



Escuela
Politécnica
Superior

Demo técnica para PC



Grado en Ingeniería Multimedia

Trabajo Fin de Grado

Autor:

Luis González Aracil

Tutor/es:

Francisco José Gallego Durán

Junio 2019

Demo técnica para PC

La demoscene

Autor

Luis González Aracil

Tutor/es

Francisco José Gallego Durán

Ciencia de la Computación e Inteligencia Artificial



Grado en Ingeniería Multimedia



Escuela
Politécnica
Superior



Universitat d'Alacant
Universidad de Alicante

Resumen

Para poder usar una herramienta en toda su capacidad y funcionalidad, es necesario cono-
cerla a fondo. Con la complejidad creciente de los ordenadores, está apareciendo una tendencia
por despreocuparse de los detalles del bajo nivel, considerando que los detalles de implemen-
tación no son o deberían ser parte del problema.

Este trabajo es, en primer lugar, una reivindicación de la importancia del bajo nivel. A lo
largo del mismo, se tratará la subcultura informática de la *demoscene*.

El *demoscening* se originó a primeros de los años 80, con el auge de los primeros ordena-
dores personales, y su principal objetivo era el de crear demostraciones técnicas de gráficos
y sonido por computador en tiempo real, buscando siempre resultados sorprendentes que
consiguieran sobrepasar las limitaciones técnicas de las máquinas de la época. Eran por tanto
demostraciones artísticas de ingenio y de conocimiento del bajo nivel.

A lo largo de este trabajo se explorarán algunas de las técnicas gráficas más comúnmente
usadas en el mundo de la *demoscene*, revisadas desde la actualidad. Se hará un análisis previo
de cada una de estas técnicas para a continuación proceder a su implementación y posterior
optimización. Además, para concluir, se hará una propuesta de una demo aplicando todos los
conocimientos explorados previamente.

Abstract

In order to be able to use a tool at its maximum capability and functionality, it's important to have an in-depth knowledge of it. Given the always increasing complexity of computers, there's a trend in software development to look away from low level implementation details, considering that these details should not be part of the problem's solution.

This essay pretends to emphasise the importance of the low level. Throughout this document, I will be writing about a computer's subculture called *demoscene*.

The *demoscene* originated back in the 80's, when personal computers' popularity was in a crest. *Demosceners*'s main goal was to create technical graphical demos in real time. These demos always tried to show off the abilities of the programmers, who tried to overcome the technical limitations of the machines at the time, with surprising results.

This document will explore and revise some of the graphical techniques that were commonly used by *demosceners*. Before the implementation of every technique, an in-depth analysis will take place. After it, the demo will be revised and optimized if possible. Furthermore, there will be a last proposal consisting on a final demo gathering all the knowledge previously exposed.

Motivación y objetivo general

Hablando con un compañero del trabajo sobre el lenguaje ensamblador, yo estaba intentando argumentarle su utilidad, a lo que él me contestó *"Hoy en día, saber ensamblador es como saber latín"*. Su afirmación zanjó el tema, pues a partir de ese momento no tuve ganas de seguir discutiendo, pero me hizo reflexionar. ¿Es inútil el ensamblador? ¿Sirve para algo el bajo nivel?

Para mí, la pregunta *"¿Para qué sirve saber ensamblador?"* es perfectamente equiparable a la pregunta *"¿Para qué le sirve a un arquitecto conocer las herramientas y materiales con los que va a construir una casa?"*.

Si una edificación cayera por una mala elección del material de los cimientos por parte del arquitecto, no habría duda en a quién culpar. Nadie abogaría que la culpa no es del arquitecto porque no es su responsabilidad conocer las bases. Sin embargo, hoy en día hay una enorme tendencia en el mundo del desarrollo software por menospreciar o infravalorar los cimientos de la programación, considerándolo algo arcaico y de carácter puramente didáctico, pero no práctico.

Yo me opongo radicalmente a esta visión, no sólo porque estoy convencido de la importancia de conocer el bajo nivel, si no que también encuentro cierta belleza en él. Cómo instrucciones en apariencia tan simples pueden construir sistemas tan complejos. A ello, se suma una gran curiosidad por saber cómo las cosas están hechas, desde el principio.

Una de las cosas que encuentro más apasionantes de la computación es la capacidad de los ordenadores, máquinas inertes y carentes de inteligencia real -por el momento- para reproducir nuestra realidad a partir de modelos matemáticos.

Los gráficos por computador son, por lo general, complejos. Sin embargo, hoy en día es posible crear con un ordenador imágenes que parecen fotografías y son capaces de engañar al ojo humano.

El objetivo principal de este trabajo es ir a las raíces, y revisar algunas de las técnicas que se usaban en los orígenes de los gráficos por computador para, a partir de operaciones con bajo coste computacional, generar escenas complejas.

A mis padres, por estar ahí, siempre.

A mi hermana, por ser mi recordio y mi alegría.

A mi familia y amigos, por apoyarme y alegrarme los días.

A mi tutor,

*por la visión que me ha dado sobre el mundo de la programación
y que tan valiosa es.*

A Lola, por haber marcado la dirección cuando estábamos perdidos

*If you give people
the choice of writing
good code or fast code,
there's something wrong.
Good code should be fast*

Bjarne Stroustrup

*When the whole world is silent,
even one voice becomes powerful.*

Malala Yousafzai

Índice general

1	Introducción	1
2	Estado del arte	5
2.1	La demoscene	5
2.1.1	Qué es la demoscene	5
2.1.2	Orígenes de la demoscene	6
2.1.3	Composición y cultura de la demoscene	6
2.1.4	La demoscene en la actualidad	7
2.2	Eventos de demoscening	8
2.3	Grupos de demoscening	9
2.3.1	Farbrausch	9
2.3.2	Future Crew	11
2.3.3	PoPsY TeAm	12
2.3.4	Equinox	13
2.3.5	Fairlight	13
2.3.6	RGBA	14
2.3.7	Batman Group	15
2.4	Portales de demoscening	16
2.5	Demos destacables	17
2.6	Efectos gráficos más comunes	18
2.7	Influencia de la demoscene en la industria	21
3	Objetivos	23
4	Metodología	25
4.1	Software	25
4.2	Tests de rendimiento	26
4.3	Entorno: motor gráfico	26
4.4	Las demos	27
4.4.1	Búsqueda de información	27
4.4.2	Planteamiento formal	27
4.4.3	Implementación	27
4.4.4	Refinamiento	27
5	Tests de rendimiento	29
5.1	Planteamiento inicial	29
5.2	Implementación	30
5.3	Resultados	31
5.4	Ánalisis de los resultados	34

5.5 Conclusiones	37
5.6 Posibles mejoras	37
6 El motor gráfico	39
6.1 Investigación inicial	39
6.2 Características	41
6.2.1 La textura de dibujado y el píxel	43
6.2.2 Detectar input	45
6.2.3 Dibujar texto	46
6.2.4 Dibujar puntos	49
6.2.5 Dibujar rectángulos	50
6.2.6 Dibujar líneas	51
7 Efectos clásicos	53
7.1 Fuego	53
7.1.1 Investigación inicial	53
7.1.2 Planteamiento formal	53
7.1.3 Implementación	54
7.1.4 Refinamiento	57
7.1.5 Resultado	58
7.2 Túnel de puntos	59
7.2.1 Investigación inicial	59
7.2.2 Planteamiento formal	60
7.2.3 Implementación	60
7.2.4 Refinamiento	67
7.2.5 Resultado	70
7.3 RotoZoom	70
7.3.1 Investigación inicial	70
7.3.2 Planteamiento formal	71
7.3.3 Implementación	72
7.3.4 Refinamiento	75
7.3.5 Resultado	76
7.4 Deformaciones de imagen	76
7.4.1 Investigación inicial	76
7.4.2 Planteamiento formal	77
7.4.3 Implementación	78
7.4.4 Resultado	79
7.5 Plasma	81
7.5.1 Investigación inicial	81
7.5.2 Planteamiento formal	81
7.5.3 Implementación	82
7.5.4 Resultado	83
7.6 Planos infinitos	84
7.6.1 Investigación inicial	84
7.6.2 Planteamiento formal	85
7.6.3 Implementación	86

7.6.4	Refinamiento	88
7.6.5	Resultado	89
7.7	Geometría	90
7.7.1	Investigación inicial	90
7.7.2	Planteamiento formal	90
7.7.3	Implementación	92
7.7.4	Refinamiento	95
7.7.5	Resultado	97
8	Demo final	99
8.1	Introducción	99
8.2	Planteamiento inicial	99
8.3	Generar sonido	100
8.4	Crear la demo	112
8.5	Conclusiones de la demo	128
9	Conclusiones	129

Índice de figuras

2.1 Assembly 2004 - Fuente: Wikipedia	10
2.2 Farbrausch 41: Debris - Fuente: YouTube	10
2.3 Videojuego de 96kB: .kkrieger - Fuente: YouTube	11
2.4 Farbrausch 63: Magellan - Fuente: YouTube	11
2.5 Second Reality - Fuente: YouTube - En esta captura se puede ver un efecto de reflexión en dos esferas en tiempo real, mediante <i>raytracing</i>	12
2.6 VIP2 - Fuente: YouTube	13
2.8 Dead Ringer (por FairLight) - Fuente: YouTube - Demo 64k ganadora de Assembly 2006	14
2.9 Elevated - Fuente: YouTube - Intro 4K ganadora en Breakpoint 2009	15
2.10 Batman Forever - Fuente: YouTube	15
2.11 Heaven Seven (por Exceed) - Fuente: YouTube	18
2.12 State of the Art (por Spaceballs) - Fuente: YouTube	19
6.1 Diagrama simplificado de la estructura del motor gráfico	41
6.2 Estructura de un píxel en el motor gráfico	44
6.3 Funciones y estructuras del sistema de input	46
6.4 Pintado de líneas en pantalla (adaptación de una línea a una cuadrícula) - Fuente: Wikipedia	51
7.1 Matriz de convolución para generar efecto de fuego	54
7.3 Degradado en 32 celdas dadas 5 marcas de color	55
7.4 Fuego estático, usando el algoritmo en [7.3]	56
7.5 Fuego dinámico, con intensidad por píxel aleatorizada	58
7.8 Estructura básica de un círculo	61
7.9 Túnel de puntos básico	64
7.10 Órbitas de los planetas vistas desde la Tierra (por Giovanni Cassini) - Fuente: Wikipedia	65
7.11 Estructura de cada órbita que determina el camino	66
7.12 Tunel básico siguiendo un camino	67
7.13 Túnel final	71
7.14 Efecto de RotoZoom - Second Reality (by Future Crew) - Fuente: YouTube	71
7.15 Escalado, rotación y traslación en el espacio 2D	72
7.16 Diagrama del sistema para cargar imágenes	73
7.17 Representación de los píxeles para una imagen 5x4 en BMP	75
7.18 RotoZoom en distintos estados	76
7.19 Representación gráfica de la onda y su ecuación	77
7.21 Distintos efectos de deformación a partir de ondas	80

7.22 Cada una de las ondas que son sumadas (con su correspondencia a línea de código)	82
7.23 Efecto de plasma	83
7.25 Modo 7 (efecto de planos infinitos) en la SNES - Fuente: Wikipedia (por Anomie)	84
7.26 Nuestro acercamiento a los planos infinitos	85
7.27 Posición relativa de la textura antes y después de ser transformada	87
7.28 A la izquierda el punto de fuga está en el 0 de las coordenadas horizontales, a la derecha está en el centro	87
7.29 De izquierda a derecha, de mayor a menor campo de visión	88
7.30 Resultado final, con efectos de niebla y relieve aplicados	89
7.31 Cubo 3D que se proyecta contra el plano (proyección paralela)	92
7.32 Estructura básica de Point3D y Object3D	93
7.33 Cubo 3D con proyección paralela	96
7.34 Distintos modelos en perspectiva	97
7.35 Pirámide rotada y escalada	97
 8.1 Onda sinusoidal y su discretización	100
8.2 Estructura de una nota y su envolvente	105
8.3 Envolvente de un sonido - Fuente: Wikipedia, por Abdull	106
8.4 Distintas formas de onda - Fuente: Wikipedia, por Omegatron	109
8.5 Filtros de pasa baja y de pasa alta	111
8.6 Inicio de la demo	114
8.7 Generación de una malla de vértices	116
8.8 La malla de vértices ondulada y coloreada	117
8.9 Esfera formada por anillos	118
8.10 Relación entre una malla de puntos y una esfera	119
8.11 Hallar r a partir de R y h	121
8.12 Esfera a partir del plano	122
8.13 Distintas capturas del efecto de plasma + deformaciones + RotoZoom	124
8.14 Distintas capturas del efecto de planos infinitos	126
8.15 Distintas capturas del efecto de túnel de puntos	128

Índice de tablas

5.1	Operaciones con enteros de 32 bits	32
5.2	Operaciones con enteros de 64 bits	33
5.3	Operaciones en coma flotante con 32 bits	33
5.4	Operaciones en coma flotante con 64 bits	33
5.5	Llamadas a funciones matemáticas	34
5.6	Accesos a memoria	34
5.7	Acceso a memoria directo e indirecto	35

Índice de Códigos

5.1	Constructor de nuestra clase para manejar tests	30
5.2	Método para ejecutar un test	30
5.3	Método para calcular la media del tiempo transcurrido	30
5.4	Método para calcular la media del tiempo transcurrido	31
5.5	Ejemplo de uso de test template	31
5.6	Resultado del test	31
6.1	Método que renderiza un sólo carácter	47
6.2	Método que renderiza un sólo carácter	48
6.3	Método que renderiza una cadena de caracteres	48
6.4	Método para dibujar puntos	49
6.5	Métodos de repintado en pantalla	50
6.6	Versión simplificada y reducida del código para dibujar líneas en pantalla . .	51
7.1	Método para crear gradientes de color	55
7.2	Creación e inicialización del mapa de valores	55
7.3	Algoritmo básico de efecto de fuego	56
7.4	Algoritmo final de efecto de fuego	58
7.5	Algoritmo básico de dibujado de circunferencias	61
7.6	Inserción y eliminación de círculos	62
7.7	Actualización del túnel	62
7.8	Creación de un camino de turbulencia	65
7.9	Actualización de un camino de turbulencia	66
7.10	Generación de tablas precalculadas y uso en código	70
7.11	Código para cargar una imagen BMP en nuestro sistema	73
7.12	Código para convertir una cadena de 4 bytes en un entero de 32 bits	74
7.13	Dibujar un píxel cuya textura se desplaza en función de un ángulo, una escala y una traslación	75
7.14	Dibujado de un pixel aplicando una función que modifica el acceso a textura	78
7.15	Funciones de deformación en X	78
7.16	Combinación de distintas ondas	82
7.17	Código para generar un efecto básico de planos infinitos, con escalado, rotación y translación	86
7.18	Código para calcular el desplamiento de la textura	89
7.19	Código para dibujar un objeto 3D	93
7.20	Métodos para rotar en torno a los ejes, en el espacio 3D	94
7.21	Ciclo de actualización de un objeto en pantalla	95
7.22	Cálculo de la perspectiva cónica	97

8.1	Código necesario para inicializar PortAudio	101
8.2	Función delegada que pasamos a PortAudio	103
8.3	Actualización y obtención del valor de las notas	107
8.4	Cálculo de una onda sinusoidal con una frecuencia determinada	110
8.5	Aplicación de un filtro de pasa baja y uno de pasa alta a la generación de ruido	112
8.6	Generación de un sonido retro	114
8.7	Aplicación de una deformación de onda a nuestra malla de puntos	115
8.8	Generación de un sonido de olas	117
8.9	Método que establece una relación entre los vértices de un plano y una esfera	118
8.10	Otros sonidos empleados en la demo	124
8.11	Generación de la escala de Shepard	125
8.12	Generación de un sonido etéreo	126

1 Introducción

Sin saber muy bien cómo, hemos llegado a un punto en el que conocer las bases del funcionamiento de un computador nos parece algo obsoleto, e incluso arcaico. Tecnologías de hace 20 años se tachan de reliquias en un mundo que aún no cuenta un siglo de antigüedad.

El irrefrenable avance de la tecnología y velocidad de evolución es innegable, pero a menudo, cuando se avanza muy rápido, también se pierde muy rápido.

La abstracción en el mundo de la computación ha sido un factor clave, de hecho es el factor que ha permitido que un set reducido de instrucciones como el que tienen los ordenadores sea capaz de imitar la realidad. Abstraer el software y llevarlo hacia modelos más cercanos al ser humano ha permitido pensar más en términos de nuestro día a día y menos en términos de mover memoria y realizar sumas y restas. Y esto es bueno, si para realizar cualquier mínima tarea tuviéramos que preocuparnos de hasta el más mínimo detalle de implementación, la curva de aprendizaje sería demasiado inclinada, y la eficiencia de la producción del software caería en picado.

Sin embargo, a más nos alejamos del hardware y más capas de abstracción añadimos, las instrucciones que escribimos se alejan más y más del reducido set de instrucciones que nuestro ordenador puede ejecutar. Como dijo una vez David J. Wheeler, *"Todo problema en computación puede resolverse con otra capa de indirección, excepto el problema de tener demasiada indirección"*¹.

Esta frase, además de tener un punto cómico, plantea un problema más serio del que muchas veces nos damos cuenta. Hoy en día hay aplicaciones construidas dentro de webs. Para ejecutar código en el cliente de una web se usa *JavaScript*, un lenguaje interpretado. Esto significa que para ejecutar una instrucción de código máquina proveniente de *JavaScript* es necesario, primero, interpretar la línea de código, compilarla y traducirla al lenguaje de la máquina virtual de *JavaScript* que integra el navegador y que esta máquina virtual interactúe con el sistema operativo para ejecutar la orden necesaria. Esto obviando una gran cantidad de pasos intermedios e ignorando las propias capas de abstracción de la memoria, el funcionamiento del procesador, los posibles fallos de la caché del procesador... Y si a todo este proceso, ya de por sí complejo y con muchas capas de por medio, añadimos una aplicación web compleja que añade nuevas capas de abstracción sobre el propio código que se ejecuta en *JavaScript*... ¿No parece demasiado?

Y sin embargo hoy en día prácticas como esta son perfectamente aceptadas, e incluso a veces, son punteras en la industria. Se justifica el hecho de tener una gran capacidad de

¹http://www.stroustrup.com/bs_faq.html#really-say-that

cómputo para poder añadir más y más carga computacional. Se habla de las ventajas en la velocidad de producción, o en la sencillez de manejo del alto nivel en comparación del bajo nivel. Y es cierto que en muchos casos se gana eficiencia o productividad, ¿pero y lo que estamos perdiendo a cambio?

Podemos estar reduciendo la velocidad de nuestro programa cientos de veces, y aún así muchas veces no importa, porque la diferencia entre que algo tarde en ejecutarse 0,001s a que tarde 0,1s se nota, pero tampoco importa tanto. Sin embargo, pensamientos como este son peligrosos, y son los que han llevado a que programas aparentemente sencillos y ligeros incluyan tiempos de espera al iniciarlos, tal y como argumenta Mike Acton².

Y lo que es más, cabe preguntarse, ¿de verdad tanta abstracción simplifica el problema?

La realidad es que hay software donde la gran cantidad de capas que lo forman no solo reduce su tiempo de ejecución, si no que aumenta su complejidad de forma innecesaria, hecho que al que se llama popularmente *overengineering*.

De hecho, hoy en día existen hasta aplicaciones gráficas y juegos dentro de la web. Capas de abstracción dentro de capas de abstracción. Pero en aplicaciones tan computacionalmente costosas como aquellas que manejan gráficos y/o modelos matemáticos, toda esta abstracción tiene un coste que pasa factura. Quizá es precisamente por este motivo, que empiezan a surgir iniciativas interesantes, como Wasm³.

Los gráficos siempre han requerido grandes capacidades de cómputo, aumentar la resolución de pantalla supone un coste cuadrático. Es por ello, que en el terreno de los gráficos, la simplicidad, la sencillez y la eficiencia priman. En una web, la diferencia entre 10 o 100 fotogramas por segundo puede no ser relevante, pero en una aplicación gráfica, en una reproducción de vídeo o en un videojuego, es un factor clave.

Y sin embargo, a pesar de su altísimo coste computacional, los gráficos por computador nos acompañan desde principios de los años 50, (dónde los ordenadores eran miles de veces menos potentes) en forma de hacks rudimentarios que permitían generar gráficos a partir de ficheros de texto⁴. No obstante, el boom de los gráficos por computador se produciría en los años 80, con la aparición y popularización de los primeros ordenadores personales, así como del videojuego. Es en este marco en el que se originaría la ***demoscene***.

El propósito de este estudio es, pues, visitar el arte del *demoscening* e investigar y exponer algunas de las técnicas gráficas que más comúnmente se usaban en los orígenes de la *demoscene*. Pretende ser una vuelta a los orígenes, donde se exploren los gráficos desde una perspectiva actual pero cercana al bajo nivel.

Cabe destacar además que este trabajo no dispone de sección de referencias bibliográficas.

²<https://www.youtube.com/watch?v=rX0ItVEVjHc&t=4620s>

³<https://webassembly.org>

⁴<http://www.catb.org/jargon/html/D/display-hack.html>

Esto se debe a la práctica carencia de publicaciones formales relacionadas con el mundo de la *demoscene*. Las pocas publicaciones que existen respecto a esta subcultura informática abordan principalmente la historia de la misma, sin centrarse en el apartado técnico que tanto la caracteriza, y son extremadamente difíciles de encontrar, pues se trata de libros que ya no se editan, como *Freak⁵* o *Demoscene: The Art of Real-time* de Mikael Schustein, 2004. Además, en lo que se refiere a documentación formal en español -si ya en otros idiomas es extremadamente escasa- parece nula.

La *demoscene* es una cultura de origen social asentada a través de Internet, y es por ello que la mayor fuente de información sobre la misma se halla en la red. Esta es la información que los propios *demosceners* y grupos de *demosceners* han publicado o publican sobre ellos mismos en foros, blogs, páginas personales, discusiones abiertas, redes sociales y de forma muy ocasional, entrevistas.

Es por esto mismo que uno de los ambiciosos objetivos de este trabajo es el de constituir una base sólida y de referencia para la materia de estudio. Aportar un documento referenciable en español que ponga en valor la subcultura de la *demoscene*, analizándola a partes iguales como el fenómeno social que supone así como aportando una base teórica y técnica acerca de los efectos gráficos más populares que se han empleado en la misma.

⁵<http://freak.intro.hu/about.html>

2 Estado del arte

2.1 La demoscene

2.1.1 Qué es la demoscene

La *demoscene* es una subcultura informática cuyo principal objetivo es la creación de demostraciones técnicas llamadas *demoscenes*. Una *demoscene* es un programa autocontenido y por norma general de peso ligero que intenta explotar al máximo el *software* y el *hardware* de la máquina que la ejecuta, con el fin de generar efectos visuales y sonoros. El objetivo de una *demoscene* suele consistir en mostrar el ingenio y las habilidades del programador, así como tratar de impresionar al público.

Además, aunque el *demoscening* en sí mismo no se puede considerar una forma de arte, sí que es cierto que muchas demos poseen un cierto componente artístico.

Se distinguen principalmente dos tipos de demo¹:

- **Demo:** programa que genera gráficos y sonido en tiempo real. Suele tener una extensión superior a 5 minutos y normalmente no tienen límite de tamaño. Una demo suele ser creada por un grupo de personas que incluye al menos un programador, un diseñador gráfico y un músico. Las demos actuales suelen estar realizadas en 3D y cuentan con aceleración gráfica por hardware. Las demos más antiguas o realizadas para plataformas más antiguas (conocidas como "demos *oldskool*") son procesadas de forma íntegra por la CPU (pues las plataformas para las que se desarrollan no poseen GPU) y suelen combinar ilusiones 3D con efectos gráficos en 2D.
- **Intro:** una demo de corta duración. Una intro suele ser temática (mientras que una demo suele compilar distintas escenas/temáticas). Además, las intros no suelen superar los 5 minutos de duración y su tamaño tiende a estar restringido. Las principales categorías de intros son 64K (65536 bytes), 4K (4096 bytes) y 1K (1024 bytes).

Existen otras categorías de demo, aunque son mucho menos comunes, como las **mega demos** (demos de gran duración/extensión, compuestas por múltiples partes) o las **dentros** (intros cuyo propósito es ofrecer un avance de una demo por llegar, todavía en desarrollo).

Además, existen muchas otras categorías derivadas o relacionadas con la *demoscene*, como la creación de gráficos procedimentales. Una de las subcategorías más populares dentro de esta son las **4K images**, imágenes complejas y de alta resolución generadas proceduralmente por programas de 4096 bytes. También es posible encontrar categorías similares relacionadas

¹http://www.oldskool.org/demos/explained/demo_reference.html

con la generación procedural de archivos de música o vídeo. Las mayores diferencias entre estas producciones y las demos son su tiempo de ejecución (no se ejecutan en tiempo real) y las técnicas que usan (al no ser el tiempo una limitación, pueden usar algoritmos computacionalmente más costosos, pero que generan resultados más complejos).

2.1.2 Orígenes de la demoscene

A principios de los años 80, con la popularización de los primeros ordenadores personales, la computación dejó de ser algo que sucedía en universidades para pasar a abrirse al gran mercado. Con ello, llegó también la distribución del software, aunque en aquella época no se producía por internet, si no tan sólo por medios físicos, como los disquetes. Estos programas venían con protecciones de copia por parte de los desarrolladores para evitar su distribución ilegal. Poco tardaron, no obstante, en aparecer los primeros *crackers*, personas que se dedicaban a eliminar las protecciones de copia del software para su distribución gratuita. Esto llevó a la creación de una subcultura informática basada en el *cracking* de videojuegos y otros tipos de software, al margen de la legalidad. Esto se hacía no solo con la intención de poder distribuir el software de forma gratuita, si no que también suponía una fuente de diversión y competición para los *crackers*².

Es por ello que los denominados *crackers* empezaron a "firmar" el software que *crackeaban* con seudónimos que aparecían en los menús o en las intros de los juegos. Con el tiempo, la competición y la ambición de los *crackers* fue aumentando, y llegó un punto en el que no solo se limitaban a quitar las protecciones de copia del software, si no que también creaban sus propias intros para los programas.

Es en este punto cuando la *demoscene* empieza a tomar forma, cuando una parte de los *crackers* deciden retornar a la legalidad pero sin dejar atrás la competición y la diversión. De este modo, este nuevo sector se empieza a dedicar a la creación de intros y demos cuyo objetivo es mostrar sus habilidades al resto de *demosceners*³.

2.1.3 Composición y cultura de la demoscene

La *demoscene* es una subcultura informática muy centrada en el trabajo en equipo y en compartir. Con el desarrollo y popularización de la *demoscene*, a partir de principios de los 90 se popularizó y se estandarizó, con la creación de eventos y competiciones.

Heredado de las *copyparties*, eventos en los que *crackers* y *demosceners* se juntaban para conocerse y compartir software, al margen de la legalidad, nuevos eventos empezaron a crearse a principios de los noventa. Estos eventos sí eran legales y se centraban únicamente en el aspecto de las demos. Pasan a ser eventos sociales en los que los *demosceners* se conocen, comparten y compiten.

Estos eventos, conocidos como *demoparties*, pasan entonces a ser concursos y tener distintas categorías y premios. Para concursar, normalmente los competidores se juntaban en grupos,

²<https://web.archive.org/web/20170726063815/http://tomaes.32x.de/text/faq.php#2.3>

³<http://widescreen.fi/assets/reunanen-wider-1-2-2014.pdf>

usualmente compuestos por al menos un programador, un diseñador gráfico y un músico. Estos grupos tenían su propio nombre e identidad. A su vez, cada uno de los componentes del grupo también solía usar un seudónimo. El uso de un seudónimo es una herencia de los orígenes de la *demoscene* en el *cracking*, aunque el propósito de usar un alias cambia. Mientras que los *crackers* usaban un nombre falso para ocultar su identidad, pues realizaban actividades ilegales, los *demosceners* usan este alias como una forma de expresión.

2.1.4 La demoscene en la actualidad

Si bien la *demoscene* siempre se ha mantenido como una subcultura y nunca ha llegado a tener una popularidad masiva, su auge se dio en los años noventa. En la actualidad, muchos de los eventos de *demoscening* que se crearon en los 90 han desaparecido, y otros tantos han derivado en eventos dedicados a los ordenadores de forma mucho más genérica, derivando en eventos de software o en *LAN-parties*.

Del mismo modo que muchas ferias y eventos han desaparecido, muchos otros también se han ido creando. No obstante, parece que hay una tendencia general hacia el olvido.

Las *demoparties* en la actualidad suelen ser eventos locales, normalmente humildes, donde participan apasionados y nostálgicos.

Es difícil atribuir las causas de la lenta caída en popularidad de la *demoscene*, aunque hay varios factores que pueden contribuir a ello, del mismo modo que hay una serie de factores que evitan que se pierda.

Por un lado, la *demoscene* es cada vez más pequeña debido a:

- Siempre ha sido una subcultura, y nunca ha destacado especialmente por encima de otras subculturas informáticas.
- Para poder participar en la *demoscene* hace falta una gran cantidad de conocimiento matemático y de programación de bajo nivel, cualidades que no abundan.
- Con el auge de los ordenadores, otro tipo de espacios más accesibles al público como las ferias tecnológicas, las ferias de videojuegos o las ferias de programación han eclipsado parcialmente a la *demoscene*.
- Una parte de los *demosceners* eran reacios a la incorporación de gente nueva e inexperta a la *demoscene*, pues si bien aportaban sangre nueva, sus metas y conocimiento se alejaban de los originales de la escena.

Todos estos factores han contribuido a colocar la *demoscene* como una práctica de nicho. Quizá una crítica válida sería que no ha sabido adaptarse de forma conveniente a los nuevos tiempos. El mundo de la *demoscene* no ha sabido publicitarse o venderse suficientemente bien como para llegar a ser conocido por un público mayor. Sin embargo, la *demoscene* sigue viva, y estos son algunos de los factores que contribuyen a ello:

- El auge de lo *retro*. En esta última década ha empezado a masivizarse un cierto reconocimiento y nostalgia hacia los orígenes de la computación y del videojuego. La popularización de los juegos *indie*, técnicas como el *pixel art* y tributos a videojuegos antiguos han llevado a destapar obras olvidadas y a suscitar un nuevo interés por todo lo *retro*.
- La pasión y dedicación de los *demosceners*, que siguen produciendo y compartiendo su obra, extendiendo así su pasión y abriéndola al público general.
- La masivización de la informática. Hoy en día hay muchísimos más informáticos que en los años 80 y 90. Si bien es cierto que hay una tendencia de abandono hacia el bajo nivel, también hay mucha más gente que se hace preguntas, investiga y se interesa por el mismo.

Como reflexión final me gustaría añadir que creo que es posible mantener el panorama de la *demoscene* vivo, pero pienso que esto va a ser muy complicado si no se intenta abrir al público, cosa que algunas *demoparties* ya están haciendo. Está claro que al abrir la *demoscene* al público general se pierden cosas por el camino, y se gana gente que tiene un interés mucho más casual y mucho menos pasional. Creo que en el panorama actual esa gente es necesaria. No todo el mundo tiene el tiempo o la capacidad como para meterse de lleno en la *demoscene*, pero hay muchas personas que sí pueden tener curiosidad por ella de una forma mucho más básica, y esta gente también cuenta. Cada vez es más común encontrar en ferias de videojuegos puestos *retro*, y hay incluso museos del videojuego. La nostalgia y la veneración por los orígenes se abre paso, y pienso que es en este lugar donde la *demoscene* puede buscar su camino.

2.2 Eventos de demoscening

A continuación se listan y describen brevemente algunas *demoparties* que aún se celebran en la actualidad, en el año 2019.

- **Revision:** tiene lugar en durante Pascua, en Alemania. Es la sucesora de la *demoparty* Breakpoint⁴. El evento se estableció en 2011, tras el fin de Breakpoint, y mantiene a muchos de los organizadores. El evento congrega a más de 800 personas de todo el mundo cada año, y es el mayor evento dedicado exclusivamente a la *demoscene* en el mundo⁵.
- **Assembly [2.1]:** tiene lugar en verano en Finlandia, y es el mayor evento de *demoscening* en el país. Además, es uno de las *demoparties* más antiguas que siguen en activo, cumpliendo 25 años el pasado 2017. Sin embargo, el evento no está dedicado exclusivamente a la *demoscene*, si no que es también un evento de videojuegos y *esports*. No obstante, la *demoscene* sigue siendo importante en él, y se celebran anualmente concursos en 5 categorías distintas (Demo, Oldskool demo, 64K intro, 4K intro y 1K intro)⁶.

⁴<http://breakpoint.untergrund.net>

⁵<https://2019.revision-party.net>

⁶<https://www.assembly.org/summer19/demoscene>

- **VIP (Very Important Party)**: es la *demoparty* más longeva de Francia, cumpliendo este año su 20 aniversario. Congrega a personas de todas partes de Europa, aunque es una *demoparty* principalmente francesa⁷. Fue fundada por el grupo de *demosceners* PoPsY TeAm⁸.
- **Nova**: esta *demoparty* fue creada en 2017, por lo que este año se celebra su tercera edición. Es la mayor *demoparty* en Reino Unido, con una salón principal con capacidad para hasta cien *demosceners*⁹.
- **Alternative Party**: es uno de los eventos de *demoscening* más grandes de Finlandia. Es un festival bastante peculiar, (o alternativo, como su nombre indica) y su objetivo es motivar a los programadores y artistas a explorar su creatividad y nuevos puntos de vista. Suele mezclar ordenadores de distintas épocas y capacidades. Aunque mantiene una categoría constante a la mejor demo, el resto de categorías y premios cambian cada año, suponiendo así un reto aún mayor para los programadores. Ha tenido distintos invitados célebres dentro del mundo de la informática, como Al Lowe, creador del juego *Leisure Suit Larry*. Además, la primera noche del evento incluye un concierto de música en el que participan artistas del mundo de la *demoscene*. Este año se celebra su 20 aniversario¹⁰.
- **Chaos Constructions**: es el festival de *demoscenes* más longevo y popular que se celebra en Rusia. Tiene lugar en verano y se creó en 1995, aunque en aquel momento se llamada *Enlight*. La celebración de esta *demoparty* en el año 1997 congregó a más de 1200 personas. En la actualidad, su temática se ha ampliado, incluyendo nuevas áreas como exposiciones y charlas empresariales. En 2018, el festival tuvo lugar durante dos días sin interrupción, contó con participantes internacionales y se emitió de forma íntegra por *Twicht*¹¹

2.3 Grupos de demoscening

A continuación se listan y describen brevemente algunos de los grupos de *demosceners* más populares.

