicos. Según

[1] Es asociar a una malla de vértices un esqueleto articulado mediante pesos y posiciones de los vértices

la lógica del juego también habrá que ir cambiando esas animaciones según las acciones del juego.

3.6.4.5. Audio

Este componente se encargará de ofrecer una serie de llamadas que desencadenarán en sonidos. Se va a usar una librería externa, Buzz para la gestión de los audios en HTML5. Se explicará con más detalle la integración en la parte de implementación.

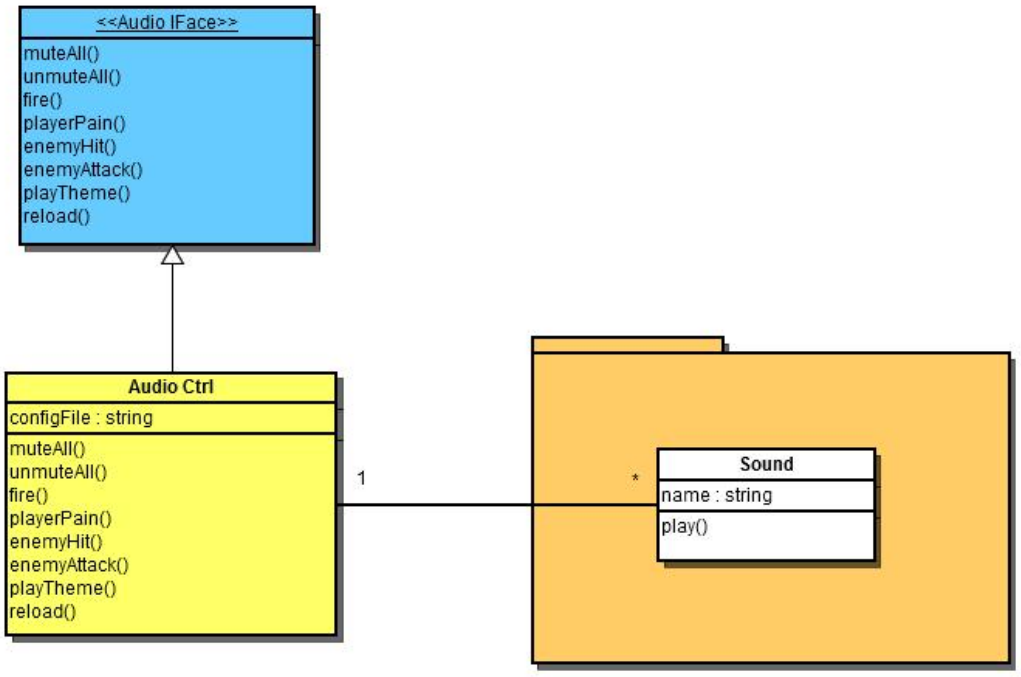


Imagen 3.16: Diseño Componente Audio UML.

La librería nos proporciona una clase sound. El controlador según su fichero de configuración cargara todos los sonidos del juego y ofrecera una interfaz de acceso y reproducción.

3.6.4.6. Física

En el componente de la física se usará una librería externa llamada JigLibJs. Como las típicas librerías de física, necesita que le representemos toda la escena mediante sus componentes, cajas, esferas, planos y gravedad. Mediante una llamada iterativa a "Simulate" la librería simulará como se comportarian los cuerpos que le hemos representado. La librería ofrece métodos para aplicar fuerzas a los cuerpos que hemos representado.

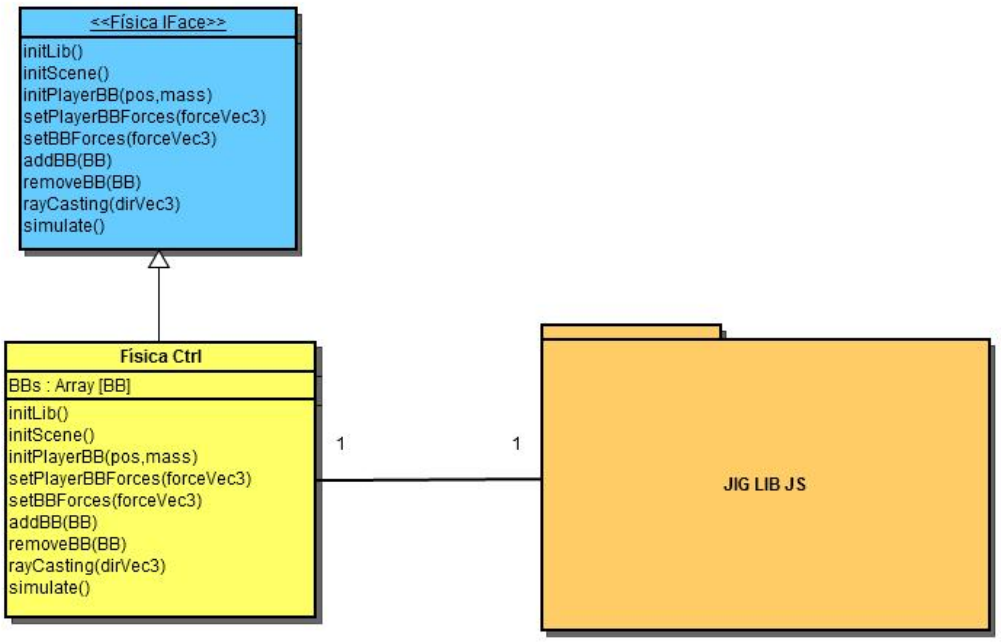


Imagen 3.17: Diseño Componente Físicas UML.

La idea es que cada modelo del Sistema tenga una Bounding Box, una caja englobante que lo represente, que limite su volumen colisionable. Esta Bounding Box, se le pasará al modulo de física que gestionará su posición según su simulación. La Física devolverá las posiciones y rotaciones de esa Bounding Box y el Modelo se acabará pintando donde la Bounding Box diga. De esta forma pintaremos los modelos de la escena en las posiciones que la librería de la física ha simulado.

Si nos fijamos, todos los modelos del Render tienen una Bounding Box asociada para que puedan ser representadas como un volumen. El controlador de físicas ofrecerá todos los métodos necesarios para que el juego pueda simular las físicas. Cada vez que añadamos un modelo a la escena, este se tendrá que añadir a la física mediante addBB(BB), pasando como parámetro su Bounding Box. Durante la vida de este modelo, una referencia a su Bounding Box quedará guardada en el controlador de físicas que simulará su posición hasta que no la eliminemos.

De esta manera separamos la lógica de renderizado de la lógica de física mediante una clase intermedia, las Bounding Boxes.

3.6.4.7. Shaders Factory

Este componente es una factoría de Shaders. Su objetivo es proporcionar programas ya compilados, los Shaders, a la aplicación. Cualquier componente de la aplicación puede pedir un programa de renderizado por nombre. Mediante un fichero de configuración la factoría cargará y compilará todos los shaders definidos y guardará los programas para que cuando alguien se los pida estén preparados para su uso.

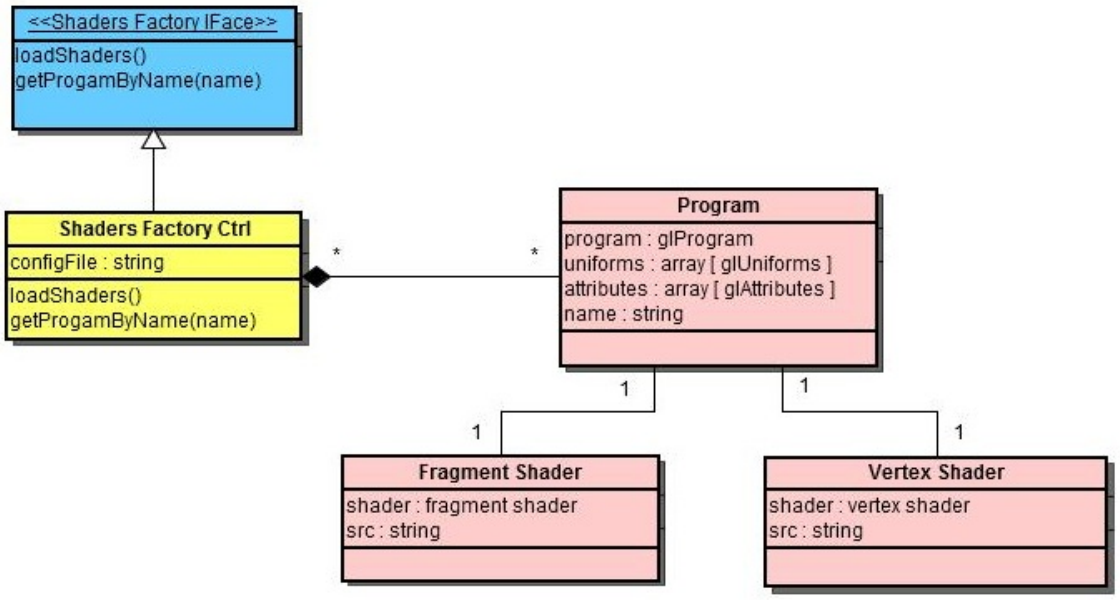


Imagen 3.18: Diseño Componente Shaders Factory UML.

3.6.4.8. Iluminación

Este componente se encarga de gestionar la iluminación de la escena mediante las definiciones de las luces y los materiales.

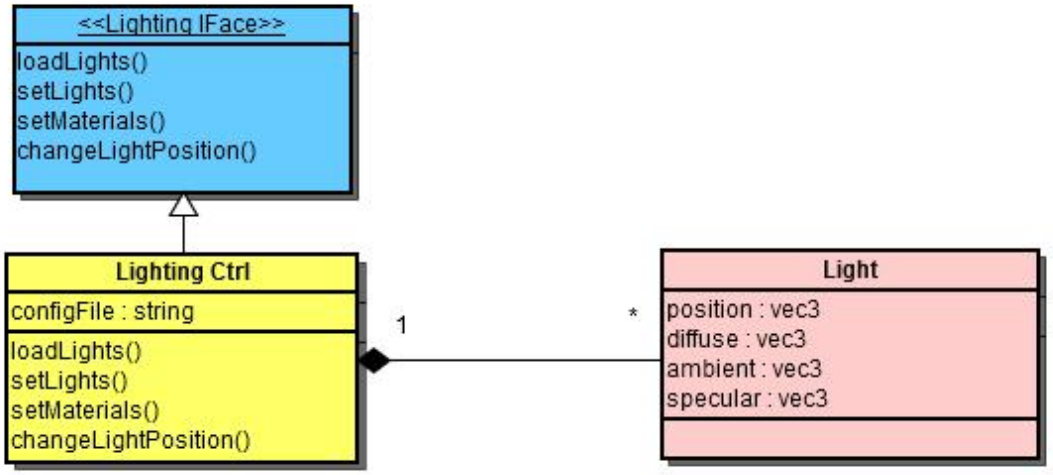


Imagen 3.19: Diseño Componente Luces UML.

Durante el renderizado habrá que controlar qué luces y materiales se usan en la escena para dar los efectos deseados. Mediante un fichero de configuración se definirán las características de las luces.

3.6.5. Sistema Global

En la siguiente imagen se muestra cómo interactúan todos los componentes en sus actividades más importantes. Cada actividad está especificada por una letra mayúscula para su posterior explicación:

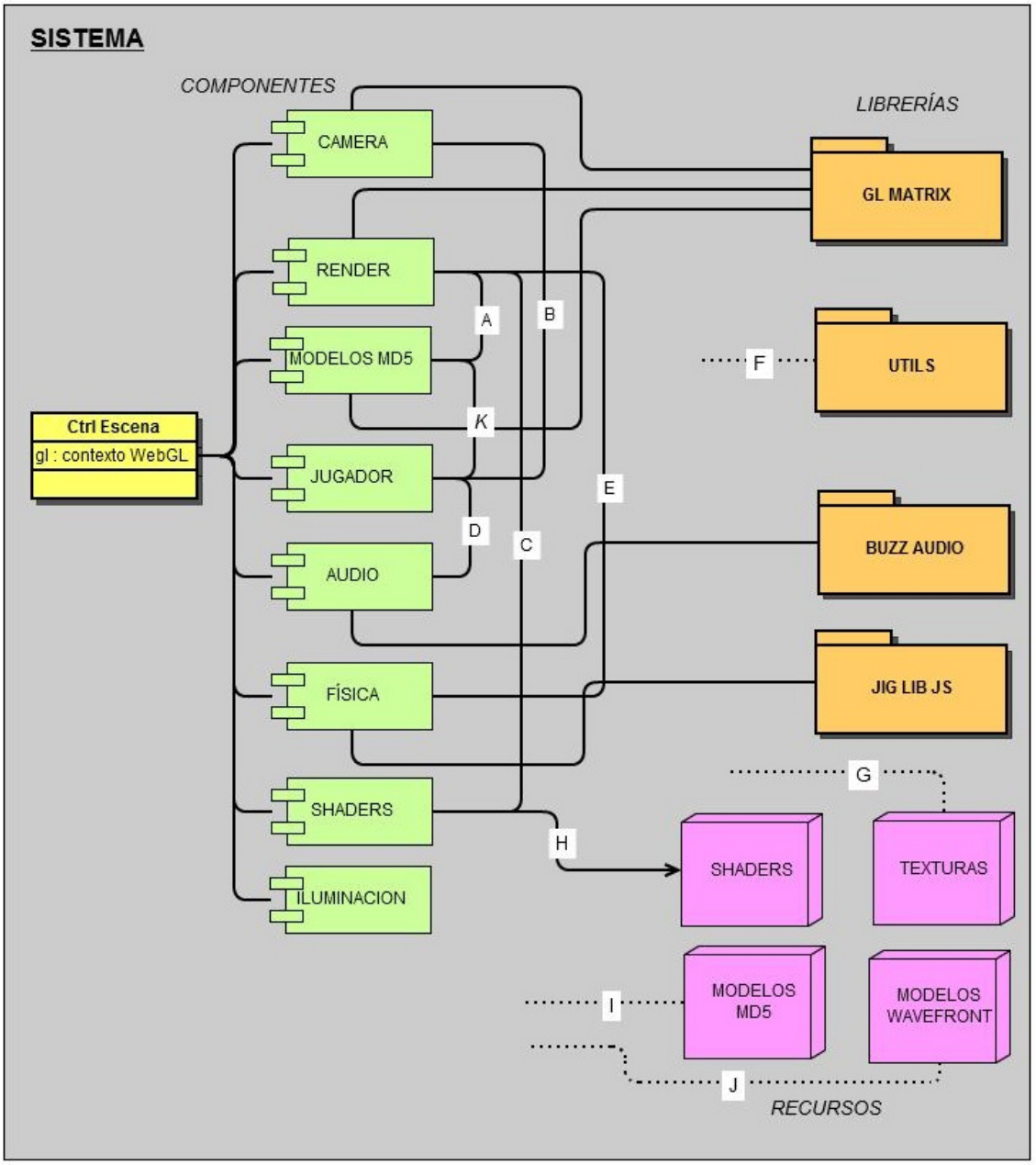


Imagen 3.20: Diseño Sistema UML.

Actividades más importantes del sistema:

- A - El componente de Render pedirá al componente de Modelos MD5 que ejecute el Skinning de los modelos animados antes de pintarlos.
- B - La cámara deberá de pedir la posición del jugador principal para posicionarse en la posición correcta.
- C - El componente de Render necesita de varios programas de renderizado según el tipo de modelo, la factoría de Shaders es la encargada de proporcionarlos.
- D - El componente del jugador principal deberá usar el componente de Audio para reproducir aquellas acciones que necesiten una representación sonora.
- E - El componente de Render deberá de encargarse de añadir los modelos y sus Bounding Boxes a la física para después leer las posiciones simuladas y pintarlas en sus posiciones correctas.
- F - Hay muchos métodos comunes entre todos los componentes, para por ejemplo trabajar con la máquina de estados de WebGL, o por ejemplo para cargar ficheros. Todos estos métodos de ámbito global se han agrupado en una clase común, llamada Utils.
- G - Recursos en forma de imagen **.png, *,bmp, *.jpg*
- H - Código fuente de los Sshaders, archivos **.c*
- I - Recursos de los modelos MD5. Modelos: **.md5Mesh* , Animaciones: **.md5Anim*
- J - Recursos de los modelos Wavefront : **.json*
- K - El jugador principal activará ciertas animaciones según las acciones del usuario.

3.6.6. Diagramas de Secuencia de un Frame

En esta diagrama de secuencia se especifica las actividades entre componentes más importantes durante la ejecución de un frame.

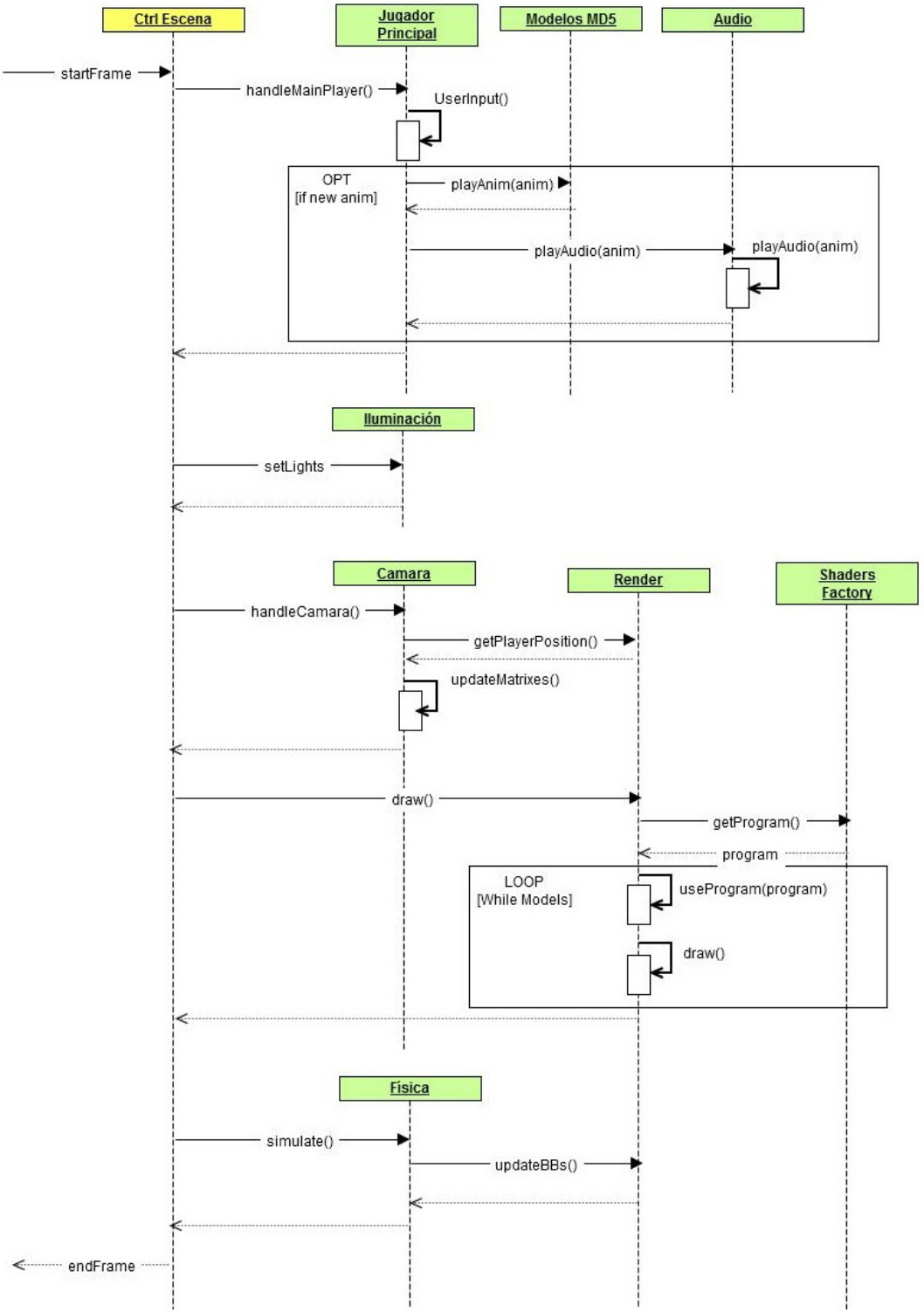


Imagen 3.21: Diagrama de Secuencia de un frame.

3.6.7. Ficheros de Configuración

Todos los componentes que cargan recursos externos o necesitan de una configuración requieren de un fichero que contenga esa información. Tales como propiedades, rutas, valores, etc. De esta forma es posible cambiar el comportamiento de la aplicación sin tener que retocar el código fuente de la aplicación.

También de la puerta abierta a que se puedan crear herramientas o *Tools* que mediante una aplicación que modifique ese fichero de configuración y la aplicación sólo tenga que leerla y cargar los elementos necesarios. Así podemos separar la lógica de dominio de la lógica de datos y configuración.

3.7. Implementación

En esta sección se va a proporcionar la solución extrictamente tecnológica al diseño proporcionado previamente. Antes de hablar como se ha implementado el diseño, hace falta analizar Javascript para entender cual es el propósito de este lenguaje y cómo hay que adaptarse correctamente desde un principio para no cometer errores.

3.7.1. Javascript

Los desarrolladores, en general, tienen una mala impresión del lenguaje de Javascript asociado principalmente a la programación en el navegador que tradicionalmente ha sido difícil por culpa de la integración con DOM o con Internet Explorer 6. El lenguaje se ve penalizado por la opinión pública cuando no debería porque es un gran lenguaje pero incomprendido, ya que acepta desde programadores que no saben lo que hacen, hasta grandes desarrolladores con habilidad para programar.

Javascript tiene buenas y malas cosas, así que habrá que intentar programar sólo usando las buenas partes.

Empezamos por ver que influencias de lenguajes tiene el origen de Javascript:

- Self - herencia prototypal y objetos dinámicos.
- Scheme - lambda (funciones anónimas o Closures) y flojo prototipaje.
- Java - Sintaxis.
- Pearl - expresiones regulares.

3.7.1.1. Partes Malas

Ahora hablemos de las partes malas de Javascript con las que tendremos que lidiar para no cometer errores al implementar el diseño:

- Variables Globales - Uno de los grandes problemas que hay en Javascript es que al no haber una unidad de lincaje para los ficheros fuente todo el asemblado se comete bajo un mismo namespace global donde las variables pueden colisionar.
- No namespaces - Afectan a la modularidad del código.
- No hay control de visibilidad en los objetos.

- Operador "+" - Este operador sirve para sumar enteros y concatenar strings. Habiendo un protipaje flojo no hay diferencia entre los strings y los números así que puede generar resultados inesperados.

- Typeof - Es una macro del propio lenguaje que nos dice de que tipo es la variable pero hay ciertos tipos incromprensibles como:
  typeof Object == Object , typeof Array == Object y typeof null == Object.

- Demasiados valores vacíos - Javascript proporciona muchos Bottom Values que parecen ser lo mismo, pero tiene pequeñas diferencias que pueden generar confusión: *false, null, undefined y NaN.*

3.7.1.2. Partes Buenas

Herencia

Hay dos formas de representar la herencia en los lenguajes actuales, una es la Clásica, la que todos conocemos y la otra es la Prototypal o Herencia Diferencial, que es casi únicamente usada por Javascript.

La Herencia Prototypal o Diferencial es tan potente que puedes usarla como si fuera clásica pero no al revés. No hay clases, sólo objetos que heredan de otros objetos mediante links que marcan las diferencias. En la herencia clásica se define un objeto mediante la declaración de una clase y se marca de que otra clase hereda. Cuando creas una instancia se crea todo un cojunto de instancias según la cadena de herencia y todas son iguales. Por lo tanto la clase nos ofrece el comportamiento, la estructura y la instancia de los datos.

En Herencia Prototypal, todos los objetos del mismo tipo comparten un objeto llamado Prototype. Y todo lo que esté declarado en Prototype de un objeto lo heredarán el resto de objetos que contengan el mismo tipo de prototype para así compartir el comportamiento. No existe el concepto clase del cual tú heredas, sino que coges directamente el comportamiento del objeto que quieras.

Funciones

Las funciones en Javascript son objetos de primera clase[1]. La única forma que hay de crear un scope en Javascript es creando una función, ya que este lenguaje usa function scope y no block scope como Java o C++. Un ejemplo claro sería éste:

```
1
2  // Function Scope
3  function getCounter(){
4    var i = 0;
5    return function() {
6        console.log(++i);
7      }
8  }
9
10 // Crea un scope al invocar la función con i = 0 y
11 // se guarda en la variable counter, el return de la función getCounter,
12 // que es otra función que imprime i pre incrementada por consola
13 var counter = getCounter();
14
15 // Invocamos counter ya que esta guardaba una función.
16 // Se recupera el scope, se incrementa i, resultado : 1
17 counter();
18
19 // Si la volvemos a llamar, recuperamos el scope
20 // donde fue creada la funcion del return, recupera i,1,
21 // y se incrementa en 1, igual a 2.
22 counter();
```

Código Fuente 3.1: Ejemplo Data Hiding Javascript

Esta forma de crear ámbitos privados mediante funciones son comúnmente llamadas Closures. Para trabajar cómodamente con las Closures hay que cambiar la forma de pensar. En otros lenguajes estamos acostumbrados a crear una clase, añadirle datos y algún tipo de comportamiento también. Las clases son estado con datos y funciones, veamos un ejemplo de cómo se implementaría una clase en Javascript:

[1]Se tratan a las funciones como objetos de primer orden. Se pueden pasar funciones como argumentos a otras funciones, devolverlas como valores de otras funciones y asignarlas a variables.

```
1  // Classes en Javascript
2  function Comportamiento(config)
3  {
4    this.config = config;
5    this.doIt = function(param)
6    {
7      if(this.config.flag){
8      ...
9      } else {
10     ...
11     }
12   }
13 }
14
15 var b = new Comportamiento({flag:false});
16 b.doIt("item");
```

Código Fuente 3.2: Clases en Javascript

En este ejemplo, lo que estamos haciendo es crear una función que se comporte como una clase. Las funciones son objetos por lo tanto podemos extender de ellas y crear los miembros. this.config guardará la config pasado en construcción y this.doIt será una función miembro de la clase. Así es como trabajaríamos con otros lenguajes, creamos una clase, añadimos datos a la clase y creamos métodos que trabajen con los datos locales.

Para trabajar con Closures, de forma idónea, vamos a cambiar la forma de expresarlo. En vez de crear una clase de la cual voy a instanciar, se crea un generador de comportamiento.

```
1  // Closures en Javascript
2  function dameComportamiento(config)
3  {
4    return function(){
5      if(config.flag){
6      ...
7      } else {
8      ...
9      }
10   }
11 }
12
13 var b = dameComportamiento({flag:false});
14 b("item");
```

Código Fuente 3.3: Closures en Javascript

Cuando se llama a dameComportamiento éste me a devolver un objeto función, la que está en el return y va a guardar el estado con el que fue llamado(flag:false). O sea siempre que llame a dameComportamiento va a crear una funcion return nueva con su propio scope que guardará el valor pasado en la llamada. Esto pasa gracias a que la función invocada crea un scope nuevo y cada scope nuevo pueda guardar sus valores propios. Por lo tanto la variable b va a contener una función que esta guardando el estado con la que fue declarada. Y por último podemos llamar a la variable "b", que contiene una función que internamente guarda los datos, para que ejecute el comportamiento.

Las Closures son el pensamiento inverso a lo que estamos acostumbrados. No creamos clases que son estado con comportamiento sino que creamos comportamiento que tiene el estado que necesita. Y la gracia, es que puedes pasar ese comportamiento por todo el programa porque las funciones en Javascript son de primera clase.

Objetos

Todo en Javascript son objetos, funciones y arrays también. Y todos los objetos son mutables, quiere decir, que puedes cambiar sus propiedades en tiempo de ejecución. Y no sólo eso, la potencia real, puedes cambiar todo el comportamiento de una sola vez.

3.7.2. Render Loop

Casi todos los juegos en HTML5 que han surgido en los últimos meses funcionan con un Game Loop mediante *setInterval() o setTimeout()*. Estos métodos son funciones propias de Javascript para manejar el tiempo. Por ejemplo "setInterval(function,1000)" ejecutará automáticamente "function" cada segundo.

Desde que nació el renderizado mediante la GPU con WebGL las aplicaciones son consumidoras de muchos recursos. Pongamos el caso que tenemos un navegador con 5 pestañas abiertas y las 5 cada una con un juego ejecutándose. El navegador estará ejectuando los Game Loops de cada uno de ellos estando sólo visible uno de ellos porque cada uno de ellos definio su game loop son setIntervals, y el navegador no es capaz de descubrir para qué lo estas usando.

Los navegadores Web han proporcionado una solución a este problema se llama ***requestAnimationFrame***. En cada navegador se llama diferente. Funciona casi igual que la función setTimeOut(). Tú le pides al navegador una animación con un callback dentro de la llamada, esta llamada será la llamada que ejecutará tu juego o Game Loop. De esta forma el navegador te irá dando frames según crea conveniente, por ejemplo cuando tu canvas sea visible.

La forma correcta de implementar un Game Loop, en nuestro caso, el Render Loop es esta:

```
1
2  // Creamos una función requestAnimationFrame que sea un wrapper cross platform
3  window.requestAnimationFrame = (function(){
4      //Comprobamos para cada Navegador su propio requestAnimationFrame
5      return  window.requestAnimationFrame       ||   //Chromium
6              window.webkitRequestAnimationFrame ||   //Webkit
7              window.mozRequestAnimationFrame    ||   //Mozilla Geko
8              window.oRequestAnimationFrame      ||   //Opera Presto
9              window.msRequestAnimationFrame     ||   //IE Trident?
10             function(callback, element){            //Fallback function
11                 window.setTimeout(callback, 1000/60);
12             }
13
14 })();
15
16 // Esta es la función que se ejecutará iterativamente
17 function tick(){
18   requestAnimFrame(tick); // Pedimos la siguiente animación con el mismo callback
19   doSomething();          // Lo que se ejecutará en cada frame.
20 }
```

Código Fuente 3.4: Game Loop en HTML5

3.7.3. Instanciación

Antes de entrar en detalle en la implementación de los módulos tenemos que tener en cuenta que durante el pintado de un frame hay que enviar mucha geometría a la GPU. Mucha de esta geometría puede estar repetida y tenemos que evitar la redundancia. Pongamos por ejemplo que queremos pintar un modelo Wavefront:

```
1  StaticModelsCtrl.prototype.draw = function draw()
2  {
3      var model = null;
4      // Recorrer todos los modelos de la escena
5      for(var i = 0; i < this.models.length; i++)
6      {
7          // Coger el modelo actual
8          model = this.models[i];
9          // Chequear que esta completamente cargado
10         if(this.isModelLoaded(model))
11         {
12             // Setear los Buffers en GPU
13             this.setBuffers(model);
14             // Setear los materiales
15             this.setMaterials(model);
16             // Pintar la geometría
17             this.draw(model);
18         }
19     }
20 }
```

Código Fuente 3.5: Pintado sin instanciación

De esta forma por ejemplo, si tenemos un modelo Wavefront representando un coche y lo queremos pintar 4 veces, estaremos seteando los Buffers y los materiales 4 veces. En otras palabras, estaremos enviando a la GPU los mismos datos 4 veces. Si este modelo llega a tener 60000 caras, esto es muy costoso e innecesario.

Por eso hace falta que cada modelo tenga la posibilidad de instanciar tantos modelos de su mismo tipo y en el Draw aprovechar los seteos en GPU solo una vez. Para hacer esto necesitamos que la entidad Modelo tenga un Array de instancias que contenga la información única a esa instancia, rotación, posición y escala.

```
1  StaticModelsCtrl.prototype.draw = function draw()
2  {
3      var model = null;
4      // Recorrer todos los modelos de la escena
5      for(var i = 0; i < this.models.length; i++)
6      {
7          // Coger el modelo actual
8          model = this.models[i];
9          // Chequear que esta completamente cargado
10         if(this.isModelLoaded(model))
11         {
12             // Setear los Buffers en GPU
13             this.setBuffers(model);
14             // Setear los materiales
15             this.setMaterials(model);
16             // Pintar la geometría
17             this.drawInstances(model);
18         }
19     }
20 }
21
22 StaticModelsCtrl.prototype.drawInstances = function drawInstances(model)
23 {
24     var scale = null;
25     // Recorrer todas las instancias del modelo
26     for(var i = 0; i < model.instances.length; i++)
27     {
28             //  Pintar la instancia i
29             this.draw(model.instances[i]);
30     }
31 }
```

Código Fuente 3.6: Pintado con instanciación

De esta forma conseguiremos que pintar 4 coches sea más barato en cuanto a rendimiento. Ya que los 4 coches serán 4 instancias del modelo coche y solo se setearán los Buffers y los Materiales una vez, ya que comparten las mismas propiedades. En el Draw específico de cada instancia podemos modificar la posición de cada instancia.

3.7.4. Componentes

En esta sección se va a presentar el código más importante de los componentes principales. El código está modificado para la presentación escrita y su entendimiento. Para consultar el código real de producción referirse al código fuente adjunto con esta memoria. El objetivo de esta sección es ver la solución de implementación de las partes más importantes de cada componente según la tecnología descrita.

3.7.4.1. Patrón Módulo

Javascript no proporciona nada para encapsular el código directamente sino las herramientas para que lo hagas tú mediante un patrón de construcción propio. Mediante las propiedades dinámicas, las closures y sus scopes vamos a intentar crear un módulo que encapsule la lógica interna. Mediante las closures podemos esconder los datos privados que necesitamos y publicar solo lo que queramos, dejando limpio el ámbito global de la aplicación.

```
1  // Singleton Module
2  var singletonModule = function(){
3    // Priavate and unique part
4    var privateVar;
5    function privateFunc(){
6      ...
7    }
8    // Public Part
9    return {
10     firstPublicMethod: function(a,b){
11       ...privateVar...
12     },
13     secondPublicMethod: function(c){
14       ...privateFunc()...
15     }
16   }
17 }();
```

Código Fuente 3.7: Patrón Módulo en Javascript

Ésta es la forma de poder crear un módulo Singleton en Javascript con datos internos no visibles. Creamos una función anónima y directamente la ejecutamos (l.17 del código fuente anterior). Al ejecutarla, se crean las partes privadas y se devuelve a la variable singletonModule un literal con los métodos que queremos retornar. Estos métodos tendrán acceso a las partes privadas porque están definidas internamente en

la closure pero desde fuera sólo se podrá acceder a los métodos expuestos en el return, encapsulando así el código.

3.7.4.2. Camera

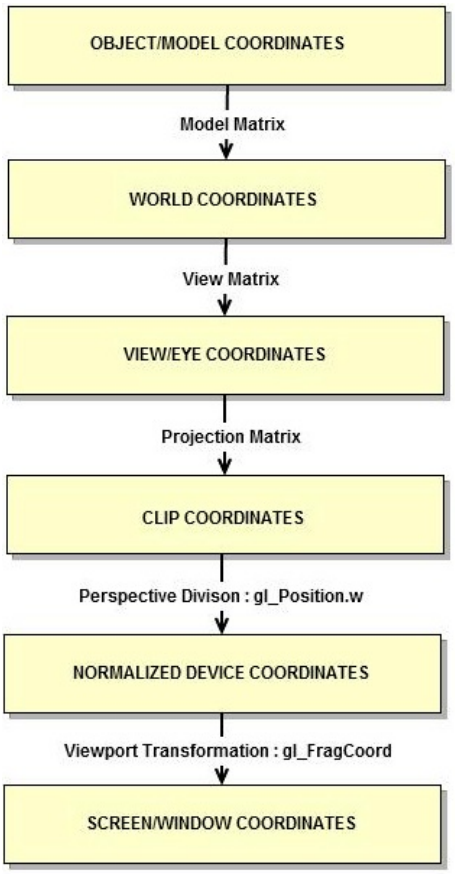


Imagen 3.22: Secuencia de transformaciones de los vértices en los diferentes sistemas de coordenadas

El primer componente que hay que implementar es la cámara. De esta forma tendremos el posicionamiento que queramos desde un principio y no nos tendremos que preocupar más por ello. Antes de explicar la implementación de la cámara de este juego hay que entender cómo vamos a trabajar con las matrices y la transformación de coordenadas. Los modelos que vamos a pintar estarán en **coordenadas de modelo**, coordenadas respecto su centro la posición (0,0,0). Según como queramos transformar ese modelo (posición, rotación y escalado) definiremos una matriz de modelo para cada transformación. Esta matriz será multiplicada por los vértices del modelo y así conseguir que ese objeto ahora esté en **coordenadas de mundo**, con su posición real respecto al centro de la escena. La escena no siempre es vista desde la posición 0 y menos en este juego que la cámara está en constante movimiento. La cámara definirá otra matriz, la view Matrix, que multiplicará al resultado anterior consiguiendo que las posiciones de mundo sean vista en el sistema de coordenadas de la cámara, **view/eye coordinates**. Ahora tenemos toda la escena en coordenadas de la cámara. Según el viewport de proyección, la cámara tendrá que difinir un fustrum de visión que genere una projección en 2D, esto se consigue mediante una matriz de proyección, que será definida según los parámetros del viewport, projection Matrix. Ahora sí tenemos la escena en **coordenadas de clipping** que es lo que OpenGL necesita para acabar de completar la secuencia. Los últimos dos pasos son internos de WebGL.

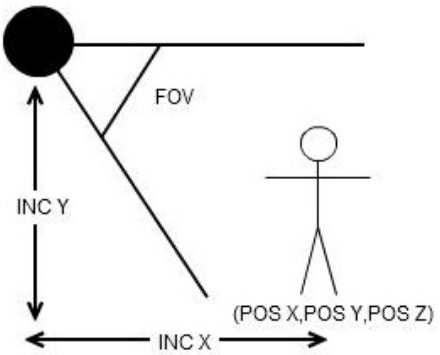


Imagen 3.23: Cámara en tercera Persona

En conclusión, la cámara tiene que proporcionar a la aplicación dos matrices: la viewMatrix y la projectionMatrix. Empecemos por la projectionMatrix, esta matriz va a definir la perspectiva de proyección. Usaremos una librería llamada glMatrix para la gestión de matrices. También nos proporcionara un método para generar una matriz de perspectiva con los típicos parámetros: campo de visión, Z Far , Z near, y relación de aspecto. La viewMatrix tendrá que posicionar la cámara en el lugar adecuado de la escena. En este caso estamos usando una cámara en tercera persona que seguirá al jugador allí donde se mueva. Por lo tanto, tal y como se muestra en la imagen, la cámara habrá que posicionarla en el lugar del jugador (posX, posY, posZ), separarla un incremento de X, subirla en el eje Y, un incremento Y, y por último, inclinarla para que mire hacia el jugador. En el siguiente código vemos como se crean las dos matrices.

```
1  CameraCtrl.prototype.setProjectionMatrix= function(){ // PROJECTION MATRIX
2    // Create perspective Matrix using glMatrix
3    // @param FOV - Angle field of view
4    // @param ra - relación de aspecto.Anchura del canvas entre la altura.
5    // @param ZN - Plano Zeta near.
6    // @param ZF - Plano Zeta far.
7    // Matrix - Matrix donde depositar la proyeccón.
8    mat4.perspective(this._FOV, canvas.width / canvas.height, this._ZN, this._ZF,
       pMatrix);
9  }
10 CameraCtrl.prototype.setViewMatrix= function(){ // VIEW MATRIX
11 // Setear una matriz identidad
12 mat4.identity(viewMatrix);
13
14 // Inclinar la cámara en el eje X tantos radianes como inclinationX
15 // y depositar la rotation en viewMatrix
16 mat4.rotate(viewMatrix, inclinationX, [1, 0, 0]);
17
18 // Trasladar la matrix a la posición del player y posicionar en tercera persona
19 // según los incrementos X e Y
20 mat4.translate(viewMatrix, [-posX-incX , -posY-incY, -posZ]);
21 }
22
```

Código Fuente 3.8: Matrices de Cámara

3.7.4.3. Render

El componente de render es el encargado de en cada frame pintar todo lo correspondiente en la escena. Es muy importante que la ejecución durante el frame sea lo más rápido posible para que la aplicación tenga un frame rate decente.

A continuación se expone como está implementado el controlador de Wavefront Models que se encarga de pintar los modelos importados Wavefront. Como hemos visto en instanciación, la idea es cargar el Modelo prototipo y a partir de ahí pintar tantas instancias como queramos. Es un ejemplo de las llamadas que habrá que hacer a WebGL para pintar.

```
1  StaticModelsCtrl.prototype.draw = function draw()
2  {
3      var model = null;
4      // Recorrer todos los modelos de la escena
5      for(var i = 0; i < this.models.length; i++)
6      {
7          // Coger el modelo actual
8          model = this.models[i];
9          // Chequear que esta completamente cargado
10         if(this.isModelLoaded(model))
11         {
12             // Setear los Buffers en GPU
13             this.setBuffers(model);
14             // Setear los materiales
15             this.setMaterials(model);
16             // Pintar la geometría
17             this.drawInstances(model);
18         }
19     }
20 }
21
22 StaticModelsCtrl.prototype.setBuffers = function setBuffers(model)
23 {
24     // Enviar Coordenadas de Textura a la GPU
25     gl.bindBuffer(gl.ARRAY_BUFFER, model.texturesBuf);
26     gl.vertexAttribPointer(currentProgram.attribute.textureCoord, model.
       texturesBuf.itemSize, gl.FLOAT, false, 0, 0);
27
28     // Enviar Normales a la GPU
29     gl.bindBuffer(gl.ARRAY_BUFFER, model.normalsBuf);
30     gl.vertexAttribPointer(currentProgram.attribute.vertexNormal, model.
       normalsBuf.itemSize, gl.FLOAT, false, 0, 0);
31
32     // Enviar las posiciones de los Vertices a la GPU
33     gl.bindBuffer(gl.ARRAY_BUFFER, model.vertexesBuf);
34     gl.vertexAttribPointer(currentProgram.attribute.vertexPosition, model.
       vertexesBuf.itemSize, gl.FLOAT, false, 0, 0);
35
36 }
37
```

```
38 StaticModelsCtrl.prototype.setMaterials = function setMaterials(model)
39 {
40     // Mediante el componente de iluminación seteamos los materiales
41     this.lights.setMaterials(currentProgram, model.materials);
42 }
43
44 StaticModelsCtrl.prototype.drawInstances = function drawInstances(model)
45 {
46     var scale = null;
47     // Recorrer todas las instancias del modelo
48     for(var i = 0; i < model.instances.length; i++)
49     {
50         // Push State
51         utils.mvPushMatrix();
52
53         // Coger la instancia
54         scale = model.instances[i].scale;
55
56         // Coger su posición según la Bounding Box simulada por la Física
57         mat4.multiply(mMatrix, model.instances[i].bb.getDrawPositionMat());
58
59         // Escalar el modelo en la matriz de modelo.
60         mat4.scale(mMatrix,[scale,scale,scale]);
61
62         // Enviar la Matrix a GPU
63         utils.setModelMatrixUniforms();
64
65         // Draw Instance
66         gl.drawArrays(gl.TRIANGLES, 0, model.vertexesBuf.numItems);
67
68         // Pop State
69         utils.mvPopMatrix();
70     }
71 }
```

Código Fuente 3.9: Ejemplo de pintado de un modelo completo

Este es el conjunto de llamadas que tenemos que hacer para pintar cualquier modelo en WebGL. Lo importante de este código es ver cómo el setBuffers y el setMaterials se hace una vez por modelo y luego por cada instancia se pinta reusando esos valores. El pintado de una instancia requiere que guardemos la matriz de estado de modelo, usemos los valores de posición, orientación y escalado de esa instancia y enviemos a la GPU la geometría mediante DrawArrays.

3.7.4.4. Modelos

El juego tiene varios tipos de modelos que a continuación especifico:

- **Modelos Básicos** - Estos modelos básicos representan las figuras geométricas del juego, tales como planos (paredes, suelo y techo) y prismas (columnas). Han sido creados a mano ya que no presentan gran complejidad.
- **Modelos Wavefront** - Nos servirán para decorar la escena con modelos estáticos. Los modelos Wavefront son una especificación estándard de modelos en 3D. En el apéndice A expongo todo la especificación y la importación a la aplicación. En este caso, la temática es un parking, así que usaremos modelos Wavefront de coches.
- **Modelos MD5** - Son Modelos animados para representar tanto a los enemigos como al jugador principal. En el apéndice B expongo todo la especificación, importación y modificación.

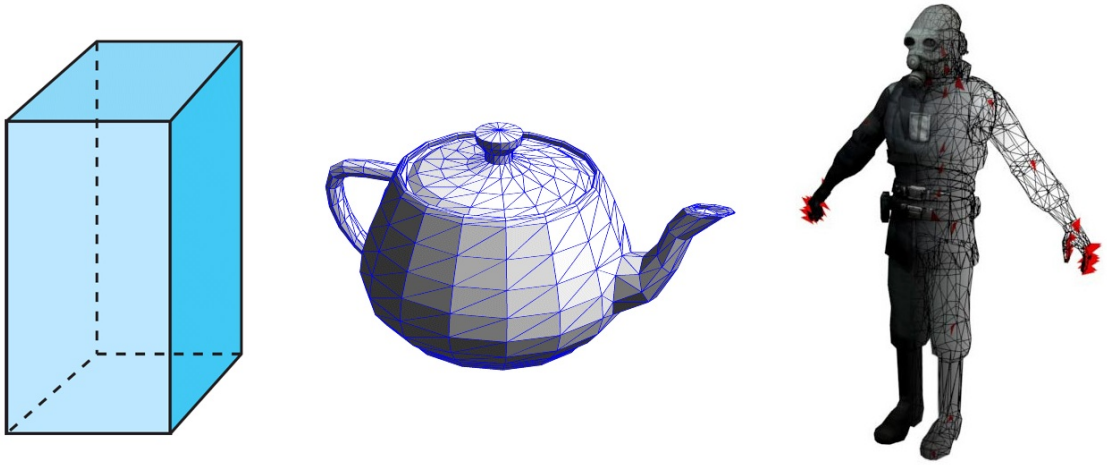


Imagen 3.24: Tipos de Modelos usados. En orden de izquiera a derecha: básicos, Wavefront y MD5.

3.7.4.5. Iluminación

Una de las grandes apuestas de este juego ha sido dotar a la escena de realismo. Para proporcionar realismo a una escena se requiere de iluminación dinámica. La iluminación dinámica afecta a todos los puntos de la escena, según los valores de la luz, según la posición del punto y según el material del punto.

Se ha usado el modelo Phong con alguna variante. El modelo Phong o conocido como Per Pixel Lighting es un modelo de iluminación que describe la forma en que una superficie refleja la luz como la combinación de reflexión difusa, especular y ambiente, según la siquientes fórmulas:

$$I = I_{ambiente} + I_{difusa} + I_{especular}$$

$$I_{ambiente} = L_{ambiente} * K_{ambiente}$$

$$I_{difusa} = L_{difusa} * K_{difusa} * max(0, N * L)$$

$$I_{especular} = L_{especular} * K_{especular} * max(0, R * V)^n$$

I es el color total acumulado en componentes RGB llamado Intensidad de la luz. L son las componentes ambiente,difusa,especular en RGB de la intesidad de la luz. K son los coeficientes ambiente,difusa,especular en RGB del material que va a reflejar la luz.