2.3.1 Farbrausch

Farbraush es un grupo de *demosceners* de origen alemán que empezó a ser notado a partir de diciembre del 2000, con su octava producción, llamada *fr-08: .the .product*¹².

El nombre del grupo se puede traducir como "éxtasis de color". Todos sus proyectos empiezan por "fr-número_del_proyecto", donde el número del proyecto se decide en el momento de

⁷https://en.wikipedia.org/wiki/Very_Important_Party

⁸<http://www.popsyteam.org>

⁹<http://www.novaparty.org>

¹⁰<http://www.altparty.org>

¹¹<https://chaosconstructions.ru/en/>

¹²<http://www.farbrausch.de/prod&which=17.py>



Figura 2.1: Assembly 2004 - Fuente: Wikipedia

empezar a trabajar en el mismo, independientemente de cuándo se produzca su lanzamiento.

Farbraush tiene un gran cantidad de demos notorias, como Debris [2.2], que está considerada en el popular portal *demoscener* <http://www.pouet.net> como la mejor demo de todos los tiempos.

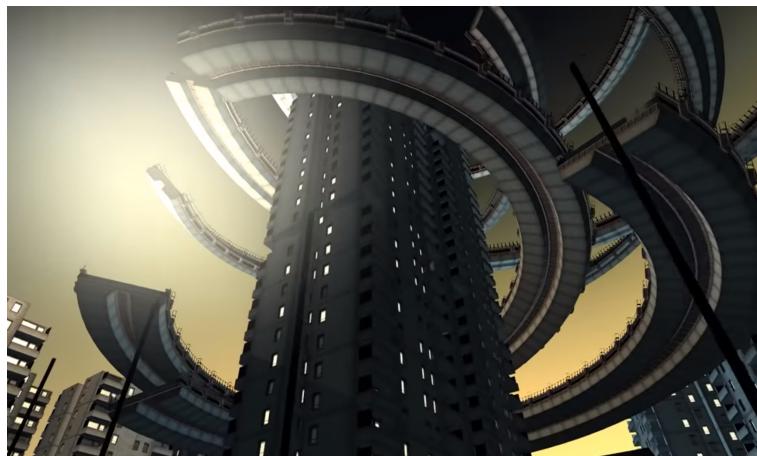


Figura 2.2: Farbrausch 41: Debris - Fuente: YouTube

Además, en el año 2004 un subgrupo dentro de Farbrausch, denominado *.theprodukkt*, lanzó *.kkrieger*, un juego de disparos en primera persona que ocupaba tan sólo 96kB. Este pequeño tamaño se consiguió mediante el uso de generación procedural para las texturas y el uso de formas básicas (cubos, esferas...) combinados y deformados para los modelos. El juego [2.3] recibió distintos premios y fue alabado por la comunidad.



Figura 2.3: Videojuego de 96kB: .kkrieger - Fuente: YouTube

El grupo sigue en activo y siguen produciendo obras de gran calidad, contando con más de diez productos que han recibido primeros premios en distintas competiciones. En general, sus demos tienden a proponer una temática bastante urbana o robótica, con un cierto aire post-apocalíptico. Sin embargo, su capacidad, imaginación y variedad de contenido [2.4] nunca deja de sorprender.

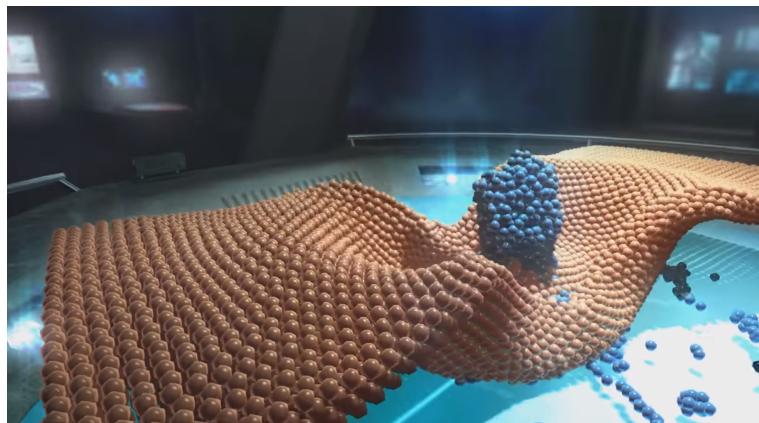


Figura 2.4: Farbrausch 63: Magellan - Fuente: YouTube

2.3.2 Future Crew

Future Crew¹³ fue un grupo de *demosceners* finés, activo principalmente entre 1987 y 1994. Su obra y legado son ampliamente conocidos en el mundo de la *demoscene*. El grupo empezó creando demos para Commodore 64, aunque no tardó en pasar a PC.

Su trabajo es especialmente conocido no sólo por su calidad, si no también porque consiguieron resultados que en aquella época parecían imposibles. Su demo, Second Reality [2.5],

¹³https://en.wikipedia.org/wiki/Future_Crew

publicada en julio de 1993, se considera una de las demos más influyentes en la historia de la *demoscene*. Además, el grupo fue coorganizador de la primera edición de la *demoparty Assembly*.

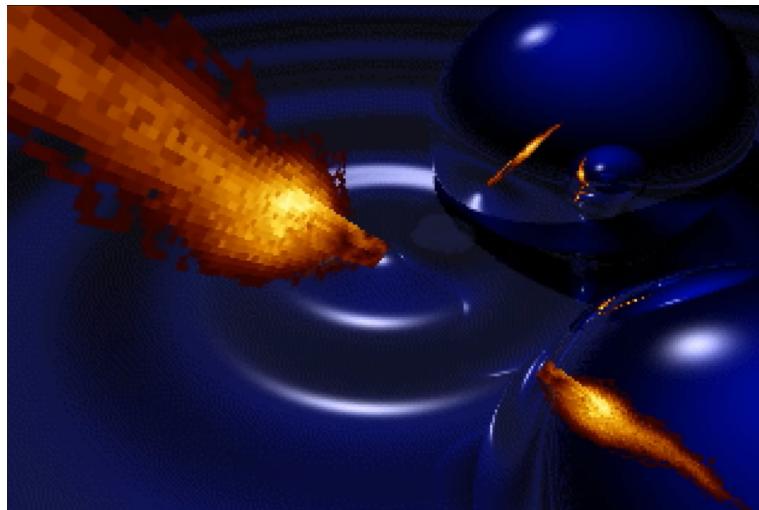


Figura 2.5: Second Reality - Fuente: YouTube - En esta captura se puede ver un efecto de reflexión en dos esferas en tiempo real, mediante *raytracing*

Si bien no se produjo una disolución oficial, el grupo se fue deshaciendo paulatinamente hacia la segunda parte de los 90. La mayoría de sus miembros pasaron a la industria del videojuego o de los gráficos por computador, muchos de ellos fundando sus propios estudios, con resultados exitosos.

2.3.3 PoPsY TeAm

PoPsY TeAm¹⁴ es un grupo de *demosceners* franceses fundado en Lyon, en julio de 1996. Empezaron produciendo demos para Atari y posteriormente para PC.

Son los creadores y promotores de VIP (Very Important Party), la *demoparty* más relevante de Francia. Además, PoPsY TeAm se estableció en 2001 como una asociación legalmente registrada en Francia.

Su demo más conocida es VIP2 [2.6], una demo que presentaron en la *demoparty* holandesa TakeOver en el 2000, resultando ganadora. El objetivo de esta demo era también el de promover su propia *demoparty*, VIP. Esta demo, además, fue la primera de PoPsY TeAm en usar aceleración gráfica por hardware.

El grupo siempre ha intentado promover la *demoscene* y entre otras cosas, han llegado a organizar viajes en bus a diversas partes de Europa para hacer posible al resto de *demosceners*

¹⁴<http://www.popsyteam.org>



Figura 2.6: VIP2 - Fuente: YouTube

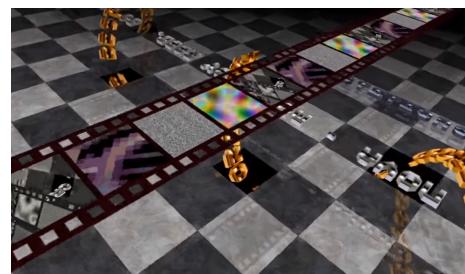
de la región de Lyon atender a *demoparties* europeas. Además, diversos miembros del equipo han participado en la industria del videojuego.

2.3.4 Equinox

Equinox¹⁵ es un grupo de *demosceners* francés que estuvo principalmente activo entre 1988 y 2007, siendo principalmente conocido por sus demos para Atari ST, aunque sus últimas demos, publicadas pasado el cambio de milenio, fueron lanzadas para PC.



(a) Pupul intro (1989) - Fuente : YouTube



(b) Sota 2004 invitation intro (64k intro) -
Fuente: YouTube

2.3.5 Fairlight

FairLight¹⁶ es un grupo de *demosceners* de origen sueco, formado en 1987. FairLight empezó creando demos para Commodore, aunque ha creado también demos para Amiga, SNES y posteriormente en PC.

FairLight fue fundado en 1987 por dos *crackers* suecos, ex-miembros de un grupo llamado "West Coast Crackers". De hecho, FairLight no solo se dedicaba a la *demoscene*, si no tam-

¹⁵ <https://equinox.planet-d.net>

¹⁶ <http://www.fairlight.to>

bién al mundo del *cracking*, rompiendo juegos para su lanzamiento gratuito de forma ilegal. De hecho, llegaron a hacerse especialmente conocidos por la velocidad a la que eran capaces de lanzar juegos *crackeados*¹⁷. Tal fue su impacto que en Abril de 2004, varios miembros del grupo fueron tomados por el FBI en una operación antipiratería denominada *Operation FastLink*. Más de 120 personas fueron arrestadas en esta operación, en la que se consideraba a los *crackers* como una organización criminal.



Figura 2.8: Dead Ringer (por FairLight) - Fuente: YouTube - Demo 64k ganadora de Assembly 2006

Este es, quizás, un grupo en el que se reflejan y mezclan los orígenes de la *demoscene*, provenientes del mundo del *cracking*. A pesar de todo, el grupo volvió a estar en activo a partir de octubre de 2006.

2.3.6 RGBA

RGBA¹⁸ es un grupo español de *demosceners* que estuvo activo entre 2001 y 2009. Todas sus producciones fueron lanzadas para PC, principalmente en Windows.

Son especialmente conocidos por su demo Elevated[2.9]. Esta demo, realizada en colaboración con TBC¹⁹, es especialmente conocida y celebrada por la comunidad *demoscener*, situándose como la segunda más popular en el portal <http://www.pouet.net>.

Los binarios de todas sus producciones se pueden encontrar en GitHub.

¹⁷<https://computersweden.idg.se/2.2683/1.444716/we-might-be-old-but-were-still-the-elite>

¹⁸<http://www.rgbaproject.org>

¹⁹<https://demozoo.org/groups/641/>

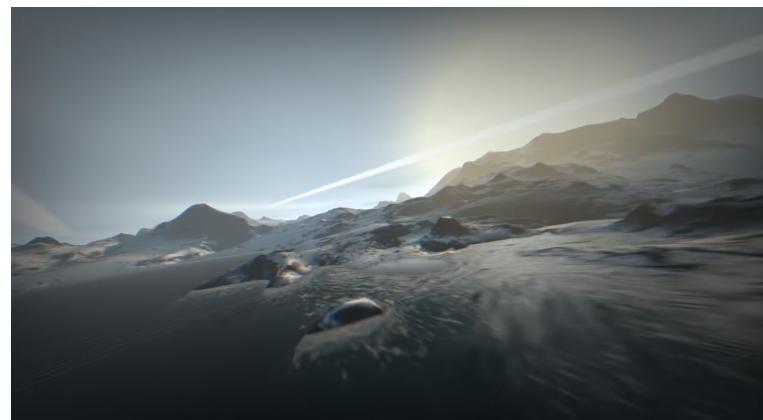


Figura 2.9: Elevated - Fuente: YouTube - Intro 4K ganadora en Breakpoint 2009

2.3.7 Batman Group

Batman Group²⁰ es un modesto grupo de *demosceners* de origen español, activo desde 1993. Han producido juegos y demos para Amstrad CPC, ZX Spectrum, Amiga, Android e iOS.

Su demo más conocida es Batman Forever[2.10], para Amstrad CPC, lanzada en 2011 para la *demoparty* eslovaca Forever, quedando en primera posición.



Figura 2.10: Batman Forever - Fuente: YouTube

²⁰<https://demozoo.org/groups/18871/>

El grupo se vio envuelto en una polémica a finales de 2018, debido a que el grupo de desarrolladores retro *4MHz* había usado para sus producciones código cedido por Batman Group sin su correcta atribución. La polémica se resolvió con la disolución de todo tipo de relación entre Batman Group y *4MHz*²¹.

2.4 Portales de demoscening

A continuación se listan y describen algunos de los portales más populares dedicados a la *demoscene*. Estos portales surgen de la pasión por la *demoscene*, pero son a su vez una interesante y valiosa fuente de conocimiento, así como un reflejo de la historia y el avance de la computación desde finales de los años 80 hasta la actualidad.

- **Scene.org:** el portal y archivo de la *demoscene* por excelencia. El archivo de Scene.org contiene una cantidad inmensa y de lo más completa de demos. En resumen, si estás buscando una demo específica, lo más probable es que esté alojada aquí.
- **Pouet:** el mayor foro dedicado de forma exclusiva a la *demoscene*. Prácticamente cualquier demo que se produce es subida a esta página, donde puede ser votada y comentada por la comunidad. El sitio aloja más de 75000 producciones, y contiene rankings de las mejores demos de todos los tiempos, votadas por los usuarios.
- **Demozoo:** portal dedicado a la *demoscene* que aloja una gran cantidad de información sobre *demogroups* y sus producciones.
- **Slengpung:** este es un curioso portal dedicado a recopilar fotografías tomadas en *demoparties*. Si se ha sido fotografiado en un evento relacionado con la *demoscene*, lo más probable es que se pueda encontrar la imagen en este sitio. Contiene tanto imágenes con más de 25 años de antigüedad como otras tomadas en las *demoparties* más recientes. Este sitio permite además familiarizarse con la cultura de la *demoscene* desde el punto de vista más social y observar su evolución a lo largo del tiempo.
- **Hornet:** se trata de un archivo de la *demoscene* para PC entre 1992 y 1998. Contiene más de 16000 documentos, en un total de más de 7GB. Al tratarse de un archivo, es difícil navegar o encontrar información en él, pero vale la pena dedicar un rato a perderse entre sus ficheros. Este sitio contiene una gran cantidad de conocimiento y pasión, incluyendo acceso a un magazine semanal que llegó a publicar 150 números. Este semanario contenía entrevistas, resúmenes de eventos, explicaciones matemáticas sobre demos y un sinfín más de información.
- **OldSkool:** este portal está dedicado en general a la nostalgia por los PCs y la programación de la "vieja escuela". Es una fuente de recursos que da acceso a información sobre el lado más clásico de la *demoscene*.
- **Demoscene.info:** pequeño portal que contiene información acerca de otros sitios, eventos, demos y *demogroups*. Es un sitio muy pequeño y sencillo, lo que lo convierte en un buen lugar de referencia para empezar a introducirse en el mundo de la *demoscene*.

²¹<http://www.amstrad.es/forum/viewtopic.php?t=5247>

- **Wanted:** este sitio está pensado para que aquellos *demosceners* que estén buscando a otras personas para su grupo puedan anunciararse, así como para que gente sin grupo pueda anunciar sus intereses y habilidades para ser reclutados. Establece un entorno perfecto para formar o completar grupos de *demosceners*. En otras palabras, podría definirse como el LinkedIn de la *demoscene*.
- **Demoparty.net:** pequeño y sencillo portal que lista las próximas *demoparties* y eventos en Europa.
- **Curio:** archivo dedicado a la *demoscene* moderna. Si se busca ver lo último en efectos y gráficos en tiempo real, este es el lugar adecuado.
- **SceneID:** este es un curioso portal. Ofrece un servicio de autenticación común para los portales de *demoscene*. Gracias a este servicio, es posible acceder a los principales portales de la *demoscene* bajo los mismos credenciales.

2.5 Demos destacables

La mayor parte de demos celebres han sido ya mencionadas previamente, al hablar de sus creadores. No obstante, se listan a continuación con el objetivo de la compleción del documento, así como para ampliar la información ya dado en algunos casos.

- **Debris [2.2]:** considerada como la mejor demo de la historia en el portal <http://www.pouet.net/>, fue lanzada en abril de 2007, compitiendo en la edición de Breakpoint del mismo año. Quedó en primera posición. Es una demo para Windows, con un peso de 177KB. Fue realizada usando el propio motor gráfico de Farbraush, werkzeug3, aunque con un nuevo sistema de iluminación y materiales. Usan además sombreado volumétrico, lo cual llegó a suponerles un pequeño *handicap* debido al alto consumo de CPU de esta técnica.
- **Elevated [2.9]:** considerada la segunda mejor demo de la historia en <http://www.pouet.net/>, supuso toda una revolución en cuanto a la cantidad y calidad de contenido que se puede generar con 4KB. Fue lanzada en abril de 2009, para la edición de Breakpoint de ese año, quedando la primera en su categoría. Esta demo tiene una base fuertemente matemática, usando en sus cálculos las derivadas de una función de ruido²². Su principal creador, Íñigo Quílez²³, tiene una presentación explicando el proceso de creación de esta demo²⁴.
- **Heaven Seven [2.11]:** tercera mejor demo de la historia en <http://www.pouet.net/>, se trata de una demo producida por el grupo Exceed, lanzada para MS-Dos y Windows. Es una demo para PC de 64KB que fue primera en la *demoparty* Mekka & Symposium, en abril del 2000. La mayor característica de esta demo es probablemente su uso del raytracing en tiempo real, de una forma que nunca antes se había visto.

²²<http://iquilezles.org/www/articles/morenoise/morenoise.htm>

²³<http://www.iquilezles.org>

²⁴<http://www.iquilezles.org/www/material/function2009/function2009.pdf>

- **Second Reality [2.5]**: demo clásica, lanzada por Future Crew en octubre de 1993 para MS-Dos. Fue la ganadora de Assembly ese mismo año. Está considerada como una de las mejores demos de la historia. Dura más de 15 minutos y cuenta con un sinfín de efectos, incluyendo efectos de geometría, túnel de puntos, patrones de Moiré, rotozoom, efecto de plasma, efecto de agua, raytracing e incluso una nave volando en un entorno 3D. En resumen, una demo completa, que hace uso de todos los recursos disponibles en la época para traer en tiempo real efectos para muchos considerados imposibles por aquél entonces.
- **State of the Art [2.12]**: una demo clásica, lanzada en diciembre de 1992 para Amiga 500. Quedó en primera posición en The Party, el mismo año de su lanzamiento. Sus creadores son el grupo noruego Spaceballs²⁵. Esta demo en ocasiones ha sido criticada por su sencillez en comparación a otras demos de la época, como Nexus 7²⁶ lanzada en 1994 por Andromeda. No obstante, State of the Art se erige como todo un clásico de la *demoscene*, y además, debido a su fuerte componente artístico, es una demo que ha envejecido especialmente bien.

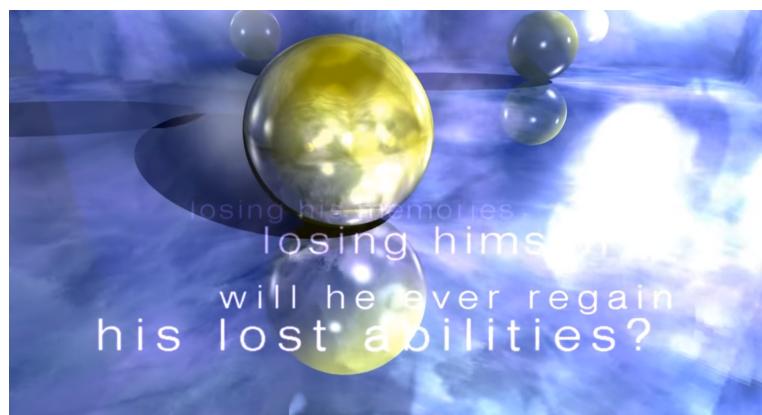


Figura 2.11: Heaven Seven (por Exceed) - Fuente: YouTube

2.6 Efectos gráficos más comunes

A continuación se listan algunos de los efectos clásicos más comunes. Estos son efectos gráficos que llevan presentes desde el inicio de la *demoscene* y que se pueden ver en una gran cantidad de producciones^{27 28 29}.

- **Fuego**: efecto clásico por excelencia [2.13a], podría considerarse casi el "Hola Mundo" de la *demoscene*, debido a su sencillo pero consistente resultado. El efecto de fuego en

²⁵<https://demozoo.org/groups/3/>

²⁶<http://www.pouet.net/prod.php?which=402>

²⁷http://www.oldskool.org/demos/explained/demo_graphics.html

²⁸<http://demo-effects.sourceforge.net>

²⁹https://en.wikipedia.org/wiki/Demo_effect

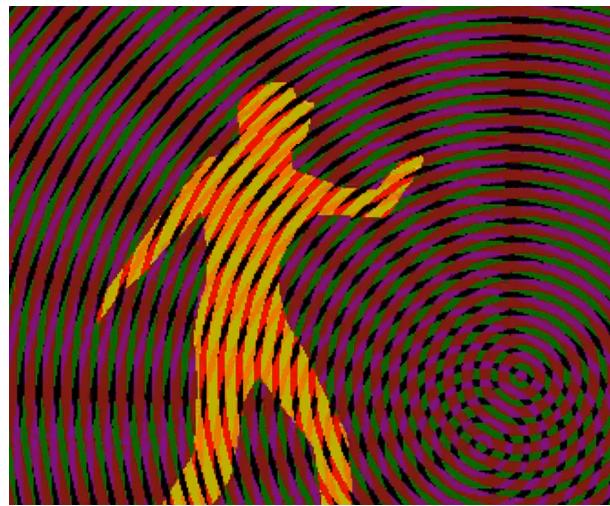


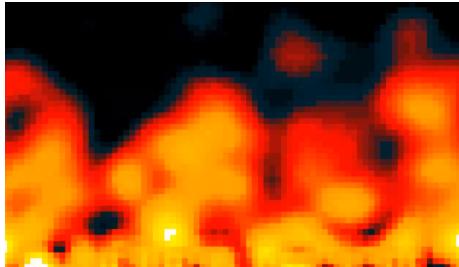
Figura 2.12: State of the Art (por Spaceballs) - Fuente: YouTube

tiempo real es muy sencillo a nivel de implementación, aunque computacionalmente costoso.

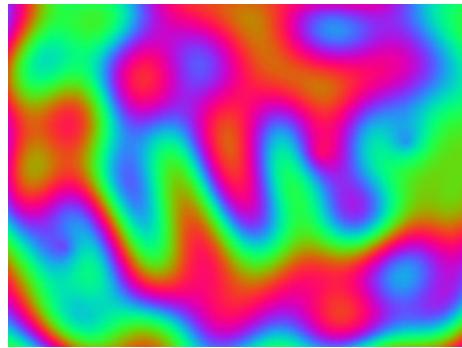
- **Plasma:** este efecto es otro de los grandes clásicos [2.13b]. Es nuevamente un efecto muy sencillo de implementar, pero con alto coste computacional, por lo que normalmente son necesarias optimizaciones o aproximaciones. Puede implementarse mediante la combinación de funciones de seno o mediante funciones de ruido.
- **Partículas:** un tipo de efecto muy socorrido es el de partículas. Debido a las limitaciones de los ordenadores, tener que actualizar miles de píxeles en pantalla, a veces con operaciones matemáticas complejas de por medio, era muy a menudo inviable. Sin embargo, aplicar efectos con trayectorias o transformaciones complejas a sets reducidos de objetos (partículas) en pantalla era perfectamente viable. En esta demo de Equinox [2.7a] cada una de las esferas se podría considerar de hecho una partícula con una trayectoria compleja.
Otro uso muy común era el de crear "nubes de puntos". Eran efectos como túneles o formas de apariencia tridimensional compleja (como paraboloides) formados por puntos. La aplicación de estas fórmulas píxel a píxel era computacionalmente inviable, sin embargo, al aplicarlas únicamente sobre un set reducido de píxeles, podían lograrse resultados en tiempo real de lo más sorprendentes³⁰.
- **Deformaciones de imagen:** las deformaciones de imagen son y han sido efectos muy usados en la demoscene. Muchas de estos tipos de transformación tienen incluso nombre propios. Uno de los efectos más conocidos es el rotozoom (textura que rota y escala simultáneamente) y se logra mediante la aplicación de transformaciones de escalado (multiplicaciones) y transformaciones de rotación (senos y cosenos). Otros efectos de deformación muy populares son las deformaciones de lente o las torsiones [2.13c].

³⁰https://www.youtube.com/watch?v=L_xycbyI1vU

- **Vectores y geometría:** hoy en día, puede que los entornos virtuales en 3D se den como algo garantizado, pero detrás de ello lo único que hay son transformaciones matemáticas mediante el uso de matrices y vectores. Cuando no había tarjetas gráficas ni librerías para gráficos 3D, crear figuras geométricas con sensación de tridimensionalidad era todo un reto, no hablemos ya de escenas complejas [2.13d].
- **Raytracing:** el *raytracing* es una técnica de iluminación usada especialmente para el cálculo de reflexiones. Esta técnica consiste básicamente en hallar la trayectoria de la luz lanzando haces de rayos. Hoy en día existen muchos algoritmos de *raytracing*, y aún así, sigue siendo una operación computacionalmente costosa. Sin embargo, se puede encontrar demos de hace más de 20 años que ya conseguían aplicar esta técnica en tiempo real [2.5].
- **Otros:** existen una infinidad de efectos y variaciones más. Entre otras destacan los efectos de planos infinitos (mediante el uso de transformaciones afines, es conocido como el Modo 7³¹ de Nintendo, usado en juegos como el Mario Kart de la Super Nintendo). El efecto de planos infinitos emula un entorno 3D mediante el uso de una textura 2D.



(a) Efecto de fuego - Inconexia (por Iguana)
- Fuente : YouTube



(b) Efecto de plasma - Fuente : Wikipedia



(c) Deformación de imagen - Batman Forever (por Batman Group) - Fuente: YouTube



(d) Mundo vectorial - Airframe (por Prime) - Fuente: OldSkool

³¹https://en.wikipedia.org/wiki/Mode_7

2.7 Influencia de la demoscene en la industria

La *demoscene* siempre se ha mantenido de forma discreta. Algunas de las razones de que esto sea así se han listado anteriormente, como el hecho de que hace una gran cantidad de conocimiento y pasión para poder participar de forma activa en ella. Sin embargo, esto no ha impedido dejar su huella en la industria informática, especialmente en la del videojuego.

La lista de personalidades que vienen del mundo de la *demoscene* o se han visto influidos por ella es extensa³². A continuación se listan algunas de las más destacables:

- **DICE**³³: La compañía *Digital Illusions*, conocida por juegos como varios de los títulos de la saga *Battlefield* o *Mirror's Edge Catalyst*, cuenta con una gran plantilla proveniente de la *demoscene*, entre los que podemos contar miembros de FairLight.
- **Remedy**³⁴: Esta compañía es especialmente conocida por la saga Max Payne. Fue cofundada por dos miembros de Future Crew. Además, esta compañía mantenía una estrecha relación con Futuremark, creadores de 3DMark, un software de pruebas de rendimiento (*benchmarks*). Esta última compañía también poseía una gran cantidad de miembros provenientes de la *demoscene*, contando con varios de Future Crew.
- **Starbreeze, Ascaron, 49Games, Techland, Lionhead Studios, Guerrilla Games**: Todas estas compañías también cuentan o han contado con miembros de la *demoscene*.
- Will Wright, creador del videojuego Spore, afirma que la *demoscene* fue una gran influencia para el juego, debido a que este está fundamentalmente basado en la generación procedural de contenido³⁵.
- John Carmack, *lead programmer* de videojuegos como Wolfenstein 3D, Doom y Quake, afirmó en la QuakeCon de 2011 que tiene en alta consideración a aquellos programadores que desarrollan demos de 64K, pues tienen que hacer frente a grandes limitaciones y se obtiene mucho conocimiento en el proceso³⁶.
- Jaakkko Iisalo, principal diseñador de Angry Birds, fue un *demoscener* activo y reconocido durante los 90.

Además, hay algunas otras subculturas informáticas que están estrechamente relacionadas con la *demoscene* o derivan de la misma, como por ejemplo la música por *tracker* (hay toda una comunidad de músicos que crean producciones a través del uso de *trackers*, software para la producción de música).

³²<https://chipflip.wordpress.com/2015/06/12/famous-people-who-came-from-the-demoscene/>

³³<http://www.dice.se>

³⁴<https://www.remedygames.com/>

³⁵<http://www.gamespy.com/articles/595/595975p1.html>

³⁶https://www.youtube.com/watch?v=4zgYG-_ha28#t=4827s

3 Objetivos

A continuación se listan los objetivos principales del siguiente trabajo:

- **Entender la *demoscene*:** entender y exponer la subcultura informática de la *demoscene*, sus orígenes y motivación, los rasgos culturales compartidos por sus participantes, sus principales grupos y eventos, así como su legado.
- **Entender la importancia de conocer el bajo nivel:** qué es el bajo nivel y su relevancia en el contexto actual. Cómo influye el conocimiento del bajo nivel y del hardware en el rendimiento de un programa informático.
- **Crear pruebas de rendimiento:** realización de pruebas con el fin de conocer qué operaciones son más costosas de realizar en un computador y por qué. Exponer posibles mejoras y optimizaciones en el código para mejorar el rendimiento de un programa.
- **Entender e implementar los efectos más usados por la *demoscene*:** recopilar las técnicas gráficas más comúnmente usadas en el origen de la *demoscene*. Ofrecer una comprensión profunda y detallada de las mismas, desde un punto de vista tanto teórico como práctico.
- **Crear una breve *demoscene* original:** entender y compilar todo el conocimiento aprendido en una sola demo, que combine todos los efectos y mejoras de rendimiento estudiadas.

4 Metodología

4.1 Software

- **Toggle¹**: Se utilizará la herramienta online Toggl para contabilizar el tiempo dedicado a cada parte del proyecto. Esto permitirá poder analizar qué partes del trabajo han requerido más dedicación y por qué, ayudando a completar el estudio.
- **Git²**: Se utilizará Git para el control de versiones. El uso de Git nos permitirá, además, tener un registro detallado de la evolución del código. El código para este proyecto se aloja en GitHub³.
- **Make⁴**: El código de este proyecto será compilable tanto en las plataformas Windows como Linux. Para ello, el uso de la herramienta Make facilitará la compilación de los proyectos así como su portabilidad.
- **MinGW⁵**: MinGW es un entorno de desarrollo para Windows que ofrece un entorno similar al de GNU. Se usará para compilar tanto el código como las librerías en Windows, haciendo la portabilidad más consistente y sencilla.
- **GCC⁶**: GCC es una colección de compiladores con soporte para C++. Se usará para compilar el código de este proyecto, tanto en Windows (a través de MinGW) como en Linux (de forma nativa).
- **GLFW⁷**: GLFW es una librería multiplataforma para OpenGL que facilita los procesos de creación de ventana, generación de contexto y manejo de input.
- **OpenGL⁸**: OpenGL es una extendida librería para la creación y manipulación de gráficos bidimensionales y tridimensionales. En este proyecto, sin embargo, su uso será mínimo y restringido. Se utilizará tan solo para dibujar una textura en nuestra ventana. Será mediante la manipulación de esta textura que generaremos gráficos. OpenGL, por tanto, será un mero mediador, redibujando constantemente la misma textura en pantalla.

¹<https://toggl.com/>

²<https://git-scm.com>

³<https://github.com/donluispanis/TFG>

⁴<https://www.gnu.org/software/make/>

⁵<http://www.mingw.org>

⁶<https://gcc.gnu.org>

⁷<https://www.glfw.org>

⁸<https://www.opengl.org>

- **Valgrind**⁹: Valgrind es una herramienta de depuración y perfilado. Se utilizará para realizar pruebas de rendimiento y para comprobar el correcto funcionamiento del programa.
- **Compiler Explorer**¹⁰: Compiler Explorer es una herramienta online que permite ver la salida en ensamblador del código escrito de forma instantánea. Resultará muy útil para analizar y entender mejor el código que se ejecuta.

4.2 Tests de rendimiento

Se pretende recopilar una serie de resultados cuantificables que muestren el coste de distintos tipos de operaciones computacionales, a nivel de coste temporal.

Se realizará un análisis exhaustivo de los resultados obtenidos, exponiéndolos y razonando acerca de los mismos.

Para realizar estas pruebas, se procederá a la ejecución de pequeños programas que contengan pruebas concretas. Las pruebas que se propone realizar son: operaciones matemáticas con números enteros, operaciones matemáticas con números en coma flotante, coste del acceso a memoria y coste de funciones matemáticas básicas.

Tras analizar los resultados, se elaborarán una serie de directrices para escribir código que sea generalmente más rápido y/o más fácilmente optimizable. Para poder analizar correctamente los resultados obtenidos, se tendrá en cuenta el código ensamblador generado por el compilador.

4.3 Entorno: motor gráfico

El objetivo final de este proyecto es recopilar e implementar una serie de efectos gráficos. Sin embargo, para poder realizar esta tarea, es necesario disponer de un entorno que nos permita realizar labores básicas, como gestión de la ventana o de input.