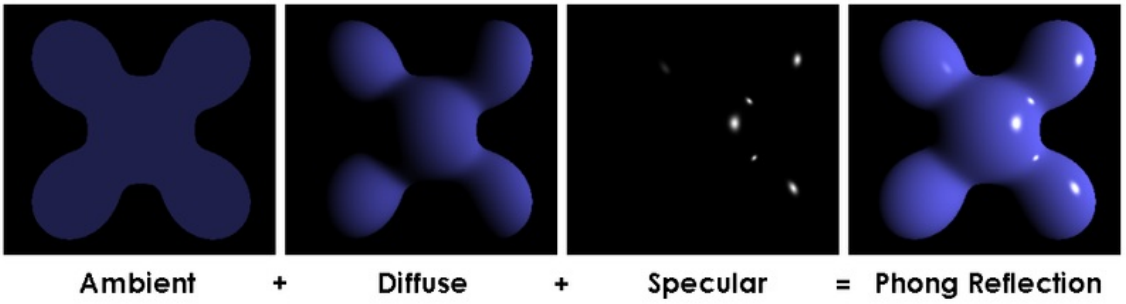


Imagen 3.25: Modelo Phons de Iluminación.

Intensidad Luz Ambiente

La luz ambiente es el resultado de multiplicar las reflexiones de luz en todas las direcciones. Se modela usando un coeficiente constante para cada material. Es la luz constante en una escena.

Intensidad Luz Difusa

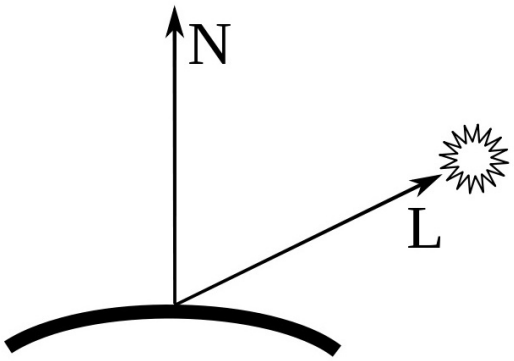


Imagen 3.26: Intensidad Luz Difusa

La intensidad de la luz difusa, es la intensidad de luz que un material refleja en todas direcciones. Esto significa que la cantidad de luz reflejada por una material no depende de la posición de visión solo de los ángulos entre la normal de la superficie y la dirección de la luz. Por lo tanto, la intesidad de luz difusa es el resultado de multiplicar la componente de luz difusa por el coeficiente difuso del material por el coseno entre la normal N y la dirección de la luz desde la misma posición. El coseno de dos vectores normalizados se puede expresar como la multiplicación de los dos vectores, simplicficando cálculos:

$$N * L = |N| * |L| * cos(N, L)$$

El coseno puede dar negativo, valor que no queremos lo que quiere decir que la luz está en el lado contrario a la normal y por lo tanto no refleja la luz difusa. Lo podemos arreglar con el máximo de 0 y el valor del coseno, quedando la fórmula así:

$$I_{difusa} = L_{difusa} * K_{difusa} * max(0, N * L)$$

Intensidad Luz Especular

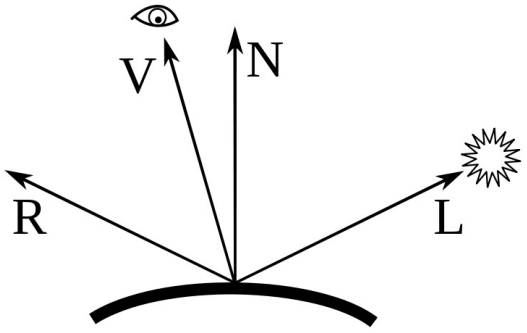


Imagen 3.27: Intensidad Luz Especular

Casi todas las superficies en la vida real no son totalmente difusas sino que dependiendo de cómo se miren, hay zonas que tienen más reflexión que otras y hasta puede a ver puntos especulares generados por materiales muy brillantes y máxima reflexión según el punto de vista. Este punto de vista es el vector R, que es la direcciión de reflexión, la invertida al dirección de la luz L según la normal N. R se calcula como:

$$R = 2 * N * (N * L) - L$$

Y el vector V es la dirección de visión desde donde se está mirando el objeto. Por lo tanto la luz especular depende del ángulo de reflexión de la luz con el vector de visión. Se eleva a n, que especifica el brillo del material. Y se acaba multiplicadno por la intensidad de la luz especular y por el coeficiente especular del material:

$$I_{especular} = L_{especular} * K_{especular} * max(0, R * V)^n$$

Definición de la luz y de los materiales

Según este modelo de iluminación tenemos que definir en nuestro programa una luz con una posición y unos valores de intensidad. Y para cada modelo de la escena unos coeficientes de reflexión para que tengan un aspecto real.

Definición de la entidad Luz según unos parámetros que vendrán del fichero de configuración del módulo de luces que definirá las componentes:

```
var Light=function( pX,pY,pZ,   // Posición
                    aR,aG,aB,   // Intensidad Luz Ambiente
                    dR,dG,dB,   // Intensidad Luz Difusa
                    sR,sG,sB){  // Intensidad Luz Especular
  // Guardar los Valores en el objeto
  this.position=[pX,pY,pZ];
  this.ambient=[aR,aG,aB];
  this.diffuse=[dR,dG,dB];
  this.specular=[sR,sG,sB];
}
```

Código Fuente 3.10: Entidad Luz

En el siguiente código mostramos un método que crea un material. Los valores de entrada vendrán dados por las propiedades del objeto. Este código está alojado en el componente Utils para que cualquier otro componente que requiera crear un material, lo pueda hacer:

```
createMaterial: function (ka,kd,ks,s,texture)
{
    var material = [];      // Definir un Material Vacío
    material.ka = [];
    material.ka.r = ka[0];  // Coeficientes RGB ambiente
    material.ka.g = ka[1];
    material.ka.b = ka[2];
    material.ka.a = 1.0;
    material.kd = [];
    material.kd.r = kd[0];  // Coeficientes RGB difusos
    material.kd.g = kd[1];
```

```
12      material.kd.b = kd[2];
13      material.kd.a = 1.0;
14      material.ks = [];
15      material.ks.r = ks[0];  // Coeficientes RGB especulares
16      material.ks.g = ks[1];
17      material.ks.b = ks[2];
18      material.ks.a = 1.0;
19      material.shininess = s; // Coeficiente de brillo
20      material.mapDiffuse = texture;  // Textura del Objeto difuso
21
22      return material;
23  }
```

Código Fuente 3.11: Entidad Material

En este punto ya tenemos definido el modelo de iluminación, las luces y los materiales. Falta enviar esta información a GPU y realizar los cálculos en los Shaders. En cada frame tenemos que enviar esta información a GPU. La luz es la misma para todos los modelos de la escena por lo tanto solo habrá que enviarla una vez. Y para cada modelo habrá que enviar un material específico, que se comporte como deba. El cálculo de luz hay que hacerlo en cada frame ya que el punto de visión cambia todo el rato dependiendo donde está el jugador. Los cálculos se expondrán en la parte de Shaders. A continuación se especifica como enviar esta información a GPU.

```
1
2   // Program - Es el Programa de shading que usaremos en la GPU.
3
4   // Enviar Luz a GPU
5   LightsCtrol.prototype.setLight  = function(program){
6       //Abmient color uniform
7       gl.uniform3fv(program.uniform.ambientColor,this.ambientLight);
8       gl.uniform4fv(program.uniform.lights[i].position, this.light[i].position );
9       gl.uniform4fv(program.uniform.lights[i].diffuse, this.light[i].diffuse);
10      gl.uniform4fv(program.uniform.lights[i].specular, this.light[i].specular);
11      gl.uniform4fv(program.uniform.lights[i].ambient, this.light[i].ambient);
12  }
13
14  // Enviar Material a GPU
15  Lights.prototype.setMaterials  = function(program,material){
16    gl.uniform3fv(program.uniform["material.ka"],
17      [material.ka.r,material.ka.g,material.ka.b]);
18    gl.uniform3fv(program.uniform["material.kd"],
19      [material.kd.r,material.kd.g,material.kd.b]);
20    gl.uniform3fv(program.uniform["material.ks"],
21      [material.ks.r,material.ks.g,material.ks.b]);
22    gl.uniform1f(program.uniform["material.shininess"], material.shininess);
23    gl.uniform1i(program.uniform.sampler, 0);
24
25
26
```

```
27    // Enviar Textura si tiene
28    if(material.mapDiffuse != undefined)
29    {
30      gl.activeTexture(gl.TEXTURE0);
31      gl.bindTexture(gl.TEXTURE_2D,material.mapDiffuse);
32
33      gl.uniform1f(program.uniform["material.hasMapDiffuse"], 1.0);
34    }
35    else
36    {
37      gl.uniform1f(program.uniform["material.hasMapDiffuse"], 0.0);
38    }
39  }
```

Código Fuente 3.12: Entidad Material

Con este código y el render ya tenemos preparada la geometría de todos los modelos, la luz dinámica que haya en el juego y los materiales. Sólo queda completar el cálculo en los shaders.

3.7.4.6. Factoría de Shaders

Los Shaders de la aplicación son recursos externos escritos en GLSL ES, un lenguaje de programación basado en C, por eso su extensión es ".c". Este componente tiene un responsabilidad muy concreta e importante. En carga de la aplicación tiene que cargar todos los shaders, compilarlos, linkarlos y ofrecerlos al resto de la aplicación para su uso inmediato.

Hay dos tipos de Shaders: *Vertex Shaders* y *Fragment Shaders*. La mejor forma de pensar en qué es un Shader es entenderlo como un código en C y su compilador. Aquí funciona muy parecido. La máquina interna de OpenGL ES compilará ese código en un objeto, después de la compilación ese objeto puede ser linkado en un programa final, un binario. En OpenGL ES el programa final necesitará ser generado con dos objetos, un Vertex Shader y un Fragment Shader y así generar finalmente un programa final. Ese programa final podrá ser enviado a GPU cuando lo necesitemos usar. Igual que en los compiladores de C puede haber errores de compilación y errores de linkado ya que tiene que haber una cierta lógica entre ambos Shaders, por ejemplo que las variables varying sean las mismas entre ambos.

Ahora expondremos el código más importante de este componente, la creación de un programa, para conocer las llamadas principales de WebGL y la representación del diseño:

```
1  /* Representación de la entidad lógica Programa*/
2  var Program = function ( vs_src,
3              fs_src,
4              program_name,
5              attribs,uniforms,
6              subUniforms){
7  // Declarar Variables
8  this._program = null;
9  this._program_name = program_name;
10
11 // Rutas Código Fuente
12 this._vS_src = vs_src;
13 this._fS_src = fs_src;
14
15 // Cargar Programa
16 this._program = loadProgramFromPath(this._vS_src,this._fS_src);
17 }
18
19 /*Generar un programa GLSL ES*/
20 function loadProgramFromPath (vxPath,fsPath){
21   // Cargar Código Fuente
22   var vxStr = loadFile(vxPath);
23   var fsStr = loadFile(fsPath);
```

```
24
25    // Generar Objetos Compilados de Cada Shader
26    var vertexShader = this.getShader( vxStr, gl.VERTEX_SHADER);
27    var fragmentShader = this.getShader( fsStr, gl.FRAGMENT_SHADER);
28
29    // Crear un programa vacío
30    var program = gl.createProgram();
31
32    // Ascociarles los objetos vertex shader y fragment shader
33    gl.attachShader(program, vertexShader);
34    gl.attachShader(program, fragmentShader);
35
36    // Lincar los objetos en el programa
37    gl.linkProgram(program);
38
39    // Chequear errores
40    if (!gl.getProgramParameter(program, gl.LINK_STATUS)) {
41      alert("Could not initialise shaders");
42    }
43
44    return program;
45  }
46
47  /*Generar un objeto shader*/
48  function getShader (str,type){
49    var shader;
50
51    // Crear un Shader Vacío
52    shader = gl.createShader(type);
53
54    // Asociarle el código fuente
55    gl.shaderSource(shader, str);
56
57    // Compilar el shader
58    gl.compileShader(shader);
59
60    // Chequear errores
61    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
62      alert(gl.getShaderInfoLog(shader));
63      return null;
64    }
65
66    // Devolver el objeto shader compilado
67    return shader;
68  }
```

Código Fuente 3.13: Carga y representación de los Programas de Shading

3.7.4.7. Shaders

En esta sección se expone y se explican los Shaders usados para calcular todo el modelo de iluminación, posicionamiento y texturización. En este punto del desarrollo tenemos en GPU todo lo necesario para implementar los cálculos. Para entender los Shaders primero hay que decir cómo se ha partido la lógica entre los vértices y los fragmentos.

Cálculos Vertex Shader

- Calcular posición del vértice en Eye space.
- Calcular normal del vértice en Eye space.
- Calcular posición del punto de visión en Eye space.
- Calcular Intensidad Luz Ambiente $I_{ambiente} = L_{ambiente} * K_{ambiente}$
- Calcular parte constante de la luz Difusa $T_{difusa} = L_{difusa} * K_{difusa}$

```
1  // DEFINITIONS
2  struct Light {
3      vec4 position;
4      vec4 ambient;
5      vec4 diffuse;
6      vec4 specular;
7  };
8
9  struct Material {
10   vec3 ka;
11   vec3 kd;
12   vec3 ks;
13   float shininess;
14   float hasMapDiffuse;
15 };
16
17 // ATTRIBUTES
18 attribute vec3 vertexPosition;
19 attribute vec2 textureCoord;
20 attribute vec3 vertexNormal;
21
22 // UNIFORMS
23 uniform vec3 ambientColor;
24 uniform Light lights[10];
25 uniform Material material;
26 uniform mat4 mvMatrix;
27 uniform mat4 pMatrix;
28 uniform mat3 nMatrix;
29 uniform mat4 vMatrix;
30
31
```

```
32 // VARYINGS
33 varying mat4 vMtlLighting;
34 varying vec2 vTextureCoord;
35 varying vec3 vTransformedNormal;
36 varying vec4 vPosition;
37 varying vec4 vCamPos;
38
39 void lighting_per_fragment()
40 {
41   vec3 _ambient;
42   vec3 _diffuse;
43   vec3 _ambientGlobal;
44
45
46   // Normales a Eye Space mediante la inversa de la ModelViewMat
47   vTransformedNormal = normalize( nMatrix * vertexNormal ).xyz;
48
49   // Coordenadas Textura
50   vTextureCoord = textureCoord;
51
52   // Intensidad Luz Difusa - término constante
53   _diffuse = material.kd * vec3(lights[0].diffuse);
54
55   // Intensidad Luz Ambiente
56   _ambient = vec3(material.ka) * vec3(lights[0].ambient);
57   _ambientGlobal = vec3(ambientColor) * vec3(material.ka);
58
59   // Setear las intensidades en una Varying Mat4
60   // para optimizar espacio
61   vMtlLighting[0] = vec4(_ambient+ _ambientGlobal,1.0);
62   vMtlLighting[1] = vec4(_diffuse,1.0);
63
64   // Calcular Punto de Vision en Eye Space para la luz especular
65   vCamPos = vMatrix * vec4(0.0,0.0,0.0, 1.0);
66
67   // Calcular la Posición en World Coordinates
68   vPosition = mvMatrix * vec4(vertexPosition, 1.0);
69
70   // Calcular la Posición en Clipping Coordinates
71   gl_Position = pMatrix * vMatrix * mvMatrix * vec4(vertexPosition, 1.0);
72 }
73
74 void main(void)
75 {
76   lighting_per_fragment();
77 }
```

Código Fuente 3.14: Vertex Shader

Cálculos Fragment Shader

- Calcular el resto del término difuso no constante: $I_{difusa} = T_{difusa} * max(0, N * L)$
- Calcular la intensidad especular:$I_{especular} = L_{especular} * K_{especular} * max(0, R * V)^n$
- Calcular la Intensidad Total: $I = I_{ambiente} + I_{difusa} + I_{especular}$
- Calcular el color final: aplicar textura, si la tiene, y combinarla con la luz.

```
1  // DEFINITIONS
2  struct Light {
3      vec4 position;
4      vec4 ambient;
5      vec4 diffuse;
6      vec4 specular;
7  };
8
9  struct Material {
10   vec3 ka;
11   vec3 kd;
12   vec3 ks;
13   float shininess;
14   float hasMapDiffuse;
15 };
16
17 // UNIFORMS
18 uniform vec3 ambientColor;
19 uniform Light lights[10];
20 uniform Material material;
21 uniform sampler2D sampler;
22 uniform bool useLighting;
23 uniform bool useTextures;
24 uniform int renderType;
25
26 // VARYINGS
27 varying mat4 vMtlLighting;
28 varying vec2 vTextureCoord;
29 varying vec3 vTransformedNormal;
30 varying vec4 vPosition;
31 varying vec4 vCamPos;
32
33
34 void lighting_per_pixel(void){
35   // Variables Temporales
36   vec3 _diffuse,_specular,lightDir;
37   float NdotL,NdotHV;
38   float attenuation;
39
40   // Extraer los cálculos del Vertex Shader
41   vec3 ambient = vec3(vMtlLighting[0][0],vMtlLighting[0][1],vMtlLighting[0][2]);
42   vec3 diffuse = vec3(vMtlLighting[1][0],vMtlLighting[1][1],vMtlLighting[1][2]);
43   vec3 specular = material.ks;
```

```
44  float shininess = material.shininess;
45
46  // Tiene Textura?
47  float hasMapDiffuse = material.hasMapDiffuse;
48
49  // Normal interpolada
50  vec3 normal = normalize(vTransformedNormal);
51
52  // Calcular Dirección de la luz e intensidad según
53  // posición del vértice
54  if (lights[0].position.w == 0.0) // directional light?
55  {
56     attenuation = 1.0; // no attenuation
57     lightDir = normalize( vec3(lights[0].position) );
58  }
59  else // point or spot light
60  {
61     vec3 vertexToLightSource = vec3(lights[0].position - vPosition);
62     float distance = length(vertexToLightSource);
63     attenuation = 20.0 / distance; // linear attenuation
64     lightDir = normalize(vertexToLightSource);
65  }
66
67  // Calcular max(N*L,0) para la luz difusa
68  NdotL = max(dot(normal,lightDir),0.0);
69
70  // Calcular la intensidad difusa total
71  _diffuse = attenuation * diffuse * NdotL;
72
73  // Cálculos Luz Especular
74
75  // Calcular Vector punto de Vision
76  vec3 viewDirection = normalize(vec3(vCamPos-vPosition));
77
78  // Comprobar si el lado de reflexión es correcto
79  if(NdotL >  0.0)
80  {
81      // Calcular (R*V)
82      NdotHV = max(0.0,dot(reflect(-lightDir,normal),viewDirection));
83
84      // Calcular Intensad Especular Total
85      _specular = attenuation * vec3(specular) *vec3(lights[0].specular)*pow(
      NdotHV,shininess);
86  }
87  else
88  {
89      _specular = vec3(0.0,0.0,0.0);
90  }
91
92  vec4 fragmentColor = vec4(1.0, 1.0, 1.0, 1.0);
93
94
95
96
97
```

```
98    // Si tiene Textura
99    if(floatToBool(hasMapDiffuse))
100   {
101       // Extraer colores de la textura según las coordenadas.
102       fragmentColor = texture2D(sampler, vec2(vTextureCoord.s, vTextureCoord.t));
103   }
104
105   // Cáculo Color total, combinación de la luz con el color de la textura si
        tiene.
106   gl_FragColor = vec4( (ambient + _diffuse + _specular) * vec3(fragmentColor),
        fragmentColor.a);
107
108 }
109
110 void main(void) {
111     lighting_per_pixel();
112 }
```

Código Fuente 3.15: Fragment Shader

Estos dos Shaders descritos son los más importantes del juego. En el desarrollo del juego y en modo de Debug se han usado muchos otros con otros propósitos. Por ejemplo, para pintar los modelos MD5 se han usado otros Shaders, con los mismos principios pero cambiando el cálculo de vértices porque entraban cambios de posiciones dependiendo de las animaciones. También para pintar la línea de disparo se ha usado un Shader mucho más sencillo porque sólo había que pintar una línea con efecto láser semi-transparente. Pero la idea principal de nuestro modelo de iluminación, texturización y cálculo de posición ha sido éste.

3.7.4.8. Jugador Principal

Veamos como están implementadas las funciones más importantes de este componente. Principalmente la gestión del input del usuario.

Empecemos por ver el código de cuando se pulsa el botón izquierdo del ratón, el disparo del jugador:

```
1  md5PlayerCtrl = function()
2  {
3    ...
4    ...
5
6    // Mouse click handler
7    canvas.addEventListener('mousedown', function(event) {
8      // Comprobar que no esté disparando
9      if(self.md5CurrentAnimType != AnimType.SHOOT)
10     {
11       // Actualizar estados
12       self.stats.bullets_shot++;
13
14       // Pedir a la física un ray Casting según el disparo
15       physics.rayCasting( self._phy_idx,
16         self.pos[0],
17         0,
18         self.pos[2],
19         self.orientation,
20         0);
21
22       // Parar la animación actual
23       self.md5CurrentAnim.stop(self.instance);
24
25       // Setear animación de disparo
26       self.md5CurrentAnim = self.md5Anims[AnimType["SHOOT"]];
27       self.md5CurrentAnimType = AnimType.SHOOT;
28
29       // Activar Animación
30       self.md5CurrentAnim.play(self.instance);
31
32       // Reproducir Audio Disparo
33       audio.fire();
34     }
35   },false);
36
37   ...
38   ...
39 }
```

Código Fuente 3.16: Fire Handler

Lo principal es ver cómo se asocia un evento DOM al controlador del player, en este caso el evento mousedown. Solo cogemos el evento si se produce dentro del canvas, dentro de la pantalla de juego. El resto del código esta auto explicado.

Otra de las partes interesantes de este controlador es manejar los moviemientos del jugador. Leer el input de teclado y asociarlo a un movimiento y a una animación. El siguiente código se ejecuta cada frame, en el render loop, y mira que teclas hay presionadas y actúa en consecuencia:

```
1  /* Draw Player Generico*/
2  md5PlayerCtrl.prototype.drawPlayer = function(){
3    // Guardar Estado
4    utils.mvPushMatrix();
5    // Aplicar fuerzas de movimiento
6    this.setPlayerForces();
7    // Recoger resultados de la física
8    this.setPhysicsChanges();
9    // Renderizado
10   this.draw();
11   // Eliminar Estado
12   utils.mvPopMatrix();
13 }
14 /*Posicionar al Player*/
15 md5PlayerCtrl.prototype.setPhysicsChanges = function(){
16   // Leer la posición del player según la simulación de la física
17   physics.readObject(this._phy_idx, this.pos,this._rotationMat);
18 }
19 /*Aplicar Fuerzas de Movimiento*/
20 md5Player.prototype.setPlayerForces = function () {
21   timeNow = new Date().getTime();
22   elapsed = timeNow - lastTime;
23   // calcular el incremento de movimiento según la velocidad
24   // Y según el tiempo anterior.
25   inc = this.speed * elapsed;
26   // Handle Movimiento según las teclas W,A,S,D
27   if (pressedKeys['W'.charCodeAt(0)]){
28     incZ = -inc;
29     nextAnim = AnimType.WALK_STRAIGHT;
30   }
31   if (pressedKeys['S'.charCodeAt(0)]){
32     incZ = inc;
33     nextAnim = AnimType.WALK_BACKWARDS;
34   }
35   if (pressedKeys['A'.charCodeAt(0)]){
36     incX = inc;
37     nextAnim = AnimType.WALK_LEFT;
38   }
39   if (pressedKeys['D'.charCodeAt(0)]){
40     incX = -inc;
41     nextAnim = AnimType.WALK_RIGHT;
42   }
```

```
43
44    // Enviar a la física los incrementos de movimiento como fuerzas
45    physics.setPlayerForces(this._phy_idx,incX,incY,incZ);
46
47    // Calcular animación según la orientación
48    nextAnim = this.calcNextAnimOrientation(nextAnim);
49
50    // Aplicar Animación
51    if(this.md5CurrentAnimType != nextAnim)
52    {
53      if((this.md5CurrentAnimType!= AnimType.SHOOT)||
54        (this.md5CurrentAnimType== AnimType.SHOOT &&
55          (this.instance.currentFrame%this.instance.currentMaxFrame) > Math.floor(
      this.instance.currentMaxFrame/2) ))
56          {
57            this.md5CurrentAnim.stop(this.instance);
58            this.md5CurrentAnimType = nextAnim;
59            this.md5CurrentAnim = this.md5Anims[nextAnim];
60            this.md5CurrentAnim.play(this.instance);
61          }
62    }
63    lastTime = timeNow;
64  }
```

Código Fuente 3.17: Movement Handler

Como se aprecia en el cálculo del incremento de movimiento, hace falta calcularla en función del tiempo, transucrrido desde la última vez que se ejecutó, para que máquinas con un frame Rate más rápido, no generen un movimiento más rápido.

3.7.4.9. Física

Este componente ha sido implementado alrededor de 7 veces. Con diferentes librerías y diferentes formas de ejecutarlo. Todas las librerías de física en Javascript penalizan tanto el rendimiento que el frame rate cae en picado haciendo imposible un renderizado decente además de funcionar a una frecuencia diferente a la de render. Por eso se ha visto la necesidad de implementarlo en un hilo de ejecución externo. En HTML5 hay la posibilidad de ejectuar Javascript scripts en un hilo aparte, Web Workers. Cada navegador los implementa de una forma pero la especificación dice que tienen que correr independientemente al hilo principal. En el capítulo 4, se entrará más en detalle en la forma de trabajar con los Web Workers para dar a la aplicación un entorno bueno de ejecución. Por lo tanto en lo que al componente se refiere, se va a ejecutar en segundo plano y se va comunicar con el render loop o el hilo principal mediante un controlador que hará de handler con el Web Worker, así aislamos dentro del componente el core de la física para que en un futuro cambiar la librería o cambiar la implementación sea solo cambiar las llamadas que ejecuta el handler y no tener que cambiar todo el componente.

La gran responsabilidad de la librería de física que hemos usado,JigLibJS , es representar el mundo virtual de renderizado en un mundo físico. El mundo físico se configura con una gravedad y con volumenes o planos. Para cada elemento físico se le asocia una forma, plano, cubo o esfera y se le asigna una masa y una fricción como atributos más importantes. Según los requerimientos de tu aplicación puedes simular como se comporta ese mundo físico según una frencuencia. Una frecuencia muy alta generá simulaciones cada poco tiempo y puede que bloquee tu aplicación porque no da tiempo a ejectuar todos los cálculos de colisión. Pero una frecuencia muy baja generará simulaciones cada poco tiempo y creará una simulación que no parezca real, a trompos. Después de varias sesiones de testing, según los volúmenes de nuestra apliación, se ha conseguido establecer un intérvalo de ejecución de 30 pasos por segundo, la mitad del frame rate. A esta frecuencia la física simula las posiciones cada 2 frames de render sin verse penalizado.

Por lo tanto, el componente estará formado por un controlador de física que se encargará de publicar la física a la aplicación y de manejar el Web Worker. Para comunicarse con un Web Worker la única forma, a día de hoy, es mediante cadenas de texto. Para un mejor manejo se usarán cadenas de texto en JSON, así la conversión y la extracción de datos es muy fácil.

El código de este componente es bastante complicado de resumir y no sería didáctico exponerlo en la memoria. Para consultar la implementación exacta referirse al archivo *physics.js y worker.js* del código fuente. Se expondrá un diagrama de funcionamiento para explicar como trabaja el componente.

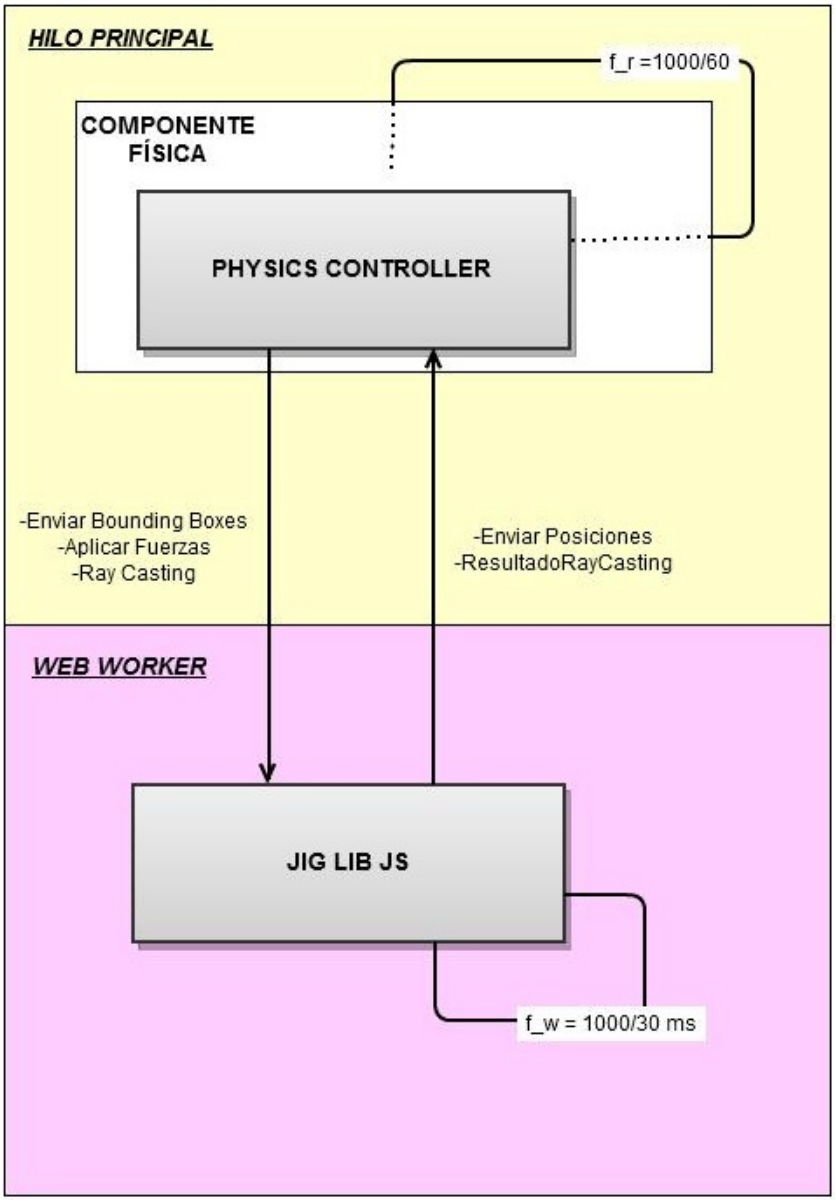


Imagen 3.28: Físicas en un Web Worker.

Lo importante de este componente es ver cómo la librería de físicas trabaja en un Web Worker aislada del hilo principal de ejecución a una frecuencia $f_w = 1000/30$ ms (milisegundos) y va devolviendo los resultados de la emulación al controlador del hilo principal. Este controlador guardará las posiciones de todas las simulaciones para que el resto de la aplicación consulte a cada frame, en este caso el frame rate de renderizado ideal es: $f_r = 1000/60$ ms (milisegundos), las posiciones donde pintarse.

3.7.4.10. Audio

Este componente expone la librería Buzz al resto de la aplicación. No se expone la lógica sino directamente los sonidos que se quieren reproducir. Se es consciente de la poca escalabilidad de este diseño pero no se requería más para el juego.

```
1  // Construcción del componente
2  define([
3      "js/globals.js",    // Importar Variables Globales
4      "js/util/buzz.js" // Importar la librería Buzz
5      ], function () {
6
7        "use strict"
8        // Preload los sonidos
9        buzz.defaults.preload = 'auto';
10       // No autoplay
11       buzz.defaults.autoplay = false;
12       // No loop sounds
13       buzz.defaults.loop = false;
14       // Tipos de Audio - Cross Browser ones
15       buzz.defaults.formats = ['ogg', 'wav'];
16
17       ...
18
19       // Cargar el sonido de disparo
20       var sgfire1 = new buzz.sound("data/sounds/wetfire3", {
21             formats: [ "wav" ],
22             preload: true,
23             autoload: false,
24             loop: false
25       });
26
27       // Función de reproducción del sonido de disparo del jugador principal
28       function fireSound()
29       {
30         // Sonido de Recarga del arma si llega al límite de disparos
31         if(shoots%nshoots == nshoots -1){
32             shoots = shoots +1;
33             reload.play();
34         }
35         // Si el sonido de recarga ha acabado, reproducir disparo
36         if(reload.isEnded()){
37           shoots = shoots +1;
38                   sgfire1.play();
39         }
40       }
41
42       ...
43
44
45
46
```

```
47      // Métodos públicos al resto de la aplicación
48      return {
49              checkTypesSupported: checkTypesSupported,
50              muteAll : muteAll,
51              unmuteAll : unmuteAll,
52              fire : fireSound,       // Ejemplo de implementación de arriba
53              playerPain: playerPainSound,
54              enemyHit:enemyHitSound,
55              enemyAttack:enemyAttackSound,
56              playTheme: themeSound,
57              fadeTheme:fadeTheme,
58              reload: reloadSound,
59              isPartyStarted: isPartyStarted,
60              playerDeath: function(){
61                  playerDeath.play();
62              }
63
64          };
65  });
```

Código Fuente 3.18: Implementación Componente Audio.

En este fragmento de código se aprecia cómo se crea el componente incluyendo la librería Buzz. Una vez incluida, se inicializa. Se ha puesto de ejemplo la creación de un sonido, en este caso el de disparo. Además de la función de reproducción que maneja el sonido creado anteriormente también se ha añadido cómo se publica ese método en el retorno del componente.

Capítulo 4

Técnicas, Optimizaciones y Rendimiento

Tenemos que ser conscientes que este tipo de aplicaciones van a ser ejecutadas por muy diferentes tipos de Hardware y tendrán un límite. Es responsabilidad del desarrollador adaptarse y ofrecer la mejor respuesta posible, si es que existe.

4.1. Javascript

4.1.1. Arrays

Es muy común en Javascript usar Arrays para almacenar datos. Los Arrays en Javascript son objetos y por lo tanto están dotados de todas las propiedades que estos tienen, además de la herencia diferencial de los Arrays. Al tener tantas posibles operaciones el compilador de Javascript tiene diferentes formas de representarlo y estas representaciones puede ser costosas si no tenemos cuidado. El compilador tiene dos formas de almacenar los Arrays: Sparse Arrays y Dense Arrays. Como desarrollador no puedes elegir cual de las dos implementaciones usar, se hará automáticamente. Sparse Arrays son como los Arrays del lenguaje C, trozos de memoria contigua. Los Dense Arrays, en cambio, son como tablas de Hash. Dense Arrays tienen un coste mucho más alto así que lo que tenemos que hacer es intentar dejar el código de tal forma que el compilador nos genere Sparse Arrays. Veamos un ejemplo

```
1 var a = new Array(); // Declarar un array no inicializado
2 // Acceder dinámicamente a la posición 1000.
3 a[1000] = 8;
```

Código Fuente 4.1: Javascript Arrays Mala Idea

```
1 //Buena idea - Sparse Arrays
2 // Declarar un array inicializado
3 var a = new Array(1001);
4 // Acceder dinámicamente a la posición 1000.
5 a[1000] = 8;
```

Código Fuente 4.2: Javascript Arrays Buena Idea

El primer código se implementara como Dense Arrays porque el compilador no sabe cuanto ocupa el array y encima vamos a acceder a posiciones no inicializadas, nada parecido a un típico array en C, por lo tanto directamente se implementa como un Dense Arrays y no queremos que sea así. En el segundo ejemplo declarado un Array con un tamaño específico por lo tanto el compilador puede tratarlo como un Array en C y pedir una memoria fija para todo el almacenamiento, opción mucho más eficiente que usar tablas de Hash. Así que en conclusión, deberíamos siempre declarar los tamaños de los Arrays para que su implementación interna sea la más ajustada posible. Es algo muy sencillo a tener en cuenta que ofrece un gran redimiento a la aplicación.

4.1.2. Objetos

Los objetos en Javascript son representados internamente como un array de clave - valor. Las propiedades pueden cambiar dinámicamente al igual que la cadena de herencia diferencial. Cuando declaras un objeto este va a crear una estructura interna llamada Hidden Class que lo representa y todas las instancias de ese objeto van a estar representadas por la misma Hidden Class. Si, a continuación, añadimos propiedades dinámicamente, ya que Javascript lo permite, esa Hidden Class deja de ser útil para esa instancia y tenemos que pagar el coste de volver a crearla para representar esa propiedad dinámica que estamos añadiendo, pogamos un ejemplo.

```
1  // Declaración de un Objeto
2  function Vec2(x,y)
3  {
4    this.x = x;
5    this.y = y;
6  }
7  // Creación e inicialización
8  var v0 = new Vec2(5,7);
9  // Añadir propiedad dinámicamente cambiando la HiddenClass solo de v0.
10 v0.z = 12;
```

Código Fuente 4.3: Javascript Objects Mala Idea

Este código es ineficiente porque estamos añadiendo la propiedad z dinámicamente a una instancia v0 que tiene una Hidden Class de Vec2. Como Vec2 no tiene definida z tenemos que crear otra Hidden Class para v0 siendo este proceso muy costoso. La forma correcta para hacer nuestro código eficiente es definir objetos fijos y no añadir y quitar propiedades dinámicamente para que todas las instancias compartar la misma Hidden Class.

```
1  // Declaración de un Objeto
2  function Vec3(x,y,z)
3  {
4    this.x = x;
5    this.y = y;
6    this.z = z;
7  }
8  // Declaración de un Objeto
9  function Vec2(x,y)
10 {
11   this.x = x;
12   this.y = y;
13 }
14 // Creación e inicialización de vec2
15 var v0 = new Vec2(5,7);
16 // // Creación e inicialización de vec3
17 var v1 = new Vec3(v0.x,v0.y,12);
```

Código Fuente 4.4: Javascript Objects Buena Idea

De esta forma tenemos dos objetos diferentes para representar los datos que necesitemos y cada uno de ellos con su Hidden Class y sin adición ni eliminación de propiedades dinámicas.

Los objetos son como Arrays por lo tanto tienen dos posibles representaciones internas igual que los Arrays: Sparse y Dense. Como se ha comentado, en la sección anterior lo que queremos es que se implementen como Sparse por la eficiencia de acceso y creación. ¿Qué es lo que convierte nuestros objetos en Arrays de C o en tablas de Hash? Una de las causas, que puede generar que nuestro objeto sea representado como una tabla de Hash, es que el objeto tenga muchas propiedades. También puede ser causado por cambiar las propiedades dinámicamente ya que la representación no encaja como un Array fijo de memoria y tiene que ser implementado por una tabla de Hash dinámica. Así que lo que necesitamos hacer es crear objetos pequeños y fijos. De esta forma el compilador nos ofrecerá la mejor implementación posible, Arrays parecidos a los de C.

4.1.3. Optimizaciones internas de Javascript

Los motores de Javascript, aparte de interpretar tu código, intentan optimizarlo y buscar patrones de ejecución para optimizarlo al máximo. Hay que intentar hacer código lo más apto posible para que estos compiladores puedan optimizarlo al máximo. Hay ciertos aspectos que tenemos que evitar para que el compilador no se estrese y decida no optimizar:

- Funciones muy grandes no son candidatas para ser optimizadas.
- ForIn bucles.
- Conversión de números a String.
- Try-Catch.

Las optimizaciones que se hacen son especulativas ya que no siempre tu código va a ser ejecutado de forma parecida en todos los casos. Pero es bueno que sea así. Quiero decir, hay que intentar programar para que las funciones se ejecuten de la misma forma intentando no cambiar su comportamiento dinámicamente para que esas optimizaciones especulativas se conviertan en código optimizado y válido. Si en algún caso, habiendo ya substituido el código lento por el optimizado, se cambia el comportamiento de esa función, el motor tendrá que deoptimizarlo porque ya no valdrá y no queremos vernos penalizados por esto: Optimizar, aplicar, deoptimizar y ejecutar el nuevo código no optimizado, situación bastante trágica.

En conclusión, tenemos que dejar al optimizador código sencillo, con tipos fijos, predicible y no dinámico porque aunque nosotros no estemos escribiendo el código nativo, el compilador sí y hay que dejarselo fácil para que lo asimile y lo traduzca. Una pequeña regla general es que cuanto más tu código se parezca al lenguaje C o C++ más rápido va a ser.

4.1.4. Garbage Collector

Uno de los grandes problemas de aplicaciones grandes en Javascript es el Garbage Collector, el encargado de limpiar toda esa memoria ya no referenciada en tu código. Normalmente los motores de Javascript tiene dos áreas de memoria, una para los objetos de corta vida y otra zona para los objetos de larga vida para optimizar la búsqueda. Lo que hay que intentar es minimizar el coste del Garbage Collector. Por ejemplo los objetos se pueden promocionar de corta a larga vida y este paso es muy costoso porque se copia

memoria de un lado a otro. Lo que no queremos es que nada más ser promociando a la memoria de larga vida este objeto tenga que ser borrado porque ya no se usa. Así que se tiene que programar intentado que los objetos: tengan una vida larga o una vida corta. No queremos objetos con vida media porque consumirán rendimiento innecesariamente.

Teniendo en cuenta lo comentado, el Gargage Collector se dedicará a limpiar esos objetos no referenciados. Siendo conscientes de esto hay que intentar dejar de referenciar esos objetos cuando no los queramos y no dejarlos referenciados si no los vamos a usar. Puede ser una tarea dura si no eres consciente desde un principio porque la generación de Closures pueden esconder muchísimas referencias sin que tú lo sepas una vez el código ya está escrito.

Pero la mejor forma de evitar el impacto del Garbage Collector en su limpieza es no generar basura. No generar basura quiere decir que si el Garbage Collector va a estar mirando todas las referencias de los objetos de tu programa, no generes objetos nuevos innecesarios. Pogamos un ejemplo:

```
1 // Función que suma dos vectores y devuelve el resultado en un vector nuevo
2 function add(vecA, vecB) {
3     return new Vector(
4         vecA.x + vecB.x,
5         vecA.y + vecB.y,
6         vecA.z + vecB.z,
7     );
8 }
```

Código Fuente 4.5: Javascript Garbage Collector Mala Idea

En este código cada vez que sumamos dos vectores se crea un tercero. Proceso que en muchos casos va a ser innecesario y encima costoso si ejecutamos esta tarea en cada Frame muchas veces. Normalmente este tipo de operaciones se pueden hacer dejando el resultado en una de los parámetros de entrada ahorrándonos el coste de crear nuevos objetos:

```
1 // Función que suma dos vectores y devuelve el resultado en el primer vector
2 function addTo(vecA, vecB) {
3     vecA.x = vecA.x + vecB.x;
4     vecA.y = vecA.y + vecB.y;
5     vecA.z = vecA.z + vecB.z;
6 }
```

Código Fuente 4.6: Javascript Garbage Collector Buena Idea

4.1.5. Render Loop Memory

Al ser una aplicación de renderizado en tiempo real la ejecución de esta aplicación es continua en un bucle de Frames. Hay que ser conscientes de que cada 16 milisegundos se va a ejecutar el mismo código continuamente por lo tanto cualquier optimización por pequeña que sea en el bucle va a tener un gran impacto en el rendimiento general de la aplicación ya que se va a ejecutar muchísimas veces durante el proceso de ejecución.

Un ejemplo claro para entender como trabajar con bucles de renderizado es la gestión de matrices. En cada Frame tenemos que hacer el cálculo de los diferentes sistema de coordenadas ya que las posiciones cambian continuamente. Para hacer el cambio de sistemas de coordenadas hay que generar una serie de matrices, como se explicó en el componente de la cámara. Este cálculo se va a ser exactamente el mismo pero con diferentes números así que nos tenemos que asegurar que el cálculo no genere memoria nueva. Una de las pequeñas técnicas para esto es usar variables de módulo para realizar cálculos en vez de declarar nuevas variables cada frame. Otro ejemplo en código:

```
1  // vec3.direction
2  // Generates a unit vector pointing from one vector to another
3  //
4  // Params:
5  // vec - origin vec3
6  // vec2 - vec3 to point to
7  // dest - Optional, vec3 receiving operation result. If not specified result is
       written to vec
8  //
9  // Returns:
10 // dest if specified, vec otherwise
11 vec3.direction = function(vec, vec2, dest) {
12   if(!dest) { dest = vec; }
13
14   var x = vec[0] - vec2[0];
15   var y = vec[1] - vec2[1];
16   var z = vec[2] - vec2[2];
17
18   var len = Math.sqrt(x*x + y*y + z*z);
19   if (!len) {
20     dest[0] = 0;
21     dest[1] = 0;
22     dest[2] = 0;
23     return dest;
24   }
25
26   len = 1 / len;
27   dest[0] = x * len;
28   dest[1] = y * len;
29   dest[2] = z * len;
30   return dest;
31 };
32
```

```
33 // Función que retorna un vector dirección
34 function getEnemyDirection(enemy){
35   var direction= vec3.create();
36   direction=vec3.direction([cX,0.0,cZ],enemy.bb.getDrawPosition(),direction);
37   return direction;
38 }
```

Código Fuente 4.7: Memoria Render Loop Mala Idea

En este código para cada enemigo del juego y para cada frame tenemos que calcular la dirección de movimiento con la función getEnemyDirection este código tiene varios puntos negros que por si solos no son importantes pero si somos conscientes de que se va a jecutar tantas veces por enemigo y por Frame es un coste que penaliza en muchos aspectos. Primero en rendimiento y segundo en el Garbage Collector. Veamos como sería la forma correcta:

```
1 // Declaración de direction fuera del hilo de ejecución de render
2 var direction= vec3.create();
3
4 // Función que retorna un vector dirección
5 function getEnemyDirection(enemy){
6   vec3.direction([cX,0.0,cZ],enemy.bb.getDrawPosition(),direction);
7   return direction;
8 }
```

Código Fuente 4.8: Memoria Render Loop Buena Idea

En este código hemos extraído la creación de “direction” fuera del hilo principal para que sea una variable global del módulo y no se tenga que crear cada vez que ejecutamos esta función. Aparte del ahorro de memoria, si nos fijamos en la llamada a vec3.direction el último parámetro es donde se depostiva el valor resultado, por lo tanto no hace falta retornarlo y volverlo a asignar. Es un ejemplo de como una pequeña corrección puede afectar al cómputo global de la aplicación ya que tanta creación de memoria puede generar un parón en la limpieza de basura y pequeños arreglos de rendimiento acaban generando un buen resultado.

4.2. Reglas Generales

En esta sección nos vamos a centrar en reglas generales de como trabajar en una aplicación con WebGL que todo programa debería de cumplir.

4.2.1. Llamadas a WebGL

glDraw: Reducir al mínimo posible las llamadas de pintado. Cuando hacemos esta llamada estamos enviando un flujo de datos a la GPU y tenemos que aprovecharlo al máximo posible para no parar el Stream de datos. Así que si por ejemplo tenemos que pintar 50 columnas es mejor empaquetarlas en único glDraw con el inconveniente que el cálculo de posición habrá que hacerlo en GPU y no en CPU.

glGet o glRead: Todas las llamadas de WebGL que empiezan por glGet o glRead son costosas de por si porque estamos leyendo datos de la GPU usando un circuito inverso, ya que normalmente se envían datos a la GPU, no se leen de ellos. Como se explicó en la sección del funcionamiento interno de los navegadores, el mecanismo de lectura de datos de la GPU funciona mediante un forma extraña y costosa. Sólo es recomendable usarlas en modo de Debug o para propósitos muy específicos en los que se sabe el bajo rendimiento de estas llamadas porque generan comandos Flush en la GPU para acabar de ejecutar el estado actual y poder leer el dato demandado.

glGetError: Esta llamada es muy típica para recoger los errores en las llamadas de WebGL. Como se ha dicho anteriormente en entornos de producción hay que quitarlas.