Mediante la creación de un entorno se asegura un flujo de trabajo consistente entre todas las demos, así como la reutilización de código. Será necesaria, por tanto, la creación de un motor gráfico. Este motor deberá de ser lo más sencillo posible, pues debe limitarse a facilitar tareas básicas, pero no debe ofrecer soluciones a problemas complejos o específicos a una demo. Este motor debe ser conciso y ligero, pues debe influir lo mínimo posible en el rendimiento de la demo.

Este motor debe darnos soporte para: manejo de la ventana, acceso a un espacio de memoria donde sea posible manipular gráficos, manejo de entradas de teclado, dibujado básico en ventana (puntos, líneas, áreas rectangulares y texto)

⁹<http://valgrind.org>

¹⁰<https://godbolt.org>

4.4 Las demos

Para afrontar cada una de las demos, se seguirá un procedimiento común que se expone a continuación.

4.4.1 Búsqueda de información

En una primera fase, se procederá a la búsqueda de documentación e información sobre la demo. Esto incluye tanto vídeos como imágenes, además de tutoriales o explicaciones teóricas. En esta fase, se plantea recopilar tanta información acerca de la demo como sea posible, y lograr un modelo teórico básico acerca de cómo debería funcionar.

4.4.2 Planteamiento formal

Una vez se ha recopilado información sobre la demo a estudio y se posee un conocimiento suficiente sobre la misma, se procede a un planteamiento formal, previo a su implementación. Este planteamiento incluye entender en profundidad la base matemática de la demo, si la hay. Se debe realizar un análisis y explorar distintos puntos de vista desde los que se podría implementar la demo.

4.4.3 Implementación

Analizada la demo, y se habiendo razonado sobre el mejor modo de desarrollarla, se procede a la fase de implementación en código de la demo. Esta es una fase experimental y que permite flexibilidad. Es posible que sea necesario probar distintos acercamientos, buscando el que más se adecúe al resultado que se busca y que ha sido definido en el planteamiento formal de la demo.

4.4.4 Refinamiento

Cuando la demo ya está completada a nivel de funcionalidad, se procede a su refinamiento y refactorización. En esta fase se busca hacer el código más legible (nombres de variables y funciones explícitos, correcta documentación del código...) así como hacer el código más eficiente (identificar los factores críticos del programa y buscar e implementar soluciones más eficientes).

5 Tests de rendimiento

5.1 Planteamiento inicial

Siguiendo el carácter definido previamente, el objetivo de este trabajo es poder hacerlo todo desde cero, en la medida de lo que resulta posible o razonable.

Es por ello, que siguiendo este mismo espíritu busqué crear mi propio marco de trabajo para ejecutar pequeños tests de rendimiento. Esto es todo lo que debería ser capaz de hacer mi herramienta:

- Ser capaz de ejecutar un test provisto de forma externa
- Ejecutar cada test múltiples veces, para poder obtener un abanico de tiempos de ejecución
- Extraer una media de los resultados obtenidos en la medición
- Mostrar los resultados obtenidos por terminal, así como almacenarlos en un fichero de texto

Al ser pocos los requerimientos de mi herramienta, el desarrollo de la misma debería ser breve y sencillo.

Además, llegó también el momento de plantear el carácter los tests a realizar:

- Medir el coste de la suma, resta, multiplicación, división y módulo para enteros de 32 bits
- Medir el coste de la suma, resta, multiplicación, división y módulo para enteros de 64 bits
- Medir el coste de la suma, resta, multiplicación y división para números en coma flotante de 32 bits
- Medir el coste de la suma, resta, multiplicación y división para números en coma flotante de 64 bits
- Medir el coste de llamar a una función generadora de números aleatorios, llamar una función de seno, de arco seno, de raíz cuadrada y de raíz cúbica en comparación al coste de llamar a una función vacía
- Medir el coste de acceso aleatorio a memoria
- Medir el coste de acceso directo o indirecto a memoria

5.2 Implementación

Para inicializar la clase que manejará nuestros tests, debemos crearla pasándole por parámetro una cadena de caracteres que represente la ruta del fichero donde queremos que se guarde el resultado de nuestros tests. Tras mostrar por terminal un código de "bienvenida", abrimos el fichero con la ruta especificada y, a continuación, inicializamos dos *buffers*. Será redirigiendo la salida de `std::cout` a cada uno de estos *buffers* como conseguiremos escribir tanto en terminal como en fichero.

Código 5.1: Constructor de nuestra clase para manejar tests

```

1 TestTemplate::TestTemplate(const char *logPath)
2 {
3     std::cout << std::endl;
4     std::cout << "Running tests... this may take a while" << std::endl;
5     std::cout << "Output will be stored in " << logPath << std::endl;
6     std::cout << std::endl;
7
8     file = std::ofstream(logPath);
9     writeToScreen = std::cout.rdbuf();
10    writeToFile = file.rdbuf();
11}
```

A continuación, llega el momento de crear un método que nos permita ejecutar nuestros tests, como podemos ver en el código [5.2]. Este método recibe una función por parámetro, que ejecutará internamente 100 veces el test pasado, y almacenará el tiempo de cada ejecución en un vector.

Código 5.2: Método para ejecutar un test

```

1 void TestTemplate::ExecuteTest(std::function<void(void)> test)
2 {
3     timeCounts.clear();
4     std::chrono::duration<double> elapsedSeconds;
5
6     for (int i = 0; i < 100; i++)
7     {
8         time_point = std::chrono::system_clock::now();
9
10        test();
11
12        elapsedSeconds = std::chrono::system_clock::now() - time_point;
13        timeCounts.push_back(elapsedSeconds.count());
14    }
15}
```

Será entonces, tras haber ejecutado nuestro test, que podremos obtener el tiempo medio de ejecución con el siguiente código:

Código 5.3: Método para calcular la media del tiempo transcurrido

```

1 double TestTemplate::CalculateAverageTime()
2 {
3     double time = std::accumulate(timeCounts.begin(), timeCounts.end(), 0.0) / (double)timeCounts.size();
4     return time;
5 }
```

Proveeremos además con un método [5.4] que dado el nombre del test, una descripción del mismo y el tiempo que ha tardado en ejecutarse, escribe de forma fácilmente legible para el

ojo humano los resultados de la ejecución. El resultado se escribe tanto en fichero como por terminal, redirigiendo la salida de texto.

Código 5.4: Método para calcular la media del tiempo transcurrido

```

1 void TestTemplate::WriteTestResultsIntoScreenAndFile(const char *testName, const char *testDescription, ↵
   ↵ double testTime)
2 {
3     auto output = [] (const char *testName, const char *testDescription, double testTime) {
4         std::cout << "| Test " << testName << ":" << testDescription << std::endl;
5         std::cout << "| Time spent: " << testTime << " seconds" << std::endl;
6         std::cout << "|" << std::endl;
7     };
8
9     std::cout.rdbuf(writeToFile);
10    output(testName, testDescription, testTime);
11
12    std::cout.rdbuf(writeToScreen);
13    output(testName, testDescription, testTime);
14}

```

Tal vez lo mostrado hasta ahora pueda parecer un poco confuso, pero es al combinar todo este código para ejecutar un test cuando se ve la potencia de la clase que hemos creado, que nos permite manejar tests con facilidad, de forma legible y sencilla:

Código 5.5: Ejemplo de uso de test template

```

1 TestTemplate T("PerformanceResults.txt");
2
3 T.ExecuteTest([]() {for (int j = 0; j < 10000000; j++); });
4 T.WriteTestResultsIntoScreenAndFile("Loops", "Void For Loop executed 10000000 times", T.←
   ↵ CalculateAverageTime());

```

Código 5.6: Resultado del test

```

1 | Test Loops: Void For Loop executed 10000000 times
2 | Time spent: 0.0168712 seconds
3 |

```

El test de ejemplo mostrado en el código [5.5] es bastante trivial, pues es un código que se limita a ejecutar un bucle. No obstante, se exemplifica lo sencillo y limpio que se vuelve escribir un test, con un resultado también fácil de leer e interpretar [5.6].

5.3 Resultados

El resultado de cada test que se muestra a continuación ha sido hallado a partir de la media de 5 ejecuciones individuales de cada conjunto de tests para cada plataforma. A su vez, cada conjunto de tests ha sido ejecutado internamente un total de 100 veces, como podemos ver en [5.2], habiéndose hallado la media de tiempo de estas ejecuciones.

Los resultados que se muestran en la columna izquierda vienen dados en segundos, siendo el tiempo de ejecución medio del test. Los resultados que se muestran a la derecha son el resultado ponderado con respecto a un resultado base (por ejemplo, cuántas veces es más costosa una división que una suma).

Cada conjunto de tests ha sido compilado a su vez con tres objetivos distintos, siendo estos:

- Ejecutable de **Windows x86** compilado con **g++ 7.4.0** para **MinGW32**
- Ejecutable de **Windows x64** compilado con **g++ 8.2.1** para **MinGW64**
- Ejecutable de **Linux x64** compilado con **g++ 8.2.1**

Los tests han sido ejecutados en los siguientes sistemas operativos:

- **Windows 10 Home x64** (versión 1803)
- **Manjaro Linux x64** (versión 18.0.4)

Los tests han sido ejecutados bajo el siguiente hardware:

- CPU: **Intel Core i7-6700HQ** 2.60GHz, x64 processor
- RAM: **16 GB DDR4-2133**

Cada operación matemática que muestra a continuación se ha realizado dentro de un bucle que se ejecuta 10000000 millones de veces, por lo que el resultado que vemos no se debe usar tanto de forma cuantitativa como para establecer comparaciones o relaciones.

En la tabla [5.1] se muestran los resultados de realizar operaciones matemáticas con números enteros de 32 bits. En la columna de resultados relativos, se compara el coste de las operaciones con respecto al coste de la suma de enteros de 32 bits.

	Results (s)			Relative results		
	Win x86	Win x64	Lin x64	Win x86	Win x64	Lin x64
Sum	0,0232331	0,0286147	0,0284548	0,0	0,0	0,0
Subtraction	0,0232895	0,0268971	0,0278414	0,0	-0,2	-0,1
Multiplication	0,0236683	0,0272116	0,0274386	0,1	-0,1	-0,1
Division	0,0353099	0,0400042	0,0391396	2,7	1,1	1,0
Modulo	0,0361311	0,0392958	0,0451121	2,9	1,0	1,6

Tabla 5.1: Operaciones con enteros de 32 bits

En la tabla [5.2] se muestran los resultados de realizar operaciones matemáticas con números enteros de 64 bits. En la columna de resultados relativos, se compara el coste de las operaciones con respecto al coste de la suma de enteros de 32 bits.

En la tabla [5.3] se muestran los resultados de realizar operaciones matemáticas con números en coma flotante de 32 bits. En la columna de resultados relativos, se compara el coste

	Results (s)			Relative results		
	Win x86	Win x64	Lin x64	Win x86	Win x64	Lin x64
Sum	0,0233056	0,0234745	0,0241097	0,0	-0,5	-0,4
Subtraction	0,0234734	0,0238337	0,0256926	0,1	-0,4	-0,3
Multiplication	0,0240932	0,0245130	0,0265930	0,2	-0,4	-0,2
Division	0,0457833	0,0462163	0,1060608	5,1	1,6	7,6
Modulo	0,0392737	0,0393487	0,1016039	3,6	1,0	7,2

Tabla 5.2: Operaciones con enteros de 64 bits

	Results (s)			Relative results		
	Win x86	Win x64	Lin x64	Win x86	Win x64	Lin x64
Sum	0,0235983	0,0247058	0,0243978	0,1	-0,4	-0,4
Subtraction	0,0236995	0,0245818	0,0247821	0,1	-0,4	-0,4
Multiplication	1,0653020	0,0246152	0,0242643	233,6	-0,4	-0,4
Division	1,2454660	0,0248546	0,0251386	273,9	-0,3	-0,3

Tabla 5.3: Operaciones en coma flotante con 32 bits

de las operaciones con respecto al coste de la suma de enteros de 32 bits.

En la tabla [5.4] se muestran los resultados de realizar operaciones matemáticas con números en coma flotante de 64 bits. En la columna de resultados relativos, se compara el coste de las operaciones con respecto al coste de la suma de enteros de 32 bits.

	Results (s)			Relative results		
	Win x86	Win x64	Lin x64	Win x86	Win x64	Lin x64
Sum	0,0240914	0,0250733	0,0253800	0,2	-0,3	-0,3
Subtraction	0,0239746	0,0246671	0,0253605	0,2	-0,4	-0,3
Multiplication	0,2191568	0,0249607	0,0251803	43,9	-0,3	-0,3
Division	1,2480080	0,0256489	0,0261598	274,5	-0,3	-0,2

Tabla 5.4: Operaciones en coma flotante con 64 bits

En la tabla [5.5] se muestran los resultados de llamar a una función vacía, una función de generación de números aleatorios, una función de coseno, una función de arco coseno, una función de raíz cuadrada y una función de raíz cúbica. En la columna de resultados relati-

vos, se compara el coste de las funciones con respecto al coste de la suma de enteros de 32 bits.

	Results (s)			Relative results		
	Win x86	Win x64	Lin x64	Win x86	Win x64	Lin x64
empty	0,0244699	0,0249656	0,0242798	0,3	-0,3	-0,4
rand()	0,1180654	0,1169742	0,0821240	21,3	8,2	5,3
cos()	0,6730564	0,4900506	0,2550030	145,6	42,7	22,2
acos()	2,7986080	0,3835644	0,0584450	622,0	32,8	2,9
sqrt()	0,3691394	0,1106834	0,0743690	77,5	7,6	4,5
cbrt()	0,8855260	0,5628924	0,3532132	193,3	49,4	31,8

Tabla 5.5: Llamadas a funciones matemáticas

En la tabla [5.6] se muestran los resultados de realizar un millón de accesos a memoria, en el primer lado siendo memoria contigua y en los siguientes casos accediendo a memoria aumentando la distancia entre accesos en potencias de 10. En la columna de resultados relativos, se compara el coste de los accesos con el coste de acceder a memoria contigua.

	Results (s)			Relative results		
	Win x86	Win x64	Lin x64	Win x86	Win x64	Lin x64
Lineal access	0,0025534	0,0024107	0,0023643	0,0	0,0	0,0
Jumps 10	0,0039727	0,0039288	0,0038412	0,6	0,6	0,6
Jumps 100	0,0060442	0,0061442	0,0059900	1,4	1,5	1,5
Jumps 1000	0,0063647	0,0064300	0,0058503	1,5	1,7	1,5
Jumps 10000	0,0095576	0,0096220	0,0056557	2,7	3,0	1,4

Tabla 5.6: Accesos a memoria

En la tabla [5.7] se muestran los resultados de acceder 10000000 a un elemento de una variable, siendo el primer acceso directo y el segundo indirecto (a través de un puntero). En la la columna de resultados relativos, se compara el coste del acceso directo con el acceso indirecto.

5.4 Análisis de los resultados

A partir de los resultados obtenidos, se pueden extraer las siguientes conclusiones:

- Ejecutar código compilado para una máquina de 32 bits bajo una arqui-

	Results (s)			Relative results		
	Win x86	Win x64	Lin x64	Win x86	Win x64	Lin x64
Direct access	0,0166824	0,0169107	0,0156102	0,0	0,0	0,0
Indirect access	0,0220008	0,0223851	0,0208514	0,3	0,3	0,3

Tabla 5.7: Acceso a memoria directo e indirecto

Arquitectura de 64 bits puede tener penalizaciones severas: como podemos ver, la mayoría de las operaciones corren más lento para el código compilado en 32 bits, siendo especialmente notorio en multiplicaciones y divisiones en coma flotante, así como en todas las operaciones matemáticas. Estas diferencias se deben principalmente al código ensamblador que se genera. Por ejemplo, para realizar una multiplicación en coma flotante para un ejecutable x32, se usa la instrucción `fmul`{, mientras que para realizar una multiplicación para un ejecutable x64, se utiliza la instrucción `mulss`{. Cada una de estas instrucciones viene de un set de instrucciones y de una unidad distinta en la CPU, viéndose la primera de la unidad de coma flotante (FPU) mientras que la segunda viene del conjunto de instrucciones de extensión de SIMD (SSE). Las instrucciones SIMD (*Single Instruction, Multiple Data*) son instrucciones que, como su nombre indica, se pueden ejecutar sobre múltiples datos de forma paralela, de modo que son mucho más rápidas que las que se ejecutan en la FPU. No obstante, las instrucciones SSE tienen precisión fija, mientras que las instrucciones para la FPU son más flexibles al poder tener precisión variable.

- **El sistema operativo es un factor determinante:** si comparamos los resultados entre las ejecuciones de los tests en x64, veremos que en general los resultados entre Linux y Windows son consistentes (como cabría esperar). No obstante, podemos ver algunas diferencias significativas, por ejemplo, en la tabla [5.2], la división de enteros de 64 bits toma mucho más tiempo en Linux que en Windows. Se da el caso contrario en la tabla [5.5], donde todas las funciones resultan significativamente más rápidas en Linux. Ambos tests han sido ejecutados bajo la misma versión del compilador, aunque compilando en distintos sistemas operativos. Como podemos ver, los detalles de implementación son muy relevantes, así como la interacción con el sistema operativo.
- **La suma, multiplicación y división de enteros tiene un coste similar, la división y el módulo son al menos el doble de costosas:** como podemos ver en las tablas [5.1] y [5.2], no existe gran diferencia a nivel de coste entre realizar sumas, restas o multiplicaciones, mientras que la división y el módulo siempre tienen una penalización notoria, por lo que debe tenerse en cuenta.
- **Las operaciones en coma flotante tienen un coste pequeño:** un mito muy extendido es que las operaciones en coma flotante son muy costosas. Como podemos ver en las tablas [5.3] y [5.4], este no tiene por qué ser el caso, ya que todas las operaciones en coma flotante para sistemas de 64 bits resultan entre un 20% y 40% más rápidas que la suma de dos enteros. Este no es un mito sin fundamento, no obstante, ya que si observamos el coste de estas operaciones para sistemas de 32 bits, es muy distinto. Por

tanto, nuevamente, la respuesta está condicionada por los detalles de implementación y el set de instrucciones utilizado. Si nuestra máquina no es capaz de ejecutar instrucciones SIMD, entonces es probable que las operaciones en coma flotante tengan un coste elevado.

- **Toda llamada a una función matemática tiene un coste elevado:** nuevamente, el coste de ejecución depende de muchas variables, principalmente de los detalles de implementación, pero como podemos ver en la tabla [5.5], mientras que el coste de llamar a una función vacía es equiparable al coste de realizar una suma, algunas funciones pueden llegar a tener un coste más de 50 veces superior. Esto es un claro indicador para nuestro sistema de que realizar operaciones matemáticas complejas tiene un alto coste, y por tanto, hay también espacio para optimización, sustituyendo las funciones matemáticas dadas por otras propias, sacrificando así precisión por velocidad.
- **La memoria debe ser accedida de la forma más lineal posible:** acceder a memoria no es gratis. Cuando la CPU accede a memoria, carga en una caché una porción de memoria, de forma que pueda acceder a la misma de forma rápida y (casi) directa. Cuando solicitamos un recurso de memoria a la CPU, si accedemos de forma lineal, iremos accediendo continuamente a la caché de la CPU, y será en el momento en el que hayamos recorrido toda la caché cuando se producirá un fallo de caché y entonces la CPU solicitará una nueva porción de memoria, que guardará en su caché. Cuando se produce un fallo de caché, la CPU tiene que cargar nuevos datos en la caché antes de poder acceder a ellos, y este proceso lleva tiempo. Si accedemos a la memoria de forma lineal, minimizamos los fallos de caché, pues sólo se producirá uno cuando hayamos operado sobre toda la memoria que hay en caché. Sin embargo, si accedemos a memoria de forma totalmente aleatoria o inconexa, se producirá un fallo de caché por acceso a memoria, lo que ocasionará tener que cargar cada vez un bloque de memoria completo para realizar una operación únicamente sobre unos pocos bytes de la caché. Tener en cuenta por tanto, que el acceso sea lo más lineal posible, es muy importante, como podemos ver en la tabla [5.7], donde podemos ver como los accesos a memoria saltando de 100 en 100 elementos son un 150% más lentos que accediendo a esos mismos elementos de forma lineal.
- **El uso de punteros no es gratuito, la indirección tiene coste:** en la tabla [5.7] podemos ver, de forma bastante consistente en apariencia, como el acceso indirecto a memoria es un 30% más lento. En otras palabras, el uso de variables mediante punteros es un 30% más lento que el uso de variables mediante valor. Esto tiene sentido, pues al acceder mediante un puntero se ejecuta una instrucción extra, una dereferencia. En otras palabras, mientras que para acceder a una variable de forma normal tan sólo tenemos que obtener el valor que se encuentra en la dirección de memoria de la variable, para acceder a un valor a través de un puntero debemos acceder a una variable que contiene el valor de la dirección de memoria a la que debemos acceder para obtener el valor que queremos. Es por tanto una doble referencia, por tanto una instrucción extra y por tanto, un coste añadido.

5.5 Conclusiones

¿Significa por tanto que no debemos usar punteros porque son un 30% más lentos? ¿Que si nuestro programa accede constantemente a direcciones de memoria situadas en distintos puntos de la memoria interna, nuestro programa va a ser el doble de lento? ¿Que usar cosenos va a realentizar más de 50 veces la velocidad de nuestros programas?

Sí. Y no.

Por supuesto, el uso de punteros, el acceso no lineal a memoria o el uso de funciones matemáticas va a ralentizar nuestros programas, y no hay nada que se pueda hacer para evitarlo. La cuestión no es saber si deben utilizarse ciertas cosas, la cuestión es **cuándo**.

Si tenemos un programa en un sistema embebido de potencia humilde, probablemente el uso excesivo de operaciones en coma flotante podrá ser motivo suficiente como para ralentizarlo. Sin embargo, en nuestro sistema, donde disponemos de SSE, resultan hasta más rápidas que una suma normal y corriente.

Y por supuesto, el uso de punteros es extremadamente útil y necesario, no se trata de evitar un puntero a toda costa por ese 30% de ganancia en rendimiento. Se trata de ser consciente que no se debe usar un puntero por usarlo, pues un uso sin causa justificada sólo implica que el código sea más lento.

Este set de conclusiones que se ha obtenido a partir de los tests es, como ya se ha dicho anteriormente, dependiente del sistema bajo el que se han ejecutado los mismos, y no deberían extrapolarse de forma universal bajo ninguna circunstancia.

Lo importante, pues, es poder identificar qué métodos o prácticas conllevan un coste considerable en nuestro sistema y cuáles no, y ser capaces de discernir con criterio cuándo conviene utilizar cada herramienta.

5.6 Posibles mejoras

Como se ha comentado previamente, haber creado nuestro propio entorno para ejecutar tests de rendimiento presenta tanto ventajas como desventajas. Es por ello que los tests desarrollados tienen validez en el entorno de desarrollo enmarcado, pero tienen una serie de limitaciones:

- **Los tests han sido ejecutados bajo un solo hardware:** esto implica que los resultados obtenidos no pueden aplicarse de forma universal, si no tan solo para el hardware (y software) bajo el que han sido ejecutados. Los tests de rendimiento no eran el foco principal de este proyecto, y haber invertido tiempo en contrastar resultados en otras máquinas hubiera requerido una inversión demasiado grande. Su objetivo principal, que era otorgar una mayor comprensión de cómo funciona la maquina bajo la que se ejecutan, no obstante, han sido cumplidos con creces.
-

- **Los tests han sido ejecutados solo bajo dos sistemas operativos distintos:** de forma paralela a lo explicado para el hardware, se aplica también al software, de forma que estos resultados no se pueden tomar como universales.
- **Los tests son dependientes del sistema operativo:** el resultado de tiempo obtenido de los tests está condicionado por el sistema operativo, y cómo el mismo ha distribuido los recursos y los tiempos para la ejecución del programa. En otras palabras, los tests son *subjetivos* al sistema operativo, pues dependen de los tiempos del sistema.
- **Los tests son dependientes del compilador:** los tests son dependientes del compilador utilizado y las opciones de optimización especificadas. En nuestro caso, todos los tests han sido ejecutados sin optimización alguna, por lo que los resultados pueden variar al compilar con otras opciones o compiladores.

¿Hace todo lo listado anteriormente inválidos los tests realizados?

En absoluto. Pero hace los resultados subjetivos. En otras palabras, los resultados obtenidos en esta máquina no deberían ser extrapolados de forma directa a cualquier otra (aunque puedan coincidir en muchos casos, no deben ser asumibles a cualquier sistema). Para tener unos tests más objetivos se podrían seguir varias estrategias:

- **Distintos compiladores:** ejecutar un mismo set de tests en distintos compiladores y bajo distintas opciones de optimización (en modo depuración, sin optimizaciones y con un nivel de optimización, como mínimo).
- **Distintos sistemas operativos:** ejecutar un mismo set de tests bajo los principales sistemas operativos y en diversas versiones o distribuciones de los mismos.
- **Distintas arquitecturas:** ejecutar los tests bajo distintas arquitecturas de procesador (x86, x64). Actualmente los tests se han compilado para dos arquitecturas distintas en Windows, pero ejecutado bajo la misma.
- **Usar parámetros de medición cuantificables, independientes del sistema operativo:** el tiempo de ejecución, como ya hemos visto, depende en gran medida del sistema operativo y de su sistema de tareas, basarse en otras medidas cuantificables y que sean independientes del sistema, como por ejemplo cantidad de instrucciones, pueden ayudar a obtener resultados más fácilmente cuantificables o extrapolables.

Con todo lo dicho, llevar a cabo todas estas mediciones en si mismas puede conformar un estudio completo, ya no hablamos de la complejidad de crear un sistema capaz de llevar a cabo todas estas mediciones.

Es por ello que se ha mantenido un ámbito más reducido y humilde para los resultados que se presentan en este trabajo, pues si bien estos resultados nos servirán de gran ayuda en el desempeño de las demos y optimizaciones para las mismas, un análisis de los tests en mayor detalle o profundidad está fuera del ámbito de este estudio.

6 El motor gráfico

Antes siquiera de poder empezar a desarrollar la primera demo, es necesario crear un entorno que sea capaz de automatizar las tareas más básicas que no son competencia directa de la demo, como por ejemplo gestión de la ventana y de las entradas de teclado. Este código será común y necesario a todas las demos, pues independientemente de sus características concretas, todas ellas necesitarán una ventana y un espacio en el que poder volcar datos.

Es por ello que antes de empezar con la primera demo, se hizo necesario el desarrollo de un pequeño *framework* que permitiese gestionar de la forma más rápida y sencilla posible aquellas tareas que no debían ser responsabilidad directa de la demo.

6.1 Investigación inicial

Una de las principales influencias en el desarrollo de la versión inicial del motor gráfico fue OneLoneCoder¹. Este programador británico tiene una colección de tutoriales con alto valor educativo y en muchos de ellos explica incluso técnicas de programación de la vieja escuela. Fue a raíz de visualizar estos vídeos donde vi expuestos muchos de los problemas a los que me tendría que enfrentar en el futuro.

El ejemplo más claro: en sus primeros vídeos, este programador siempre repite el mismo código para poner en marcha una consola usable, hasta que decide crear un modelo básico que le permita reutilizar este código.

Este canal ha tenido un gran valor formativo para mí, ya que me permitió identificar una serie de problemas que de otro modo sólo hubieran aparecido en un momento más avanzado del desarrollo, y que sin embargo, hubieran resultado costosos de solventar.

Inicialmente, estos fueron los objetivos que pretendía cubrir el motor gráfico:

- **Reutilización de código:** tareas como abrir y cerrar la ventana o gestionar el dibujado son necesarias en absolutamente todas las demos, por lo que todo código relacionado con la ventana y/o el dibujado debería poder ser reutilizado sin tener que duplicarse.
- **Encapsulación de toda lógica no relacionada con la demo:** uno de los principales objetivos que se persiguen con la creación de un motor gráfico es la claridad. La implementación de una demo **sólo debe contener lógica que está directamente relacionada con sus detalles de implementación**, es decir, con los algoritmos o técnicas de los que la demo hace uso. De este modo, el código de una demo sólo refleja

¹<https://www.youtube.com/channel/UC-yuWVUp1UJZvieE1gKBkA>

la lógica de la misma, sin exponer la lógica necesaria para la de gestión de ventana, que no es responsabilidad de la misma. Esto permite un código más claro y conciso, más fácil de implementar, usar, refinar y entender.

- **Encapsulación de la ventana:** una demo no debe ser consciente de qué es necesario para crear o borrar una ventana, todo lo que debe hacer es ser poder decir "quiero crear una ventana" o "quiero cerrar la ventana" pero no debe ocuparse de los detalles de implementación.
- **Encapsulación del dibujado:** una demo no debe tener responsabilidad de gestionar el dibujado en ventana. Todo lo que una demo necesita saber es en qué lugar de memoria debe escribir para que esos datos sean dibujados en pantalla, pero no debe encargarse de la gestión del dibujado.
- **Abstracción de la plataforma:** el código de una demo no debe contener detalles de implementación relativos a la plataforma en que se ejecuta. Desde el punto de vista de la demo, todo lo que importa es el algoritmo, y este debe ser el mismo independientemente del sistema operativo y del *hardware* sobre el que se ejecuta.

Durante el desarrollo, no obstante, nuevas necesidades se irían añadiendo, ya fuera por nuevas decisiones de diseño, refinamiento de código o por nuevas necesidades de las demos:

- **Abstracción de las librerías y tecnologías utilizadas:** tras varias iteraciones sobre el desarrollo inicial, fue necesario un refinamiento. El motor gráfico contenía demasiada lógica, y era lógica acoplada a la gestión de la ventana o del dibujado. Esto levantó una pregunta: ¿y si en algún momento necesito cambiar las librerías que utilizo o incluso prescindir de las mismas? Esta era una posibilidad bastante probable, dado que a lo largo del desarrollo de un proyecto y conforme surgen nuevas necesidades, puede que las tecnologías elegidas inicialmente no satisfagan las condiciones actuales. Por tanto, el motor gráfico no debía estar acoplado a las tecnologías que usaba, si no que debía mediar con ellas mediante el uso de interfaces.
- **Abstracción de los eventos de teclado:** conforme el desarrollo avanzó, se hizo aparente que en muchas ocasiones era útil permitir al usuario modificar parámetros de la demo en tiempo real, en cierto modo permitir "jugar" con la demo. Era necesario por tanto permitir el manejo de eventos de teclado, aunque su uso debía estar abstraído de su implementación, de forma que desde el punto de vista de la demo, todo lo que se pudiera hacer es "quiero saber el estado de esta tecla".
- **Abstracción del dibujado de texto en pantalla:** una vez más, al continuar con el desarrollo, se hizo aparente la necesidad de poder dibujar texto en pantalla. El motor gráfico debía ser por tanto capaz de abstraer o enmascarar las rutinas de dibujado del texto, de modo que desde la perspectiva de la demo todo lo que importase fuera dibujar un texto con un color, posición y tamaño determinados, independientemente de la implementación.
- **Uso de mecanismos de dibujado seguros para formas básicas:** aunque inicialmente parecía que cualquier tipo de dibujado debía ser responsabilidad de la demo, pronto se hizo aparente que ciertas rutinas se repetían de forma constante. Además,

mientras que en un inicio el dibujado de un punto o una línea era responsabilidad de la demo, pronto se vio que desde el punto de vista de la demo, estas responsabilidades no tienen interés, ya que la capacidad de poder dibujar una línea es importante, pero no cómo se dibuja.

- **Dibujado de puntos:** desde la perspectiva de una demo, tan sólo importan la posición, color y tamaño de un punto que se quiera dibujar en pantalla. La gestión de si ese punto está dentro o fuera de los límites de pantalla o la gestión del tamaño del punto no debería ser competencia de la demo, si no del motor.
- **Dibujado de líneas:** una demo debe ser capaz de solicitar el dibujado de una línea dados dos puntos, un color y un tamaño, pero no debe responsabilizarse de la gestión de los límites en pantalla ni del algoritmo de dibujado para una línea.
- **Dibujado de rectángulos:** una demo debe ser capaz de dibujar rectángulos en pantalla, especialmente útiles para el borrado de la pantalla o de regiones de la misma, pero no debe conocer sus detalles de implementación.

Con todos estos puntos en mente, y de forma progresiva, se fue desarrollando, revisando y refinando la creación de un motor gráfico que sirviera como marco de trabajo efectivo para el desarrollo de una demo.

6.2 Características

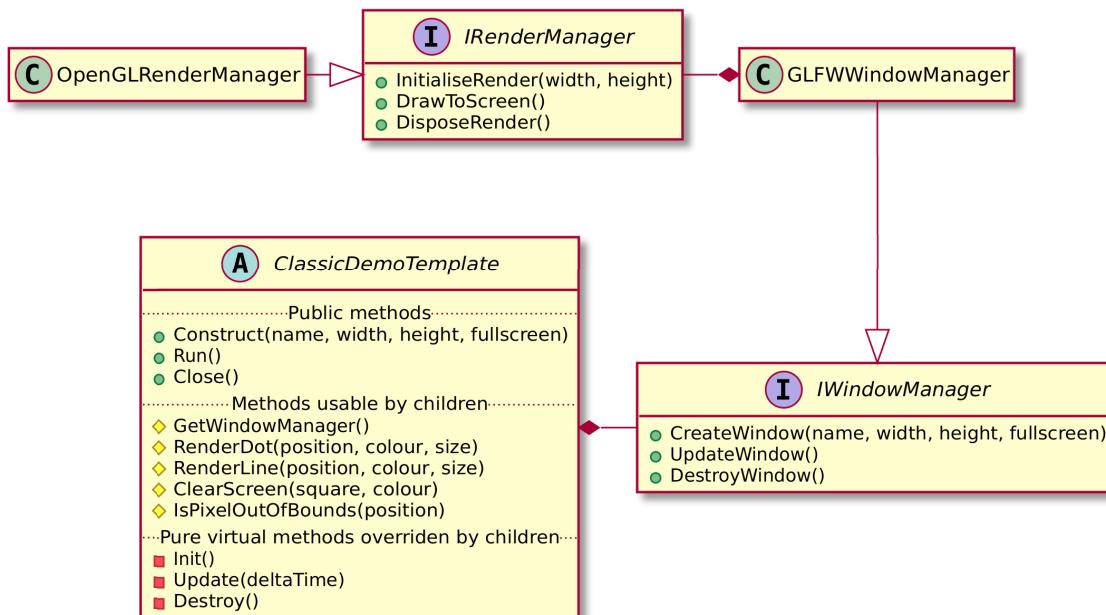


Figura 6.1: Diagrama simplificado de la estructura del motor gráfico

La figura [6.1] presenta la estructura simplificada del motor gráfico.