Redundancia: Hay que identificar todas aquellas llamadas que sean repetitivas en WebGL. Hay muchas veces que enviamos a GPU más datos de los necesarios, repitiendo por ejemplo algunos datos uniforms que han sido previamente enviados por otro módulo. Hay que enviar los menos datos posibles. En WebGL se guarda el estado entre Frame y Frame así que podemos ahorrarnos ciertos envíos de datos si sabemos que no van a cambiar y solo enviar los cambios cuando convengan.

glCompileShader y glLinkProgram: Estas instrucciones son muy costosas y no deberíamos hacerlas durante el bucle de renderizado es mejor crear, compilar y linkar los shaders en carga de nuestro programa y tenerlos preparados para cuando se necesiten y no hacerlo dinámicamente en tiempo de ejecución.

4.2.2. Buffering

Hay varias etapas en la vida de un Frame. Como vemos en la imagen lo primero que se ejecuta el RAF (Request Animation Frame) durante este proceso se ejecuta el código de tu aplicación y se envían las llamadas pertinentes a GPU mediante WebGL. Al final de este proceso el navegador se encarga de recoger toda esa información y pintarla en el navegador como debe, communmente llamado Composite o compositor.

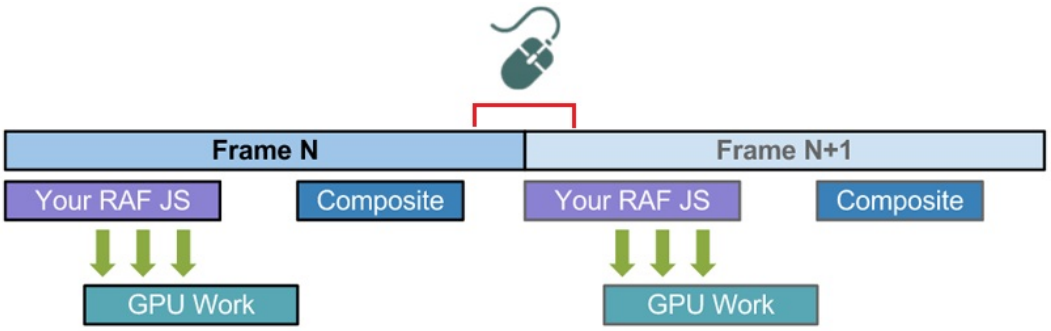


Imagen 4.1: Etapa de un frame en WebGL.

Pero por la naturaleza de los navegadores, pueden llegar eventos externos como click de ratón o temporizadores que de repente lleguen a tu aplicación y tengan que ser ejecutados en ese momento. Por ejemplo en la imagen nos llega un click de ratón al final de un Frame y ese click desencadena en algunos cálculos costosos, como el cálculo del disparo. Ese cálculo mientras esta siendo ejecutado esta relantizando el Frame siguiente generando un Frame Rate inestable y malo para la percepción del usuario que justo cuando clica hay una pequeña interrupción de pintado.

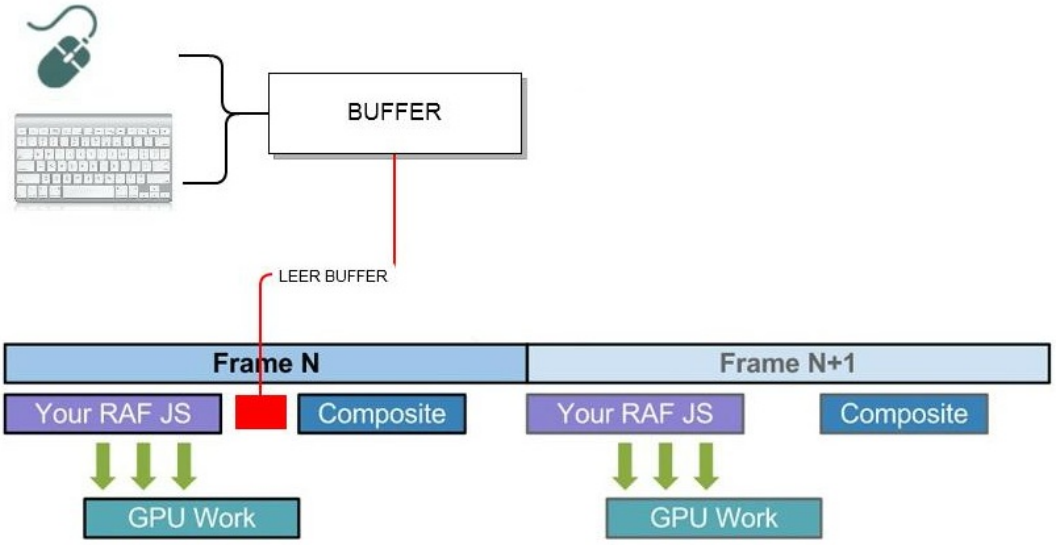


Imagen 4.2: Buffering Input Usuario.

Lo que queremos es hacer un Buffer que deje las acciones ahí guardadas y cuando nosotros queramos atenderlas. No queremos que se procesen automáticamente por el navegador. Estas acciones son el input del usuario y lo temporizadores internos de nuestra aplicación. De esta forma tenemos el control del input del usuario fuera de RAF y no perjudicamos al frame rate de la aplicación aunque haya una exhaustiva cantidad de eventos externos. De esta forma leeremos el buffer para tratarlo en el siquiente Frame con una latencia de 16 milisegundos constante sin que el Frame Rate se vea afectado. La idea principal es pasar de síncrono a asíncrono.

El ejemplo siguiente muestra como se hace Buffering de teclado en Javascript y como luego se lee asíncronamente.

```
1  // Buffer de teclas presionadas
2  var pressedKeys = new Array(128);
3
4  // Tecla presionada
5  window.addEventListener('keydown', function(event) {
6      // Guardamos para la tecla keyCode que está presionada
7      pressedKeys[event.keyCode]=true;
8
9  },false);
10 // Tecla despresionada
11 window.addEventListener('keyup', function(event) {
12   // Guardamos para la tecla keyCode que NO está presionada
13     pressedKeys[event.keyCode]=false;
14 },false);
```

Código Fuente 4.9: Buffering Javascript

```
1  // Lectura del Buffer para el movimiento del usuario de las teclas WASD.
2  if (pressedKeys['W'.charCodeAt(0)]){
3      incZ = -inc;
4      nextAnim = AnimType.WALK_STRAIGHT;
5  }
6  if (pressedKeys['S'.charCodeAt(0)]){
7      incZ = inc;
8      nextAnim = AnimType.WALK_BACKWARDS;
9  }
10 if (pressedKeys['A'.charCodeAt(0)]){
11     incX = inc;
12     nextAnim = AnimType.WALK_LEFT;
13 }
14 if (pressedKeys['D'.charCodeAt(0)]){
15     incX = -inc;
16     nextAnim = AnimType.WALK_RIGHT;
17 }
```

Código Fuente 4.10: Lectura del Buffer Javascript

4.3. CPU vs GPU

Antes de embarcarse en un proyecto de estas características hay que entender bien los propósitos de cada unidad de proceso. La GPU es una CPU con un propósito específico por lo tanto tienen diferentes aspectos de rendimiento. Algunas tareas son más rápidas en CPU y otras en GPU por eso es importante saber cual es el significado real de cada uno de ellos a la hora de decidir que debe ejecutarse en CPU y que debe ejecutarse en GPU ya que muchas veces podemos decidir en cual de los dos hacerlo.

GPU: Graphics Processing Unit

La GPU está diseñada para manejar grandes cantidades de datos en flujo de operaciones sencillas que sean paralelizables. En otras palabras, trabajos no paralelizables o datos muy costosos no son el propósito de estos procesadores.

CPU: Central Processing Unit

La CPU en cambio está diseñada para ejecutar una instrucción lo más rápido posible. Y rinde bien con operaciones complejas en una única instrucción en pequeños flujos de datos.

Con estas sencillas definiciones, podemos entender que en la unidad gráfica tienen que ejecutarse todas aquellas cosas que no sean dependientes entre vértices y píxeles. De esta forma, conseguiremos paralelizar todo el trabajo y aprovechar al máximo las dos unidades de proceso. Pero el entorno de WebGL es especial. Nuestra aplicación no está corriendo en un proceso único dedicado, está bajo el mando de un navegador y encima en Javascript, que para darnos una rápida idea es la mitad de rápido que C++ [1]. Conociendo este rendimiento tan bajo de Javascript podemos ver el sistema como si tuviéramos una CPU más lenta de lo normal y una GPU potente. Entonces habrá ciertas cosas que alomejor típicamente son realizadas en CPU que deberían ir a la GPU para liberar la CPU. Estas cosas o tareas necesitan seguir unas reglas porque la GPU tiene su propósito. En reglas generales todo lo que no tenga estado irá a la GPU. Si modifica los vértices irá en el Vertex Shader y si modifica los píxeles irá en el Fragment Shader. No es una regla general, pero Javascript es lento y todo lo que se pueda hacer fuera de ese ámbito será mejor siempre que tengamos en cuenta el propósito de las operaciones.

[1] C++ vs Javascript Link

4.4. Rendimiento Pipeline

En el proceso de pintado de WebGL hay muchas etapas e intervienen muchos factores y unos dependen de otros. Así que mejorar un aspecto puede impactar negativamente en otro. Por lo tanto a la hora de optimizar el Pipeline de WebGL hay que seguir unas guías fijas. Esta sección ha sido integramente copiada del libro GPU Gems de NVidida. Lo importante es entender e identificar todos los pasos del Pipeline de WebGL, adaptarlo al siguiente gráfico y buscar donde está nuestro cuello de botella si lo hay.

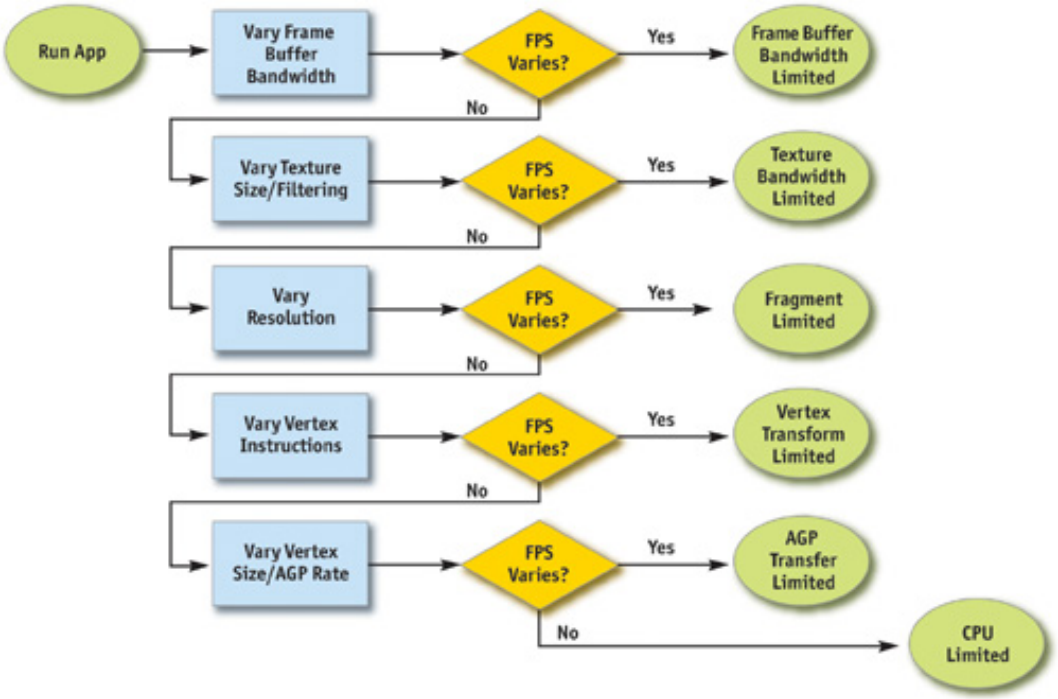


Imagen 4.3: Rendimiento Pipeline.

4.5. Web Workers

Un Web Worker es un Script Javascript ejecutado en HTML que es procesado en segundo plano e independientemente.

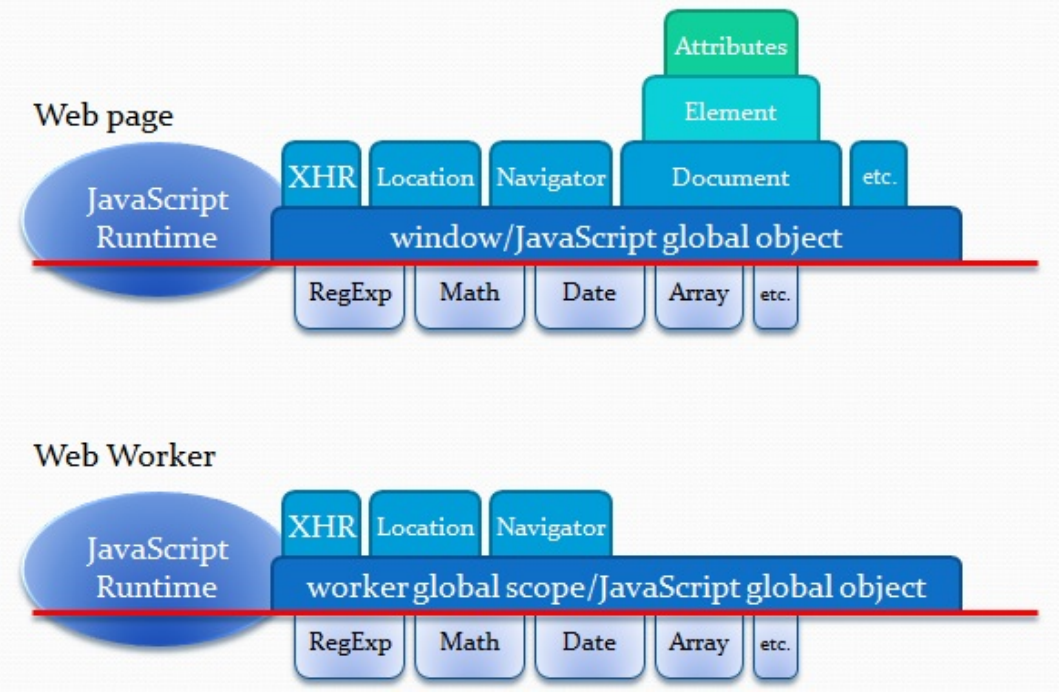


Imagen 4.4: Esquema Web Worker. Imagen de un Blog de Microsoft.

Como vemos en la figura un Web Worker trabaja en paralelo con el hilo principal. No tiene acceso al elemento Document del navegador con lo que no se puede comunicar con la interfaz gráfica HTML pero nos proporciona un hilo de ejecución extra para hacer trabajos en segundo plano.

Una aplicación de WebGL tiene un bucle de renderizado dependiente del Navegador. Durante cada iteración tenemos que ser capaces de pintar la escena con WebGL, simular la física, ejecutar la lógica del juego y simular la Inteligencia Artifical entre otras muchas cosas. Cuando por ejemplo integré la física en este bucle no era capaz de simular tantas cosas en un frame. 16 milisegunos no eran suficientes.

Así que gracias a los Web Workers es posible ejectuar en paralelo tareas extras como la física. De esta forma hacemos que la física en vez de trabajar síncronamente con el renderizado trabaje asíncronamente con el sistema de render. Este forma es perfecta porque si uno de los dos componentes se relantiza no impacta en el otro, aparte de separar la lógica y no generar cohesión. Sabemos que el bucle de renderizado tiene que ir a 60 FPS porque es lo óptimo para refrescar la pantalla en sincronización (VSync). Pero no hay necesidad de que la física simule a esta velocidad. Gracias a los Web Workers

podemos simular la física a una frecuencia diferente que se adapte con las características del cliente y sea adaptativa.

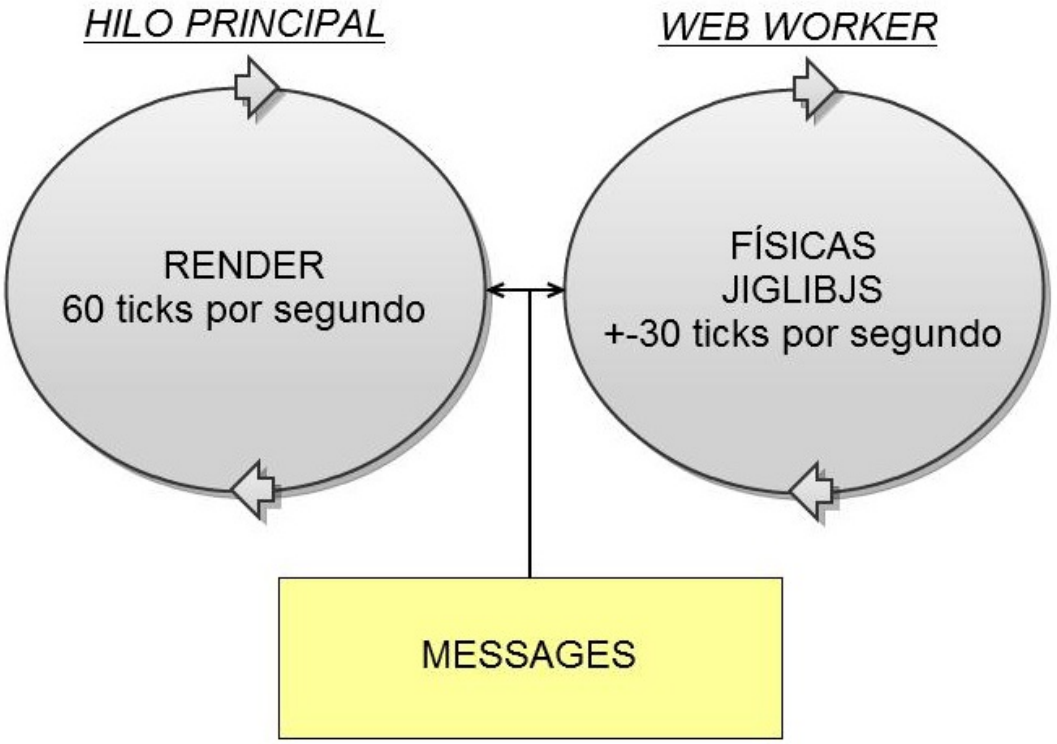


Imagen 4.5: Esquema Web Worker con la física.

De esta forma el hilo principal de la aplicación solo se tiene que encargar de tratar el estado actual y pintar las posiciones de los elementos simulados en el WebWorker. ¿Pero como tratamos tanta información? Mediante las Bounding Boxes. Cada elemento de la escena tiene una Bounding Box que engloba su volumen. Este volumen será pasado al WebWorker que lo añadirá a su sistema y simulará a cada paso su posición. Esa posición sera devuelta a las Bounding Boxes originales y el hilo principal solo se dedicará a leer esa posición para pintarlo ahí. Si quitáramos las físicas todos los elementos estarían en su posición original sin ningún tipo de movimiento.

La caja Messages representa la forma en que el Web Worker se comunica con la aplicación principal. Estos Messages son JSON, estructuras de datos de Javascript que representan objetos. Así que en este caso es muy fácil representar un Array de Posiciones que es lo que necesitamos para pintar los elementos. En conclusión, existen dos mundos en esta aplicación el mundo del Render, que se dedica a pintar los modelos en las posiciones de sus Bounding Boxes y el mundo físico, una representación de esas Bounding Boxes que según su masa, fricción y fuerzas externas simulará esas posiciones.

4.6. Optimizar Pintado

Normalmente cuando queremos pintar varios elementos en una escena hay que ordenar la escena de atrás a adelante para conseguir una visibilidad correcta y que los elementos más cercanos sean los últimos en ser pintados. Por ejemplo en Canvas 2D se usa mucho. Se ordena por profundidad (z-index) y se pinta en orden.

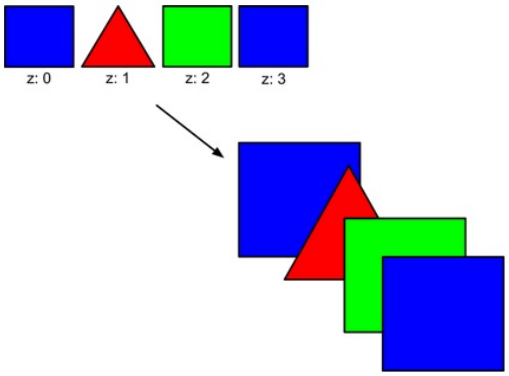


Imagen 4.6: Pintado Lógico en Canvas.

Pero esta sería la peor forma de pintar en WebGL porque estaríamos llamando dos veces al pintado del cubo cuando podríamos enviarlo solo una vez. Y es aquí donde entra también el Test de Profundidad. En OpenGL es común pintar de adelante a atrás primero ordenado por estado. Primero se ordenan los objetos de la misma forma para optimizar las llamadas de pintado en una sola y luego se ordena de lo que se ve primero a lo que ve último. De esta forma todos los fragmentos que no superen el test de profundiad no serán tratados y por lo tanto no se ejecutarán en el Fragment Shader. Si lo hiciéramos alrevés como en Canvas cada forma geométrica que pintamos estaría por delante de la anterior y todos los fragmentos se ejecutarían en los shaders innecesariamente. Es la forma lógica de aprovecharse del test de profundidad.

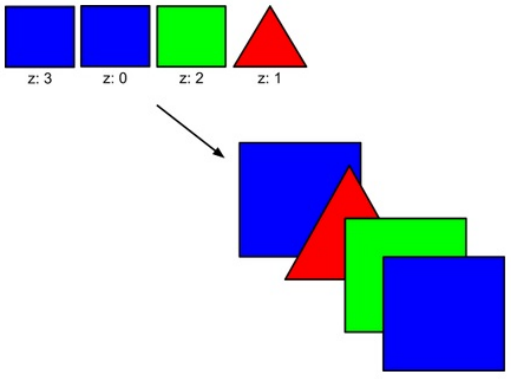


Imagen 4.7: Pintado Lógico en WebGL.

La ordenación por estado quiere decir que se ordenen los elementos de la siguiente manera:

- 1º - Por FrameBuffer or por Contexto.
- 2º - Por Programa/Buffer/Textura
- 3º - Por Uniforms/Samplers

Una vez tenemos esta ordenación, lo siquiente sería ordenarlos por profundidad como mostraba la imagen anterior. En el caso de tener elementos traslucidos, como es nuestro caso con el rayo láser de disparo, hay que pintarlos de atrás hacia delante para que se acumule el color transparente a cada capa que añadamos. En conclusión hay que seguir este orden:

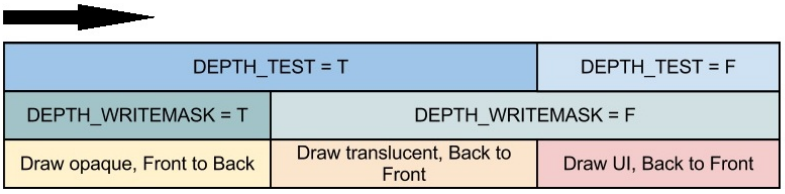


Imagen 4.8: Orden de Pintado en WebGL.

4.7. Optimizar Geometría

Hay una lista de cosas que tenemos que tener en cuenta cuando lidiamos con mucha geometría. Las recomendaciones que hago son las siguientes:

- Reducir el número de Vértices - usar buffers de índices.
- Reducir datos por vértice - Más rápido de enviar a GPU.
- Intentar alinear los datos de los atributos a 4 bytes.
- Usar los datos más pequeños posibles: $BYTE < SHORT < FLOAT$

Por ejemplo para enviar dos triángulos a GPU podemos hacerlo de la siguiente forma.