Como se puede observar, el motor (*ClassicDemoTemplate*) delega las tareas de gestión de la ventana en una interfaz cuyos métodos más relevantes permiten crear, actualizar y destruir la ventana. De este modo, el motor gráfico está completamente desacoplado de las tareas concretas de gestión de la ventana.

La implementación concreta de la interfaz (*GLFWWindowManager*) utiliza, como su nombre indica, la librería GLFW para gestionar la ventana. Esta es una librería de código abierto y multiplataforma que hace más sencilla la gestión. No obstante, la implementación concreta está completamente desacoplada del sistema, por lo que si fuera necesario migrar a una tecnología distinta (como SDL, SFML o accediendo directamente a la API gráfica de Windows (WinAPI) o Linux (X11)), se podría hacer siempre y cuando esta nueva clase implementase la interfaz definida.

A su vez, *GLFWWindowManager* hace uso de la interfaz *IRenderManager*, que implementa *OpenGLRenderManager*. Esto permite, una vez más, cambiar la tecnología de dibujado sin tener que cambiar necesariamente el sistema de ventanas. De este modo también se separa de forma efectiva todo el código relativo a la gestión de la ventana con respecto al código relativo al dibujado, lo que facilita la claridad y mantenimiento del código.

En nuestro caso, el dibujado se hace mediante OpenGL, una especificación para gráficos 3D multiplataforma. No obstante, OpenGL es utilizado como un mero mediador, cuyo único uso es el dibujado de una textura en pantalla. Es esta textura que se *renderiza* de forma cíclica a la que el usuario tiene acceso y puede modificar, dibujando así en pantalla.

De este modo, la clase principal de nuestro motor (*ClassicDemoTemplate*) no tiene responsabilidad directa sobre la gestión de la ventana y el dibujado, de modo que aunque las librerías o tecnologías utilizadas cambiasen, toda la lógica contenida en el motor seguiría siendo usable.

Como se puede observar en el diagrama, esta clase principal es una clase abstracta, lo que implica que debe ser implementada por una clase concreta para poder instanciarse. Toda demo que use este motor gráfico debe heredar de *ClassicDemoTemplate*. Esto permite definir una estructura que todas nuestras demos deberán satisfacer para hacer un uso efectivo de nuestro motor.

En primer lugar, se exponen únicamente tres métodos, *Construct*, *Run* y *Close*. Esto implica que cualquier demo ha de ser completamente usable mediante estos tres métodos.

A continuación, *ClassicDemoTemplate* expone una serie de métodos que pueden ser utilizados únicamente por las demos, que heredan de esta clase. Estos métodos aportan funcionalidad común que resultan útiles en la mayor parte de las demos, como dibujar puntos y líneas o comprobar si un píxel determinado está dentro de los límites de la ventana.

Por último, hay tres métodos virtuales y privados que toda demo debe implementar: *Init*, *Update* y *Destroy*. La llamada a estos método es gestionada por *ClassicDemoTemplate*, por lo que el usuario tan sólo debe preocuparse de implementarlos. Los métodos *Init* y *Destroy*

permiten inicializar y destruir las variables los datos propios de la demo. El método *Update* es llamado en el bucle de ejecución del programa y recibe el tiempo sucedido desde el último fotograma. Este método contendrá toda la lógica necesaria para actualizar los datos que maneja nuestro programa a lo largo del tiempo.

6.2.1 La textura de dibujado y el píxel

Para dibujar en pantalla en este proyecto, lo único que nos interesa es tener una zona de memoria en la que sepamos que podemos volcar datos. Es decir, no se usarán librerías externas para manipular los gráficos que se muestran en pantalla, si no que todas las transformaciones y el pintado en pantalla corre a nuestra cuenta.

Para ello, no obstante, el motor debe tener la capacidad de proveernos con un espacio de memoria en el que podamos pintar. Esto es responsabilidad de la implementación del *IRenderManager*.

En nuestro caso, la clase *OpenGLRenderManager* crea una textura basada en la altura, anchura y número de canales que queremos usar para la demo.

La altura y anchura se define en píxeles, y el número de canales se corresponde con la cantidad de canales de color que queremos. Por ejemplo, si quisiéramos un ventana monocromática, bastaría con definir un solo canal. Para crear una ventana que pueda representar (casi) cualquier color, necesitamos definir tres canales: rojo, verde y azul. Estos son los tres colores primarios de la luz. A través de combinaciones de los mismos podemos crear cualquier color. También podríamos definir incluso cuatro canales, tres para el color y un canal *alfa* para la transparencia. Por defecto, se crea una textura asumiendo tres canales, cada uno de 8 bits, por lo que disponemos de 256 intensidades de color por canal y un total combinado de 16777216 colores posibles. La memoria reservada es pasada cada fotograma a OpenGL, como una textura. Es entonces esta librería la que se encarga de volcar estos datos en pantalla.

Sin embargo, desde el punto de vista del usuario, trabajar con la memoria directamente se hace, cuanto menos, incómodo. Para poder modificar cualquier píxel en pantalla es necesario tener en cuenta el número de canales, la posición y orden de cada color para un píxel determinado... y de pronto operar con píxeles, que debería ser algo sencillo, se convierte en una tarea compleja. Para poder sumar dos píxeles de pronto hay que sumar manualmente cada canal por separado, ¿pero qué pasa si la suma de dos canales supera el valor máximo que puede tener un color por canal, 255? La suma binaria de 128 más 128 resultaría en 0, con lo que si intentamos sumar dos rojos de intensidad media, obtendríamos como resultado el color negro (0), en lugar de un rojo intenso (255). Manejar esta textura se convierte de pronto en una tarea poco intuitiva, que nos llevará a cometer errores con mayor facilidad. Es por ello que se introduce el concepto del píxel a nivel del código.

Un píxel [6.2] es una clase que contiene únicamente tres variables cuyo acceso es público. Estas variables son R(ed), G(reen) y B(lue), haciendo referencia al nombre del color (en inglés) asociado a cada canal. De este modo, la representación de un píxel coincide en memoria con la textura que hemos definido. De este modo, si bien a OpenGL le pasamos un conjunto

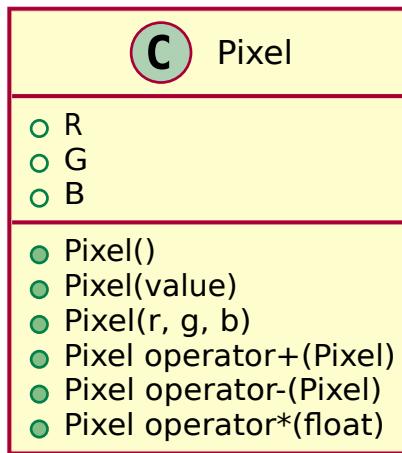


Figura 6.2: Estructura de un píxel en el motor gráfico

de bytes, desde nuestro sistema los podemos organizar en píxeles.

Se proveen tres posibles constructores para un píxel. El constructor por defecto inicializa todos los valores a 0 (es equivalente a crear un píxel negro). También podemos proveer un solo valor al constructor, en cuyo caso se asignará este valor a cada canal (resultando en un color gris de mayor o menor intensidad) y por último podemos inicializar un píxel proveyendo un valor específico a cada canal. Es importante notar que por razones de eficiencia y simplicidad, no se hace ningún tipo de comprobación en la construcción, por lo que el intento de asignación de cualquier valor fuera de rango será equivalente al módulo de 256 de dicho valor. Por ejemplo, si intentamos asignar el valor 512 a un canal, será equivalente a asignarle un valor de 0.

Tras los tres métodos de construcción, se proveen tres operaciones matemáticas que se pueden realizar con píxeles, suma y resta de píxeles y multiplicación de un píxel por un escalar. La división de un píxel por un escalar no se provee ya que no resulta necesaria, en caso de que se quiera dividir el valor de un píxel, se multiplica por el inverso, consiguiendo el mismo resultado.

Por razones de consistencia, la suma y resta de píxeles tienen comprobaciones para evitar el desbordamiento. No sería deseable en nuestro sistema que la suma de dos píxeles cualesquiera, por ejemplo (128, 129, 130) y (200, 201, 202), resultara en un color oscuro. La suma de estos dos píxeles resultaría en (255, 255, 255), es decir, en blanco. Del mismo modo, la resta de dos pixeles siempre debería producir un color más oscuro, pero nunca más claro, por lo que cualquier resta de píxeles que pudiera producir desbordamiento ($123 - 124 = -1$, que equivale a 255, siendo este valor máxima intensidad) se resuelve en 0 (mínima intensidad o negro).

La multiplicación de un píxel por un escalar no se comprueba, ya que dado su uso en el sistema, no resulta práctico. El usuario ha de ser consciente, sin embargo, que sólo las

operaciones en que el valor del escalar está entre 0 y 1 son seguras, ya que la multiplicación de un píxel por cualquier valor inferior a 0 o superior a 1 puede producir, potencialmente, desbordamiento ($64 * 2 = 128$, no desborda, comportamiento esperado; $128 * 2 = 256$, resulta en 0, comportamiento inesperado).

El motivo por el que la suma y la resta son operaciones comprobadas mientras que la construcción y la multiplicación no lo son está justificado. Los valores de construcción y multiplicación en nuestro sistema están siempre controlados, y en la mayor parte de los casos predefinidos, por lo que se asegura de antemano que no se va a producir un desbordamiento, y por tanto poner comprobaciones para estas operaciones tan sólo supondría en la pérdida de eficiencia (ahora bien, es necesario hacer un uso consciente de las mismas). La suma y la resta necesitan comprobaciones ya que la mayor parte de sumas y restas que se producen son sobre valores generados dinámicamente, sobre los que no tenemos control directo, por lo que no podemos asegurar que no se va a producir desbordamiento, y debemos, por tanto, prevenirllo.

Del mismo modo, se provee acceso directo al valor de cada píxel, no solo para su lectura, si no también para su escritura. Esto se hace por conveniencia, pues en ocasiones puede que solo queramos modificar un canal concreto. Debe tenerse en cuenta, no obstante, que estas operaciones pueden provocar desbordamiento. La razón de esta decisión es nuevamente el hecho de que cuando se accede de forma directa al valor de un canal, se hace de forma controlada, por lo que se programa en torno a la posibilidad de desbordamiento.

Este sistema busca ser eficiente, por lo que sólo se realizan comprobaciones cuando se consideran estrictamente necesarias. Esto fuerza, no obstante, a hacer un uso consciente del sistema.

6.2.2 Detectar input

Por básica que pueda parecer la detección de entradas de teclado, esta característica no fue implementada hasta un estado relativamente avanzado del desarrollo del proyecto. Aunque esto pueda resultar difícil de entender desde el punto de vista del usuario, desde el punto de vista del desarrollador, las entradas de teclado sirven principalmente para "jugar" con la demo y para ajustar valores, es decir, que resultan útiles en el proceso de refinamiento de la demo, pero son irrelevantes en el proceso de creación de la misma. Es por ello que disponer de *input* para las demos no es algo que se priorizara, ya que estaba mucho más interesado en el desarrollo de los algoritmos y métodos necesarios para cada demo que no en poder "jugar" con los resultados.

No obstante, llegó un punto en el desarrollo en el que las demos, además de ser funcionales, debían ser manipulables, modificables de forma dinámica, y fue en este momento cuándo se planteó la pregunta de cómo gestionar las entradas de teclado.

Las entradas de teclado se gestionan desde la ventana, por lo que tenía claro que su gestión debería ser parte de la responsabilidad de *GLFWWindowManager*. Era importante, sin

embargo, saber en qué eventos de teclado estaba interesado.

Estos eran los eventos que me interesaban: saber el momento en que la tecla se pulsa por primera vez, saber el momento en el que la tecla se suelta y saber si la tecla se está manteniendo pulsada o no. Sin embargo, GLFW sólo daba acceso directo a saber si la tecla estaba pulsada o no, por lo que la lógica para el resto de eventos debía ser implementada por mi parte.

GLFW permitía otra opción, además, en lugar de preguntar por el estado de una tecla concreta, es posible pasarle un método delegado que sea llamado cada vez que se produce un evento, de forma que este método sobre el que nosotros tenemos control pueda gestionar los eventos en los que estamos interesados. Por razones de simplicidad y mantenibilidad, decidí optar sin embargo por la opción de preguntar por el estado de las teclas.

Esta opción sin embargo planteaba un problema de eficiencia: la única forma de saber si una tecla cualquiera está pulsada o no es almacenando y actualizando el estado de todas las teclas. Esto implicaría tener que estar actualizando el estado de más de 100 teclas cuando tan sólo tenemos interés en unas pocas. Y precisamente lo que decidí fue añadir un método que permitiera registrar interés en una tecla. Esto hace que la gestión de *input* en una demo sea ligeramente más compleja (para poder preguntar por el estado de una tecla, esta debe haberse registrado previamente). Sin embargo, a cambio de esta ligera complejidad añadida, hay una gran ganancia en rendimiento, ya que en lugar de actualizar el estado de *todas* las teclas del teclado por fotograma, sólo actualizaremos el estado de aquellas que nos interesan, y en caso de no tener interés en ninguna, simplemente no se tendrán en cuenta las entradas de teclado, no impactando en ningún modo al rendimiento del programa.

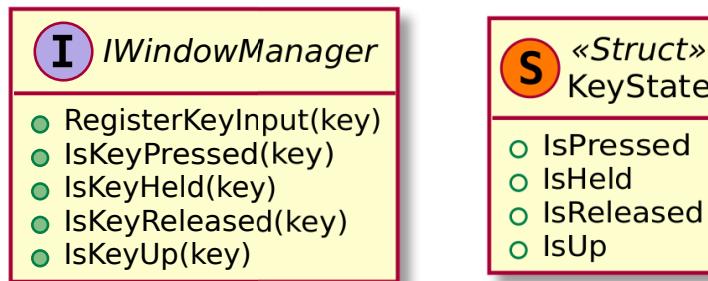


Figura 6.3: Funciones y estructuras del sistema de input

6.2.3 Dibujar texto

Al igual que con el input, la necesidad de dibujar texto no se hizo aparente hasta un estado más avanzado de desarrollo del proyecto. Una vez en la tesitura en la que poder dibujar texto en pantalla era necesario, fue necesario dedicar un tiempo de análisis.

Inicialmente pensé en usar una librería y estuve investigando cuál podía satisfacer mejor

mis objetivos. `glfreetype`² parecía bastante adecuada, tratándose de una librería pequeña, concisa y aparentemente sencilla de utilizar.

A mitad camino del proceso de integración de esta librería, no obstante, me empecé a plantear hasta qué punto tenía sentido utilizar una librería externa para el dibujado de texto en pantalla. Al fin y al cabo, uno de mis objetivos con este trabajo era que todo aquello que estuviera dibujado en pantalla hubiera sido desarrollado de forma exclusiva por mi, sin usar librerías externas. Las únicas librerías que usa este trabajo (GLFW y OpenGL) son utilizadas con el único objetivo de facilitar la implementación y no salirse del ámbito de este estudio. No obstante, este no parecía ser el caso para el dibujado de texto en pantalla.

No obstante, implementar un sistema de dibujado de texto parecía complejo, *a priori*. Es por ello que me plantée, ¿cuál es la forma más sencilla y rápida en la que podría dibujar texto en pantalla?

Tras un tiempo de análisis, llegué a la conclusión de que implementar un sistema de dibujado de texto era una tarea sencilla y que se podía considerar dentro del ámbito de este proyecto. Por tanto, no sería necesario usar librerías externas. Este sistema de texto debería ser lo más sencillo posible, no obstante, y tendría ciertas limitaciones.

La mayoría de librerías de texto son capaces de leer formatos de fuente como TrueType o OpenType, además, para cada letra dibujan un *quad* mediante OpenGL al que aplican una textura con transparencia representando la letra. Este proceso es más complejo de lo asumible para este proyecto, pero por suerte, no era necesario seguirlo.

Mi sistema estaría basado en las siguientes ideas:

- Es necesario un método para dibujar caracteres individuales
- Es necesario un método para dibujar cadenas de caracteres
- Todo lo que necesita un carácter para ser dibujado es posición, color y tamaño.
- La forma más sencilla de usar una fuente es crear nuestra propia fuente, que vaya embebida en el código
- Cualquier carácter puede ser definido dentro de una cuadrícula de 5x5 unidades como en [6.1]
- No es necesario diferenciar entre mayúscula y minúscula (tener doble representación para las letras supone una pérdida de espacio y tiempo)

Código 6.1: Método que renderiza un sólo carácter

```

1 const char *Characters::A{
2     "###_"
3     "#____#"
4     "######"
5     "#____#"

```

²<https://github.com/benhj/glfreetype>

```

6     "#____#
7 };

```

Así pues, la tareas que realmente más tiempo llevan en un sistema como el definido es crear todos los caracteres que se necesitan, que en mi caso son todas las letras del alfabeto inglés, los números y algunos caracteres especiales. Una vez hecho esto, se inserta todos estos caracteres dentro de un mapa estático cuya clave sea un carácter y el valor sea la cuadrícula 5x5 que lo representa.

Código 6.2: Método que renderiza un sólo carácter

```

1 void ClassicDemoTemplate::RenderCharacter(char character, int x, int y, int scale, const Pixel &colour)
2 {
3     if (character < 0 || character == ' ')
4     {
5         return;
6     }
7
8     const char *c = Characters::GetCharactersMap()[character];
9
10    for (int i = x; i < x + 5 * scale; i++)
11    {
12        for (int j = y; j < y + 5 * scale; j++)
13        {
14            int offsetX = (i - x) / scale;
15            int offsetY = (j - y) / scale;
16
17            if (c[offsetY * 5 + offsetX] != ' ')
18            {
19                screen[j * width + i] = colour;
20            }
21        }
22    }
23}

```

Una vez tenemos el mapa de caracteres, dibujar un carácter en pantalla es de lo más sencillo [6.2]. Tras una comprobación inicial para saber si el carácter no es válido o es un espacio (que obviamente no se dibuja), lo primero que hacemos es obtener la cuadrícula 5x5 que se asocia con el carácter a dibujar.

Una vez hecho esto, recorremos la cuadrícula en vertical y horizontal. La cantidad de píxeles que recorremos en cada dirección es equivalente al tamaño de la cuadrícula (5) multiplicado por la escala del carácter. Así pues, un carácter con escala 1 tendrá un grosor de línea de un píxel y ocupará un espacio de 5x5 píxeles mientras que un carácter con escala 2 tendrá 2 píxeles de grosor de línea y ocupará un espacio de 10x10 píxeles.

Tras esto hacemos una conversión sencilla para hallar, dadas unas coordenadas cualesquiera dentro del bucle, las coordenadas de la cuadrícula que le corresponden. Una vez halladas, se comprueba la posición de la cuadrícula. Si es un espacio en blanco (en el ejemplo [6.1] se sustituyen los espacios por barras bajas por cuestión de claridad visual) no se rellena, mientras que si esa posición de la cuadrícula no es un espacio, se pinta el píxel con el color correspondiente.

Código 6.3: Método que renderiza una cadena de caracteres

```

1 void ClassicDemoTemplate::RenderText(const char *text, int posX, int posY, int scale, const Pixel &colour)
2 {
3     std::string txt(text);
4     for (auto &c : txt)
5     {
6         c = toupper(c);
7     }
8
9     for (auto c : txt)
10    {
11         RenderCharacter(c, posX, posY, scale, colour);
12         posX += 6 * scale;
13     }
14 }
```

El método para dibujar texto es también muy sencillo, como se puede ver en el ejemplo [6.3]. Lo primero que hacemos es poner todos los caracteres en mayúscula, pues como ya habíamos decidido antes, es más práctico tener un único conjunto de letras. A continuación, por cada carácter que forma el texto invocamos a la función de dibujado de carácter [6.2]. Por cada dibujado, aumentamos la posición horizontal, de modo que el próximo carácter se dibuje a 6 unidades de distancia del inicio del carácter anterior (es decir, se deja una unidad de 1 espacio entre uno y otro carácter, ya que un carácter ocupa 5 unidades).

6.2.4 Dibujar puntos

La necesidad de dibujar puntos en el motor gráfico no llegó hasta un punto algo más avanzado del desarrollo. Al fin y al cabo, dibujar un punto de una unidad de tamaño en pantalla equivale a dibujar un píxel, y para dibujar un píxel, basta con acceder a nuestra textura y modificar los valores deseados.

Argumentablemente se podría decir que mediante un control en el dibujado de puntos podemos asegurar que nuestro programa nunca pinte fuera de pantalla, pudiendo provocar errores, pero nuevamente, si podemos asegurar un entorno controlado en el que sabemos que ningún píxel estará fuera de pantalla, hacer esta comprobación es redundante, una pérdida de rendimiento para nada.

Sin embargo, todo esto cambia cuando consideramos que un punto no es necesariamente un píxel. Un punto de dos unidades ocupará $2 \times 2 = 4$ píxeles, y un punto de tres unidades ocupará $3 \times 3 = 9$ píxeles, si bien a nivel conceptual sigue siendo un punto. Y en este momento se presenta otra pregunta, ¿qué pasa si un punto de por ejemplo 4 unidades tiene solo medio "cuerpo" en pantalla? Lo lógico sería que se viera la parte del mismo que está en pantalla. Sin embargo, si no controlamos los límites de dibujado, esto puede llevar a comportamientos inesperados (al intentar volcar datos fuera de la memoria de la textura, pudiendo reescribir datos cualesquiera).

Código 6.4: Método para dibujar puntos

```

1 void ClassicDemoTemplate::RenderDot(int x, int y, const Pixel &colour, int dotSize)
2 {
```

```

3   for (int i = 0; i < dotSize; i++)
4   {
5       for (int j = 0; j < dotSize; j++)
6       {
7           int offsetX = x + i;
8           int offsetY = y + j;
9
10          if (IsPixelOutOfBounds(offsetX, offsetY))
11          {
12              continue;
13          }
14
15          screen[offsetY * width + offsetX] = colour;
16      }
17  }
18}

```

Es en este momento cuando se crea un método que permita el dibujado de puntos en pantalla [6.4] de forma controlada y segura. Este método permite dibujar puntos de cualquier tamaño con la garantía de que sólo se accederá a memoria dentro de pantalla.

Pero el uso de este método es nuevamente un compromiso. A cambio de tener la posibilidad de tener puntos con escala y que no se dibujan fuera de pantalla, se compromete la eficiencia, pues una operación de una sola línea [15] pasa a convertirse en un método de complejidad cuadrática y con comprobaciones que interrumpen un flujo de ejecución lineal. Es por ello que esta funcionalidad sólo debe usarse cuando tenga sentido, es útil si queremos pintar unos pocos puntos de tamaño variable o que pueden estar dentro o fuera de la pantalla, pero no tendría sentido utilizar este método si se requiere repintar toda la pantalla píxel a píxel.

6.2.5 Dibujar rectángulos

El dibujado de rectángulos en pantalla es una funcionalidad especialmente útil para el borrado de pantalla o de ciertas áreas de la misma.

Código 6.5: Métodos de repintado en pantalla

```

1 void ClearScreen(const Pixel &colour);
2
3 void ClearScreen(int x1, int y1, int x2, int y2, const Pixel &colour);

```

ClassicDemoTemplate provee dos funciones de dibujado diferentes [6.5], llenar toda la pantalla dado un color o llenado de una subsección de la pantalla.

Para llenar una subsección de la pantalla, es necesario aportar, además de un color, las coordenadas de dos puntos que representen la esquina superior izquierda del rectángulo y la esquina inferior derecha. Este método comprueba la validez de la coordenadas pasadas y si alguna de las coordenadas está fuera de pantalla, pinta la sección correspondiente, pero sin sobrepasar los límites. El uso de esta segunda función se recomienda siempre que se pueda frente al redibujado completo de toda la pantalla, pues si bien recorrer todos los píxeles en pantalla es una operación sencilla, es costosa a nivel temporal.

6.2.6 Dibujar líneas

El dibujado de líneas no fue necesario hasta un estado avanzado del desarrollo, cuando para poder representar modelos geométricos era como mínimo necesario poder dibujar sus aristas mediante líneas.

El código para dibujar líneas lo reutilicé de un proyecto anterior que yo mismo había desarrollado³. Este código, a su vez, estaba inspirado en el algoritmo de pintado de líneas de Bresenham⁴, si bien era ligeramente distinto en su implementación.

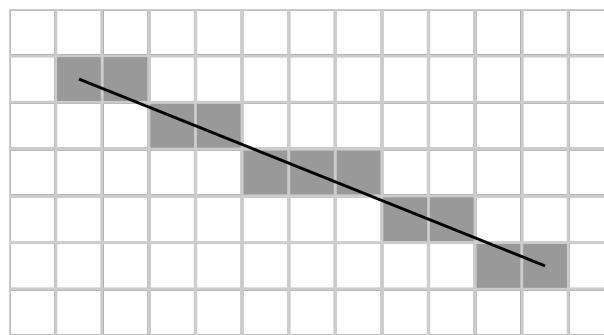


Figura 6.4: Pintado de líneas en pantalla (adaptación de una línea a una cuadrícula) - Fuente: Wikipedia

En esencia, esta es la lógica que sigue el código de pintado:

- Dados el punto de inicio y el punto de fin de la recta, calcular la pendiente de la recta a partir de los mismos.
- Si la pendiente es inferior a 45° , iterar desde el punto de inicio de la recta hasta el punto final, incrementando siempre la posición horizontal e incrementando sólo la posición vertical cuando el error acumulado es mayor que uno.
- Si la pendiente es superior a 45° , iterar desde el punto de inicio de la recta hasta el punto final, incrementando siempre la posición vertical e incrementando sólo la posición horizontal cuando el error acumulado es mayor que uno.

Código 6.6: Versión simplificada y reducida del código para dibujar líneas en pantalla

```

1 void RenderLine(int x1, int y1, int x2, int y2, const Pixel &colour)
2 {
3     float slope = GetSlope(x1, y1, x2, y2);
4
5     //Slope < 45°
6     if (slope <= 1.f)
7     {
8         DrawLineWithSmallSlope(x1, y1, x2, y2, colour, slope);
9     }
10    //Slope > 45°

```

³<https://github.com/donluispanis/PaintLike>

⁴https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

```

11     else
12     {
13         DrawLineWithBigSlope(x1, y1, x2, y2, colour, slope);
14     }
15 }
16
17 void DrawLineWithSmallSlope(int x1, int y1, int x2, int y2, const Pixel &colour, float slope)
18 {
19     float accumulatedError = 0.f;
20     int auxX = x1, auxY = y1; //auxiliar point that we will increment from the old point to the new one
21
22     int signX, signY;
23     GetSigns(x1, y1, x2, y2, signX, signY);
24
25     //While we don't reach the desired pixel position
26     while (auxX != x2 || auxY != y2)
27     {
28
29         RenderDot(auxX, auxY, colour);
30
31         auxX += signX; //Increment X until it reaches the target
32
33         accumulatedError += Fast::Abs(slope); //When this reaches 1, we increment Y
34
35         if (accumulatedError >= 1.f)
36         {
37             auxY += signY;
38             accumulatedError -= 1.f;
39         }
40     }
41 }
```

El código que se muestra en [6.6] es tan solo una versión simplificada, reducida y no funcional de la implementación en código del algoritmo para el dibujado de rectas en pantalla, pero que pretende ser suficiente para ejemplificar cómo se traduce la lógica anteriormente descrita al código.

Como se ha comentado previamente, este código se inspira en el algoritmo de Bresenham, pero no lo sigue. Una de las mayores diferencias es que el algoritmo de Bresenham sólo utilizaba números enteros, pues cuando fue desarrollado, las operaciones con números en coma flotante eran lentas y se realizaban por *software*. Hoy en día, no obstante, se realizan por *hardware*, y no suponen en muchos casos un coste significativo con respecto a las operaciones con enteros.

Además, según las necesidades han ido variando, se ha ido añadiendo funcionalidad adicional al pintado de líneas:

- Posibilidad de definir un grosor de línea
- Posibilidad de definir un color de inicio y de final, sobre los que se interpola (dando una sensación de degradado)

7 Efectos clásicos

7.1 Fuego

7.1.1 Investigación inicial

Una demo que simule un efecto de fuego se podría considerar algo así como el "hola mundo" de la demoscene. Es un ejercicio sencillo con un resultado final bastante espectacular.

Tras una búsqueda de información inicial, pude encontrar también dos sitios diferentes en los que se explicaba cómo crear un efecto de fuego.

En el canal de YouTube de Creature Mann¹ se explica la base teórica para crear un efecto de fuego sencillo². A este vídeo le siguen un par de vídeos de este mismo creador³⁴ en los que itera sobre el efecto anteriormente creado, añadiendo complejidad (como la posibilidad de controlar la dirección del fuego o trazar un camino que se prende fuego). Por desgracia, los enlaces provistos al código que estos vídeos muestran están caídos, por lo que el código no es accesible. No obstante, la parte argumentablemente más importante, la explicación teórica del efecto, se hace en el primer vídeo.

Otra página que ofrece una descripción muy buena del efecto es Lode's Computer Graphics Tutorial⁵. Esta página sí que aporta código, aunque decidí ignorar la implementación (para evitar que condicionara mi propio desarrollo) y centrarme únicamente en la explicación teórica que se ofrecía, muy similar a la del vídeo anterior aunque más técnica.

7.1.2 Planteamiento formal

El fuego es un efecto muy sencillo tanto a nivel teórico como de implementación. Consiste en la convolución de una matriz como la de la figura [7.1] a lo largo de una matriz que tan sólo contiene valores en su fila inferior. Al aplicar esta operación de abajo a arriba, se obtiene un conjunto de valores [7.2a] que al ser asociados a un set concreto de colores [7.2b], dan una sensación similar al fuego.

Otra forma de entender esta operación es la siguiente: el valor de cada píxel se deduce de la media del valor de los tres píxeles adyacentes por debajo de él. Para que este efecto se

¹<https://www.youtube.com/user/kjlg74/featured>

²https://www.youtube.com/watch?v=_SzpMB0p1mE

³<https://www.youtube.com/watch?v=iezD8B1ym3w>

⁴<https://www.youtube.com/watch?v=206TEPB0nLc>

⁵<https://lodev.org/cgtutor/fire.html>

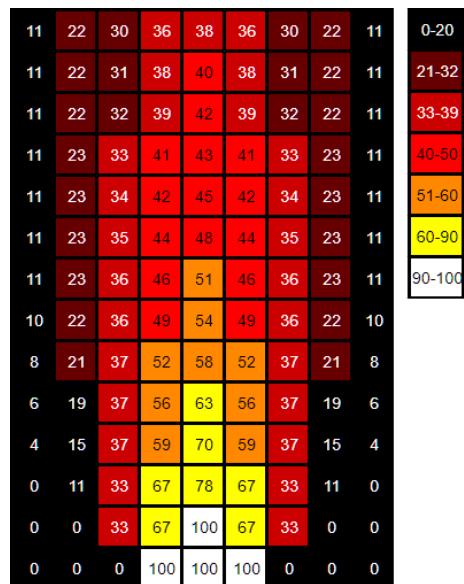
$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix} \quad (7.1)$$

Figura 7.1: Matriz de convolución para generar efecto de fuego

produzca de forma efectiva, la fila inferior de píxeles se suele llenar con valores aleatorios. De esta forma, la tendencia natural de este modelo es la de disiparse.

El único caso en el que no se produciría disipación sería en el que todos los valores de la fila inferior se inicializaran al máximo (en cuyo caso todos los píxeles acabarían con el mismo valor). Para cualquier otra situación, se produce una atenuación progresiva de los valores.

11	22	30	36	38	36	30	22	11
11	22	31	38	40	38	31	22	11
11	22	32	39	42	39	32	22	11
11	23	33	41	43	41	33	23	11
11	23	34	42	45	42	34	23	11
11	23	35	44	48	44	35	23	11
11	23	36	46	51	46	36	23	11
10	22	36	49	54	49	36	22	10
8	21	37	52	58	52	37	21	8
6	19	37	56	63	56	37	19	6
4	15	37	59	70	59	37	15	4
0	11	33	67	78	67	33	11	0
0	0	33	67	100	67	33	0	0
0	0	0	100	100	100	0	0	0



(a) Valores resultantes tras convolucionar iterativamente la matriz [7.1] por una matriz de ceros, con valores solo en la fila inferior

(b) Efecto resultante de asociar determinados rangos de valores a un set de colores preestablecido

Dado el comportamiento descrito, será necesario realizar los siguientes pasos:

- Implementar una forma de realizar degradados (o mapas de color)
- Reservar e inicializar una matriz a cero, con valores aleatorios en su fila inferior
- Implementar el comportamiento de convolución de la matriz [7.1]

7.1.3 Implementación

Para poder crear degradados, se crea la función *GenerateGradient* que dado un conjunto de *ColourStamp* y un tamaño, interpola los valores de colores pasados para generar un de-

gradado continuo en *colourMap*.

Código 7.1: Método para crear gradientes de color

```
1 static void GenerateGradient(std::vector<ColourStamp> colours, Pixel* colourMap, int colourMapSize);
```

ColourStamp (marca de color) es una estructura formada por dos variables: un color y un número decimal (que puede oscilar entre 0 y 1). Este número señala la posición de este color en el gradiente o mapa de color, siendo 0 el inicio y 1 el final.

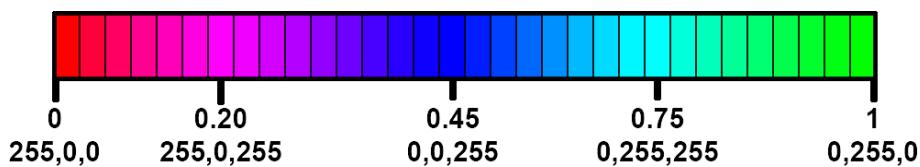


Figura 7.3: Degradado en 32 celdas dadas 5 marcas de color

De este modo, si llamamos a la función *GenerateGradient* con los valores para las marcas de color representados en la figura [7.3] y pasamos un *array* con capacidad para 32 valores, este método producirá un set de colores similar al ilustrado. Esto se hace mediante una interpolación lineal entre los valores que se pasan, creando un degradado de forma progresiva.

Una vez tenemos implementado este método, ya podemos crear un degradado de colores de forma sencilla y cómoda, automatizando el proceso de crear un mapa de color para el fuego.

Para nuestra implementación, en lugar de asignar a un rango de valores un color específico, asignaremos un color por valor entero. Esto permitirá un resultado más realista, al disponer de una mayor cantidad de colores. Cada celda de nuestra matriz de valores estará formada por un byte. Esto implica que cada celda podrá tener 256 valores distintos, y que por tanto necesitaremos generar un degradado que nos devuelva 256 colores.

Creamos un mapa de valores de un byte de tamaño, lo inicializamos a 0 e inicializamos aleatoriamente algunas de las celdas de la fila inferior a su valor máximo (255).

Código 7.2: Creación e inicialización del mapa de valores

```
1 screenMapping = new unsigned char[width * height];
2
3 for (int i = width * (height - 1), n = width * height; i < n; i++)
4 {
5     if (Fast::Rand() % 10 == 0)
6     {
7         screenMapping[i] = 255;
8     }
9 }
```

Como podemos ver en la línea [5], hacemos uso de una función propia para la generación de números aleatorios. Esta función está basada en el algoritmo *Xorshift*⁶ de George Marsaglia.

⁶<https://en.wikipedia.org/wiki/Xorshift>

La aproximación usada está basada en una respuesta de *StackOverflow*⁷.

La decisión de usar nuestro propio generador de números aleatorios se debe a que el usualmente provisto por la STL⁸ es innecesariamente lento y complejo para nuestras necesidades. Para nuestro caso, no necesitamos un algoritmo que pase todos los tests de aleatoriedad⁹, con tal de que sea suficientemente "aleatorio al ojo" y sea rápido, nos basta.

Una vez hecho esto, tan sólo queda implementar el algoritmo principal, que cree el efecto de fuego sobre el mapa de valores previamente creado, de acorde a la operación descrita en la figura [7.1] y asocie dichos valores con un color generado por la función *GenerateGradient* [7.1].

Código 7.3: Algoritmo básico de efecto de fuego

```

1 for (int i = width * (height - 1); i >= 0; i--)
2 {
3     int lowerCell = width + i;
4     int newCellValue = screenMapping[i] = (screenMapping[lowerCell + 1] + screenMapping[lowerCell] + ↵
5         ↵ screenMapping[lowerCell - 1]) / 3.0;
6     pixels[i] = colourMap[newCellValue];
7 }
```

En el código [7.3] podemos ver el algoritmo de generación de fuego en su forma más simple. Recorremos la pantalla de abajo a arriba y operamos por cada píxel. En la línea [3] obtenemos, para una posición dada en el mapa de valores, la posición del valor inmediatamente por debajo del mismo. Una vez obtenida esta posición, operamos haciendo la media usando su valor y los adyacentes. Guardamos el resultado como nuevo valor y usamos a la vez este nuevo valor para asociar un nuevo color al píxel en pantalla (línea [5]). Los valores más altos (255) representan tonos blancos y/o amarillentos, mientras que los valores intermedios representarán valores rojizos y los valores más bajos tendrán asociados colores más oscuros.

El resultado de aplicar este algoritmo resulta en una imagen estática y de aspecto poco realista [7.4], pero que ya se empieza a parecer al efecto que buscamos.

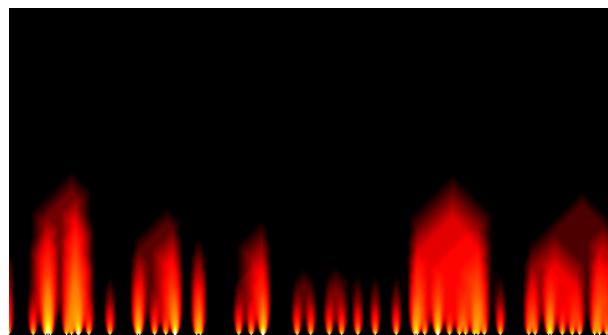


Figura 7.4: Fuego estático, usando el algoritmo en [7.3]

⁷<https://stackoverflow.com/questions/1640258/need-a-fast-random-generator-for-c>

⁸https://en.wikipedia.org/wiki/Standard_Template_Library

⁹https://es.wikipedia.org/wiki/Pruebas_de_aleatoriedad

7.1.4 Refinamiento

Una vez tenemos el efecto de fuego a nivel básico, es el momento de iterar sobre la idea y ver cómo mejorarla. A continuación se listan y explican las medidas tomadas, mostradas en el orden en que se aplicaron:

- **Hacer fuego dinámico:** una vez obtenido un fuego estático, era el momento de darle movimiento, y que tuviera un efecto realista. Inicialmente probé con una técnica que se sugería en algunos de los tutoriales que había seguido: en lugar de dibujar el fuego de abajo a arriba, dibujarlo de arriba a abajo y aleatorizar la base, de forma que las variaciones en el fuego derivaran a partir de las variaciones en la base.

El resultado no me dejó convencido. Cuando uno observa un fuego o una llama, la parte más cambiante del fuego no es la base, la base es siempre estable y es la parte superior la que más titila / oscila. Por tanto no tenía para mí sentido generar dinamismo aleatorizando la base.

Una llama se caracteriza a menudo por tener una intensidad variante, y fue esto lo que me decidí por implementar: la base permanecería estable, la llama tendría un factor de aleatoriedad.

Tras un ensayo de prueba y error ajustando valores, y teniendo que retocar la posición y tono de los colores en el gradiente, se llegó al siguiente código (`Fast::Rand()%4 ↪ ↪ == 0 ? 2 : 0`) que se puede ver aplicado en [7.4]. Básicamente este código altera levemente el valor de intensidad del píxel de forma aleatoria, con un 25% de posibilidades de que el valor del píxel se vea incrementado.

El resultado puede verse en [7.5]. La base sobre la que se aplica es la misma que en la figura [7.4], sin embargo, el resultado resulta más convincente / natural gracias a que se introduce una pequeña aleatoriedad en la intensidad del píxel, introduciendo cierta dispersión.

- **Más colores:** una vez tenía el fuego básico creado llegó el momento de ponerse creativo y añadir más degradados, que pudieran ser aplicados para crear fuegos de distintos colores. Al degradado de fuego básico (blanco - amarillo - rojo - negro) se le añadieron dos nuevos degradados, un fuego estilo neón (rosa - verde - azul - negro) y un degradado en blanco y negro (blanco - gris - negro).

Además, el código para generar degradados, que inicialmente estaba en el mismo archivo que el fuego, se separó a su propio archivo y clase. Por último, y en vistas de que el código para crear un degradado ocupaba mucho espacio (debido a la necesidad de definir las marcas de color) y que potencialmente sería reutilizado por otros efectos que usasen degradados, el código se movió a una clase común con inicialización estática. De este modo, desde el inicio de la ejecución están disponibles los vectores de marcas de color necesarios para generar distintos patrones (fuego, neón, blanco y negro, arcoiris...) mediante la función `GenerateGradient`.

- **Manipulación del fuego:** una vez teníamos distintos colores, era necesario poder cambiar entre ellos. Fue en este momento cuando se incorporó al motor la capacidad de gestionar entradas de teclado.
-

Una implementada esta funcionalidad, lo que se hizo fue, al inicio de la ejecución del programa, crear un vector contenido patrones de degradado. Luego, al pulsarse una tecla determinada, se actualiza el patrón de degradado en uso al siguiente en el vector.

También se añadió la posibilidad de cambiar levemente la intensidad del fuego, añadiendo la variable *fireIntensity*, cuya aplicación se puede ver en [7.5].

Código 7.4: Algoritmo final de efecto de fuego

```

1 for (int i = width * (height - 1); i >= 0; i--)
2 {
3     int sum = width + i;
4     sum = screenMapping[i] = (screenMapping[sum + 1] + screenMapping[sum] + screenMapping[sum - 1]) / ←
        ↪ (3.03 + fireIntensity) + (Fast::Rand() % 4 == 0 ? 2 : 0);
5     pixels[i] = colourMap[sum];
6 }
```

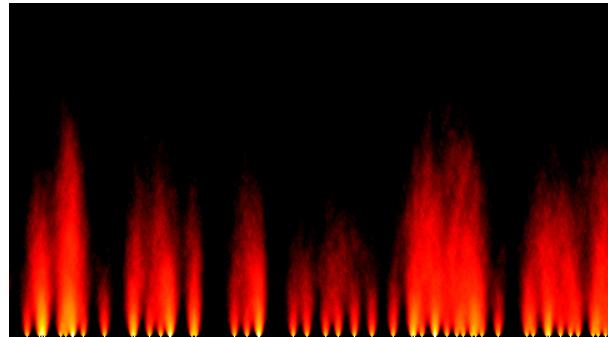
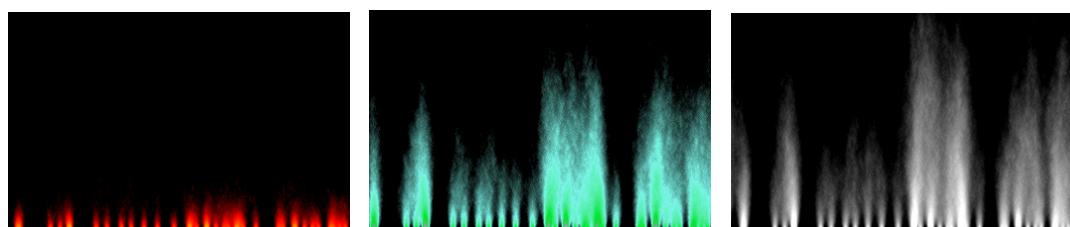


Figura 7.5: Fuego dinámico, con intensidad por píxel aleatorizada

7.1.5 Resultado

A continuación se presenta el resultado final del efecto de fuego: un fuego dinámico, de corte y comportamiento realista, con la posibilidad de cambiar su color y su intensidad.



(a) Fuego rojo con intensidad mínima (b) Fuego neón con intensidad media (c) Fuego en blanco y negro con intensidad máxima

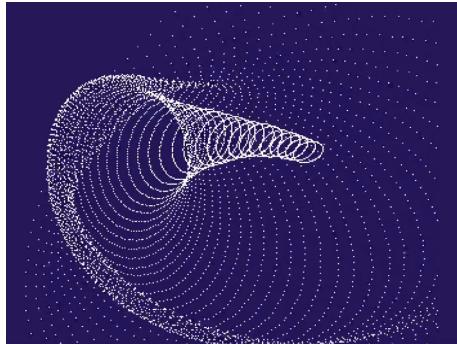
7.2 Túnel de puntos

7.2.1 Investigación inicial

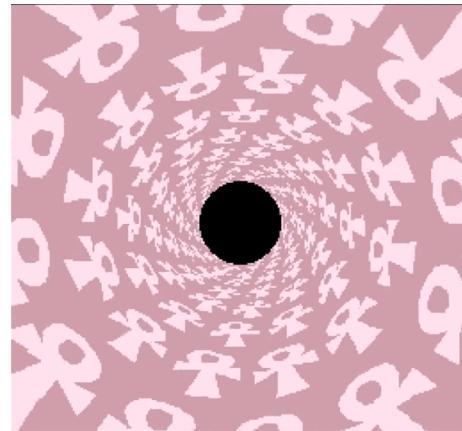
Un efecto también muy común en el mundo de la *demoscene* es el efecto de túnel o vórtice, entre otras causas por su relativa sencillez sumada a su espectacularidad (por la sensación de profundidad y de dinamismo, evocando a escenas futuristas o situadas en el espacio).

Es por ello que el efecto de túnel parecía un candidato perfecto para ser el segundo efecto a implementar; más complejo que el efecto de fuego pero aún así sencillo, y con un resultado visual más complejo.

Una vez decidido, llegó el momento de recabar información acerca de este efecto y cómo implementarlo. Mi idea inicial era generar un túnel de puntos como el de la figura [7.7a], sin embargo, conforme fui ahondando en mi búsqueda, descubrí que también era un efecto bastante común generar túneles como el que se muestra en la figura [7.7b].



(a) Túnel de puntos - Cyberdance (por Vir-
tual Dreams y Fairlight, 1993) - Fuente:
YouTube



(b) Túnel mediante deformación de textura
- D.A.N.E (por Kefrens, 1993) - Fuente:
YouTube

De hecho, de cara a la búsqueda de explicaciones teóricas y detalles de implementación, fue más fácil encontrar información acerca del efecto mostrado en [7.7b] que del túnel de puntos. Páginas como benryves.com o lodev.org ofrecían tutoriales detallados, en los que se explica paso a paso la base matemática del efecto así como su implementación en código.

En resumen, el efecto que se muestra en la figura [7.7b] es el resultado de deformar una imagen o una textura de forma que toda la textura tienda hacia un punto central, de modo que se produce una textura circular a partir de un patrón plano. Se deforma la imagen aplicando algo de trigonometría básica. Además, como en el ejemplo de lodev.org, se pueden usar tablas precalculadas para así evitar tener que realizar operaciones trigonométricas complejas y/o lentas de forma repetida.

No fui, sin embargo, capaz de encontrar tutoriales o detalles de implementación para lograr el efecto de la figura [7.7a]. Tras una búsqueda a conciencia con resultado infructuoso, me decidí por implementar este efecto. Mi objetivo con este trabajo no es el de seguir tutoriales ya existentes, si no el de crear efectos visuales partiendo de cero, y guiado por la intuición y la razón. No tiene sentido alguno que trate de implementar un efecto de túnel basado en una textura cuando ya hay tutoriales que desgranan (con detalle y acierto) cómo hacerlo, tanto a nivel matemático como de código.

Por tanto, resolví por implementar el efecto de túnel de puntos.

7.2.2 Planteamiento formal

¿Cómo se consigue el efecto de generar un túnel de puntos en movimiento?

La respuesta en realidad es bastante sencilla si observamos con atención cualquier demo que implemente este efecto: consiste en la superposición de circunferencias de distintos tamaños y en distintas posiciones.

El dibujado de estas circunferencias (o elipses en el caso de la figura [7.7a]) se simplifica a dibujar solo varios puntos pertenecientes a la circunferencia, en lugar de dibujar todo el perímetro. De este modo se consigue un tiempo de dibujado controlable y constante (si decidimos que una circunferencia será representada mediante 16 puntos, se usarán 16 puntos ya sea el radio de la circunferencia 50, 100 o 200 píxeles, de modo que aunque el perímetro aumente, la esfera es representada con una cantidad de puntos constante -eso sí, cada vez más separados entre sí-).

Por tanto, será necesario implementar las siguientes funcionalidades:

- Capacidad para dibujar un círculo dadas una posición, un radio y por cuantos puntos debe estar formado.
- Capacidad para mantener un conjunto de circunferencias simultáneamente
- Capacidad para gestionar el ciclo de vida de una circunferencia
- Implementar un mecanismo mediante el que el túnel siga una ruta de apariencia natural y fluida

7.2.3 Implementación

Empezando por el principio, era necesario poder dibujar circunferencias en pantalla. Estos círculos, además, debían poder variar su posición y tamaño a lo largo del tiempo, por lo que tenía sentido crear una estructura que los representara [7.8].

Como podemos ver en el código [7.5], para dibujar una circunferencia, lo que hacemos es ir de 0 a $2 * \pi$. Esto es trazar una circunferencia completa en radianes. Definimos, no obstante, un incremento variable, que depende de la cantidad de puntos que queremos dibujar. De este modo, si queremos dibujar una circunferencia de 4 puntos, este incremento será

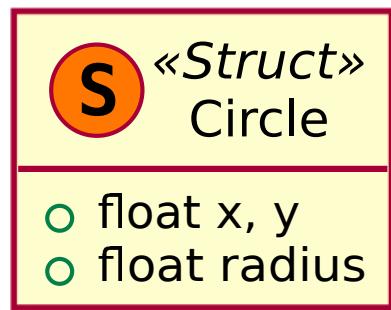


Figura 7.8: Estructura básica de un círculo

$(2 \times \pi) \div 4 = \frac{\pi}{2}$, o lo que es lo mismo, dibujaremos de cuarto en cuarto de circunferencia, dibujando así 4 puntos para completar una circunferencia completa.

La operación para determinar la posición del punto es muy básica, e implica simplemente un poco de trigonometría. La posición de cada punto de la circunferencia viene determinada por el ángulo (por el coseno del mismo para la coordenada horizontal y por el seno para la coordenada vertical) multiplicado por el radio y sumada la posición del centro de la circunferencia.

Una vez obtenida la posición para un punto de la circunferencia, lo dibujamos usando nuestra función del motor para dibujar puntos, que se encarga además de comprobar que el punto esté dentro de los límites de pantalla, y nos permite además pasarle parámetros para el color y el tamaño del punto (blanco y de un píxel para el ejemplo).

Código 7.5: Algoritmo básico de dibujado de circunferencias

```

1 void DotTunnelDemo::DrawCircle(const Circle &c)
2 {
3     const float increment = (2 * Fast::PI) / float(pointsPerCircle);
4     int x, y;
5
6     for (float angle = 0, n = 2 * Fast::PI; angle < n; angle += increment)
7     {
8         x = cos(angle) * c.radius + c.x;
9         y = sin(angle) * c.radius + c.y;
10        RenderDot(x, y, Pixel(255), 1);
11    }
12}
13}

```

Una vez podemos dibujar círculos, llega el momento de poder gestionar una serie de círculos y su ciclo de vida (creación, actualizaciones periódicas con radio creciente y destrucción alcanzado un cierto tamaño).

Para ello primero debemos preguntarnos, ¿cómo se comporta nuestro túnel? La respuesta es que para la estructura del túnel, lo que queremos es ir añadiendo nuevos círculos al principio

del túnel y vamos eliminando los círculos que están al final del túnel cuando alcanzan cierto tamaño. Así pues, esta es una estructura FIFO (*first-in, first-out*) o el primer elemento en crearse es el primer elemento en borrarse. Podríamos pensar que con usar una estructura de cola (`std::queue`¹⁰) nos bastaría, sin embargo, una cola sólo permite crear un nuevo elemento al final de la estructura y borrarlo al principio, y además no permite el acceso a los elementos intermedios (que nosotros necesitamos para poder actualizarlos).

Por suerte, no obstante, hay una estructura similar que cumple todos los requisitos que necesitamos: `std::deque`¹¹ (*double-ended queue*). Aunque el nombre puede resultar algo extraño (cola con doble final), esta estructura satisface coste constante para todas las operaciones que necesitamos! El coste de la inserción y borrado de elementos son constantes al principio y al final de la cola, y además el acceso aleatorio (acceso a un elemento cualquiera de la cola) es también constante. Por tanto, usaremos esta estructura para contener los círculos que formarán nuestro túnel.

Código 7.6: Inserción y eliminación de círculos

```

1 void DotTunnelDemo::PopulateCircleQueue()
2 {
3     if (circles.front().radius > minCircleRadius)
4     {
5         circles.push_front(defaultCircle);
6     }
7     if (circles.back().radius > maxCircleRadius)
8     {
9         circles.pop_back();
10    }
11 }
```

Como podemos ver en el código [7.6], usando esta estructura es muy sencillo añadir y eliminar elementos de nuestro túnel. La lógica que seguimos es: si el último círculo que ha sido añadido alcanza cierto tamaño, entonces añadimos un nuevo círculo al túnel. Del mismo modo, cuando el círculo que más tiempo lleva en la cola alcanza cierto tamaño, es eliminado. Cabe notar que para añadir nuevas circunferencias a la cola, se hace una copia de una instancia que creamos en la inicialización del programa, y que contiene los valores de creación de un círculo por defecto. Además, para que este código funcione correctamente, debe haber al menos un elemento previamente insertado en la cola, es decir, no puede estar vacía. Es por eso que durante la inicialización de la demo también se añade un círculo a la cola, copiado de la instancia por defecto.

Código 7.7: Actualización del túnel

```

1 void DotTunnelDemo::UpdateCircleQueue(float deltaTime)
2 {
3     for (auto c : circles)
4     {
5         EraseCircle(c);
6     }
7 }
```

¹⁰<https://en.cppreference.com/w/cpp/container/queue>

¹¹<https://en.cppreference.com/w/cpp/container/deque>

```

8   PopulateCircleQueue();
9
10  for (auto &c : circles)
11  {
12      UpdateCircle(c, deltaTime);
13      DrawCircle(c);
14  }
15 }
```

Una vez que podemos dibujar círculos [7.5] y tenemos una estructura que representa nuestro túnel sobre la que podemos insertar y eliminar elementos [7.6], llega el momento de dibujar nuestro túnel en pantalla, el proceso es bastante sencillo [7.7].

Lo primero que hacemos es recorrer todos los círculos que conforman el túnel y borrarlos en pantalla, en la línea [5]. Es importante notar que esta función no está eliminando los círculos de la cola, si no que simplemente *borra en pantalla*. Internamente esta función llama a la función de dibujado [7.5] pero con una copia del círculo en color negro. Así, lo que hacemos al inicio de cada actualización es borrar los círculos *que se dibujaron en el fotograma anterior*, previo a la actualización de los valores de los círculos. De este modo, en lugar de tener que poner a negro todos píxeles de la pantalla (que en alta definición son más de un millón), ponemos a negro sólo aquellos píxeles que fueron modificados en el fotograma anterior (que son cientos de píxeles, pero no miles o millones). Al borrar de este modo, se produce una gran optimización.

Tras haber borrado la pantalla, llamamos a la función [7.6], que puede añadir o eliminar círculos del túnel si se cumplen las condiciones necesarias.

A continuación, actualizamos (línea [12]) y redibujamos todos los círculos en pantalla. Actualmente, la actualización de un píxel es un proceso muy sencillo que consiste simplemente en ir aumentando el radio de cada círculo a lo largo del tiempo, en función de su radio anterior y una velocidad ajustable por el usuario (`c.radius += c.radius * deltaTime * ↪ radiusVelocity;`).{

El resultado de todo lo aplicado hasta ahora se puede ver en la figura [7.9].

Ahora todo lo que nos queda por hacer es dotar de movimiento al túnel, es decir, que no sólo los círculos aumenten de radio, si no que su posición también varíe. No obstante, no es necesario modificar la posición de los círculos una vez han sido creados, basta con que el centro de cada círculo esté desplazado en cuanto a posición con respecto al círculo anterior en su momento de creación. En otras palabras, cada círculo se crea en una posición distinta con respecto al punto de creación del círculo anterior, pero el centro del círculo no se modifica posterior a su creación. Aunque pueda resultar curioso, no es necesario mover el centro del círculo una vez creado, basta con ir haciendo el radio progresivamente más grande para crear una sensación de movimiento convincente.

Así pues, tan solo necesitamos crear cada círculo con una posición distinta pero coherente con respecto a la anterior, para dar sensación de continuidad. ¿Cómo podemos hacer esto?

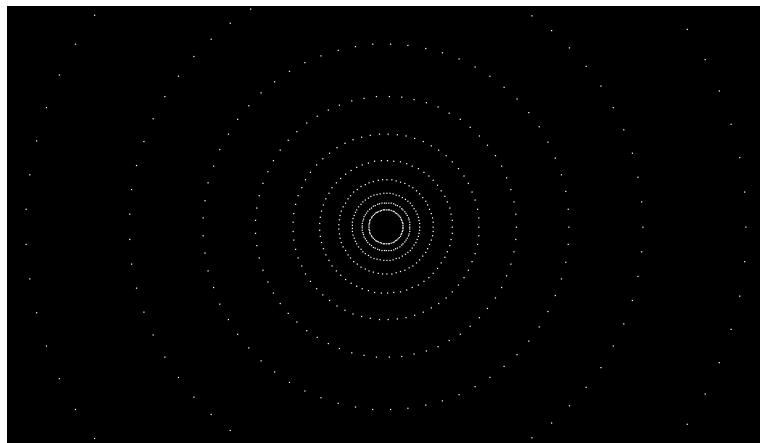


Figura 7.9: Túnel de puntos básico

No podemos decidir las direcciones de forma aleatoria, pues de este modo el movimiento no parecerá coherente. Cabe la posibilidad, no obstante, de seguir direcciones que se deciden aleatoriamente cada cierto tiempo, pero que se mantienen fijas durante un intervalo. Si hacemos esto el movimiento tendría coherencia pero aún así parecería poco natural cada vez que cambiáramos de dirección, pudiéndose producir cambios de dirección bruscos. Una posibilidad ante esto es interpolar la dirección actual con la dirección futura, de modo que los cambios de dirección queden suavizados. Pero aun así surge un problema más, no podemos controlar si el túnel se sale de los límites de la pantalla. Podríamos hacer entonces que si el túnel está cerca del límite de la pantalla, la nueva dirección elegida fuera hacia el centro de la pantalla. Aunque esto podría causar sensación de que cuando el túnel se acerca al límite de la pantalla rebota... Podríamos seguir con este modelo, a partir de direcciones o movimientos aleatorios, pues es factible, pero el código y la cantidad de situaciones con las que hay que lidiar para que el resultado parezca natural aumenta en complejidad por momentos.

Fue por ello que cuando estaba pensando en cómo implementar este efecto, descarté esta opción. Pensando en otras opciones se me ocurrió la que sería la definitiva, construir un camino a partir de un punto que gira en torno a una "órbita" virtual que a su vez "orbita" en torno a otros puntos. Un poco de forma parecida a como orbitan los planetas, que si bien lo hacen bajo rutas perfectamente definidas, las combinaciones de órbitas a distintas velocidades dan resultado a trayectorias coherentes pero difíciles de predecir desde el punto de vista terrestre [7.10].

Así, con el modelo geocéntrico en mente, me dispuse a crear una clase que fuera capaz de emular este tipo de movimiento: controlado pero difícil de predecir.

Todo lo que necesitaba era un algoritmo que me devolviera un valor numérico entre -1 y 1 (al tener un rango controlado el túnel no podría salirse de pantalla) y se calculara como el resultado de la suma de distintos puntos rotando (o como la suma de ondas con distintas frecuencias, según el punto de vista). El resultado fue un algoritmo sencillo pero satisfactorio.

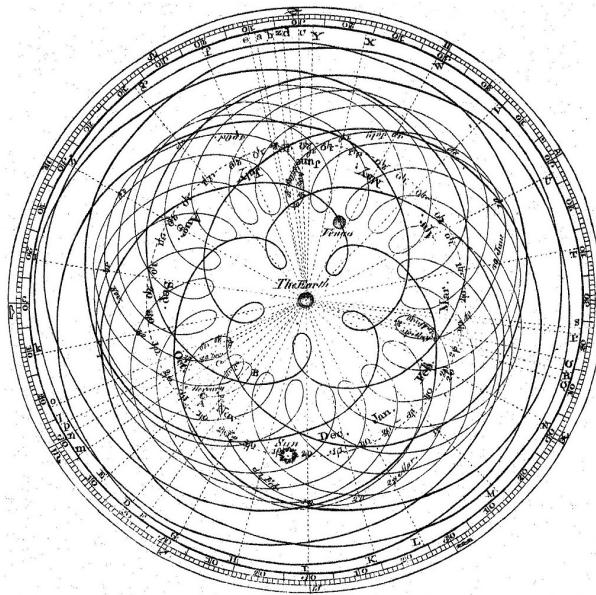


Figura 7.10: Órbitas de los planetas vistas desde la Tierra (por Giovanni Cassini) - Fuente: Wikipedia

Como podemos ver en la figura [7.11], creamos una estructura sencilla que nos permite representar un punto orbitando. Un punto orbitando se ve definido por su radio de órbita, su fase (punto inicial en el que comienza a orbitar, la dirección en la que orbita -sentido horario o antihorario- y el avance o camino actual que el punto en órbita ha recorrido. Otras características como la velocidad de órbita se definen de forma global, y no por órbita, para nuestro ejemplo, aunque la velocidad de órbita viene condicionada por el radio de órbita.

Código 7.8: Creación de un camino de turbulencia

```

1 void TurbulencePath::CreateTurbulencePath(float pathVelocity, int pathRadius, int pathComplexity)
2 {
3     this->pathVelocity = pathVelocity;
4     this->pathRadius = pathRadius;
5     this->pathComplexity = pathComplexity;
6
7     for(int i = 0; i < pathComplexity; i++)
8     {
9         paths.push_back(Path{
10             (pathRadius * 2) / (i + 2),
11             Fast::Rand() * 2 * Fast::PI,
12             i % 2 == 0 ? 1 : -1,
13             0});
14     }
15 }
```

En el código [7.8] podemos ver cómo podemos inicializar lo que definí como "camino de turbulencia". Para crear un "camino de turbulencia" necesitamos especificar la velocidad global que tendrá el camino, el mayor radio posible que el camino pueda generar, en píxeles, y la complejidad del camino, que es equivalente a la cantidad de órbitas (o ondas) superpuestas que formarán nuestro camino. En la línea [9] podemos ver cómo añadimos caminos, la cantidad que añadimos dependiendo de la complejidad del mismo. Además, el radio de cada órbita

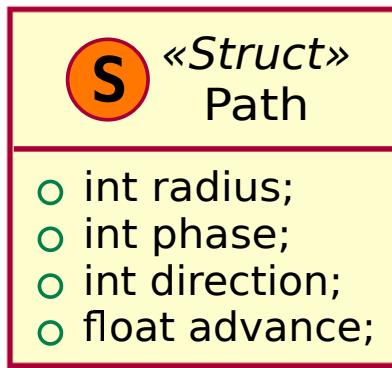


Figura 7.11: Estructura de cada órbita que determina el camino

depende de su orden (órbita 0 tendrá radio 1 ($\frac{2}{2}$), órbita 1 tendrá $\frac{2}{3}$ del radio, órbita 2 tendrá $\frac{1}{2}$ ($\frac{2}{4}$) del radio...). A continuación definimos la fase de forma aleatoria, la dirección (positiva para órbitas de orden par, negativas para impares) y el avance inicial, que es obviamente 0.

Código 7.9: Actualización de un camino de turbulencia

```

1 void TurbulencePath::UpdateTurbulencePath(float deltaTime, float &pathX, float &pathY)
2 {
3     pathX = 0.f;
4     pathY = 0.f;
5
6     for (auto& path : paths)
7     {
8         path.advance += deltaTime * 0.1;
9         float waveFrequency = path.radius * pathVelocity;
10        pathX += cos(waveFrequency * path.advance + path.phase);
11        pathY += sin(waveFrequency * path.advance + path.phase);
12    }
13
14    pathX /= (float)pathComplexity;
15    pathY /= (float)pathComplexity;
16
17    pathX *= pathRadius;
18    pathY *= pathRadius;
19}

```

Ahora llega el momento de ser capaces de actualizar nuestra estructura, como muestra el código [7.9]. A esta función le pasamos el tiempo transcurrido desde el último fotograma y nos devuelve el punto actual en el que el camino se encuentra. Para ello, por cada camino, actualizamos el avance en función del tiempo y calculamos la frecuencia en función del radio y la velocidad del camino. Una vez hemos sumado todos los caminos, dividimos entre la complejidad (línea [14]) para normalizar el resultado (de este modo aseguramos que oscila entre -1 y 1). Tras ello, tan solo nos queda multiplicar el resultado normalizado por el radio del camino para obtener el punto en el que nos hallamos actualmente (línea [17]).

Una vez tenemos nuestro "camino de turbulencia" creado, tan solo tenemos que incorpo-

rarlo en nuestro código del túnel e ir actualizándolo periódicamente. Para ello, modificamos la función `PopulateCircleQueue()` [7.6] para que cada vez que añadamos un círculo, desplazemos su centro por la posición actual del "camino de turbulencia".

Podemos ver el resultado en la figura [7.12].

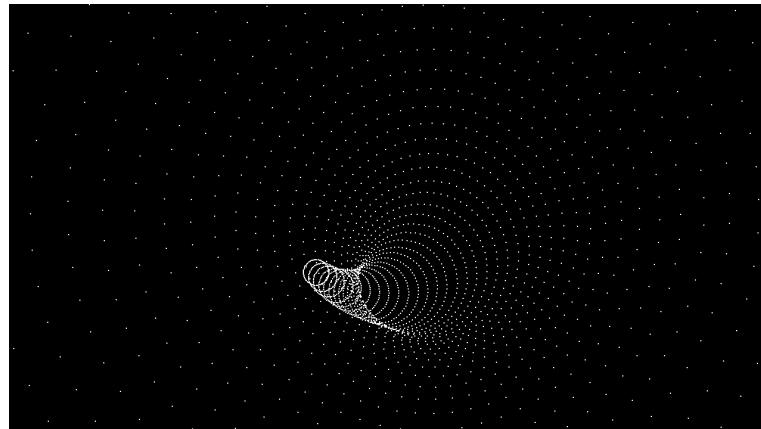


Figura 7.12: Tunel básico siguiendo un camino

7.2.4 Refinamiento

Una vez tenemos nuestro efecto de túnel acabado, llega el momento de iterar sobre la idea para mejorar el resultado, tanto a nivel visual como de rendimiento. A continuación se exponen las medidas que se tomaron:

- **Sustituir operaciones trigonométricas por tablas precalculadas:** como pudimos comprobar con los tests de rendimiento, las operaciones trigonométricas tienen un coste de computación alto. En este efecto hacemos un uso periódico de operaciones de seno y coseno, para calcular la posición de cada punto del túnel, así como para el camino de turbulencia. Si en lugar de usar funciones trigonométricas de forma directa, usamos tablas precalculadas, el coste de estas operaciones pasará a ser el de un acceso constante. No obstante, como todo, usar tablas precalculadas tiene también su coste. La principal ventaja de usar tablas precalculadas es la eficiencia, no obstante lo hacemos al coste de complejidad espacial (tenemos que guardar cada valor precalculado en memoria, y las tablas pueden llegar a ser bastante grandes), pérdida de precisión (al ser valores precalculados, solo podemos acceder a aquellos que hemos calculado, no siendo posible acceder a valores intermedios) y aumento en la complejidad del código (en lugar de pasar un valor en radianes a una función, debemos calcular a partir de un ángulo el índice de acceso a la tabla para el valor que queremos).