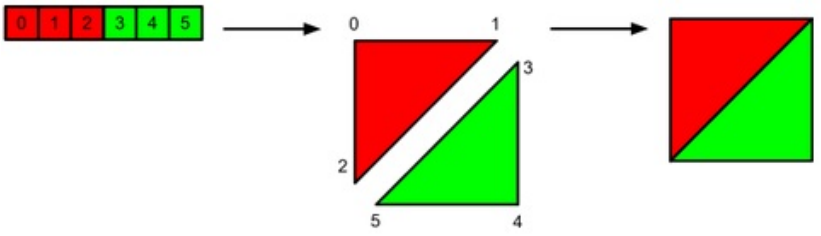


Imagen 4.9: Triángulos sin índices.

De esta manera estamos enviando la información de los 6 vértices que pueden ser su posición, sus coordenadas de textura, su material y demás cosas. Si usamos índices podemos reusar los vértices que comparten arista reduciendo a 4 el número de vértices que tenemos que enviar y solo referenciarlos con un array de índices de tipo pequeño:

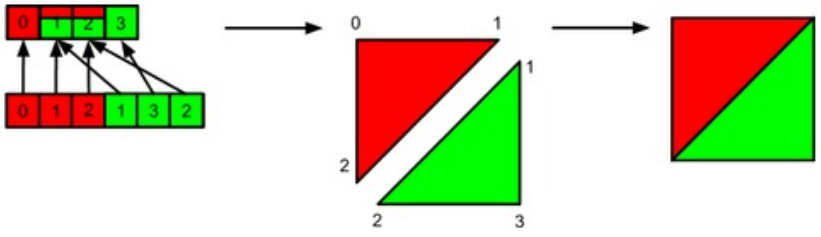


Imagen 4.10: Triángulos con índices.

A la hora de recorrer los Arrays de Datos para enviarlos a GPU hay dos formas típicas. Tener una serie de Arrays (*Struct of Arrays*) o un Array con los datos contiguos (*Array of Structs*). Para aprovechar la localidad espacial de los datos es mejor usar un único Array así cuando la CPU cargue los datos se podrá aprovechar de localidad espacial de los siguientes datos y aprovechar el "cacheo".

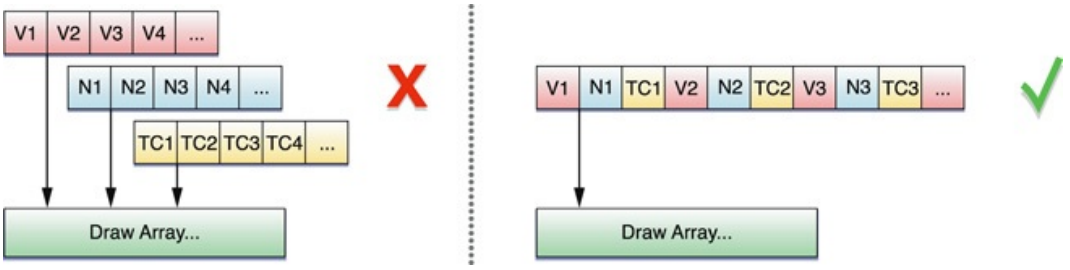


Imagen 4.11: Estructura de datos.

4.8. Optimizar Shaders

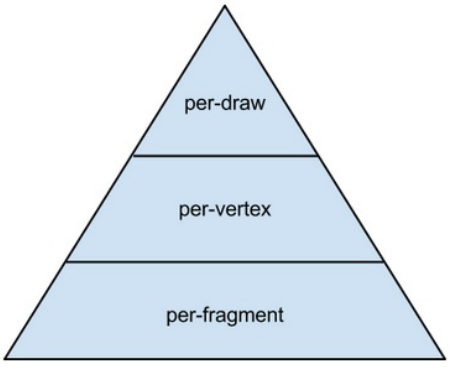


IMAGEN 4.12: Pirámide de número de llamadas.

Pongamos por ejemplo que queremos pintar un cubo. Hay que hacer **una** llamada a glDraw. Esa llamada desenvoca en **6** ejecuciones del Vertex Shader por los 6 vértices de un cubo. Y finalemente esos 6 vértices se convierten en **muchos** fragmentos, muchos más que vértices. Así que siempre tendremos menos llamadas de pintado que vértices y que fragmentos. Lógicamente todo lo que computemos antes mejor. Si es posible hacer el cálculo de matrices una vez en CPU que miles de veces en GPU mejor. Y normalmente muchos de los cálculos que se hacen en el Fragment Shader se pueden pasar al Vertex Shader con interpolación. En este proyecto mucha de la lógica de iluminación se ha pasado al Vertex Shader por puro rendimiento, como el cálculo de luz difusa y ambiente.

Los Shaders son códigos muy delicados y deben ser tratados con detalle. Hay que intentar siempre usar la precisión mínima posible tanto en los atributos como en las variables Varying. También es aconsejable cambiar de programas para proporcionar diferentes niveles de detalle. Hay muchos elementos en una escena que si nosotros sabemos que están lejos del punto de visión alomejor no necesitan un cálculo tan preciso de la luz o de Skinning. Es recomendable usar trucos matemáticos que reduzcan los ciclos de ejecución mientras el resultado esté bien. Unos ejemplos:

- Multiplicar por dos :

```
a<1
```

- Mínimo entre dos valores:

```
r = y ^ ((x ^ y) & -(x < y))
```

- Si es par:

```
(((x)&1)==0)
```

- Intercambiar valores:

```
(((a) ^ (b)) && ((b) ^= (a) ^= (b), (a) ^= (b)))
```

Estos pequeñas tiras de código reducen el cálculo de operaciones comunes y lo más importante es que no usan condicionales. Los condicionales tienen un coste muy grande y más en la GPU. Las GPUs son muy buenas paralelizando operaciones pero siempre que le dejemos hacerlo. Por ejemplo veamos este código:

```
1 vec2 value = texture2D(s_lookupSampler, uv).st;
2
3 // GPU Parada esperando el valor
4
5 gl_FragColor = texture2D(s_textureSampler,value);
```

Código Fuente 4.11: Dependencia de Datos GLSL

En este código hay una dependencia de datos, en este caso, mirar el valor de una textura para volver a mirar en otra. Este tipo de código es común para hacer trucos y optimizar el envío de coordenadas a la GPU en texturas, pero es muy costoso ya que es poco paralelizable porque tenemos a la GPU esperando un valor y sin hacer nada mientras tanto. La única solución a lecturas de datos tan dependientes es poner cálculos de otras cosas en medio para que mientras llega el valor de la primera textura se puedan hacer otros cálculos.

4.9. Redimensionar Canvas CSS

Cuando declaramos un Canvas en el navegador por ejemplo de 600 x 800 píxeles lo que estamos haciendo en el fondo es tratar 480.000 píxeles como mínimo si no hubiera repeticiones. Nuestra aplicación puede ir bien a esta resolución pero si hacemos el Canvas a pantalla completa por ejemplo a 1600 x 900, resulta en 1,440.000 píxeles, casi el triple. Esto puede perjudicar mucho el rendimiento de nuestra aplicación en la parte de cálculo de fragmentos que ya de por sí es la más usada en la pirámide de llamadas. Hay una técnica muy usada y muy útil que se puede usar y casi sin coste. Para poder usar el juego a pantalla completa sin tener que declarar un Canvas del máximo tamaño es posible escalar el canvas con CSS.

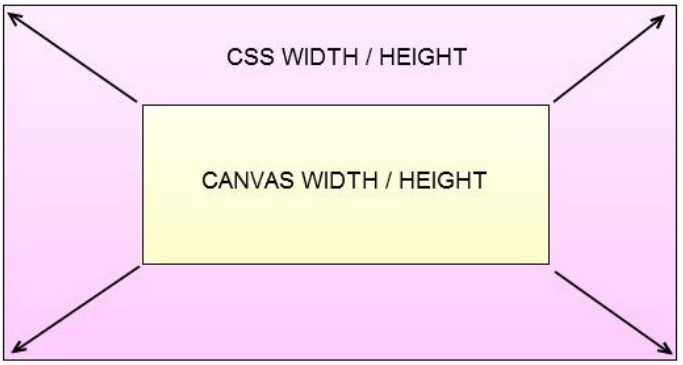


Imagen 4.13: Redimensionar Canvas con CSS.

De esta forma podemos declarar un canvas de 600 x 800 y con CSS escalarlo a 1600 x 900. El compositor del navegador hará interpolación bilineal casi sin coste. Así que podremos disfrutar de la inmersión de estar a pantalla completa pero renderizando en un canvas más pequeño, por lo tanto, usando menos fragmentos y usando menos la GPU.

4.10. Flujo de Datos

En la sección en la que examinábamos como implementan los navegadores WebGL vimos que hay varios buffers entre una aplicación y la tarjeta gráfica. Aparte de los que añade el navegador existen más buffers, por ejemplo, los que hay en DirectX antes de llegar a la GPU y otro más en los propios drivers. Lo que se tiene que evitar a toda costa es que cualquiera de estos buffers internos se llene. Si un buffer se llena se tiene que esperar a que se ejecute todo como si de un glFinish[1] se tratara. Estas limpiezas de buffers se llaman SyncFlush porque sincronizan el envío de datos y son malos en rendimiento. Lo que se quiere es que los datos fluyan como si de una cadena de montaje hablaramos. No queremos que ninguna etapa retrase a las demás.

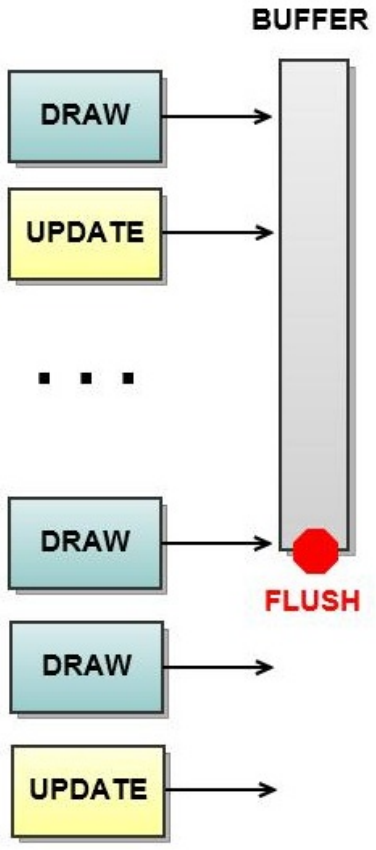


Imagen 4.14: Render Sync Flush.

Lo que llena estos buffers son los comandos y su contenido. Como regla principal lo que queremos es leer y escribir lo menos posible en GPU. Pero esto no es siempre fácil, depende de la aplicación. Si queremos hacer un juego, de alto nivel gráfico, no nos queda otro remedio que enviar mucha información a la GPU por eso hay que ajustarse bien y saber lo que se hace en todo punto para no estresar el Pipeline y reducir el Frame Rate en consecuencia. Tener en cuenta los tipos de los datos, los tamaños de las texturas y la redundancia de datos con las demás optimizaciones comentadas anteriormente.

Como se muestra en la imagen, una situación muy común con relación a lo comentado anteriormente es por ejemplo la siguiente. Enviamos llamadas a WebGL de pintado, otras de actualización, cualquier cosa que acceda a WebGL y de repente el buffer se llena y cuando hacemos la siguiente llamada no hay espacio y se bloquea (marcado con la figura roja). Pero la aplicación no ha acabado de enviar todo lo que tenía que enviar (llamadas posteriores a la marca roja) e internamente se va a generar un Sync Flush, que como hemos comentado, va a esperar a que todo lo anterior se ejecute vara vaciarlo rompiendo la cadena. Lo que veremos en los profillers de llamadas será que el DRAW que generó el derrame del buffer habrá tardado muchos

[1]Es un comando de WebGL/OpenGL que no devuelve el control hasta todas las llamadas previas han sido finalizadas.

milisegundos. Pero no es esa la llamada que genera el problema, sino que el buffer se ha llenado por culpa del resto de la aplicación.

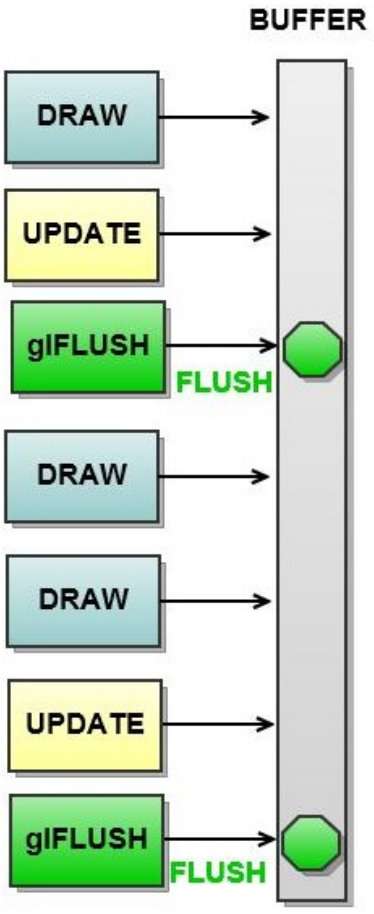


Imagen 4.15: Render gl Flush.

Lo que queremos evitar son los Sync Flush que internamente va a generar el navegador y que nos van a parar la aplicación y posiblemente perder el Frame si hay muchos. La idea es introducir Flushes manualmente para no que no lleguen Sync Flushes. Introducir un Flush es llamar a la función glFlush[1] de WebGL. Con esta llamada le decimos al sistema que vaya ejecutando todo lo que tenga guardado. Estos glFlushes tienen un coste por eso hay que ponerlos estratégicamente en tu aplicación para tampoco generarlos inútilmente. Hay que por lo tanto analizar la aplicación, ver donde están las partes más costosas de datos y ponerlos adecuadamente. De esta forma evitaremos llenar los buffers internos y no llegar a un Sync Flush.

Hay que intentar evitar los Sync Flush, pero no siempre es posible. Un cambio de programa, mucho ancho de banda, cambios del estado del contexto provocan inevitables Sync Flushes. Pero mientras seamos consciente cuando sucede, qué sucede y porqué, sabremos arreglarlo, tener el control de la aplicación y adaptarnos a ello en consecuencia.

[1]Indica a WebGL/OpenGL que se ejecuten los comandos almacenados hasta el momento lo más rápido posible sin bloquear.

Capítulo 5

Planificación y Costes

5.1. Planificación y Costes

Planificación

En el diagrama de Gantt podemos ver la planificación del proyecto tal y como ha sido desarrollada. Ha sido levemente simplificada para mostrar las tareas más importantes. Las partes en paralelo son tareas que han sido desarrolladas en paralelo por las dos personas de este proyecto. Los recursos de trabajo han sido calculados a 4 horas diarias por persona.

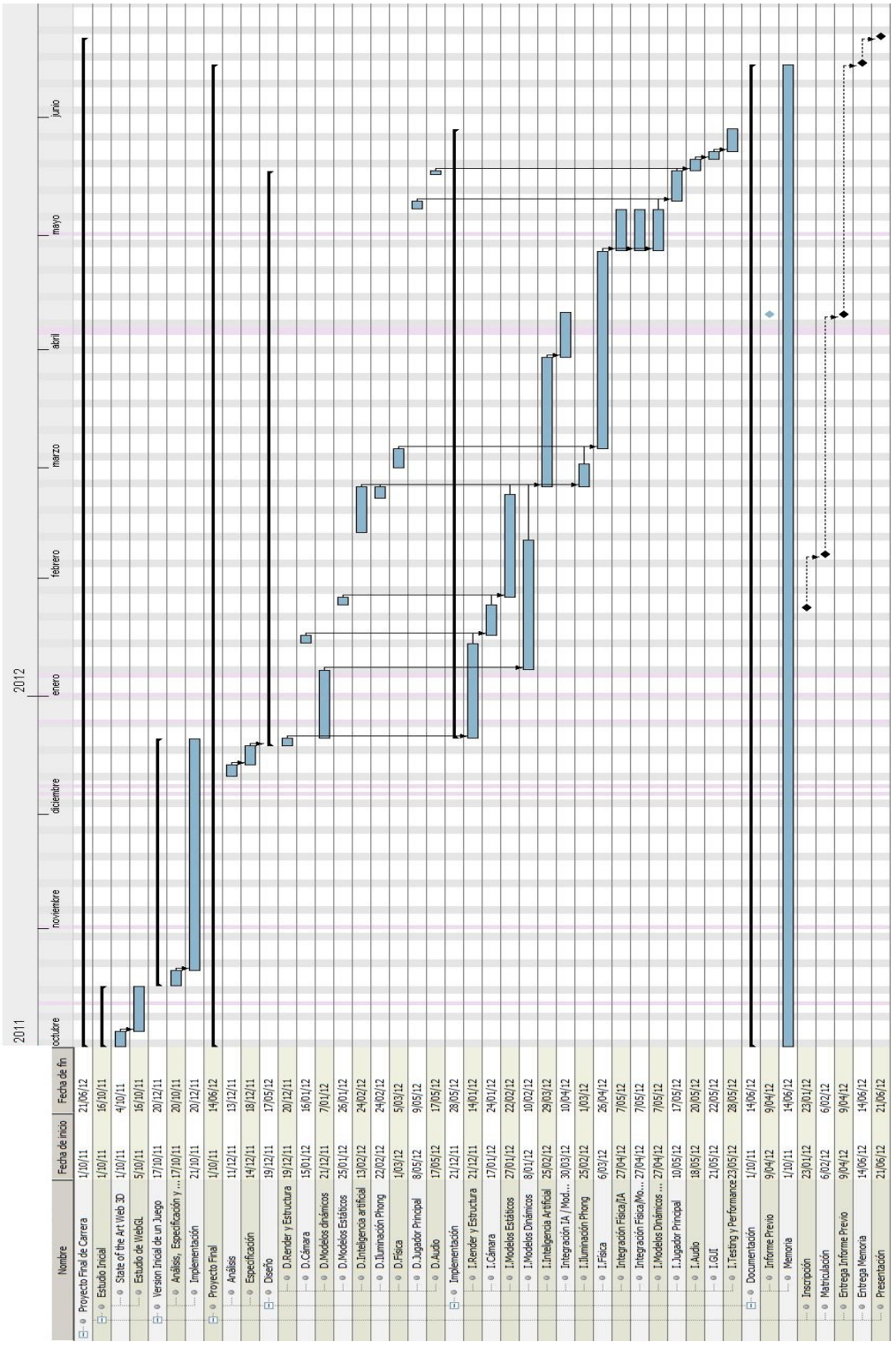


Imagen 5.1: Diagrama Gantt de la Planificación.

En el siguiente diagrama se muestran las tareas que yo personalmente he dedicado en el transcurso de este proyecto y mi rendimiento sobre ellas.

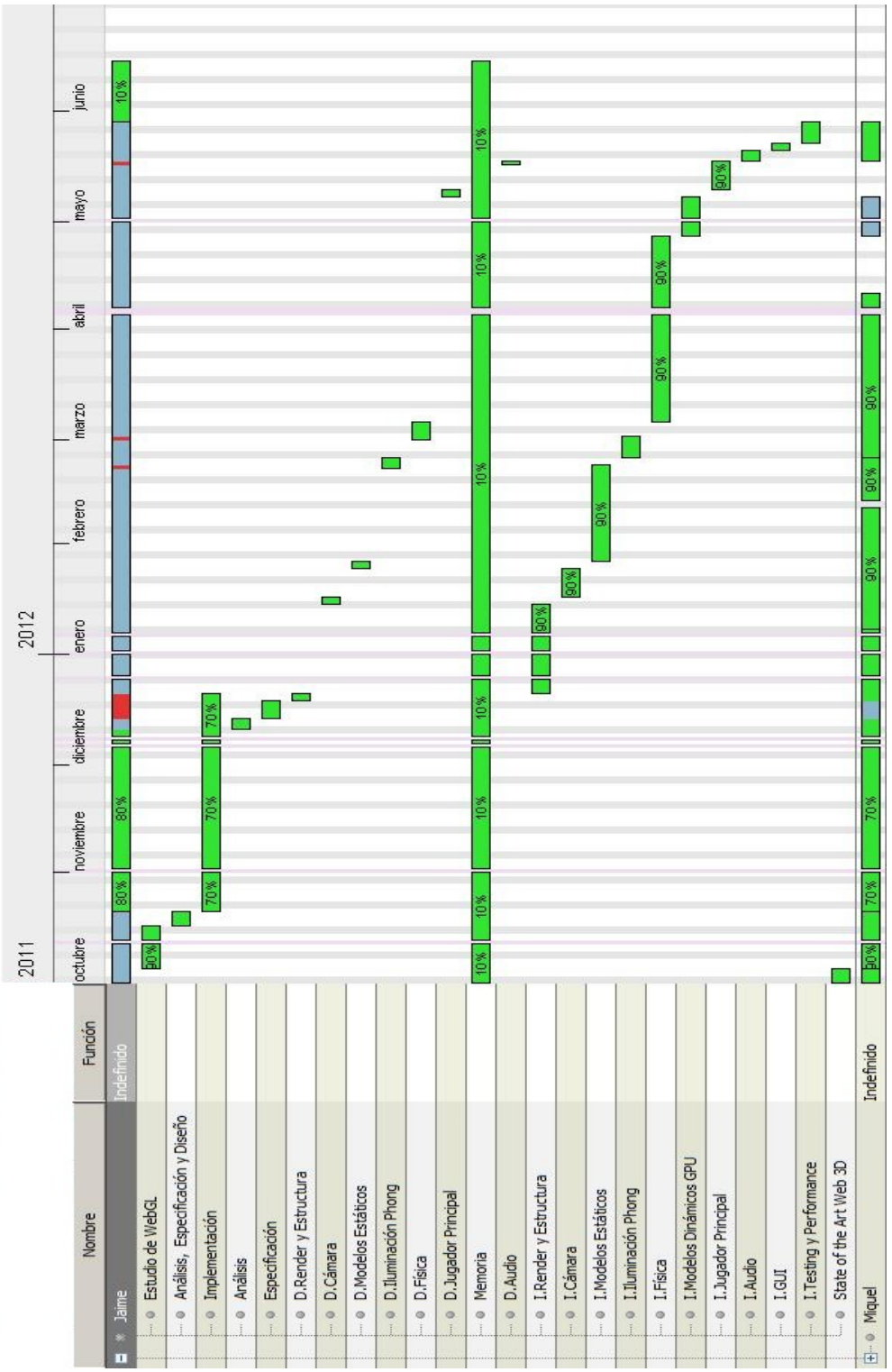


Imagen 5.2: Diagrama de Rendimiento Personal.

A partir de la planificación y del rendimeinto de cada una de las tareas podemos dar una aproximación de las horas invertidas en las tareas más importates.

Tarea	**Dedicación** ($Horas$)
Estudio Inicial	$[128 * 0{,}9] = 115$
Versión Inicial	$[496 * 0{,}9] = 446$
Análisis	$[24 * 0{,}9] = 21$
Diseño	$[360 * 0{,}9] = 324$
Desarrollo	$[1088 * 0{,}9] = 979$
Pruebas	$[64 * 0{,}9] = 58$
Documentación	$[1984 * 0{,}1] = 198$
TOTAL	2141

Tabla 5.1: Tabla de tareas y horas.

Cada tarea ha sido multiplicada por su factor de rendimiento ya que durante un día (carga de 8 horas de trabajo) se ha dedicado un 90 % a la aplicación y un 10 % a la documentación. Por lo tanto y ya que la memoria se extiende en todo el ancho del desarrollo, podemos multiplicar los factores por el total de horas de cada tarea. Por ejemplo la documentación ha tenido una duración de 248 días a 8 horas el día: 1984 horas en total que por su factor de rendimiento 0.1 (10 % al día) resulta en: 198 horas.