Podría parecer, dadas las desventajas listadas, que no vale la pena esta optimización. Esto es siempre una cuestión de interpretación y de límites, y depende de la situación y del sistema en que nos encontramos. En nuestro caso concreto, la memoria no es un limitante (disponemos de gigabytes de memoria RAM) pero sin embargo la CPU sí lo

es (tenemos que operar sobre grandes cantidades de valores en intervalos muy cortos de tiempo -cientos o miles de píxeles por fotograma-). Por tanto, vale la pena aumentar la complejidad del código a cambio de que sea notablemente más eficiente.

Como podemos ver en el código [7.10], crear una tabla precalculada es más bien sencillo. En este caso, por simplicidad se ha decidido que las tablas se calculan de forma dinámica al inicio de la ejecución de la aplicación, tomando solo así tiempo de cálculo durante la inicialización. No obstante, hubiera sido posible también implementar este mismo mecanismo mediante el uso de *templates* y *constexpr*¹². De este modo se podría hacer que las tablas se calculasen en tiempo de compilación, embebiéndose la tabla precalculada en el propio código del programa, no tomando así ningún tiempo de cálculo en ejecución. Se ha decidido crear las tablas de forma dinámica, no obstante, por simplicidad de código y reducción de los tiempos de compilación.

Así pues, en el código [7.10] vemos como para crear una tabla, tan sólo debemos pasarle un tamaño y nos devolverá una tabla del tamaño dado, conteniendo valores incrementales para el seno en una circunferencia completa (de 0 a $2 \times \pi$ radianes). Hay que tener en cuenta que la precisión de la tabla es equivalente al tamaño de la misma (a mayor tamaño, mayor precisión, pero también ocupa más espacio en memoria y tarda más en calcularse).

En la línea [14] podemos ver el resultado de usar nuestras tablas precalculadas en lugar de operaciones trigonométricas de forma directa, como en [7.5]. Como podemos ver, la complejidad del código aumenta ligeramente, aunque eso sí, a cambio de poder obtener valores de operaciones complejas en tiempo constante. Para ello, necesitamos calcular un factor (línea [17]) que nos permita marcar una correspondencia entre el ángulo que queremos y el índice que corresponde en la tabla. Luego, para acceder al valor del coseno (línea [22]), multiplicamos el valor del ángulo que queremos por el factor calculado en la línea [17]. Convertimos el resultado de esta operación en un entero (el acceso a memoria debe hacerse mediante un valor entero, pues la memoria es discreta, no se puede acceder a "medio bit"), lo que trunca el resultado y lleva a la consiguiente pérdida de precisión. A continuación hallamos el módulo en función del tamaño de la tabla. De este modo, si nos salimos del rango de la tabla, simplemente volvemos a inicio de la misma, de forma circular, por lo que no es posible de forma efectiva salirse de rango.

De este modo, conseguimos una gran optimización (operaciones trigonométricas con coste constante equivalente a un acceso aleatorio). Esta optimización puede no hacerse tan evidente en tiempo de ejecución en esta demo, donde trabajamos con cientos o miles de píxeles, pero sin llegar al orden del millón, pero será crucial en futuros efectos, para conseguir una tasa de actualización de fotogramas estable y fluida.

- **Añadir colores al túnel:** añadir colorido al túnel es muy sencillo, y sin embargo, mejora notablemente el resultado final. Basta con añadir a nuestro modelo de círculo [7.8] un campo para el color, de modo que en lugar de estar predefinido a blanco, el color con el que se dibuja cada círculo dependa del propio círculo. De este modo, podemos crear un degradado de color, como los que ya creamos para el efecto de fuego, y asignar un nuevo color a cada círculo que creamos, basándonos en los valores del

¹²<https://en.cppreference.com/w/cpp/language/constexpr>

degradado. Para este efecto en concreto, hemos decidido crear una gama de colores de efecto *arcoiris*.

- **Fundido de entrada y de salida:** en el estado actual de la demo, cuando un círculo se crea aparece de golpe y cuando un círculo se elimina desaparece de golpe. Teniendo en cuenta que actualmente el túnel es fluido tanto a nivel de movimiento (gracias al "camino de turbulencia") como a nivel de color (gracias al uso de un degradado), que los círculos aparezcan y desaparezcan de golpe rompe esta fluidez. Esto es muy sencillo de solucionar, añadiendo un fundido de entrada (opacidad creciente) cuando añadimos un nuevo círculo al túnel y un fundido de salida (opacidad decreciente) cuando estamos cerca de eliminar un círculo. De este modo, basta con crear un método que cumpla la siguiente función:
 - Cuando un círculo se añade, el valor de la opacidad es 0, y crece gradualmente hasta uno conforme el radio del círculo incrementa
 - Durante todo el ciclo de vida del círculo la opacidad es 1
 - Cuando el radio del círculo se acerca al radio máximo (radio que una vez alcanzado, el círculo es eliminado) empezamos a decrementar el valor de la opacidad, de modo que sea 0 cuando el círculo sea eliminado.

De este modo, tan solo tenemos que multiplicar el valor de la opacidad por el color del círculo para conseguir nuestros fundidos de entrada y de salida, suavizando así la creación y eliminación de los círculos.

- **Rotación:** actualmente todos los círculos tienen la misma fase inicial (0), de modo que están alineados. Modificando la fase inicial para que se incremente en función del valor del tiempo de vida el círculo, en la función `UpdateCircle()`, conseguimos que los círculos dejen de estar alineados y tengan una rotación propia, lo que da un cierto efecto de succión o de torbellino al túnel, lo que favorece el resultado visual.
- **Control por parte del usuario:** podemos hacer que el túnel sea modificable por el usuario haciendo simplemente que varias variables que ya tenemos se vean alteradas por determinadas entradas de teclado, por ejemplo:
 - **Velocidad del túnel:** modificando la variable `radiusVelocity`, que incrementa la velocidad de crecimiento de los círculos
 - **Tamaño de los puntos:** nuestra función para dibujar puntos tiene la capacidad para recibir un tamaño, tan solo debemos usar esto en nuestro favor para añadir una variable modificable que contenga el tamaño de los puntos
 - **Posición del centro del túnel:** si modificamos el centro del túnel, tenemos la sensación de poder *controlar* el túnel.

De este modo, podemos con modificaciones muy pequeñas hacer que nuestra demo sea ampliamente interactiva. Luego, basta con utilizar la funcionalidad para dibujar texto para comunicar las instrucciones de uso al usuario.

Código 7.10: Generación de tablas precalculadas y uso en código

```

1 float *Fast::GenerateSineTable(int size)
2 {
3     float *sineTable = new float[size];
4
5     for (int i = 0; i < size; i++)
6     {
7         float value = (i * 2 * Fast::PI) / size;
8         sineTable[i] = sin(value);
9     }
10
11    return sineTable;
12}
13
14 void DotTunnelDemo::DrawCircle(const Circle &c)
15 {
16     const float increment = (2 * Fast::PI) / float(pointsPerCircle);
17     const int indexFactor = mathTableSize / (2 * Fast::PI);
18     int x, y;
19
20     for (float angle = 0, n = 2 * Fast::PI; angle < n; angle += increment)
21     {
22         x = cosineTable[int(angle * indexFactor) % mathTableSize] * c.radius + c.x;
23         y = sineTable[int(angle * indexFactor) % mathTableSize] * c.radius + c.y;
24
25         RenderDot(x, y, Pixel(255), 1);
26     }
27 }
```

7.2.5 Resultado

En la figura [7.13] podemos ver el resultado final de nuestro túnel. Sigue un camino turbulento, se le aplica un degradado de color, los círculos tienen fundido de entrada y de salida y rotan, creando una sensación de vórtice.

Además, el tamaño de los puntos, la posición del centro del túnel y la velocidad del mismo son controlables por el usuario (las instrucciones no se muestran en la figura para no añadir ruido visual al resultado).

El efecto además usa tablas precalculadas para evitar tener que realizar costosas operaciones trigonométricas (la diferencia en el resultado es apenas visible, aunque prestando atención es posible notar irregularidades entre la distancia de algunos puntos -este error se podría mitigar aumentando el tamaño de la tabla, y por tanto su resolución-).

7.3 RotoZoom

7.3.1 Investigación inicial

El efecto de *RotoZoom* es otro de los grandes clásicos de la *demoscene*, y su nombre es bastante autodescriptivo. El efecto de *RotoZoom* consiste en una imagen que se rota y escala en tiempo real. Podemos ver un ejemplo del mismo en la famosa demo *Second Reality* [7.14].

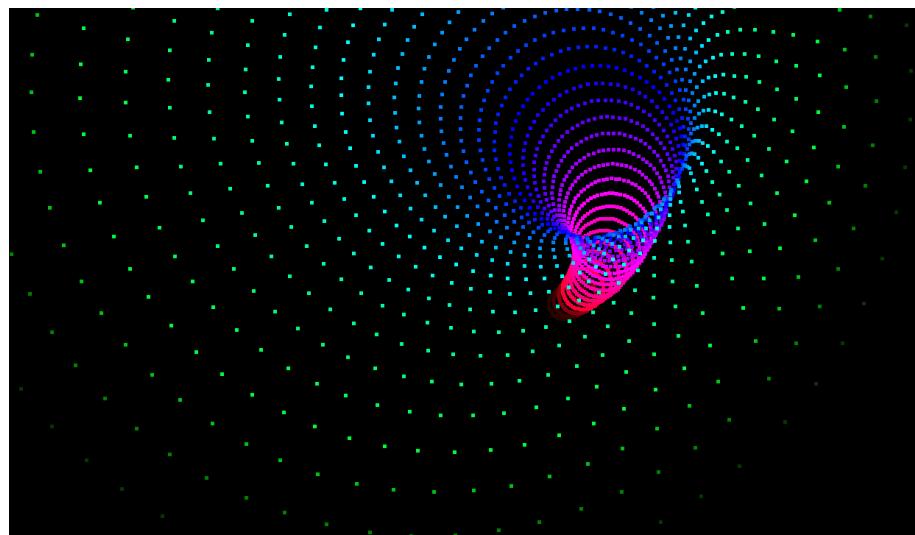


Figura 7.13: Túnel final

Existe amplia documentación para la implementación de este efecto, que además, es muy sencillo, pues se trata simplemente de transformaciones geométricas sencillas en el espacio 2D, dónde el grueso del algoritmo reside en transformar coordenadas en función de una rotación y escala variables.



Figura 7.14: Efecto de RotoZoom - Second Reality (by Future Crew) - Fuente: YouTube

7.3.2 Planteamiento formal

En la figura [7.15] podemos ver las transformaciones necesarias en el espacio bidimensional para escalar, rotar y trasladar un vector o un punto. Además, se dan también las operaciones matemáticas necesarias para hacerlo. Aparece para cada transformación, en rojo el vector inicial y en granate el vector transformado. El orden de las operaciones de escalado y rotación en el ejemplo son intercambiables, es decir, el resultado de la transformación no se verá alterado por qué operación se realice antes. La traslación, no obstante, siempre de-

be ser la última operación que se realice (ya que el escalado y la rotación son consistentes cuando se realizan en torno al origen, pero no son conmutativas si el centro se ve desplazado).

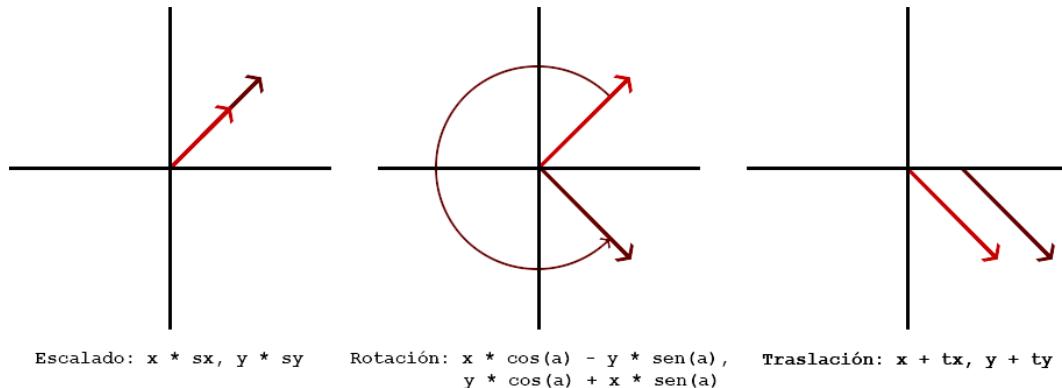


Figura 7.15: Escalado, rotación y traslación en el espacio 2D

El algoritmo principal para realizar este efecto será, por tanto, sencillo. No obstante, debemos tener algo en cuenta. Este algoritmo lo aplicaremos sobre una **imagen** y actualmente, no tenemos modo alguno de cargar imágenes en nuestras demos, por lo que tendremos que añadir esta funcionalidad.

Por tanto, deberemos seguir los siguientes pasos:

- Implementar un modo de cargar imágenes en nuestra demo
- Realizar transformaciones geométricas sencillas para emular el efecto de rotación y escalado

7.3.3 Implementación

Lo primero que necesitamos hacer es tener un modo de cargar imágenes en nuestro sistema. Siguiendo con la filosofía de este trabajo, intentamos minimizar el uso de librerías externas al máximo, intentando implementar desde cero todo aquello que esté dentro de un ámbito razonable. La *standard library* de C++ nos ofrece funcionalidades para manejar ficheros binarios, por lo que podremos usarlas para hacer nuestro trabajo más sencillo. No obstante, necesitaremos interpretar el fichero cargado en memoria para que una cadena de bytes en memoria pase a convertirse en una imagen que podemos manipular.

Es por ello que lo primero que cabe preguntarse es: ¿qué formato de imagen debemos emplear? Para no salir del ámbito de este trabajo, debemos elegir un formato que sea sencillo de leer y manipular, y este es sin duda, BMP, un formato de imagen sin transparencia ni compresión. En Wikipedia se ofrece una explicación en profundidad del formato BMP¹³, que nos será muy útil para saber cómo interpretarlo.

¹³https://en.wikipedia.org/wiki/BMP_file_format

Estas son las consideraciones que tendremos en cuenta para nuestra cargar de imágenes en BMP, a partir de un archivo binario cargado en memoria:

- Necesitamos saber la anchura de la imagen, que se puede encontrar en el byte 18 del fichero BMP y ocupa 4 bytes, en *little-endian*
- Necesitamos saber la altura de la imagen, que se puede encontrar en el byte 22 del fichero BMP y ocupa 4 bytes, en *little-endian*
- Asumimos que la profundidad de color será de 24 bits por píxel (1 byte por color)
- Asumimos que no hay ningún tipo de compresión
- En el formato BMP cada fila de la imagen está alineada a 32 bits
- En el formato BMP, la imagen está guardada de abajo a arriba y de izquierda a derecha (es decir, por filas, empezando en la inferior)

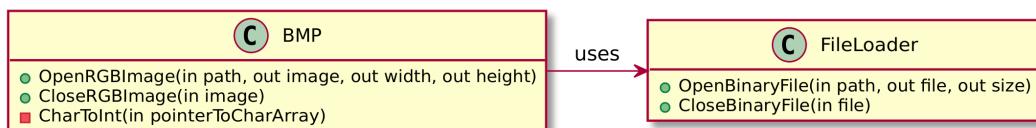


Figura 7.16: Diagrama del sistema para cargar imágenes

Como podemos ver en la figura [7.16], nos ayudaremos de una clase a la que hemos llamado *FileLoader* a la que simplemente pasaremos una cadena de texto con la ubicación de la imagen que queremos abrir y nos devolverá una copia del fichero (como una cadena de bytes `unsigned char {}`) y el tamaño en bytes del mismo. Para realizar estas operaciones, nos ayudamos de la librería *fstream* de la *standard library*.

Nuestra clase principal, *BMP* [7.16], convertirá la cadena de bytes (`unsigned char {}`) que recibe del *FileLoader* en una cadena de píxeles (`Pixel {}`) usable por nuestras demos.

Código 7.11: Código para cargar una imagen BMP en nuestro sistema

```

1 void BMP::OpenRGBImage(const char *path, Pixel *&image, int &width, int &height)
2 {
3     unsigned char *imageBinary = nullptr;
4     unsigned int imageSize;
5
6     FileLoader::OpenBinaryFile(path, imageBinary, imageSize);
7
8     width = CharToInt(imageBinary + 18); //Offset where width info is in BMP format
9     height = CharToInt(imageBinary + 22); //Offset where height info is in BMP format
10    image = new Pixel[width * height];
11
12    for (int j = 0; j < height; j++)
13    {
14        for (int i = 0; i < width; i++)
15        {
  
```

```

16     //The first part of the index "(imageSize - (j + 1) * width * 3) + i * 3"
17     //draws the image inverted in the Y axis
18     //The second part of the index "((width * 3) % 4) * j"
19     //adds the corresponding offset (in the BMP format)
20     //All rows are 32 bit aligned)
21     int index = (imageSize - (j + 1) * width * 3) + i * 3 - ((width * 3) % 4) * j;
22     image[j * width + i] = Pixel(imageBinary[index + 2], imageBinary[index + 1], imageBinary[index]);
23 }
24 }
25
26 FileLoader::CloseBinaryFile(imageBinary);
27 }
```

Aunque el código [7.11] pueda parecer algo complejo, en realidad es bastante sencillo. Una vez tenemos un puntero a nuestro archivo binario, que sabemos que será un fichero BMP, lo primero que hacemos es obtener la altura y anchura de la imagen a leer. Como sabemos la posición en la que se encuentran la anchura y la altura, dado que vienen especificadas por el formato, tan solo tenemos que acceder directamente a la posición donde se encuentran los bytes que nos interesan e interpretarlos. Es por ello que necesitamos una función que sea capaz de, a partir de una dirección de memoria dada, leer 4 bytes que representan un entero de 32 bits en *little-endian*¹⁴ y devolver como resultado un entero [7.12]. Logramos este resultado mediante el uso de operaciones a nivel de bit, creando una correspondencia byte a byte entre nuestra cadena de bytes y nuestro entero.

Código 7.12: Código para convertir una cadena de 4 bytes en un entero de 32 bits

```

1 int BMP::CharToInt(unsigned char *p)
2 {
3     int number = 0;
4
5     number = p[0];
6     number = number | p[1] << 8;
7     number = number | p[2] << 16;
8     number = number | p[3] << 24;
9
10    return number;
11 }
```

Una vez hemos obtenido de las cabeceras de nuestro fichero BMP la anchura y altura de la imagen a procesar, en píxeles, podemos calcular el tamaño de nuestra textura e inicializar una textura formada por nuestros píxeles, como podemos ver en la línea [10]. Ahora, debemos llenar nuestra recién creada textura de un modo que sea fácilmente manipulable en nuestro sistema. Para ello, tenemos que tener en cuenta que el formato BMP incluye la información fila a fila de abajo arriba, por lo que nosotros tendremos que invertirla, para poder tratar nuestra imagen de forma más cómoda, de arriba abajo. Además, tenemos que tener en cuenta que los píxeles están alineados a 32 bits por cada fila. Como podemos ver en la figura [7.17], en una imagen de 5x4 píxeles, el formato BMP incluiría un byte extra por fila, de modo que las filas estén siempre alineadas a 32 bits. En la figura [7.17], cada celda representa un byte, cada grupo de color (gris o blanco) representa un píxel (tres bytes) y el color rojo representa los bytes añadidos al final de cada fila para que la memoria esté alineada a 32 bits (4 bytes).

¹⁴<https://en.wikipedia.org/wiki/Endianness#Little-endian>

50	100	150	50	100	150	50	100	150	50	100	150	50	100	150	0
50	100	150	50	100	150	50	100	150	50	100	150	50	100	150	0
50	100	150	50	100	150	50	100	150	50	100	150	50	100	150	0
50	100	150	50	100	150	50	100	150	50	100	150	50	100	150	0

Figura 7.17: Representación de los píxeles para una imagen 5x4 en BMP

Es por ello que si bien rellenamos nuestra textura de arriba abajo, de izquierda a derecha y de forma lineal, en la línea [21] calculamos un índice que accede a nuestro fichero BMP de abajo arriba, de izquierda a derecha y saltando al final de cada fila (evitando introducir datos basura que se usan para mantener el alineamiento por fila).

Tras ello, nos aseguramos de liberar la memoria correspondiente y ¡ya hemos convertido nuestra imagen BMP a un formato manejable en nuestras demos!

Ahora tan solo queda implementar el algoritmo de *RotoZoom*, lo cual resulta bastante trivial, como podemos ver en el código [7.13]. Para hallar la posición de la textura que corresponde a nuestro píxel, aplicamos primero una rotación, tras ello un escalado y por último, sumamos un desplazamiento. Cabe denotar que para mantener la coherencia de los resultados, nos quedamos con el valor absoluto de la transformación y le aplicamos el módulo en función del tamaño de la textura, para evitar que nuestro píxel se salga de los límites de la textura.

Código 7.13: Dibujar un píxel cuya textura se desplaza en función de un ángulo, una escala y una traslación

```

1 void RotoZoom::DrawPixel(int x, int y, int offsetX, int offsetY, int angle, float scale)
2 {
3     int texX = Fast::Abs(int((x * cos(angle)) - y * sin(angle)) * scale + offsetX) % texWidth;
4     int texY = Fast::Abs(int((y * cos(angle)) + x * sin(angle)) * scale + offsetY) % texHeight;
5
6     pixels[y * width + x] = texture[texY * texWidth + texX];
7 }
```

Una vez aplicada esta transformación, ¡nuestro efecto de *RotoZoom* está acabado! Y con una tasa de fotogramas de nada más y nada menos que **5 fotogramas por segundo...**

7.3.4 Refinamiento

Como vimos en los tests de rendimiento y como hemos mencionado en el análisis del efecto del Túnel de puntos, las operaciones con funciones matemáticas son costosas. **Muy costosas**. Si operamos tan solo con unos pocos píxeles no hay problema, pero si debemos iterar sobre cada uno de los píxeles en pantalla, que en nuestro caso son más de un millón... resulta inviable.

El primer cambio que podemos hacer, si nos fijamos en el código [7.13], es extraer el resultado del seno y el coseno a una variable, ya que se realiza la misma operación dos veces. Únicamente hacer esto ya duplicará la velocidad de fotogramas, a 10 fotogramas por segundo,

lo cual sigue siendo inaceptable.

Por suerte, y con previsión, ya hemos implementado un método para generar tablas pre-calculadas [7.10]. Si en lugar de usar funciones matemáticas usamos valores precalculados, nuestra tasa de fotogramas aumenta a **28 por segundo** en modo *debug* y **40 fotogramas por segundo** en modo *release*, las cuales ya se pueden considerar tasas de fotogramas aceptables, dado que el ojo humano percibe continuidad a partir de algo más de 12 fotogramas por segundo¹⁵.

Ahora que tenemos una tasa de fotogramas adecuada y estable, llega el momento de crear una sensación de movimiento que parezca aleatorio pero continuo y fluido... por suerte, este también es un problema que se repite, pues es el mismo problema que teníamos con la ruta que debía seguir el "túnel de puntos". Así que podemos reutilizar la solución, ajustando valores. Así pues, creamos un "camino de turbulencia" que controlará los valores *x* e *y* de nuestra traslación y otro que controlará los valores del ángulo y la escala.

De este modo, las transformaciones que aplicamos sobre nuestros píxeles tendrán un efecto continuo y fluido, pero con una ruta impredecible.

7.3.5 Resultado



Figura 7.18: RotoZoom en distintos estados

7.4 Deformaciones de imagen

7.4.1 Investigación inicial

Las deformaciones de imagen son un efecto muy socorrido en el mundo de la *demoscene* y de formas muy distintas. Efectos de lupa y/o deformaciones de lente, estiramientos, efectos de muelle o rebote, torsiones, espirales, en túnel...

¹⁵https://es.wikipedia.org/wiki/Fotogramas_por_segundo

Sólo implementar todas las diferentes propuestas y efectos de deformaciones de imagen podría llevar su propia investigación y trabajo. No obstante, el denominador común de muchas de estas transformaciones es que usan operaciones trigonométricas, y al final no dejan de ser combinaciones distintas de senos y cosenos a los que se aplican distintos parámetros.

Es por ello que para las deformaciones de imagen, nos centraremos en las deformaciones de onda, pues muchos de los efectos que podemos encontrar en demos parten de estas mismas bases matemáticas.

7.4.2 Planteamiento formal

En la figura [7.19] podemos ver la ecuación de la onda, donde A es la amplitud de la onda, lo que en nuestro efecto se traducirá en la altura máxima y mínima de nuestra onda, en píxeles. f se corresponde con la frecuencia de la onda (cuántas veces por segundo se completa un ciclo de onda), λ es la longitud de onda (distancia entre dos crestas consecutivas de la onda), t es el tiempo transcurrido y ϕ es la fase inicial (o desplazamiento inicial con respecto al origen).

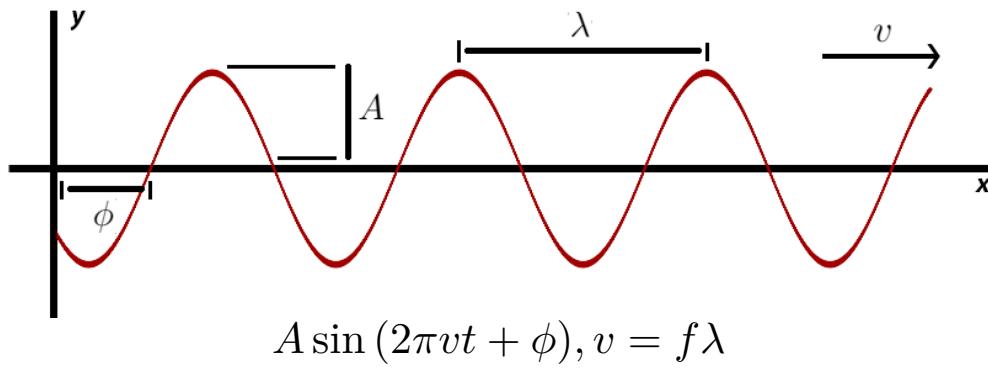


Figura 7.19: Representación gráfica de la onda y su ecuación

Además, podemos distinguir dos tipos distintos de onda, dependiendo de su forma de movimiento. Una **onda transversal** es aquella en la que la dirección de movimiento es perpendicular a la dirección de oscilación¹⁶ mientras que una **onda longitudinal** es aquella en la que la dirección de movimiento es paralela a la dirección de oscilación¹⁷. Por ejemplo, un muelle es un ejemplo de onda longitudinal, pues oscila (rebota) en la misma dirección en la que se mueve, mientras que las ondas que se producen al lanzar una piedra en el agua son un buen ejemplo de onda transversal (pues mientras que el agua oscila de arriba abajo, la onda avanza en línea recta, perpendicular a su oscilación).

En la figura [7.20a] podemos ver la fórmula de la onda transversal. Tiene sentido que el avance de esta onda sea perpendicular a su dirección de oscilación, dado que como se muestra en la figura, el valor de y depende de x , por lo que el valor en el eje y depende del valor en el

¹⁶https://es.wikipedia.org/wiki/Onda_transversal

¹⁷https://es.wikipedia.org/wiki/Onda_longitudinal

eje perpendicular al mismo. Esto mismo se extraña a la figura [7.20b], donde el avance de la onda en y depende de y , por lo que la dirección de propagación coincide con la dirección de oscilación.

$$y = A \sin(2\pi \frac{x}{\lambda} + \phi)$$

(a) Onda transversal

$$y = A \sin(2\pi \frac{y}{\lambda} + \phi)$$

(b) Onda longitudinal

Con todo lo visto, para poder crear nuestro efecto de deformaciones de imagen necesitaremos:

- Implementar un modo de aplicar deformaciones a nuestra imagen
- Implementar un sistema que permita manejar con facilidad distintas ondas
- Implementar distintos tipos de ondas, con distintos parámetros

7.4.3 Implementación

Como podemos ver en el código [7.14], empezamos por definir una función delegada que nos devuelva un valor entero en función del valor de entrada, que serán las coordenadas x e y . De este modo, podemos definir nuestro comportamiento de deformación encapsulado en sus propios métodos, generando un código más fácil de mantener y un programa fácilmente modificable en tiempo de ejecución.

Código 7.14: Dibujado de un pixel aplicando una función que modifica el acceso a textura

```

1 typedef int (Deformations::*delegate)(int, int);
2
3 void Deformations::DrawPixel(int x, int y, float deltaTime, delegate xModifier, delegate yModifier)
4 {
5     int newX = (this->*xModifier)(x, y) % texWidth;
6     int newY = (this->*yModifier)(x, y) % texHeight;
7
8     pixels[y * width + x] = texture[newY * texWidth + newX];
9 }
```

Tras ello, definimos diversos métodos que apliquen nuestras funciones de onda, tanto para x como para y . En el código [7.15] podemos ver los modificadores que definimos para x . Definimos así un método por defecto que simplemente devuelve la variable sin modificar (por si no queremos aplicar deformación en absoluto o queremos no aplicar transformación en un eje). Tras ello, definimos un método para generar ondas transversales (como podemos ver, la x depende de la y) y un método para generar ondas longitudinales (donde x depende de sí misma). La variable k corresponde al número de ondas ($k = \frac{2\pi}{\lambda}$). Además, en nuestro caso sumamos una fase inicial variable, que depende del tiempo (multiplicado por una constante definida por el usuario), de modo que se aplique un desplazamiento incremental a la deformación, generando sensación de movimiento.

Código 7.15: Funciones de deformación en X

```

1 int Deformations::DefaultXModifier(int x, int y)
2 {
3     return x;
4 }
5
6 int Deformations::TransversalWaveXModifier(int x, int y)
7 {
8     return x + amplitude * sineTable[(y * k + int(accumulatedTime * c)) % mathTableSize];
9 }
10
11 int Deformations::LongitudinalWaveXModifier(int x, int y)
12 {
13     return x + amplitude * sineTable[(x * k + int(accumulatedTime * c)) % mathTableSize];
14 }
15
16 int Deformations::FlagXModifier(int x, int y)
17 {
18     return x + bigAmplitude * sineTable[(y + int(accumulatedTime * c)) % mathTableSize];
19 }

```

Como podemos ver, ahora usamos tablas precalculadas desde el principio, pues ya hemos demostrado en el efecto anterior que la única forma de obtener una tasa de fotogramas estable y aceptable es mediante el uso de tablas precalculadas. Calculamos el módulo en función del tamaño de la tabla para asegurarnos de no salirnos de los límites de la misma.

Como podemos ver en el código [7.15], tenemos un último método al que hemos llamado *FlagModifier*. Esta función ha sido nombrada por el resultado visual que produce, en lugar de haberse elegido su nombre por el tipo de onda que es. Usamos como modificador una onda transversal con una amplitud grande y una longitud de onda también grande (k se omite, por lo que $k = 1$, lo que implica que $\lambda = 2\pi$). Esto produce, al aplicarse sobre la imagen, un efecto similar a una bandera ondeando, de ahí el nombre.

Una vez hemos definido nuestras funciones de deformación para x e y , y habiendo ajustado sus valores, podemos combinarlas y usarlas libremente para crear distintos efectos visuales. Para ello, definimos un vector de pares de modificadores (`std::vector<std::pair<delegate<→ , delegate>>{}`). Aunque esta expresión parece algo compleja, es realidad su uso es muy sencillo: al tener un vector de pares de modificadores, podemos añadir fácilmente pares de modificadores y alternarlos. De este modo, podemos aplicar modificadores para x e y y cambiarlos con facilidad en tiempo de ejecución, aplicando simplemente el siguiente par en el vector.

7.4.4 Resultado

Podemos apreciar los resultados de aplicar y combinar las distintas deformaciones en la figura [7.21].



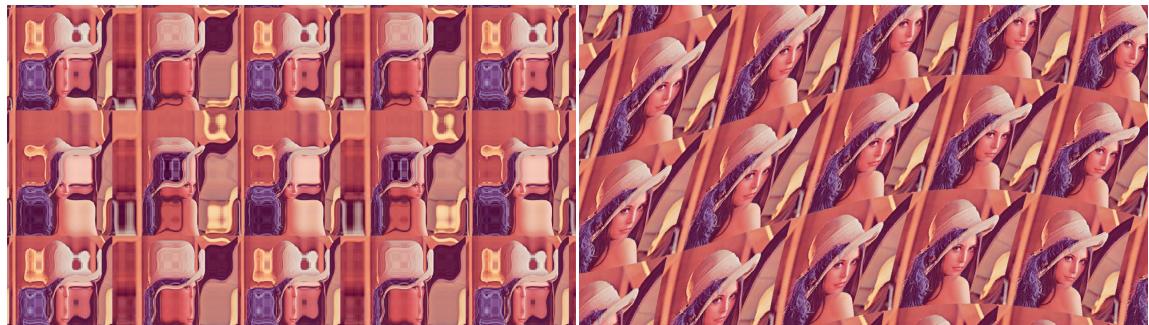
(a) Sin modificar

(b) Onda transversal en x



(c) Onda longitudinal en x

(d) Transversal en x e y



(e) Mosaico (longitudinal en x e y)

(f) Efecto de bandera

Figura 7.21: Distintos efectos de deformación a partir de ondas

7.5 Plasma

7.5.1 Investigación inicial

El efecto de plasma es otro de los grandes clásicos de la *demoscene*, como el que podemos ver en la figura [2.13b]. Este efecto existe bien documentado y podemos encontrar múltiples explicaciones y formas de implementarlo.

Un tutorial en detalle, muy bien explicado y documentado es el que podemos encontrar en la página de **Lode's Computer Graphics Tutorial**¹⁸. Este efecto es tan popular que cuenta incluso con su propia página de Wikipedia¹⁹ e incluso en Rosetta Code²⁰ podemos encontrar el código para este efecto implementado en más de 20 lenguajes de programación distintos. También podemos encontrar el código para implementar el efecto de Plasma por GPU en Bidouille.org²¹.

Como podemos ver, este no es para nada un efecto desconocido o poco documentado, si no que más bien tenemos que, ante tanta información y tantas implementaciones posibles, elegir la que más se adecue al ámbito de este proyecto.

7.5.2 Planteamiento formal

Existen dos principales acercamientos a la implementación del efecto de plasma: combinación de ondas o generación de ruido.

Ambos acercamientos son válidos, e incluso se pueden combinar para producir resultados intermedios. El acercamiento por combinación de ondas se basa en la suma o superposición de distintas funciones de onda, con distintas frecuencias, longitudes de onda y amplitudes, asignando color en función del resultado de la combinación. El acercamiento por generación de ruido se basa, por otro lado, en la generación de un ruido que tenga coherencia local, como por ejemplo el ruido Perlin²².

Los resultados de generación de ruido mediante Perlin u otros algoritmos de generación de ruido (fractal, simplex...), no obstante, suelen dar resultados más similares a nubes o turbulencias que a plasma, y suelen requerir de niveles bajos de detalle (pocas iteraciones, si se trata de un algoritmo iterativo) o de suavizados posteriores (calculando valores medios o con efectos de desenfoque) para ser suficientemente convincentes. Además de esto, a la hora de animarlos, el movimiento no resulta tan natural como en el uso de ondas, dado que aunque el ruido tenga coherencia local, sigue existiendo un factor de aleatoriedad que puede romper la coherencia del movimiento.

¹⁸<https://lodev.org/cgtutor/plasma.html>

¹⁹https://en.wikipedia.org/wiki/Plasma_effect

²⁰https://rosettacode.org/wiki/Plasma_effect

²¹<https://www.bidouille.org/prog/plasma>

²²https://en.wikipedia.org/wiki/Perlin_noise

Es por ello que se opta por implementar un efecto de plasma mediante la combinación de ondas. Estos serán los pasos a seguir:

- Hallar una combinación de ondas que produzca el efecto deseado
- Aplicar un degradado de color dado el resultado de la combinación

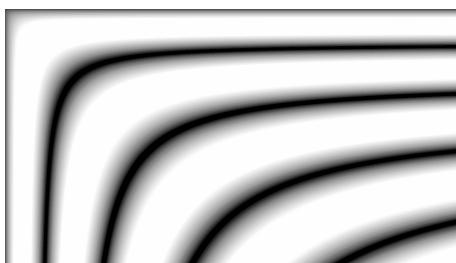
7.5.3 Implementación

En el código [7.16] se muestra una versión simplificada del código para generar ondas, de modo que sea más fácil de leer (dado que el código real emplea tablas precalculadas y guarda cada operación que se repite más de una vez en variables temporales).

Código 7.16: Combinación de distintas ondas

```
1 value += sin((j * i) / (j + i + 1) * scale + accumulatedTime * 191);
2 value += sin(((width - i) * j) / ((width - i) + j + 1) * scale + accumulatedTime * 157);
3 value += sin((i * (height - j)) / (i + (height - j) + 1) * scale + accumulatedTime * 113);
4 value += sin(((width - i) * (height - j)) / ((width - i) + (height - j) + 1) * scale + accumulatedTime * 67);
```

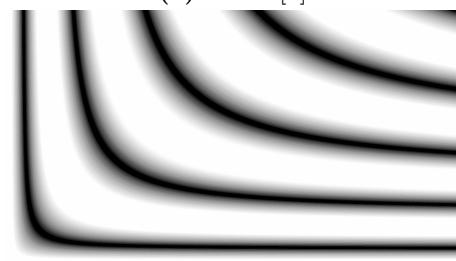
Podemos comparar este código con el resultado de la figura [7.22], donde se muestra el resultado que genera cada línea de código, la onda en que resulta de forma separada.



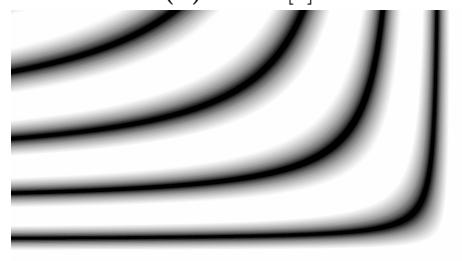
(a) Línea [1]



(b) Línea [2]



(c) Línea [3]



(d) Línea [4]

Figura 7.22: Cada una de las ondas que son sumadas (con su correspondencia a línea de código)

Si nos fijamos en el código y en las imágenes, podemos ver que todo lo que hacemos es crear cuatro ondas, yendo cada una hacia una de las esquinas de la pantalla y que están desfasadas entre ellas. Los números elegidos para la fase no son aleatorios, pues todos ellos son números primos. Esto es importante, pues si la fase de alguna onda es múltiplo de otra, se pueden

producir formas forzadas o bucles que rompan con la sensación de aleatoriedad y continuidad, dado que su desfase es proporcional. Es por ello que el mejor modo de asegurar que no se produzcan estos *acoplos* entre las ondas es multiplicando la fase por números primos, de modo que las fases de las ondas no tengan múltiplos comunes y por tanto no puedan acoplarse o dar sensación de periodicidad.

Una vez hemos sumado nuestras ondas, dividimos entre 4 para normalizar el resultado (entre -1 y 1) y a continuación convertimos este intervalo en un valor entero que oscile entre 0 y 255. De este modo obtenemos un índice con el que acceder a un color específico dentro del degradado de colores que hayamos definido previamente.

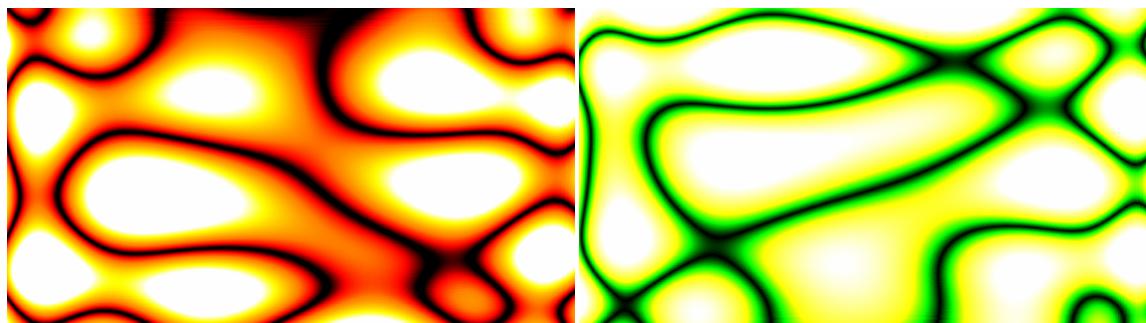
El resultado de esta combinación de ondas resulta en un efecto bastante convincente, como podemos ver en la figura [7.23].



Figura 7.23: Efecto de plasma

Ahora tan sólo nos queda añadir la capacidad de cambiar de degradado de color, como hemos hecho ya en otros efectos, para conseguir así un resultado más convincente o visualmente impactante.

7.5.4 Resultado



(a) Efecto lámpara de lava

(b) Efecto eléctrico

7.6 Planos infinitos

7.6.1 Investigación inicial

El efecto de planos infinitos es muy conocido en la *demoscene*, pero es especialmente popular y conocido en el mundo del videojuego, donde fue popularizado por Nintendo como el famoso Modo 7²³ que incluía la SNES. Este era un modo gráfico de esta consola que permitía realizar transformaciones afines, mediante las que se lograba el efecto de planos infinitos.

Este efecto básicamente consistía el uso de una textura bidimensional que se transformaba para dar efecto de profundidad o tridimensionalidad, como podemos ver en la figura [7.25].

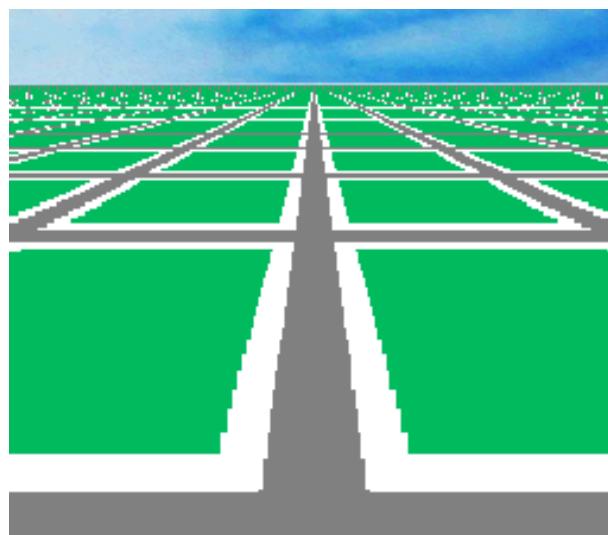


Figura 7.25: Modo 7 (efecto de planos infinitos) en la SNES - Fuente: Wikipedia (por Anomie)

Gracias a este efecto fueron posibles juegos como el primer *Mario Kart*²⁴, donde la pista o el circuito no eran más que una textura 2D de grandes dimensiones transformada, para que pareciera un plano o un circuito 3D.

Este efecto está muy bien documentado y se puede encontrar su explicación formal tanto en Wikipedia como numerosos tutoriales que ofrecen distintos acercamientos, como este tutorial de *One Lone Coder*²⁵ en que ofrece una explicación formal y da su propio planteamiento para implementar este modo o este otro tutorial en Coranac.com²⁶ dónde se nos ofrecen tres implementaciones distintas, con distintos acercamientos, y se hace una reflexión sobre los resultados obtenidos.

²³https://en.wikipedia.org/wiki/Mode_7

²⁴https://en.wikipedia.org/wiki/Super_Mario_Kart

²⁵<https://www.youtube.com/watch?v=ybLZyY655iY&t=646s>

²⁶<https://www.coranac.com/tonc/text/mode7.htm>

7.6.2 Planteamiento formal

El efecto de planos infinitos está muy bien documentado, bajo distintos acercamientos, uno de los cuales es el Modo 7, aunque también existen otras formas de causar un efecto similar.

Siguiendo el espíritu de este trabajo, intentaremos implementar el efecto de planos infinitos sin usar código de referencia, simplemente usando el material visual disponible para intentar deducir cómo podríamos implementar este efecto. En la figura [7.26], de creación propia, vemos un posible acercamiento a este efecto, que es el que seguiremos e intentaremos implementar.

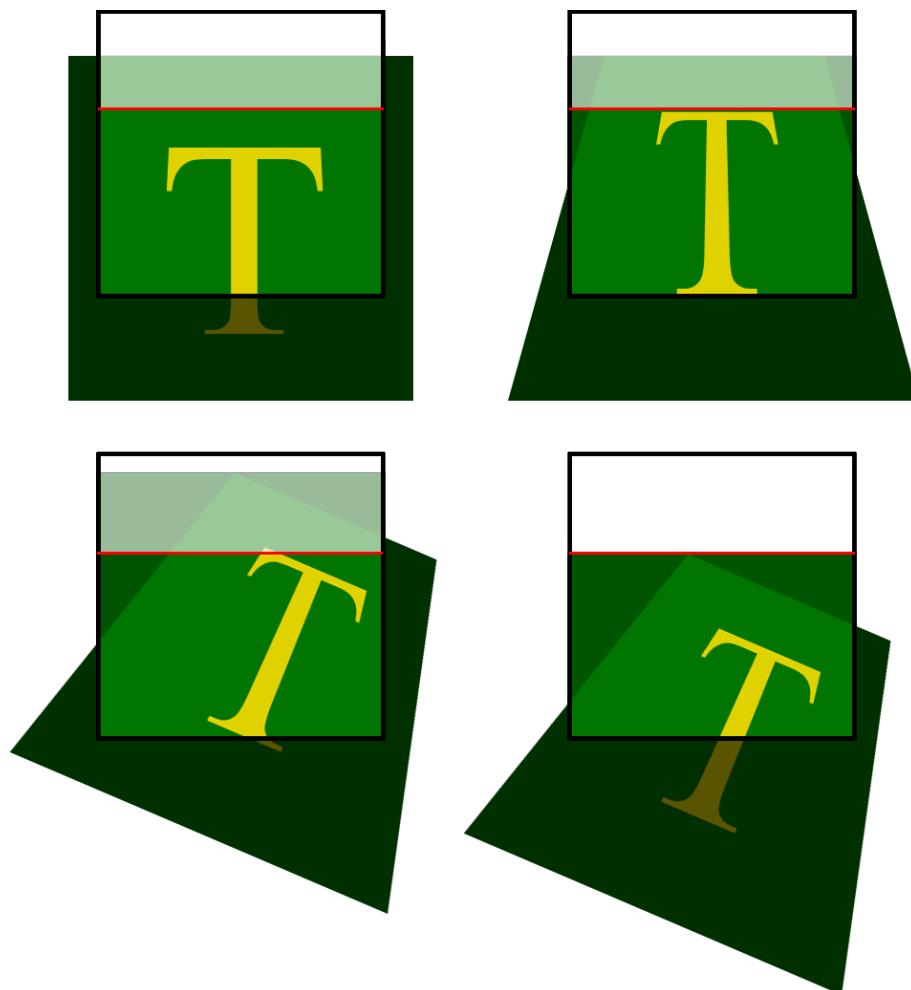


Figura 7.26: Nuestro acercamiento a los planos infinitos

En esta figura [7.26], el cuadro negro representa la pantalla o la parte visible de la textura, la línea roja representa la línea del horizonte, punto a partir del cual no se muestra nuestra textura en pantalla.

Para facilitar la visualización del contenido, la parte de la textura fuera de pantalla se ha oscurecido y la parte de la textura en pantalla pero tras la linea del horizonte se ha aclarado, de modo que la porción visible destaque. Además, se ha rellanado el espacio en de pantalla adyacente a nuestra textura con un verde más oscuro, para completar la imagen y facilitar su visualización.

Como vemos, partimos de una textura bidimensional, plana. A continuación, estiramos la textura en su parte baja y la apretamos en la parte alta. Dicho de un mejor modo: hacemos nuestra textura depender de la altura, de modo que a mayor altura, más se estreche y vice-versa. Ya solo con esto, nuestra imagen gana sensación de profundidad y tridimensionalidad. Pero además, queremos ser capaces de moverla. Para ello, una vez que nuestra textura ha sido convenientemente deformada, rotamos la imagen usando la fórmula mostrada en la figura [7.15].

Una vez que nuestra textura ha sido deformada y rotada como queremos, podemos desplazarla, de modo que así, es pantalla, dará sensación de movimiento y avance. Realmente, todo lo que hacemos es aplicarle las transformaciones 2D que ya vimos con el RotoZoom pero con un contexto y un fin distintos.

7.6.3 Implementación

Código 7.17: Código para generar un efecto básico de planos infinitos, con escalado, rotación y translación

```

1 for (int j = 0, nh = height / 2; j < nh; j++)
2 {
3     for (int i = -width / 2, nw = width / 2; i < nw; i++)
4     {
5         Point2D projectedPoint(i / (float)j, fieldOfView / (float)j);
6         projectedPoint *= textureScale;
7
8         Point2D rotatedPoint(projectedPoint.X * cosine - projectedPoint.Y * sine,
9                               projectedPoint.X * sine + projectedPoint.Y * cosine);
10
11        Pixel colour = texture[Fast::Abs((int(rotatedPoint.Y + cameraPosition.Y) % texHeight) * texWidth +
12                                int(rotatedPoint.X + cameraPosition.X) % texWidth)];
13
14        pixels[(j + nh) * width + (i + nw)] = colour;
15    }
16}

```

Como podemos ver en el código, recorremos media pantalla en altura, desde 0 hasta la mitad de la pantalla, y desde $-x \div 2$ hasta $+x \div 2$, recorriendo así toda la anchura de la pantalla. Esta decisión no es aleatoria, pues como vemos más adelante, en la línea [14] sumamos la mitad de la altura y la anchura, desplazando las coordenadas i y j . Hacemos esto por conveniencia, pues nos facilita los cálculos, ya que para transformar nuestra textura lo hacemos con respecto al origen de la pantalla (el $(0, 0)$ equivale a la esquina superior izquierda) y luego desplazamos el resultado obtenido, como podemos ver en la figura [7.27].

Esto es importante también porque además de facilitar los cálculos, operar con la coordenada horizontal de la textura centrada en el origen es el equivalente a situar nuestra "cámara", es decir, nuestro punto de vista en el origen. Si hacemos los cálculos de este modo, el punto

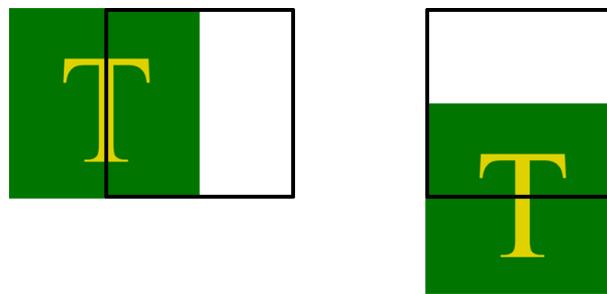


Figura 7.27: Posición relativa de la textura antes y después de ser transformada

de fuga del resultado obtenido se situará en el centro de la pantalla (ya que calculamos con respecto al origen -punto de fuga- y luego desplazamos el resultado). Esto da un resultado mucho más realista, pues en nuestra visión el punto de fuga siempre tiende hacia el centro. Podemos ver una demostración visual en la figura [7.28].

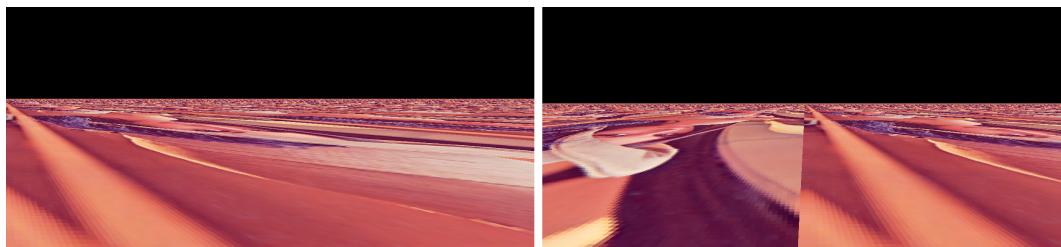


Figura 7.28: A la izquierda el punto de fuga está en el 0 de las coordenadas horizontales, a la derecha está en el centro

En la figura [7.28], la imagen de la izquierda se obtiene iterando de 0 a la anchura de la pantalla en la coordenada horizontal. Como el origen está al inicio de la pantalla, todo se siente desplazado hacia la izquierda, de forma poco natural. En la imagen de la derecha, iteramos desde $-anchura \div 2$ hasta $+anchura \div 2$ y luego desplazamos el resultado en $anchura \div 2$, situando de este modo el punto de fuga en el centro del imagen, tal y como se da en nuestra visión, en el mundo real.

Dentro del bucle, empezamos por la primera transformación (equivalente a estirar la textura en la parte inferior y estrechar en la superior, como veímos en la figura [7.26]). Para obtener este efecto, basta con dividir basándonos en la altura, como podemos ver en la línea [5]. A mayor sea la altura (recordemos que el eje Y está invertido en una pantalla, por lo que mayor altura es más hacia abajo), menor será nuestra el índice con el que accederemos a nuestra textura. Esto causa que a mayor sea la altura, menor es la distancia entre los accesos a la textura y por tanto la textura tiene una apariencia más grande. Es por esto que a menos altura (más arriba), la textura se percibirá más comprimida y a mayor altura (más abajo) los accesos a memoria serán más contiguos y por tanto la imagen resultante se percibirá más grande. Podemos ver este efecto claramente en la figura [7.28].

Además, como podemos ver en la línea [5], nuestra variable horizontal depende de la vertical (i depende de j), y para la coordenada vertical del punto que creamos, usamos una variable definida por el usuario y que también se ve afectada por la altura (que equivale a la distancia desde el observador, es decir, más arriba en pantalla implica menor altura y más distancia desde el observador si en lugar de una textura plana se tratase de un espacio tridimensional real). A esta variable la llamamos *field of view*, o campo de visión, y su modificación modifica el ángulo o la amplitud con la que vemos en pantalla, como podemos ver en la figura [7.29].

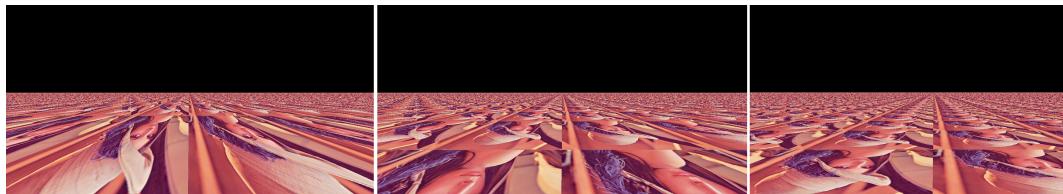


Figura 7.29: De izquierda a derecha, de mayor a menor campo de visión

A continuación, escalamos nuestra textura por un factor, como podemos ver en la línea [6]. Esto variará el tamaño de nuestra textura en pantalla, y por tanto, también el número de repeticiones de la misma. A menor sea la escala, menor será la textura y más repeticiones se darán de la misma para llenar la pantalla.

Tras el escalado, rotamos nuestra textura en función de un ángulo que se corresponderá con el giro de cámara, aplicando la fórmula matemática ya vista en la figura [7.15], y a continuación sumamos el desplazamiento, que se corresponderá con la posición de la cámara en la escena. Tras ello, aplicamos el módulo para asegurarnos que si se sobrepasan los límites de nuestra textura, se produzca un acceso cíclico a la misma, y no se den errores de acceso. Nos aseguramos además de que el resultado obtenido sea positivo (calculando el valor absoluto) para evitar así accesos negativos que puedan hacer fallar nuestra aplicación. Una vez hemos calculado las coordenadas de la textura que se corresponden con las coordenadas en pantalla, y una vez que hemos obtenido el color al que se asocian nuestras coordenadas de textura, asignamos el color.

Con esto, obtenemos una escena que produce la sensación de planos infinitos y por la que nos podemos mover como si se tratase de un videojuego en primera persona (pues podemos modificar la rotación, escala y traslación, que en nuestra escena es el equivalente a mover la cámara).

Podemos ver además que para esta demo hemos creado una estructura a la que hemos denominado `Pixel2D{}`. Esta es una estructura sencilla que simplemente nos permite almacenar dos coordenadas en coma flotante con precisión simple (x e y) y nos da facilidades para operar con las mismas (suma y resta de puntos, multiplicación por un escalar...).

7.6.4 Refinamiento

- **Fundido a negro:** actualmente, la línea del horizonte pasa directamente de tener color a un horizonte negro. Este cambio puede resultar algo brusco. Si creamos una variable

de opacidad que dependa de la altura, de modo que a menor sea la altura menos sea la opacidad (es decir, a más lejos esté la cámara del observador, más oscuro se vea) conseguiremos un efecto mucho menos brusco y mucho más ambiental.

- **Relieve:** un pequeño añadido que me pareció que podría ser curioso de implementar consiste en una falsa sensación de relieve que dependa del brillo del color. Para hacer esto, lo que hacemos es desplazar el dibujado de nuestra textura en un factor variable que dependa del brillo del color que aplicamos, de modo que los colores más oscuros se pinten más abajo y los colores más claros más arriba en pantalla, dando así una cierta sensación de relieve o terreno, sumando a la sensación de tridimensionalidad. Como podemos ver en el código [7.18], empezamos por calcular la percepción del brillo en base al color, tal y como es percibido por el ojo humano²⁷ y a continuación multiplicamos por 0.004 (equivalente de dividir por 256) para normalizar el resultado. Tras ello, calculamos el desplazamiento de la textura en función del nivel de desplazamiento (decidido por el usuario), el brillo relativo del color y la distancia al observador. Aplicamos este desplazamiento al asignar el color en pantalla.

Código 7.18: Código para calcular el desplazamiento de la textura

```

1 float colourBrightness = ((colour.R * 0.3) + (colour.G * 0.59) + (colour.B * 0.11)) * 0.004f; // * 1 / 256
2 int textureDesplacement = (bumpLevel * colourBrightness) * distanceFactor;
3 pixels[(j - textureDesplacement + nh) * width + (i + nw)] = colour;
```

7.6.5 Resultado

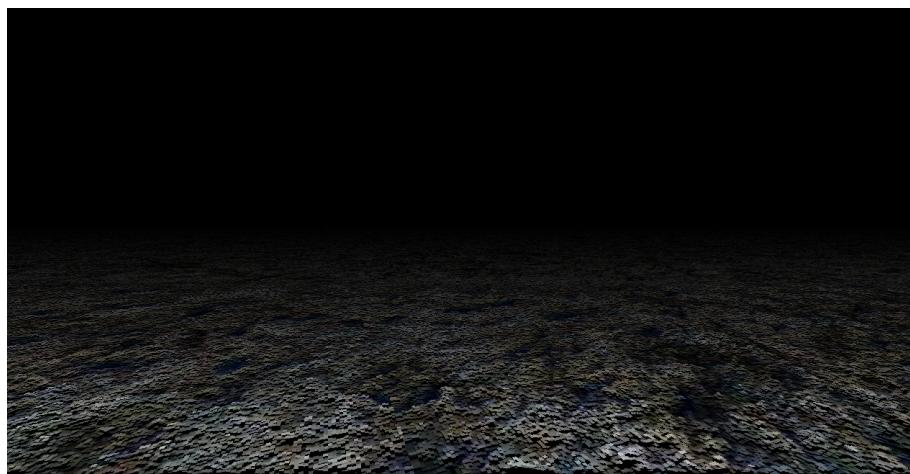


Figura 7.30: Resultado final, con efectos de niebla y relieve aplicados

²⁷https://en.wikipedia.org/wiki/Relative_luminance

7.7 Geometría

7.7.1 Investigación inicial

De cara al efecto de geometría, lo tenía claro: quería realizarlo desde 0, basándome puramente en el planteamiento matemático, y tratando de desarrollar un solución lo más simple pero efectiva posible.

Para poder realizar un efecto completo es necesario un motor gráfico que nos permita gestionar cámaras, múltiples instancias de objetos, orden de dibujado y optimizaciones para no dibujar objetos fuera de pantalla, no dibujar objetos que son tapados por otros objetos, etc...

Implementar todas estas características parecía a todas luces inviable, pues desarrollar un motor gráfico complejo conlleva su propio trabajo e investigación, y se sale del ámbito de este proyecto. Es por ello que se optó por un planteamiento basado puramente en la matemática y abogando por la simplicidad.

No obstante, y a modo de nota, dejo apuntados dos grandes canales de referencia para la implementación de un motor gráfico completo partiendo desde cero, siendo uno el ya varias veces mencionado en este trabajo *One Lone Coder*²⁸ y otro la lista de reproducción de *Chili Tomato Noodle*²⁹.

7.7.2 Planteamiento formal

Como ya hemos dicho, no vamos a implementar un motor gráfico completo, si no que vamos a tratar de implementar el motor más sencillo posible a partir de pura deducción matemática. Es por ello que vamos a sacrificar cierta flexibilidad, a cambio de una mayor sencillez:

- **Evitaremos el uso de matrices:** los motores gráficos actuales usan matrices 4x4 para el cálculo de coordenadas. Esto es muy útil pues permite un sistema de coordenadas y transformaciones homogéneas³⁰, lo que significa que el estado de transformación de cualquier objeto (escala, rotación y traslación) puede ser guardado y/o acumulado en una sola matriz, haciendo así que aplicar transformaciones geométricas sea un proceso fácilmente automatizable y mantenible. Esto viene al coste, no obstante, de tener que gestionar multiplicaciones de matrices, operación que no es trivial, pues para multiplicar una matriz 4x4 por otra, es necesario multiplicar los 16 elementos que forman una matriz por los 16 de la otra, lo que equivale a 256 multiplicaciones. El uso de matrices aporta flexibilidad, mantenibilidad y un modelo matemático único y sólido, pero lo hace al coste de eficiencia (la multiplicación de matrices no es trivial) y de complejidad (manipular matrices requiere una base matemática sólida).
- **Cámara fija en el origen:** nuestra cámara se mantendrá fija en el origen de las coordenadas y no podremos desplazarla. Normalmente para desplazar la cámara en un motor gráfico convencional, lo que se hace realmente es trasladar el *mundo* con respecto

²⁸<https://www.youtube.com/watch?v=ih2013pJoeU>

²⁹<https://www.youtube.com/watch?v=uuhGqieEbus&list=PLqCJpWy5Fohe8ucwhksiv9hTF5sfid81A>

³⁰https://en.wikipedia.org/wiki/Homogeneous_coordinates

a la cámara, que se mantiene siempre en el origen. Para hacer esto, se suele hallar la matriz inversa³¹ de la transformación de la cámara y multiplicar cada elemento en escena por la misma, para así desplazarlo en función del "movimiento" de la cámara, que se mantiene estática en el origen. Por supuesto, esta operación se puede realizar sin usar matrices, simplemente transformando cada objeto con la transformación inversa que se aplica a la cámara (si la cámara se desplaza hacia la derecha, en lugar de mover la cámara, la mantenemos en el origen y movemos nuestros objetos en escena hacia la izquierda). Sin embargo, esto aporta una complejidad añadida y una cantidad de cálculo extra que no resulta excesivamente práctica de cara al resultado final, que no se verá excesivamente impactado por tener una cámara fija.

- **Orden de dibujado dependiente del orden de los vértices:** usualmente, para evitar que objetos que están "virtualmente al fondo" se dibujen por encima de los que están "más adelante" en pantalla, se utilizan *buffers* de profundidad, también conocidos como *z-buffer*³². Estos *buffers* funcionan a nivel de píxel, y se aseguran que solo se pinte en pantalla el píxel que más cercano está al observador (si dos objetos, una verde y uno azul, se superponen en un mismo píxel, el color final del píxel dependerá de qué objeto está delante, y no de qué vértice se dibujó primero). No obstante, mantener un *buffer* de profundidad aumenta la complejidad tanto espacial como temporal de la solución, y resulta poco práctico cuando sólo vamos a tener unos pocos objetos sencillos en pantalla.
- **Pintaremos únicamente el *wireframe* de los objetos:** sólo dibujaremos las aristas de los objetos, pero no los dibujaremos como objetos sólidos ni les aplicaremos textura. Esta decisión se toma no tanto por cuestión de complejidad de implementación como por cuestión de eficiencia. En primer lugar, si nuestros objetos son sólidos, entonces se hace mucho más necesario tener un *buffer* de profundidad. Que las aristas de un objeto cualquiera intersecten con si mismas no supone un problema grave, y apenas suele ser perceptible, pero si la parte trasera del cubo se dibuja por encima de la delantera, se producen defectos visuales serios. Pero más allá de esto, dibujar tan sólo las aristas de nuestro objeto implica tener que pintar unos pocos píxeles (se dibujan líneas, únicamente), mientras que hacer nuestros objetos sólidos implica tener que llenar todo el espacio que ocupa el objeto, teniendo que pintar una gran cantidad de píxeles. Si a esto le sumásemos aplicar una textura al objeto (operación relativamente sencilla, pues consiste en un *mapping* entre la posición en el espacio del objeto y el acceso a la textura, consistiendo en una simple transformación), nos topamos con un coste temporal para nada trivial. La diferencia para dibujar un objeto pasa a ser de pintar unas pocas líneas en cualquier orden a tener que pintar todo el espacio que ocupa el objeto píxel a píxel y asegurar por píxel que se pinta el color de la cara que está más al frente y que esté correctamente *mapeado* a la textura que corresponde al objeto. Recordemos que estamos en CPU, no nos podemos permitir toda esta cantidad de cálculo a altas resoluciones de pantalla. Es por algo, al fin y al cabo, que se crearon las unidades gráficas (GPU) para manejar las operaciones con gráficos (operaciones sencillas a nivel

³¹https://en.wikipedia.org/wiki/Invertible_matrix

³²<https://en.wikipedia.org/wiki/Z-buffering>

matemático -pues suelen ser sumar, restas, multiplicaciones y divisiones- pero con gran carga computacional -pues hay que realizar miles o millones de ellas por fotograma-).

- **No dejaremos de pintar objetos por estar fuera de pantalla o ser tapados por otros objetos:** técnicas como el *clipping*³³ o el *culling*³⁴ evitan el pintado de objetos o partes del objeto que están fuera de pantalla o que no son visibles. Estas técnicas de optimización tienen sentido absoluto cuando pintamos una gran cantidad de caras sólidas con texturas, pues el coste de pintar una cara no es para nada trivial, y es mucho menor el coste de calcular qué caras u objetos deben ser pintados que pintarlo todo a fuerza bruta. En nuestro efecto, sin embargo, pintaremos pocos objetos, que rara vez se saldrán de pantalla y que no serán objetos sólidos, por lo que aplicar técnicas de optimización de este estilo sólo aumentará la complejidad del código sin ofrecer ganancias reales.

7.7.3 Implementación

Para poder dibujar un objeto tridimensional en una pantalla bidimensional, debemos encontrar un modo de representar el objeto y dibujarlo. Para ello, necesitamos almacenar los puntos que forman nuestro objeto y la relación entre ellos (qué pares de puntos están conectados formando aristas). Además, necesitamos encontrar un modo de proyectar nuestros puntos tridimensionales contra un plano, que se corresponderá a nuestra pantalla, de modo que podamos ver una representación 2D de nuestro objeto tridimensional.

En la figura [7.31] podemos ver un ejemplo de proyección paralela, donde se trazan líneas paralelas desde cada punto del objeto y se calcula su intersección con respecto a un plano bidimensional. Luego, los puntos resultado de la intersección se unen entre sí, siendo las líneas resultantes las aristas proyectadas de nuestro objeto tridimensional. Como veremos más adelante, a nivel de implementación, la proyección paralela es muy sencilla de realizar.