Como podemos apreciar en la planificación y en la tabla de tareas el resultado es 2141 horas. Al ser dos personas podemos decir que si el trabajo es entre dos son 1070 horas dedicadas por persona. Como vemos el proyecto empezó antes de un semestre entero dedicando más horas de las rigurosamente especificadas. Este tiempo extra se usó para hacer un estudio y una versión inicial. Si miramos la tabla de tareas este tiempo de estudio y versión inicial acumulan 561 horas, entre dos, 280 horas. Si a las 1070 horas totales le restamos estas 280 de estudio inicial quedan 790 horas que es la carga aproximada de 37.5 créditos a 20 horas por crédito, 750 horas.

Costes

Teniendo definidas las tareas bien desglosadas podemos adjudicarlas a diferentes perfiles profesionales que se ajusten a un coste real. Los perfiles identificados son los siguientes:

- **Analista**: Tiene como cometido describir una solución, definir requisitos funcionales y no funcionales de la aplicación, casos de uso, etc. En resumen, define qué ha de hacer la aplicación, y es también el principal responsable de escribir la documentación técnica.
- **Diseñador**: Basándose en la descripción del analista, diseña una solución utilizando una tecnología concreta. En resumen, define cómo funcionará internamente la aplicación.
- **Programador**: Encargado de implementar la aplicación partiendo de la especificación y diseño facilitados. Además será el encargado de realizar las pruebas de la aplicación.

A continuación se muestra la distribucion de las horas según la dedicación de las tareas y los perfiles descritos.

Tarea	**Dedicación** (*Horas*)	**Analista**	**Diseñador**	**Desarrollador**
Estudio Inicial	115	50	50	15
Versión Inicial	446	20	20	406
Análisis	21	15	4	2
Diseño	324	15	259	50
Desarrollo	979	9	20	950
Pruebas	58	0	0	58
Documentación	198	100	50	48
TOTAL	2141(100 %)	209 (9.8 %)	403 (18.8 %)	1529 (71.4 %)

Tabla 5.2: Tabla de dedicación por perfil.

Sabiendo las horas invertidas por cada perfil profesional, se requiere calcular el coste total por horas y por perfil.

Perfil	**Dedicación** (*Horas*)	**Sueldo €**/*Hora*	**Coste Total (€)**
Analista	209	20	4180
Diseñador	403	22	8866
Desarrollador	1529	15	22935
TOTAL	2141	-	35981 €

Tabla 5.3: Tabla de costes de trabajadores.

También hay que sumar el coste de los recursos usados durante el desarrollo de la aplicación. Uno de los objetivos de la aplicación era no usar Software privativo y se ha conseguido. Por eso la suma de recursos se reduce únicamente al coste de poseer un ordenador en Linux.

Recurso	**Coste Total (€)**
Ordenador	800
Ordenador	800
Sistema Operativo	0
Licencias	0
Modelos Gráficos	0
TOTAL	1600 €

Tabla 5.4: Tabla de costes de recursos.

Finalmente falta sumar los costes de los trabajadores con los costes de los recursos resultando en el precio total.

Tipo	**Coste Total (€)**
Trabajadores	35981
Recursos	1600
TOTAL	**37581 €**

Tabla 5.5: Tabla de costes totales del proyecto.

Capítulo 6

Conclusiones

6.1. Evaluación de Objetivos

Una vez finalizado el proyecto se exponen, los objetivos iniciales con su conclusión y valoración personal:

Crear un juego innovador, accesible a la mayoría de plataformas sin instalador, sin plugins, sólo con un navegador apto y una tarjeta gráfica.

Una vez finalizado el juego podemos decir que este objetivo ha sido completado sobretodo en las características tecnológicas. Desde que conozco el mundo WebGL me atrevo a decir que no he visto ningún juego de este tipo y de estas características. El juego no ha sido completado con el objetivo de ser jugado masivamente sino más con el objetivo de explotar el potencial de WebGL mediante él. Por eso hay puntos del GamePlay, Inteligencia Artificial y Cámara que podrían ser más acurados pero es sólo cuestión de dedicación que no afectaría en nada al análisis ni al diseño propuesto.

Usar tecnologías libres durante todo el proyecto.

Como se ha especificado en los costes del proyecto no se ha usado ningún Software de pago, más que eso, Software libre de código abierto en todos sus ámbitos. Gracias a esto demostramos que se puede llegar a hacer una aplicación Web 3D con costes de licencias igual a cero. Creo que es importante remarcar esto porque invita a muchos estudiantes y Start Ups a elegir tecnologías de este tipo.

Llegar a crear una escena realística en tiempo real

Gracias a la iluminación de la escena, el tratamiento de materiales y los Shaders de iluminación podemos comprobar en el juego como el cálculo de iluminación es en tiempo real y no mediante modelos predifinidos de luces. Si a esto le sumamos una buena definición de materiales para cada elemento de la escena se acaba creando una sensación realística muy buena.

Aparte del procesamiento de la escena, ha sido necesaria dotarla de elemtos gráficos complejos. Esto se ha conseguido con la importación de modelos Wavefront y MD5. Sólo queda comprobarlo en la aplicación.

Crear una sensación de inmersión digna de aplicaciones de escritorio.

Las juegos basados en aplicaciones de escritorio brillan más que los juegos web simplemente porque tienen más recursos del sistema y son capaces de generar más efectos. Pero desde que la Web se ha convertido en una plataforma también de aplicaciones brinda la posibilidad de generar Webs en pantalla completa. Gracias a HTML5 podemos incluir sonidos de una forma portable y , por último, gracias a WebGL tenemos una aceleración en 3D. Así que viendo el resultado del proyecto creo que está a la altura.

Conseguir una tasa de renderizado de 60FPS en el navegador.

Es un punto dependiente del Hardware totalmente. Esta aplicación se ha llevado al límite. Así que para disfrutar de un renderizado a 60 FPS hace falta una tarjeta gráfica decente dedicada y una CPU medianamente potente. El juego ha sido sometido a varias optimizaciones y somos conscientes de donde esta el coste que genera, pero hemos querido exigir más Hardware para poder introducir más elementos y más complejidad hasta donde hemos podido.

Conocer WebGL en todo su potencial.

Después de la lectura de esta memoria creo que queda demostrado el conocimiento de esta tecnología. No solo como funciona el lenguaje y su API, sino como es interpretada por el navegador, como llega a la GPU y como es ejecutado internamente. Hubiera sido imposible exponer tantas técnicas de optimizaciones sin conocer tanto WebGL. Gracias a ello puedo decir que la aplicación está a un gran nivel técnico.

Analizar WebGL como futura posibilidad 3D y multi plataforma.

Este es un punto de gran discusión en los foros dedicados a este tema. Yo, como desarrollador de esta aplicación, tengo mis convicciones en que sí. Esta tecnología abre un abanico enorme a la Web 3D y de forma libre. Esta más que demostrado que es multi plataforma pero para que se convierta en la principal tecnología 3D de la Web aún queda un camino largo. Hay grandes competidores ahí fuera que están ofreciendo casi el mismo potencial pero ligado a un Software privatido como Adobe o Unity.

Como comentaba en el apartado *State of the Art de la Web 3D*, a WebGL le faltan un par de empujones. Uno de ellos es su soporte que no es muy alto por culpa de Internet Explorer. Aunque vemos que esto está cada vez está más cerca de no ser un problema. Y otro es alguna inversión de alguna compañía de 3D tanto de videojuegos como diseño de esta tecnología que abra su conocimiento. Uno de los problemas que esta teconología tiene para integrase es la falta de desarrolladores de este perfil. Muchos de los desarrolladores Web han trabajado con Adobe Flash, la plataforma multimedia por excelencia de la Web, pero muy pocos de ellos están acostumbrados a lidiar con API's de tan bajo nivel como WebGL, que requiere de conocimientos extendidos de gráficos y de rendimiento. Normalmente este perfil de desarrollador no está en la Web. Yo mismo, antes de empezar este proyecto, venía de estar programando aplicaciones en C++ de alto rendimiento. Igual que he tenido cierta experiencia en trabajar con los temas internos de las aplicaciones , he tenido que aprender a programar y diseñar en la Web, dos perfiles muy diferentes.

Estudiar estrategias y patrones para adaptarse a la Web.

La Web es un entorno muy diferente a otros. No es ni servidor ni escritorio ambos a la mano con el Sistema Operativo. Es una plataforma atada al navegador, aplicaciones de terceros que depende totalmente del cliente que las ejecuta.

La Web trabaja siempre asíncronamente, desde la carga de recursos hasta los eventos del usuario. Como se ha visto en la implementación de este proyecto, se ha adaptado los típicos patrones de diseño básico a la Web tales como patrones de diseño, eventos de usuario, orientación a objetos y técnicas de optimización.

Explotar las posibilidades de HTML5

Aunque WebGL no es parte de la especificación de HTML5, van de la mano. Es imposible invocar un contexto WebGL sin un elemento Canvas. El elemento Canvas es propio de HTML5.

Tanto las texturas y el audio son elementos de HTML5. Gracias a ello he tenido que ir a explorar este nueva especificacón de HTML y me he dado cuenta que es mucho más potente de lo que se cree. HTML5 está totalmente preparada para albergar aplicaciones web de alto nivel. Uno de los ejemplos es esta aplicación.

Diseño de alto nivel y escalable para en el futuro poder añadir más elementos.

En el apartado de diseño de esta aplicación se ha propuesto un modelo en componentes que es totalmente escalable. El compromiso de este diseño es adaptarse a futuros cambios y nuevas funcionalidades haciendo de este proyecto casi una estructura de diseño de un juego, más que de un juego en sí.

Asentar los conociemientos de la carrera de informática.

Han sido varias las veces que he tenido que ir a buscar apuntes de la carrera durante el desarrollo de este juego desde estructura de datos, coste de algoritmos, algoritmos de búsqueda o A^*, iluminación, trigonometría, etc. Sin todos estos conocimientos hubiera sido imposible desarrollar este proyecto.

6.2. Dificultades

La primera dificultad que tuve al empezar este proyecto fue el desconomiento de las tecologías Web. Yo venía de un entorno muy diferente, desarrollo de un servidor en C++ en Linux. La Web tiene muchas tecnologías y lenguajes internos y hace falta conocerlos bien para poder llegar a hacer aplicaciones como esta. Es imposible ponerse a desarrollar en WebGL sin conocer primero HTML y Javascript. Empecé desde cero.

Una vez preparado para empezar a conocer WebGL fue la falta de información. Cuando empecé con WebGL aún no había una versión oficial y la documentación y los ejemplos eran realmente escasos, llegué a pensar sobre la inviavilidad de este proyecto. Mediante una búsqueda exhaustiva, una examinación detallada de los ejemplos y el paso del tiempo que ofreció más información empecé a creer en ello. Pero los inicios fueron muy desconcertantes.

Uno de los puntos más díficiles que he tenido con este proyecto ha sido integrar una librería de físicas 3D en Javascript. No existe ninguna librería nativa en Javascript son todo traducciones de otros lenguajes con compiladores multi lenguajes. Aparte de la dificultad de adaptarla, estas traducciones tienen una documentación mínima y un rendimiento malo ya que originalmente no están pensadas para ser ejecutadas en Javascript. Uno de los puntos más claves de este proyecto ha sido la implementación de un Web Worker dedicado a la física. Después de casi 1 mes dedicado a ésto al final fui capaz de conseguir un rendimiento bueno pero fue una tarea muy dura que al principio no le veía solución posible. Llegué a pensar que era mejor hacer un sistemas de físicas manual y en 2D. Después de multiples integraciones, pruebas y debug se consiguió. Es uno de los puntos que más orgulloso estoy porque se ha creado un pequeño precedente de cómo hacerlo.

Al igual que la física, lo mismo pasaba con los modelos MD5. La librería que usé para cargar los modelos tenía un rendimiento malísimo, era casi imposible instanciar más de un modelo. Después de identificar el problema modifiqué la librería para que hiciera el Skinning por GPU y no por CPU. Por suerte, no fue tan difícil hacerlo pero fue dura la decisión de ponerse a hacerlo porque no sabía si era posible. Creo que es la primera aplicación que conozca que haga un Skinning por GPU de esta forma. Quiero decir de esta forma porque no todos los tipos de Skinning son iguales.

GLSL es el lenguaje para crear los Shaders, los programas de la GPU. En WebGL no hay forma posible de hacer Debug. O funcionan o no funcionan. Es muy típico estar programando un Shader, probarlo y surgir una pantalla negra sin nada de información.

Es muy tedioso lidiar con un lenguaje de tan bajo nivel sin ningún tipo de información. Creo que los navegadores deberían de hacer algo al respecto. La única forma de Debuggear era usar la propia lógica del Shader como output para los valores algo muy feo pero necesario.

Ha sido una dificultad saber donde parar de desarrollar. Este proyecto podría durar mucho más tiempo del propuesto porque hay muchas áreas sin analizar como por ejemplo el uso de texturas comprimidas, adaptación a los móviles, renderizado adaptativo, multiplayer, etc. Se tuvo que poner una fecha límite y unas funcionalidades estrictas porque el desarrollo de un juego es complejo y a veces es difícil saber en que lado estás si haciendo un proyecto final de carrera ,haciendo un juego porque te gusta o programando una librería de físicas sin rumbo ninguno. Cuando abres tantos frentes y tan complejos es difícil saber parar y buscar el objetivo final.

Este proyecto ha sido desarrollado en pareja. Cuando trabajas en pareja, no estás solo. Tiene que haber mucha comunicación y muchas veces hay puntos de vista diferentes. No tiene porque ser un punto negativo pero a veces genera cierta dificultad y coste extra trabajar en grupo porque hay que saber escuchar, entender a la otra persona y cumplir ciertos tiempos que no existirían si no dependes de nadie.

6.3. Conclusiones personales

Este proyecto me ha proporcionado muchos benificios. He crecido como Ingeniero Informático porque he dado soluciones a problemas continuamente, he conocido nuevas técnicas de programación, he aprendido un lenguaje nuevo y es la primera vez que creo un juego de estas dimensiones. Desde el principio siempre he querido primar el aprendizaje y el desarrollo de un juevo innovador que cumplir los requisitos de un proyecto final de carrera. Mi objetivo siempre ha sido aprender para luego exponer mis conclusiones.

Creo que el objetivo principal de este proyecto está cumplido. Mi conclusión es que si alguien quiere hacer un juego en 3D en la Web este proyecto es un buen sitio para empezar a leer porque explica todo el proceso de desarrollo en cuanto a técnicas de implementación, diseño y optimización con un estudio amplio de mercado. Contiene desde cómo estructurar un juego grande, hasta como optimizar los Shaders de pintado, cargango modelos estáticos, dinámicos con técnicas innovadoras de Skinning por GPU, iluminación dinámica y demás.

Esto no acaba aquí, voy a dedicarle más tiempo en el futuro porque después de un desarrollo de este nivel dejar aparcado esto sería una tontería. Hay mucho futuro en lo expuesto en este proyecto. Han surgido hasta opciones profesionales derivadas de este desarrollo cosa que justifica el interés en general del momento por esta tecnología, el nivel de la aplicación y mis ganas de seguir aprendiendo más.

Apéndice A

Modelos Wavefront

A.1. Introducción

Los modelos Wavefront (archivos .obj) es un formato abierto de modelos 3D. Este formato es muy usado para representar modelos que contengan una geometría, materiales, coordenadas de texturas y texturas. Son buenos para representar objetos estáticos tales como elementos decorativos. Pueden tener un alto grado de detalle y son capaces de interactuar con la luz gracias a los materiales. A continuación se exponen unos ejemplos de modelos Wavefront:

Imagen A.1: Modelos Wavefront.

A.2. Especificación

Los modelos Wavefront tienen un especificación muy sencilla:

```
1  //Lista de vértices, con (x,y,z[,w]) coordenadas, w es opcional y  1.0 por
     defecto.
2  v 0.123 0.234 0.345 1.0
3  v ...
4  ...
5
6  //Coordenadas de textura, con (u[,v][,w]) coordenadas, v y w son opciones y 0.0
      por defecto.
7  vt 0.500 -1.352 [0.234]
8  vt ...
9  ...
10
11 //Normales en forma (x,y,z).
12 vn 0.707 0.000 0.707
13 vn ...
14 ...
15
16 //Definiciones de Cara
17 f 1 2 3      // f v1 v2 v3 v4
18 f 3/1 4/2 5/3   // f v1/vt1 v2/vt2 v3/vt3
19 f 6/4/1 3/5/3 7/6/5   // v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3
20 f ...
21 ...
22
23 // Usar una Material
24 usemtl [nombre material]
25
26 //Definir material llamado 'Texturizado'
27  newmtl Texturizado
28  Ka 1.000 1.000 1.000        // Color ambiente RGB
29  Kd 1.000 1.000 1.000        // Color difuso RGB
30  Ks 0.000 0.000 0.000        // Color especular RGB
31  d 1.0                       // transparencia
32  illum 2                     // tipo de iluminación
33  map_Ka lenna.tga            // mapa textura ambiente
34  map_Kd lenna.tga            // mapa textura difusa
35  map_Ks lenna.tga            // mapa textura especular
36  map_Ns lenna_spec.tga       // comoponente especular
37  map_d lenna_alpha.tga       // mapa texturas alpha
```

Código Fuente A.1: Especificación Modelos Wavefront

A.3. Importación

El objetivo de usar estos modelos es dotar a la escena de elementos decorativos. Por este motivo no hemos soportado todas las opciones que este formato nos proporciona. Se ha soportado toda la especificación de caras y en materiales solo los 3 colores RGB y el mapa difuso con la restricción de usar un único material por objeto.

Los modelos Wavefront vienen en archivos con la extensión *.obj* en texto plano. Nuestro aplicación está hecha en Javascript. No sería muy óptimo en cuanto a rendimiento leer un obj directamente desde el cliente ya que se tardaría mucho descartando todo lo que no queremos. La carga de una aplicación Web tiene que ser lo más rápida posible para que el usuario no sienta que pierde el tiempo y más en un juego. Lo mejor es usar un Script intermedio que nos transforme ese .obj en un archivo JSON[1] con la sólo la información que queremos. Se ha usado un Script hecho en Python para esa transformación adjunto en el código fuente llamado:

`convert_obj_three.py`

Siempre que queramos importar un archivo obj a nuestra aplicación deberemos de ejectuar este Script, lo convertirá a un JSON y despues en la aplicación ejecutaremos el comando de leer JSON que deja directamente un objeto con todas sus variables como propiedades de un objeto.

El Script lo que hará es leer todas las caras del .obj y transformarlas en un Array de vértices apto para el pintado de WebGL en triángulos. Las normales no se usarán y se deducirán por el orden de los vértices en la lista de caras siendo siempre anti-horario. Para cada vértice (x,y,z) se creará una cordenada de textura (u,v) y una normal como acabo de explicar (x,y,z). Como se ha comentado el objeto solo podrá tener un material y por lo tanto solo una textura como mapa difuso. Es una restricción muy fuerte porque muchos objetos vienen con diferentes materiales y cada material con sus texturas. Es muy común ver en un modelo por ejemplo de un coche como las ruedas tienen su propio material. Los cristales otro y el chasis otro.

Para resolver todos los objetos que tienen más de un material he tenido que hacer alguna transformación manual con Blender[2] para unificar todos los materiales en uno. Primero juntando todas las mallas del objeto y segundo haciendo una proyección de color en un plano, generando así una única textura para poder usar un único material.

[1] Acrónimo de JavaScript Object Notation, es un formato ligero para el intercambio de datos.

[2] Es un programa informático multi-plataforma, dedicado especialmente al modelado, animación y creación de gráficos tridimensionales.

Se ha adjuntado un video ejemplo con el código fuente de la aplicación para importar cualquier modelo a nuestra aplicación usando la modificación de materiales con Blender, el uso de script y la generación de la especificación en un archivo de configuración para que la aplicación sepa qué tiene que leer:

`ModelosEstaticosAWebGLJuego.mp4`

Apéndice B

Modelos MD5

B.1. Introducción

Para representar personajes en un juego 3D de forma realística se suelen usar modelos 3D animados. Normalmente el proceso de creación es el siquiente: un artista pinta digitalmente un personaje en diferentes puntos de vista. A continuación un diseñador gráfico modela ese pintado en 3D, mallas, dotándole de animación para el movimiento el ataque, la recepción de daño, lo que esté especificado. Una vez el diseñador gráfico posee el modelo en 3D se reune con el equipo de desarrollo para decidir un formato de exportación/importación a la aplicación. En otras palabras tienen que ponerse de acuerdo según los intereses de cada equipo en cómo van a traducir ese modelo 3D a una estrucutura de datos. Una vez esa estructura de datos está decidida, está ya preparada para ser importada en el juego. Estas estructuras suelen ser complejas porque un modelo 3D no solo tiene una malla sino un esqueleto de animación (estructura de por sí ya compleja) , mapas de luces, animaciones (cómo ese esqueleto se mueve). Suelen ser archivos muy grandes y muy específicos de cada juego.

En este proyecto solo poseemos la última parte, el equipo de desarrollo, por eso nos hemos tenido que adaptar a un formato ya existente e importarlo a nuestra aplicación. Casi cada empresa de videojuegos tiene en cada versión de sus motores una versión diferente de modelos para adapatarse a los requerimientos tecnológicos y la especificación propia del juego. Cómo todo juego privativo los recursos no son accesibles o públicos tienen licencias privativas.

Pero alrededor de Noviembre de 2011 la empresa Id Software[1] publicó el motor del famoso juego Doom3 y Quake4 como código libre bajo la siguiente licencia GPL:

https://github.com/TTimo/doom3.gpl

Gracias a esta liberación de código y recursos hemos sido capaces de acceder a sus modelos animados, llamados modelos MD5. La siguiente imagen representa típicos modelos md5:

Imagen B.1: Modelos MD5.

[1]Empresa estadounidense de desarrollo de videojuegos famosa por el juego Doom 3. Link.

B.2. Especificación

La especificación MD5 es compleja. Haré un resumen de las partes más importantes. Primero de todo, decir que los modelos MD5 no tienen nada que ver con la función de hash criptográfica llamada también MD5.

Los modelos MD5 representan los modelos mediante tres ficheros ASCII:

- **.md5Mesh** : datos geométricos, la malla.
- **.md5Anim** : animaciones de la malla.
- **.mtr** : materiales que definen las texturas.

Las propiedades de los modelos MD5 son:

- Animación vía esqueletos.
- Skinning por vértices.
- Usa Cuaterniones para la orientación.

¿Qué es Animación con esqueletos y Skinning por vértices?

Un esqueleto es un conjunto de huesos, una técnica usada para crear animaciones. Una animación con esqueleto consiste en una malla de vértices con una asociación a los huesos. O sea mover un hueso causará mover los vértices asociados a él. A partir de ahora llamaremos a sus elementos con el nombre inglés para mayor precisión. Huesos - Bones, Esqueleto - Skeleton y Nudo - Joint. Por lo tanto este sistema envuelve los siguiente elementos:

- Una malla de vértices y cada vértice con un peso asociado.
- Un Skeleton de la malla compuesto por una jerarquía de Bones y Joints.

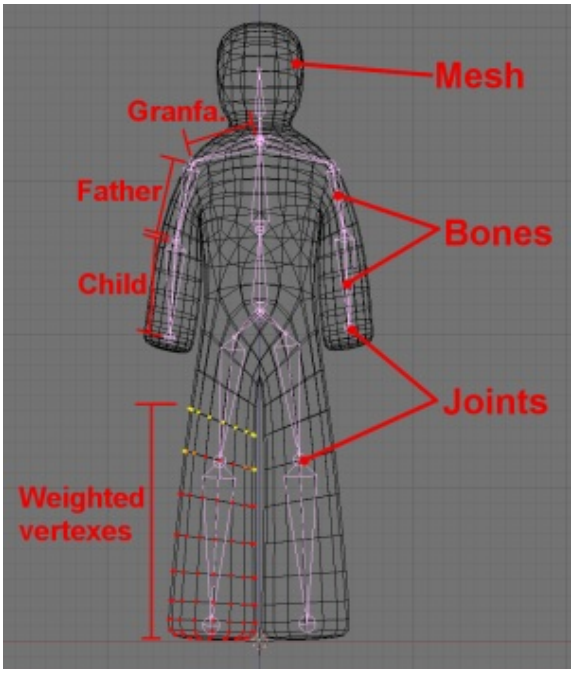


IMAGEN B.2: Representación de Skeleton y Skinning.

La malla esta conectada con el Skeleton. Cada vértice de la malla está asociado a un Bone o más, con un peso, que significa cuanto movimiento de ese Bone le afecta. Cada Bone del Skeleton está conectado a dos Joints y cada Joint está conectado al menos a un Bone. Para hacer una animación, se calculan los posiciones del Skeleton y mediante un algotirmo de interpolación de los pesos y Bones se calcula el resultado. El proceso de conectar una malla a un Bone se llama Skinning. Normalmente una vez tienes todo definido en el programa de diseño 3D (como Blender o Maya) puedes exportarlo al formato que tu quieras. Ahora hablaremos del formato MD5.

Fichero md5mesh

Lo primero que encontramos en un fichero md5mesh son algunos datos geométricos:

```
numJoints <int>
numMeshes <int>
```

NumJoints es el número de Joints del Skeleton de ese modelo y numMeshes es el número de mallas que contiene ese modelo ya que un modelo puede estar separados en diferentes mallas. Después de esto encontramos los Skeleton Joints en una posición básica.

```
joints {
    "name" parent ( pos.x pos.y pos.z ) ( orient.x orient.y orient.z )
    ...
}
```

name es el nombre del Joint. Parent es el índice del padre del Bone. Si es -1 es que no tiene padre y se le llama Root o raíz. Pos.x, pos.y y pos.z es la posición de ese Joint en el espacio y orient.x, orient.y y orient.z es la orientación del cuaternión de ese Joint.

Así repetidamente hasta completar todo el Skeleton. Ahora toca definir las malla de vértices, la Skin o piel:

```
mesh {
    shader "<string>"

    numverts <int>
    vert vertIndex ( s t ) startWeight countWeight
    vert ...

    numtris <int>
    tri triIndex vertIndex[0] vertIndex[1] vertIndex[2]
    tri ...

    numweights <int>
    weight weightIndex joint bias ( pos.x pos.y pos.z )
    weight ...
}
```

El primer String shader define el material que será usado para esta malla o Mesh. NumVerts defice el número total de vértices de la malla. Después de esta variable tenemos la lista de vértices. vertIndex es un entero especificando el índice de ese vértices. S y T son Floats representando las coordenadas de textutra, también llamadas coordenadas UV. En el formato MD5, un vértice no tiene una posición. Su posición es computada por los pesos. CountWeight es un entero que define el número de pesos, empezando por startWeight que es un índice.

Numtris es el número de triángulos de la malla. triIndex es un entero que define el índice del triángulo. Y cada triángulo está definido por 3 índices de vértices: vertIndex[0], vertIndex[1] y vertIndex[2].

NumWeights es el numero de pesos de la mesh. WeightIndex es un entero que define el índice del pesp. Joint es el joint del cual depende. Bias es un float que representa el factor entre este rango: de]0,0 a 1,0] el cual define la cantidad de contrinución de ese peso en el cálculo de la posición del vértice. Pos.x, pos.y and pos.z son las posiciones del peso en el espacio.

Para calcular las posiciones de los vértices hay que calcularlas a partir de los pesos. Cada vértice tiene uno o más pesos, cada uno de ellos dependiente de un Joint y un factor que nos dice cuanto afecta a ese vértice. La suma de todos los factores de ese

vértice debería de ser 1.0. Esta técnica es puramente lo que hemos comentado como Skinning. Para calcular la posición de un vertice en el espacio se hace así:

```
finalPos = (weight[0].pos * weight[0].bias) +
...
+ (weight[N].pos * weight[N].bias);
```

Para computar todos los vértices lo que tenemos que hacer es recoorer todos los vértices en su posición de espacio según sus pesos:

```
/* Recoger los Vértices */
for (i = 0; i < mesh->num_verts; ++i)
  {
    vec3_t finalVertex = { 0.0f, 0.0f, 0.0f };

    /* Calcular la posicion final del vértice final con los pesos */
    for (j = 0; j < mesh->vertices[i].count; ++j)
      {
        const struct md5_weight_t *weight =
         &mesh->weights[mesh->vertices[i].start + j];
        const struct md5_joint_t *joint = &joints[weight->joint];

        /* Calcular la transormación según el peso */
        vec3_t wv;
        Quat_rotatePoint (joint->orient, weight->pos, wv);

        /* la suma de todos los weight->bias debería de ser 1.0 */
        finalVertex[0] += (joint->pos[0] + wv[0]) * weight->bias;
        finalVertex[1] += (joint->pos[1] + wv[1]) * weight->bias;
        finalVertex[2] += (joint->pos[2] + wv[2]) * weight->bias;
      }

    ...
  }
```

Fichero md5Anim Este fichero contiene las animaciones del Skeleton de los modelos md5mesh de esta forma:

- Una jerarquía de Skeleton con Flags para cada Joint de los datos de la animación.
- Una Bounding Box para cada frame de la animación.
- Un Frame Base desde donde la animación comienza.
- Una lista de frames, cada uno conteniendo los datos para computar el Skeleton desde su Frame Base.

Lo primero que econtramos en el fichero es:

```
numFrames <int>
numJoints <int>
frameRate <int>
numAnimatedComponents <int>
```

NumFrames es el número de frames de la animación. Una animación está compuesta por varios frames, en cada una una posición del Skeleton. Reproduciendo una detrás de otra conseguimos una animación. NumJoitns es el número de Joints del Skeleton Frame. Tiene que ser el mismo que el número de Joints del md5mesh. FrameRate es numero de Frames por segundo de pintado de la animación. NumAnimatedComponents es el número de parámetros por frame usados para computar el Frame Skeleton.

Después de esta cabecera viene la jerarquía del Skeleton. Proporciona la información para construir los Frames del Skeleton desde el Frame Base.

```
hierarchy {
    "name"   parent flags startIndex
    ...
}
```

Name es el nombre del Joint. Parent es el índice del Joint padre. Si es -1 es que no tiene y startIndex es el índice inicial desde donde empezar a computar el Frame Skeleton.

Después de la jerarquía del Skeleton vienen las Bounding Boxes de cada frame de esta forma:

```
bounds {
    ( min.x min.y min.z ) ( max.x max.y max.z )
    ...
}
```

Representan una caja mediante dos puntos en el espacio. Las coordenadas están en sistema de coordenadas de modelo y son útiles para computar colisiones. Después de las Bounding Boxes vienen los datos del Frame Base. Contiene las posiciones y orientaciones (cuaterniones) de cada Joint desde donde el Skeleton Frame será construido. Para cada Joint:

```
baseframe {
    ( pos.x pos.y pos.z ) ( orient.x orient.y orient.z )
    ...
}
```

No se expone el código para generar una animación porque es muy largo. Se dispone de muchas formas de hacerlo si buscamos online. Hay un claro ejemplo en esta web:

http://devmaster.net/posts/loading-and-animating-md5-models-with-opengl

B.3. Importación

Cuando surgieron los modelos MD5 como código libre mucha gente se dedicó a probarlos en sus aplicaciones y también en WebGL. Hay una prueba de carga de modelos md5Mesh con animaciones md5Anims en WebGL. Hemos usado el mismo Script para cargar nuestros modelos. El código fuente original de este código es:

http://media.tojicode.com/webgl-samples/md5Mesh.html

Este código, creado por un tercero, lo que hace es exactamente leer y generar las animaciones y mallas con pesos del modelo en Javascript. Pero su rendimiento es bueno solo para un modelo. Si instanciamos cuatro o cinco modelos el rendimiento cae en picado hacia 10 FPS por modelo. O sea hace inviable que este código pueda ser usado en una aplicación que necesite varios modelos como la nuestra.

Haciendo instrumentalización del código hubo una función que destacaba en tiempo de ejecución, el Skinning. El proceso de asociar cada vértice a una posición según sus pesos. Como hemos visto en el código anteior es un bucle con muchas iteraciones e internamente con muchas operaciones. ¿Cómo lo solucionamos? Tenía dos posibilades para probar una es pasarlo el Skinning a un WebWorker y que se hiciera en un hilo aparte o aprovechar la GPU y en el Vertex Shader hacer Skinning por Vértice. Decidí probar hacer Skinning por GPU y funcionó.

B.4. Skinning GPU

Este desarrollo ha proporcionado la posibilidad de instanciar varios modelos MD5 a partir de una modificación de este código:

http://media.tojicode.com/webgl-samples/md5Mesh.html.

La modificación ha sido pasar el Skinning a la GPU. Esto quiere decir que a la hora de renderizar el modelo no vamos a enviarle la posición real de los vértices sino que vamos a enviarle la información de los vértices, los pesos y las Joints para que internamente haga el cálculo de posición en la GPU.

Empecemos por el código del Script original externo. Para hacer Skinnig se hace lo mismo que lo expuesto en el código anterior pero en Javascript.

```
for(var i = 0; i < meshes.length; ++i) {
    var mesh = meshes[i];
    var meshOffset = mesh.vertOffset + arrayOffset;

    // Calculate transformed vertices in the bind pose
    for(var j = 0; j < mesh.verts.length; ++j) {
        var vertOffset = (j * VERTEX_ELEMENTS) + meshOffset;
        var vert = mesh.verts[j];

        vx = 0;
        vy = 0;
        vz = 0;
        nx = 0;
        ny = 0;
        nz = 0;
        tx = 0;
        ty = 0;
        tz = 0;

        vert.pos = [0, 0, 0];

        for (var k = 0; k < vert.weight.count; ++k) {
            var weight = mesh.weights[vert.weight.index + k];
            var joint = joints[weight.joint];

            // Rotate position
            quat4.multiplyVec3(joint.orient, weight.pos, rotatedPos);

            // Translate position
            vx += (joint.pos[0] + rotatedPos[0]) * weight.bias;
            vy += (joint.pos[1] + rotatedPos[1]) * weight.bias;
            vz += (joint.pos[2] + rotatedPos[2]) * weight.bias;

            // Rotate Normal
            quat4.multiplyVec3(joint.orient, weight.normal, rotatedPos);
            nx += rotatedPos[0] * weight.bias;
            ny += rotatedPos[1] * weight.bias;
            nz += rotatedPos[2] * weight.bias;

            // Rotate Tangent
            quat4.multiplyVec3(joint.orient, weight.tangent, rotatedPos);
            tx += rotatedPos[0] * weight.bias;
            ty += rotatedPos[1] * weight.bias;
            tz += rotatedPos[2] * weight.bias;
        }

        // Position
        vertArray[vertOffset] = vx;
        vertArray[vertOffset+1] = vy;
        vertArray[vertOffset+2] = vz;
```

```
51
52          // TexCoord
53          vertArray[vertOffset+3] = vert.texCoord[0];
54          vertArray[vertOffset+4] = vert.texCoord[1];
55
56          // Normal
57          vertArray[vertOffset+5] = nx;
58          vertArray[vertOffset+6] = ny;
59          vertArray[vertOffset+7] = nz;
60
61          // Tangent
62          vertArray[vertOffset+8] = tx;
63          vertArray[vertOffset+9] = ty;
64          vertArray[vertOffset+10] = tz;
65      }
66  }
```

Código Fuente B.1: Skinning por CPU

Como hemos dicho este código es claramente costoso. Por cada vértice y por cada peso hay que hacer muchos cálculos cosa en que la GPU es buena en paralelizar pequeños trozos de lógica y más si son matemáticas. En vez de hacer un bucle en CPU usemos el paralelismo de la GPU para conseguir hacer el Skinning sin bucles. Ahora voy a exponer el Vertex Shader para hacer GPU Skinning, este código es de mi propia autoría y ha proporcionado que podamos usar modelos dinámicos animados vía Skinning en GPU. O sea nos hemos aprovechado del Script de carga del código externo y la parte de Skinning que era la más costosa la he pasado a GPU mejorando el rendimiento.

```
1   uniform mat4 projectionMat;
2   uniform mat4 modelViewMat;
3   uniform mat4 jointsOrientation[28];
4   uniform mat4 jointsPos[28];
5
6   attribute vec4 weights;
7   attribute vec4 weightsBias;
8   attribute mat4 weightPos;
9   attribute mat4 weightNormal;
10  attribute mat4 weightTangent;
11
12  /*Funcion para multiplicar un cuaternión por un vector*/
13  void quatMulVec3(in vec4 quat,in vec3 vec,inout vec3 dest) {
14      float x = vec[0], y = vec[1], z = vec[2];
15      float qx = quat[0], qy = quat[1], qz = quat[2], qw = quat[3];
16
17
18      // calculate quat * vec
19      float ix = qw * x + qy * z - qz * y;
20      float iy = (qw * y) + (qz * x) - (qx * z);
21      float iz = qw * z + qx * y - qy * x;
22      float iw = (-1.0)*qx * x - qy * y - qz * z;
```

```
23
24      dest[0] = ix * qw + iw * (-1.0)* qx + iy * (-1.0)*qz - iz * (-1.0)* qy;
25      dest[1] = iy * qw + iw * (-1.0)*qy + iz * (-1.0)*qx - ix * (-1.0)*qz;
26      dest[2] = iz * qw + iw * (-1.0)*qz + ix * (-1.0)*qy - iy * (-1.0)*qx;
27
28  }
29
30  /* Extraer la orientación según el índice de los Atributos */
31  void fjointOr(in int idx, inout vec4 jointOr)
32  {
33      int mat = int((float(idx)+0.1)/4.0);
34      int submat = int(float((idx - mat*4))+0.1);
35
36      mat4 jointMatOr = jointsOrientation[mat];
37
38      if(submat == 0)
39      {
40          jointOr = vec4(jointMatOr[0][0],jointMatOr[0][1],jointMatOr[0][2],
        jointMatOr[0][3]);
41      }
42      else if(submat == 1)
43      {
44          jointOr = vec4(jointMatOr[1][0],jointMatOr[1][1],jointMatOr[1][2],
        jointMatOr[1][3]);
45      }
46      else if(submat == 2)
47      {
48          jointOr = vec4(jointMatOr[2][0],jointMatOr[2][1],jointMatOr[2][2],
        jointMatOr[2][3]);
49      }
50      else if(submat == 3)
51      {
52          jointOr = vec4(jointMatOr[3][0],jointMatOr[3][1],jointMatOr[3][2],
        jointMatOr[3][3]);
53      }
54  }
55  /* Extraer la posicion según el índice de los Atributos */
56  void fjointPos(in int idx, inout vec4 jointPos)
57  {
58      int mata = int((float(idx)+0.1)/4.0);
59      int submata = int(float((idx - mata*4))+0.1);
60
61      mat4 jointMatPos = jointsPos[mata];
62
63      if(submata == 0)
64      {
65          jointPos = vec4(jointMatPos[0][0],jointMatPos[0][1],jointMatPos[0][2],
        jointMatPos[0][3]);
66      }
67      else if(submata == 1)
68      {
69          jointPos = vec4(jointMatPos[1][0],jointMatPos[1][1],jointMatPos[1][2],
        jointMatPos[1][3]);
70      }
71      else if(submata == 2)
```

```
72      {
73          jointPos = vec4(jointMatPos[2][0],jointMatPos[2][1],jointMatPos[2][2],
    jointMatPos[2][3]);
74      }
75      else if(submata == 3)
76      {
77          jointPos = vec4(jointMatPos[3][0],jointMatPos[3][1],jointMatPos[3][2],
    jointMatPos[3][3]);
78      }
79
80  }
81
82  /*Misma función que en CPU pero sólo con un bucle, el de los pesos por vértice*/
83  void skin(inout vec3 p_pos, inout vec3 p_normal, inout vec3 p_tangent)
84  {
85      // Variables Temporales
86      int j=0,idx = 0;
87      float weightJoint = -1.0,weightJointBias=-1.0;
88      debug = vec4(0.0,0.0,1.0,1.0);
89      p_pos = vec3(0.0,0.0,0.0);
90      p_normal = vec3(0.0,0.0,0.0);
91      p_tangent = vec3(0.0,0.0,0.0);
92      vec3 rotatedPos = vec3(0.0,0.0,0.0);
93      vec4 jointOr = vec4(0.0,0.0,0.0,0.0);
94      vec4 jointPos = vec4(0.0,0.0,0.0,0.0);
95      vec3 wPos = vec3(0.0,0.0,0.0);
96      vec3 wNormal = vec3(0.0,0.0,0.0);
97      vec3 wTangent = vec3(0.0,0.0,0.0);
98      int a = 0;
99      // Un máximo de 4 pesos por vértice
100     for(int i = 0; i < 4; i++)
101     {
102             weightJoint = float(weights[i]);
103             weightJointBias  = weightsBias[i];
104
105             if(weightJoint > -0.5)
106             {
107                 fjointOr(int(weightJoint+0.001),jointOr);
108                 fjointPos(int(weightJoint+0.001),jointPos);
109
110                 wPos = vec3(weightPos[i][0],weightPos[i][1],weightPos[i][2]);
111                 quatMulVec3(jointOr,wPos,rotatedPos);
112                 p_pos[0]+=(jointPos[0]+rotatedPos[0])*weightJointBias;
113                 p_pos[1]+=(jointPos[1]+rotatedPos[1])*weightJointBias;
114                 p_pos[2]+=(jointPos[2]+rotatedPos[2])*weightJointBias;
115
116                 wNormal = vec3(weightNormal[i][0],weightNormal[i][1],weightNormal
    [i][2]);
117                 quatMulVec3(jointOr,wNormal,rotatedPos);
118                 p_normal[0]+=(rotatedPos[0])*weightJointBias;
119                 p_normal[1]+=(rotatedPos[1])*weightJointBias;
120                 p_normal[2]+=(rotatedPos[2])*weightJointBias;
121
122
123
```

```
124
125                 wTangent= vec3(weightTangent[i][0],weightTangent[i][1],
    weightTangent[i][2]);
126                 quatMulVec3(jointOr,wTangent,rotatedPos);
127                 p_tangent[0]+=(rotatedPos[0])*weightJointBias;
128                 p_tangent[1]+=(rotatedPos[1])*weightJointBias;
129                 p_tangent[2]+=(rotatedPos[2])*weightJointBias;
130
131             }
132     }
133 }
134
135
136 void main(void) {
137     // Declarar los valores de salida e inicializarlos a 0.
138     vec3 _s_pos = vec3(0.0,0.0,0.0);
139     vec3 _s_normal = vec3(0.0,0.0,0.0);
140     vec3 _s_tangent = vec3(0.0,0.0,0.0);
141
142     // Hacer Skinning y conseuir los valores de pos, normal y tangente
143     skin(_s_pos,_s_normal,_s_tangent);
144
145     // Multiplicar por la model*view*proyeccion para conseguir el
146     // punto definitivo.
147     gl_Position = projectionMat * modelViewMat * vec4(_s_pos, 1.0);;
148 }
```

Código Fuente B.2: Skinning por GPU

Con este código hacemos lo mismo que en CPU pero de una forma totalmente paralela gracias al propósito de la GPU y de forma eficiente. Toda la información que no cambia de la malla de vértices es enviada como atributos por vértice, como son los pesos , su contribución, su posición, normal y tangente. Estos valores siempre son los mismos para cada vértice, por eso se puede enviar en forma de atributo, en un Buffer Estático que no cambie nunca:

```
attribute vec4 weights;
attribute vec4 weightsBias;
attribute mat4 weightPos;
attribute mat4 weightNormal;
attribute mat4 weightTangent;
```

En cambio, según el estado de la animación tenemos que enviar en cada frame el Skeleton entero, sus posiciones y orientaciones. En cada frame estos valores cambian ya que la animación va sucediendo y el Skeleton, por lo tanto, se mueve. La computación del Skeleton por frame está hecha en CPU porque no es costosa pero es enviada a GPU para hacer el Skinning en forma de uniforms:

```
uniform mat4 jointsOrientation[28];
uniform mat4 jointsPos[28];
```

Las restricciones que podemos ver en estas dos declaraciones de atributos y uniforms son los inconvenientes de pasarlo a GPU, que tienen un límite de valores. En el formato de modelos de MD5 cada vértice puede tener un número ilimitado de pesos pero hay una restriccion en el número de atributos que puedes pasar a un Vertex Shader. En este caso el límite es 4. O sea, he limitado que el número de pesos por vértice sean 4 porque caben en un vec4 y las respectivas posiciones para cada peso caben en una $mat4 = vec4 * 4$. Para todos aquellos vértices que tenían más de 4 pesos se han descartado todos aquellos pesos que contenían menos influencia, menos Bias, su factor de contribución.

En conclusión, se ha partido la lógica de los modelos en 2. Una parte, es el cálculo del Skeleton Frame que se hace por CPU leyendo el md5Anim y calculando el Skeleton resultante. Y otra, el Skinning que se ha pasado a GPU. De esta forma en el bucle de render desaparece el coste de hacer Skinning y nos da la posibilidad de instanciar más modelos, casi sin coste. El resultado de ello se puede comprobar en la aplicación.

Bibliografía

[1] Mat Buckland, *Programming game ai by example*, Programming Game AI by Example provides a comprehensive and practical introduction to the bread and butter AI techniques used by the game development industry (2004).

[2] Douglas Crockford, *Javascript: The good parts*, Considered the JavaScript expert by many people in the development community (2008).

[3] Randima Fernando, *Gpu gems: Programming techniques, tips and tricks for real-time graphics*, GPU Gems is a compilation of articles covering practical real-time graphics techniques arising from the research and practice of cutting edge developers (2004).

[4] David Flanagan, *Javascript: The definitive guide: Activate your web pages (definitive guides)*, The Definitive Guide has been the bible for JavaScript programmers (2011).

[5] Jason Gregory, *Game engine architecture*, This book covers both the theory and practice of game engine software development (2010).

[6] Eric Lengyel, *Mathematics for 3d game programming and computer graphics, third edition*, Mathematical concepts that a game developer needs to develop 3D computer graphics and game engines at the professional level (2011).

[7] Randima Fernando y Tim Sweeney Matt Pharr, *Gpu gems 2: Programming techniques for high-performance graphics and general-purpose computation*, GPU Gems is a compilation of articles covering practical real-time graphics techniques arising from the research and practice of cutting edge developers (2005).

[8] Aaftab Munshi, *Opengl® es 2.0 programming guide*, Provide start to finish guidance for maximizing the interfaces value in a wide range of high-performance applications (2008).

[9] Nehe, Rigid body physics in Javascript, http://nehe.gamedev.net/.

[10] Hubert Nguyen, *Gpu gems 3*, GPU Gems is a compilation of articles covering practical real-time graphics techniques arising from the research and practice of cutting edge developers (2007).

[11] Hugh Malan y Mike Weiblen Randi J. Rost, *Opengl shading language (3rd edition)*, Guide to writing shaders (2009).

[12] Jim Sangwine, Rigid body physics in Javascript, http://www.jiglibjs.org/.

[13] Dave Shreiner, *Opengl programming guide: The official guide to learning opengl, versions 3.0 and 3.1*, Provides definitive and comprehensive information on OpenGL and the OpenGL Utility Library (2009).

[14] Giles Thomas, Creation of lessons as a way of teaching WebGL, http://learningwebgl.com/.

[15] Eric Haines y Naty Hoffman Tomas Akenine-Moller, *Real-time rendering, third edition*, Modern techniques used to generate synthetic three-dimensional images in a fraction of a second (2008).