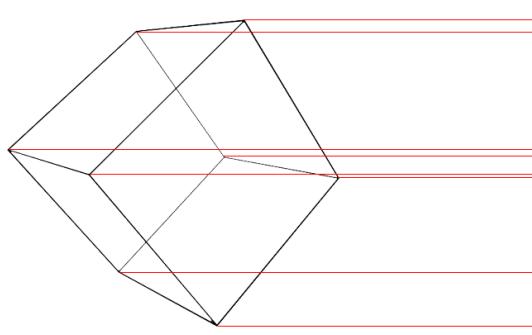


Figura 7.31: Cubo 3D que se proyecta contra el plano (proyección paralela)

Así pues, empezamos por crear dos nuevas estructuras, como podemos ver en la figura [7.32]. Son el punto 3D, muy similar al punto 2D previamente creado, pero con una dimen-

³³[https://en.wikipedia.org/wiki/Clipping_\(computer_graphics\)](https://en.wikipedia.org/wiki/Clipping_(computer_graphics))

³⁴https://en.wikipedia.org/wiki/Back-face_culling

sión extra, y el objeto 3D. Un objeto 3D consiste en una serie de puntos que representan la posición en el espacio 3D de los vértices del objeto a representar y un conjunto de pares de índices. Cada índice referencia la posición de uno de los vértices en la lista de puntos. De este modo, cada par de índices representa una arista de nuestro objeto a dibujar. Adicionalmente se incluye un conjunto de puntos 2D al que llamamos *projectedPoints*. Este vector se corresponde con el vector de vértices, y contiene los puntos tridimensionales una vez que son proyectados contra la pantalla, y por tanto convertidos al espacio 2D.

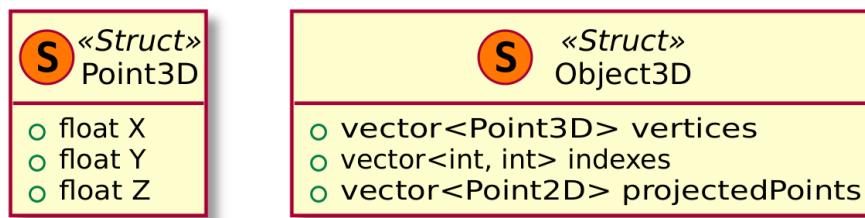


Figura 7.32: Estructura básica de Point3D y Object3D

Una vez tenemos un modo de representar nuestros objetos, es necesario hallar una forma de proyectarlos. Como hemos anticipado, la proyección de un objeto con respecto al plano $z = 0$ es extremadamente sencillo. Este es el plano que tiene su centro en el origen de coordenadas y se extiende infinitamente en x e y . La intersección de cualquier línea perpendicular a este plano con el mismo será equivalente a un punto de las características $(x, y, 0)$, dado que z siempre equivale a 0. Como para hallar nuestra proyección paralela lo que hacemos es hallar el punto de intersección de una recta perpendicular al plano de proyección que pasa por un vértice de nuestro objeto a proyectar, este punto de intersección siempre será del tipo $(x, y, 0)$. O en otras palabras, para obtener nuestro punto proyectado con respecto a $z = 0$, ¡todo lo que tenemos que hacer es tomar las coordenadas x e y e ignorar el valor de z !

Una vez tenemos una forma de representar un objeto 3D y una forma extremadamente sencilla de proyectarlo, llega el momento de dibujarlo. Como podemos ver en el código [7.19], todo lo que tenemos que hacer es recorrer todos los pares de índices y por cada par, obtener el punto de inicio y final de la arista proyectada. A continuación, dibujamos una línea que vaya de un punto al otro y ¡ya podemos dibujar figuras tridimensionales en pantalla!

Código 7.19: Código para dibujar un objeto 3D

```

1 void GeometryDemo::RenderObject(Object3D object, const Pixel &colour)
2 {
3     for (const Point2D& indexPair : object.indexes)
4     {
5         Point2D startPoint = object.projectedPoints[indexPair.X];
6         Point2D endPoint = object.projectedPoints[indexPair.Y];
7
8         RenderLine(startPoint.X, startPoint.Y, endPoint.X, endPoint.Y, colour, 1);
9     }
10 }

```

Una vez podemos crear, proyectar y dibujar nuestro objeto, todo lo que nos falta es añadir la capacidad de transformarlo. Todo lo que vimos y aplicamos para el espacio 2D es también extensible y aplicable al espacio 3D. De este modo, el orden de las transformaciones no es comutativo, y empezaremos escalando y rotando siempre en el origen, para a continuación trasladar la figura. El escalado sigue siendo tan sencillo como en el espacio 2D, consiste simplemente en multiplicar cada coordenada por un factor de escalado, y lo mismo se aplica para la traslación, que consiste en sumar un desplazamiento a cada coordenada.

El asunto cambia ligeramente para la rotación, sin embargo, y se vuelve algo más complejo. Si en el espacio 2D se rotaba con respecto a un punto (el origen), ahora, en el espacio 3D se rota con respecto a un eje (una línea) siendo estos los ejes x , y , z . Por tanto, necesitamos tres funciones distintas, para rotar nuestros vértices respecto a cada uno de los ejes. Además, algo nuevo a tener en cuenta, ¡el orden de las rotaciones tampoco es comutativo! Por tanto, el resultado de rotar primero en x y luego en y será distinto al de hacerlo primero en y y luego en x .

Código 7.20: Métodos para rotar en torno a los ejes, en el espacio 3D

```

1 void GeometryDemo::Rotate3DObjectAroundXAxis(Object3D &object, float angle)
2 {
3     for (Point3D &p : object.points)
4     {
5         p = Point3D(
6             p.X,
7             p.Y * cosf(angle) - p.Z * sinf(angle),
8             p.Y * sinf(angle) + p.Z * cosf(angle));
9     }
10}
11
12 void GeometryDemo::Rotate3DObjectAroundYAxis(Object3D &object, float angle)
13 {
14     for (Point3D &p : object.points)
15     {
16         p = Point3D(
17             p.X * cosf(angle) + p.Z * sinf(angle),
18             p.Y,
19             -p.X * sinf(angle) + p.Z * cosf(angle));
20     }
21}
22
23 void GeometryDemo::Rotate3DObjectAroundZAxis(Object3D &object, float angle)
24 {
25     for (Point3D &p : object.points)
26     {
27         p = Point3D(
28             p.X * cosf(angle) - p.Y * sinf(angle),
29             p.X * sinf(angle) + p.Y * cosf(angle),
30             p.Z);
31     }
32}
```

Recapitulando... ahora podemos crear nuestro objeto, proyectarlo, dibujarlo y transformarlo. No obstante, hay una cosa a tener en cuenta, para rotar nuestro objeto de forma coherente, siempre tiene que estar situado en el origen, pero si lo trasladamos, ¿entonces cómo podemos lograr que rote de forma coherente en el siguiente fotograma?

La respuesta es sencilla, antes de dibujar, aplicamos las transformaciones al objeto, y tras dibujar, deshacemos las transformaciones que hemos aplicado, de modo que el objeto vuelva a situarse en el origen. De este modo, nuestro objeto siempre estará virtualmente situado en el origen, y sólo lo moveremos en el paso previo al dibujado, para volver a dejarlo en su posición inicial tras el mismo. Deshacer las transformaciones aplicadas es sencillo, si movimos nuestro objeto en 10 unidades en x , ahora lo movemos en -10 unidades y así vuelve al origen. Si escalamos por 2, ahora escalamos por la inversa, $\frac{1}{2}$, para obtener la escala natural (1) y si rotamos en 90° en torno a x , ahora rotamos -90° en torno a x . Eso sí, algo muy importante a tener en cuenta: debemos deshacer nuestras transformaciones en el orden inverso del que las hicimos. En otras palabra, si para aplicar transformaciones escalamos, rotamos y trasladamos, para deshacerlas, trasladamos, rotamos y escalamos.

Código 7.21: Ciclo de actualización de un objeto en pantalla

```

1 bool GeometryDemo::Update(float deltaTime)
2 {
3     // Clear screen
4     EraseObject(objects[objectsIndex]);
5
6     // Apply transformations
7     ScaleObject(objects[objectsIndex], transformations[2]);
8     Rotate3DObjectAroundZAxis(objects[objectsIndex], transformations[1].Y);
9     Rotate3DObjectAroundYAxis(objects[objectsIndex], transformations[1].X);
10    Rotate3DObjectAroundXAxis(objects[objectsIndex], transformations[1].Z);
11    TranslateObject(objects[objectsIndex], transformations[0]);
12
13    //Draw object
14    RenderObject(objects[objectsIndex], Pixel(255));
15
16    // Undo transformations
17    TranslateObject(objects[objectsIndex], -transformations[0]);
18    Rotate3DObjectAroundXAxis(objects[objectsIndex], -transformations[1].Z);
19    Rotate3DObjectAroundYAxis(objects[objectsIndex], -transformations[1].X);
20    Rotate3DObjectAroundZAxis(objects[objectsIndex], -transformations[1].Y);
21    ScaleObject(objects[objectsIndex], transformations[2].inverse());
22
23    return true;
24 }
```

Ahora sí, ya tenemos todo lo que hace falta para dibujar un objeto y transformarlo libremente. Solo necesitamos definir los vértices y aristas de un objeto de nuestra inicialización y ya estamos listos para dibujarlo y transformarlo, como podemos ver en la figura [7.33].

7.7.4 Refinamiento

- **Proyección cónica:** si bien la proyección paralela es muy simple de calcular en nuestro efecto, también resulta poco natural, dado que no se ajusta a la forma en la que percibimos los objetos en la realidad. Es por ello que implementamos también un proyección cónica³⁵ muy sencilla, pero que es más cercana a la perspectiva con la que vemos en el mundo real. Para ello, haremos que el valor de las coordenadas x e y dependan del valor de la coordenada z . A mayor sea z y el objeto más lejos se encuentre, lo dibujaremos

³⁵https://es.wikipedia.org/wiki/Proyecci%C3%B3n_c%C3%B3nica

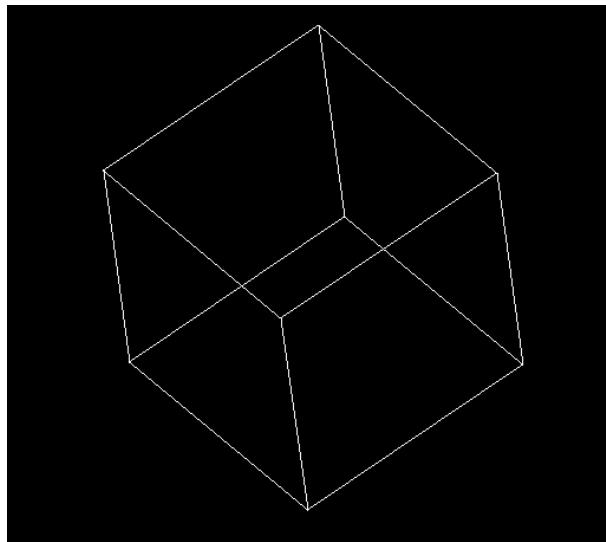


Figura 7.33: Cubo 3D con proyección paralela

más pequeño y a menor sea z (más cerca del origen, punto donde se sitúa la cámara), más grande se percibirá el objeto. Podemos ver un ejemplo simplificado en el código [7.22]. Multiplicamos la coordenada z por un factor definido por el usuario, para poder controlar así la fuerza de la proyección. Hay que tener en cuenta que hay que comprobar siempre que z no sea 0, pues la división por 0 no está definida. Esta comprobación se ha omitido en el ejemplo para hacerlo más sencillo. A continuación, dividimos nuestras coordenadas x e y por la profundidad calculada. Si nos fijamos, no obstante, antes de realizar el cálculo, añadimos la mitad de la anchura y la altura y tras el cálculo las sustraemos. Esta operación se realiza para situar el punto de fuga de la perspectiva en el centro. Si no hiciéramos esto, los objetos, al alejarse, tenderían hacia la posición $(0,0)$ en lugar de tender hacia el centro de la pantalla. Esto provocaría una sensación extraña, ya que veríamos los objetos alejándose hacia la esquina superior izquierda de la pantalla, lo que resulta antinatural. Este es el mismo artefacto que explicamos en el efecto anterior, y mostramos en la figura [7.28].

- **Múltiples objetos:** para desarrollar el efecto, añadimos la inicialización de un cubo, una figura sencilla con la que podíamos jugar libremente. Llega el momento de añadir algo más de variedad, por lo que añadimos además una pirámide y una estrella. El resultado se muestra en la figura [7.34].
- **Control del objeto:** como veíamos en el código [7.21], almacenamos nuestras transformaciones de translación, rotación y escalado en un vector de *Point3D* denominado *transformations*. Esto nos permite manipular de un modo similar cada tipo de transformación, con lo si definimos controles para alterar las coordenadas x , y y z , basta con cambiar el índice en nuestro vector de transformaciones para modificar posición, rotación o escala. Además, aprovechando que tenemos dos métodos distintos para calcular nuestra proyección, también es útil definir un control de usuario para alternarlo, así como también dar facilidad para cambiar el objeto a dibujar (almacenando nuestros

objetos en un vector). En el resultado final [7.35] se muestran también las instrucciones para manipular la demo.

Código 7.22: Cálculo de la perspectiva cónica

```

1 for(const Point3D& p : object.points)
2 {
3     float depth = p.Z * depthFactor;
4
5     object.projectedPoints.push_back({((p.X - halfWidth) / depth) + halfWidth,
6                                     ((p.Y - halfHeight) / depth) + halfHeight});
7 }
```

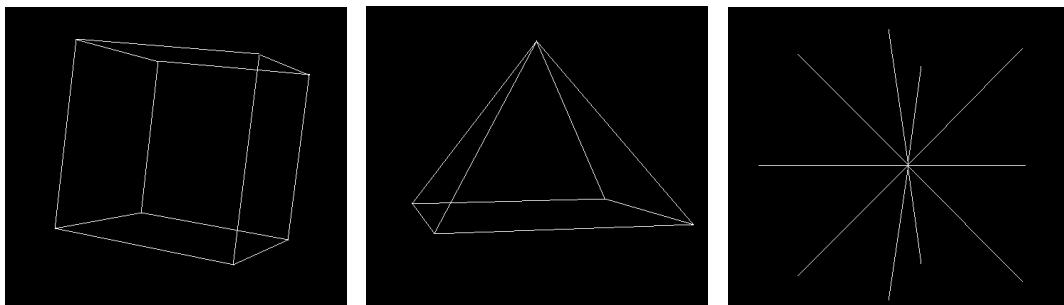


Figura 7.34: Distintos modelos en perspectiva

7.7.5 Resultado

En la figura [7.35] podemos ver un ejemplo del resultado final, donde hemos cambiado a la figura de la pirámide y la hemos rotado y escalado a voluntad. Además, tenemos disponibles las opciones adicionales para cambiar modelo y perspectiva.

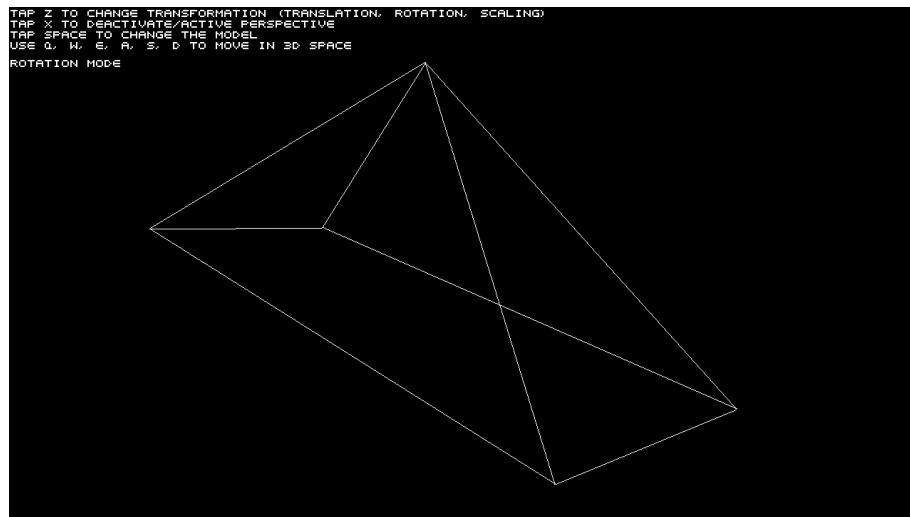


Figura 7.35: Pirámide rotada y escalada

8 Demo final

8.1 Introducción

Hasta ahora hemos estado revisando algunos de los efectos gráficos más conocidos y comunes en el mundo de la *demoscene*, analizándolos desde el punto de vista más analítico posible pero también intentando comprender su esencia y trasfondo.

Los efectos gráficos son el pilar, la base, que construye el mundo de la *demoscene*, pero un solo efecto no hace una demo, pues una demo consiste en un conjunto de efectos gráficos compilados en un solo ejecutable, normalmente acompañados además de música que se reproduce de forma sincronizada.

Tras haber estudiado todos los efectos expuestos anteriormente, llega el momento de utilizar el conocimiento adquirido para intentar generar una obra, una **demo**, lo más interesante posible. Para ello, será necesario no sólo aplicar lo aprendido, si no también saber hacerlo de una forma que tenga una cierta coherencia en conjunto, de modo que resulte visualmente agradable. No hemos de olvidar que al fin y al cabo la *demoscene* es tanto una práctica de ingeniería como de arte.

Si bien en este trabajo para nada se aspira a lograr una obra de arte, sí que se perseguirá un cierto sentido estético a lo largo de la composición, de forma que la compilación de todas las demos anteriores resulte lo más coherente y orgánica posible.

Como referencias a esta demo se pueden tomar todas aquellas que se han citado y mostrado previamente, pues esta demo pretende ser un humilde tributo y una humilde revisión de la cultura de la *demoscene*, yendo a sus orígenes y efectos más clásicos y trayéndolos de vuelta a los computadores de hoy en día, ejecutando únicamente por CPU y en tiempo real.

8.2 Planteamiento inicial

Para desarrollar esta demo hay varias limitaciones o retos de base que nos debemos plantear. En primer lugar, la música juega un factor clave en las demos, y sin embargo hasta ahora no tenemos ningún mecanismo para generar sonido.

Además, la demo se ejecutará exclusivamente en la CPU del ordenador, lo cual si bien resulta muy interesante, dado que pone en valor las capacidades de cómputo de un ordenador, también resulta un factor limitante, pues la manipulación de millones de píxeles por segundo no es una tarea trivial, y aún menos si hay operaciones matemáticas complejas de por medio.

Es por ello que deberemos aplicar todo lo aprendido para tratar de optimizar y estirar el rendimiento al máximo, y cuando esto no sea posible, buscar otras opciones o caminos que enmascaren las limitaciones técnicas de la máquina.

Por otro lado, los efectos gráficos que hemos mostrado hasta ahora son tan sólo muestras simplificadas de modo que resulten lo más explícitas y entendibles posibles, pero ahora es el momento de buscar resultados más complejos o interesantes a partir de la base que ya ha sido planteada.

Por último, los efectos gráficos creados hasta el momento son bastante distintos o inconexos entre sí, por lo que será importante encontrar un modo orgánico de generar transiciones entre los mismos o combinarlos de una forma coherente.

A grandes rasgos, estas son las tareas necesarias para elaborar nuestra demo final:

- Permitir la generación de sonido o música
- Aplicar o combinar todos y cada uno de los efectos gráficos explicados anteriormente
- Aplicar música a la demo coordinada con los efectos gráficos

8.3 Generar sonido

Antes de saltar a la implementación, conviene explicar muy brevemente como se representa el sonido de forma digital. Se asume, no obstante, que se conoce de forma básica el funcionamiento del sonido (física de ondas) y su representación matemática. Como ya sabemos, el sonido no es más que una vibración, una oscilación y por tanto, un movimiento ondulatorio. Si se requiere de un breve repaso sobre el funcionamiento de una onda, se puede encontrar en el planteamiento formal del efecto de deformaciones de imagen [7.4]

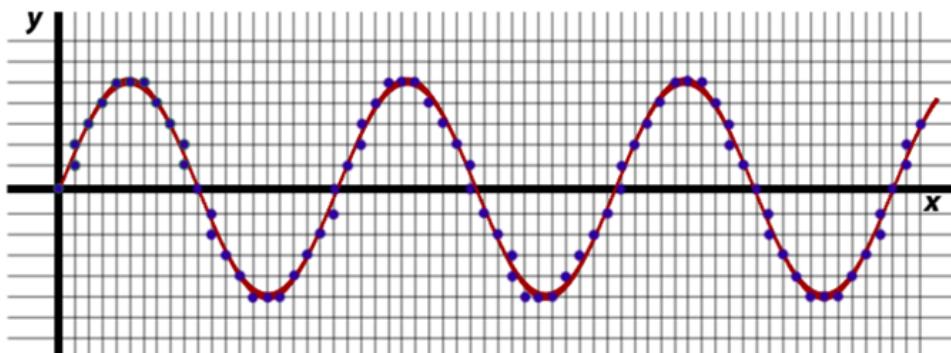


Figura 8.1: Onda sinusoidal y su discretización

Como podemos ver en la figura [8.1], un sonido se puede representar como una onda (o una combinación de ondas). En el dominio analógico (en rojo), una onda es continua, sin

embargo, en el dominio digital (en azul) solo se puede representar una cantidad discreta de valores de la onda, con la consiguiente pérdida de precisión. Es por ello, que para almacenar una onda en el dominio digital definimos una frecuencia de muestreo (equivalente a las líneas verticales de la imagen) y un formato o resolución de muestra (equivalente a las líneas horizontal en la imagen) de modo que cualquier valor intermedio, que no pueda ser representado, será aproximado al valor más cercano.

Habiendo dejado esto claro, pasamos a intentar generar sonido por computador. Siguiendo con la dinámica general de este trabajo, la generación de sonido debería ser, en la medida de lo razonable, gestionada por nosotros. La idea inicial es la de usar una librería de sonido para la música del mismo modo que usamos OpenGL para los gráficos, es decir, usar una librería que actúe simplemente como envoltorio y nos genere una capa de abstracción con respecto al sistema operativo, pero una vez hecho esto, generar el sonido desde cero.

Tras una breve investigación, las dos opciones más factibles parecían OpenAL¹ y PortAudio². Si bien OpenAL es el equivalente directo de OpenGL pero para audio, PortAudio acabó siendo la librería elegida. OpenAL es una librería más estandarizada y potente, que permite generar sonido en 3D y tiene un modo de funcionamiento similar al de su casi homónimo OpenGL. Pero esta potencia viene al coste de una mayor complejidad de uso. PortAudio, en cambio, siendo una librería también de código abierto y multiplataforma, se centra en la simplicidad. Y por ello mismo se optó por ella, ya que parecía innecesario tener que aprender a manejar toda una librería potente y completa con el mero objetivo de usarla como una abstracción de cara al sistema operativo.

Una vez tenemos la librería elegida, llega el momento de empezar a implementar nuestro sistema capaz de generar sonido. Todo lo que PortAudio necesita para empezar a funcionar es inicializar la librería y crear un flujo (*stream*) de sonido, al que se le pasa una función delegada controlada por el usuario.

Código 8.1: Código necesario para inicializar PortAudio

```
1 Pa_Initialize();
2 Pa_OpenDefaultStream(&stream, INPUT_CHANNELS, OUTPUT_CHANNELS, paFloat32, ↵
    ↵ SAMPLE_RATE, FRAMES_PER_BUFFER, AudioCallback, 0);
3 Pa_StartStream(stream);
```

Para entender no obstante, cómo funciona PortAudio y los parámetros que nos pide, debemos entender cómo funciona el audio por computador. Como podemos ver en el código [8.1], una vez inicializamos la librería, abrimos un flujo de sonido. Para hacer esto, no obstante, debemos pasar una serie de parámetros significativos. El primero de ellos, *stream*, se trata simplemente de una estructura del tipo *PaStream*. Esto es un tipo definido por los creadores de la librería y del que realmente no tendremos que preocuparnos, pues es gestionado internamente y no tendremos que realizar ningún tipo de operación con el mismo. La función principal de este tipo es la gestión de distintos canales de entrada y salida de sonido. A continuación debemos indicar los canales de entrada y de salida. Un canal de entrada se

¹<https://www.openal.org>

²<http://www.portaudio.com>

corresponde con una fuente de entrada de sonido. Aunque un canal no se corresponde necesariamente a un dispositivo, normalmente un canal de entrada se corresponde con un único dispositivo de grabación. Si en nuestra demo necesitásemos grabar audio, necesitaríamos entonces al menos un canal de entrada. Un canal de salida se corresponde normalmente, aunque no de forma necesaria, con un solo dispositivo de reproducción de audio, o en otras palabras, con un altavoz. Como en nuestra demo no necesitaremos grabar audio pero sí queremos reproducir audio estéreo, necesitaremos pues definir dos canales de salida.

Como nota, puntualizar que como se ha dicho anteriormente, un canal de entrada o salida no se corresponde necesariamente con un dispositivo físico. Esto es porque podemos por ejemplo definir dos canales de entrada que se correspondan con un único dispositivo de grabación, y sin embargo, dar a cada entrada de audio un tratamiento distinto. Por ejemplo, usar la entrada de un canal para generar eco y la del otro para generar distorsión, para posteriormente combinar los dos canales de entrada en un único canal de salida que tenga ambos efectos combinados. Del mismo modo, también es posible redirigir más de un canal al mismo dispositivo de reproducción. Por tanto, no existe una correspondencia directa entre canal y dispositivo, si bien es cierto que en muchos casos la suele haber.

Volviendo al código en [8.1], una vez hemos definido que queremos dos canales de salida, llega el momento de definir el formato de muestra. Esto es, definir qué formato tendrá una única muestra de sonido. El valor de una muestra representa el valor de la amplitud del sonido en un instante dado. El sonido de los primeros ordenadores, el tan conocido como *música de 8 bits*, tenía un formato de 8 bits interpretados como un entero por muestra. Esto quiere decir que la unidad mínima de sonido reproducible ocupaba 8 bits, y por tanto podía tener 256 valores distintos para la amplitud, que, para tratarse de sonido, podemos apreciar que es una resolución muy baja. De ahí que la música de 8 bits sonase robótica y poco orgánica, entre otras causas. De hecho, la música en 8 bits tan solo permitía 128 valores distintos, si tenemos en cuenta que en una onda que oscila en el origen, la mitad de los valores están por encima del cero y la otra mitad por debajo, por lo que de forma efectiva, contamos con 128 valores y su equivalente negativo. La música en 16 bits, que fue el siguiente paso, ya permitía definir más de 64000 valores distintos para la amplitud. Si escuchamos de hecho la diferencia entre la música de 16 bits y la música de 8 bits, se denota un cambio significativo. El formato que nosotros definimos en nuestra demo, no obstante, es el de un número en coma flotante de 32 bits. Nuestra amplitud máxima será 1 y nuestra amplitud mínima del sonido generado será -1. No obstante, como trataremos con números decimales, dispondremos de una gran resolución.

A continuación, una vez definido el formato de muestra (cuántos bits por muestra y cómo se deben interpretar -entero, coma fija, coma flotante...-) definimos el ratio de muestra, comúnmente denominado como la *frecuencia de muestreo*, o en otras palabras, cuántas muestras queremos por segundo. Tal y como indica el teorema del muestreo de Nyquist³, para generar un sonido a una frecuencia determinada, necesitamos al menos el doble de muestras por segundo que la frecuencia que se pretende muestrear. De media, el ser humano es capaz de percibir frecuencias de entre 20 y 20000 Hercios, de modo que si queremos tener la

³https://es.wikipedia.org/wiki/Teorema_de_muestreo_de_Nyquist-Shannon

habilidad de generar cualquier frecuencia audible, necesitaremos al menos 400000 muestras por segundo. En nuestra demo definimos una frecuencia de muestreo de 44100 muestras por segundo. El motivo de la elección de este número se debe a motivos históricos, ya que era la frecuencia de muestreo de los CD, ligeramente superior al espectro de sonido audible por cuestiones de formato y conveniencia⁴.

A continuación debemos definir la cantidad de muestras por *buffer*. Como acabamos de explicar, para reproducir un sonido en cualquier frecuencia audible, es necesario contar con al menos 40000 muestras por segundo, y en nuestro caso definimos 44100 muestras de sonido por segundo. Pasar estas muestras una a una sería extremadamente poco eficiente, por no decir imposible. La tarjeta de sonido es la encargada de generar y reproducir audio, de modo que cada vez que reproducimos audio, la CPU debe comunicarse con la tarjeta de sonido. 40000 accesos por segundo a la tarjeta de sonido para enviar un solo dato es una locura, y muy lento. Es por ello que se define un *buffer*. Cuando la CPU le pasa datos a la tarjeta de sonido, no lo hace de uno en uno, si no que manda la información en bloques de datos, reduciendo así la cantidad de comunicaciones con la tarjeta de sonido, que son operaciones bloqueantes. Con este parámetro, podemos definir el tamaño de los bloques de datos que se le pasan a la tarjeta de sonido. Bloques muy pequeños implican muchos accesos a la tarjeta de sonido, bloques muy grandes implican una gran cantidad de datos que transferir y una tasa de actualización muy baja (dado que nuestra función delegada se encarga de generar un bloque de datos por llamada, contando con la información en el momento de llamada, por lo que si esta información se actualiza a mitad de la generación de un bloque, la actualización no se verá reflejada hasta la siguiente llamada a nuestra función). Por tanto, conviene elegir un tamaño de *buffer* que resulte razonable, ni demasiado pequeño, ni excesivo. En nuestra demo definimos un tamaño de 256 muestras por *buffer*, lo que se traduce en unos 170 accesos a la tarjeta de sonido por segundo, y unas 170 llamadas a nuestra función delegada por segundo. Del mismo modo, el tamaño de cada *buffer* será de 2KB (4 bytes -32 bits- por muestra, dos canales, 256 muestras por canal por *buffer*), un tamaño que no resulta trivial pero que es muy pequeño.

Tras ello, los siguientes dos parámetros que hemos de pasar son una función delegada a la que PortAudio llamará de forma interna para generar sonido y, de forma opcional, una estructura definida por el usuario. En nuestro caso, realizaremos todas las operaciones necesarias desde la función delegada, y tenemos todos los datos que necesitamos en nuestra clase. Podremos acceder a estos datos desde nuestra función delegada, ya que es un miembro estático de nuestra clase para reproducir audio. Por tanto, pasaremos un 0 (también sería posible y equivalente en este caso pasar un *nullptr*) para indicar que no haremos uso de ninguna estructura de datos definida por el usuario.

Llega ahora el momento de echar un vistazo a la función delegada que puede ser definida por el usuario:

Código 8.2: Función delegada que pasamos a PortAudio

⁴https://es.wikipedia.org/wiki/Frecuencia_de_muestreo

```

1 int Imp_Audio::AudioCallback(const void *inputBuffer, void *outputBuffer,
2                               unsigned long framesPerBuffer,
3                               const PaStreamCallbackTimeInfo *timeInfo,
4                               PaStreamCallbackFlags statusFlags,
5                               void *userData)
6 {
7     float *out = (float *)outputBuffer;
8     static long int currentCount = 0;
9
10    for (unsigned long i = 0; i < framesPerBuffer; i++)
11    {
12        currentCount++;
13
14        Imp_Audio::UpdateNotes(currentCount);
15
16        *out++ = Imp_Audio::GetLeftValue(); /* left */
17        *out++ = Imp_Audio::GetRightValue(); /* right */
18    }
19    return 0;
20}

```

Aunque la cantidad de parámetros que recibe la función delegada puede abrumar a primera vista, la realidad es que apenas usamos unos pocos, como podemos ver en el código [8.2].

No usamos el *inputBuffer*, dado que no hemos definido ningún canal de entrada, del mismo modo que tampoco usamos las variables *timeInfo*, *statusFlags* o *userData*, las dos primeras porque son variables que nos aportan información extra pero que no nos resultan especialmente relevantes y la última porque es el argumento que se corresponde con la estructura de datos definida por el usuario que en nuestro caso hemos decidido no definir. Por tanto, sólo estamos interesados en dos variables, el *outputBuffer*, que se corresponde con el *buffer* para la salida de datos, es decir, es el *buffer* cuyos datos son pasados a la tarjeta de sonido para ser reproducidos por el ordenador y por otro lado, la variable *framesPerBuffer*, que recordemos que habíamos definido previamente con el valor 256 y que nos indica por cada llamada a la función, cuántas muestras por canal debemos incluir en el *buffer*. Si nos fijamos en las líneas [16] y [17], veremos que en estas líneas asignamos un valor a la posición actual del *buffer* y a continuación la incrementamos. Como podemos recordar, hemos definido dos canales de salida de audio, dado que queremos audio estéreo. Esto implica, por tanto, que nuestro *buffer* deberá ser rellenado con muestras para ambos canales. Estas muestras se leen de forma intercalada, de modo que si visualizamos nuestro *buffer* como un *array*, su primer valor se corresponderá con la primera muestra del canal izquierdo, su segundo valor con la primera muestra del canal derecho, su tercer valor con la segunda muestra del canal izquierdo, y así sucesivamente... Esto nos permite, de forma bastante sencilla, asignar valores a nuestros canales de salida de audio.

Antes de continuar ahondando en el funcionamiento del sistema de sonido, vale la pena explicar su funcionamiento de forma genérica, a partir de lo previamente construido.

En nuestro sistema, tenemos un vector estático de notas. Podemos ver la estructura de una nota en la figura [8.2]. Por cada muestra que añadimos al *buffer* de salida, actualizamos los valores de todas las notas que se están reproduciendo actualmente, basándonos en la variable entera *currentCount*, que se actualiza por iteración. De esta modo, usamos *currentCount* para

actualizar el valor de nuestras notas de forma muy similar a como usamos el valor de *deltaTime* para actualizar nuestras demos. De hecho, ambas variables tienen una relación directa con el tiempo, ya que *deltaTime* representa el tiempo transcurrido desde el fotograma anterior mientras que *currentCount* representa el número de muestra que se está actualizando, y como sabemos, actualizamos 44100 muestras por segundo, por lo que cada 44100 actualizaciones de este valor, habrá transcurrido un segundo.

Una vez actualizamos el valor de nuestras notas, llega el momento de asignarlas a la salida. Para ello, llamamos a dos funciones que respectivamente nos devolverán el valor para la salida de audio izquierda y el valor para la salida de audio derecha, basándose en la posición de las notas que se están reproduciendo actualmente.

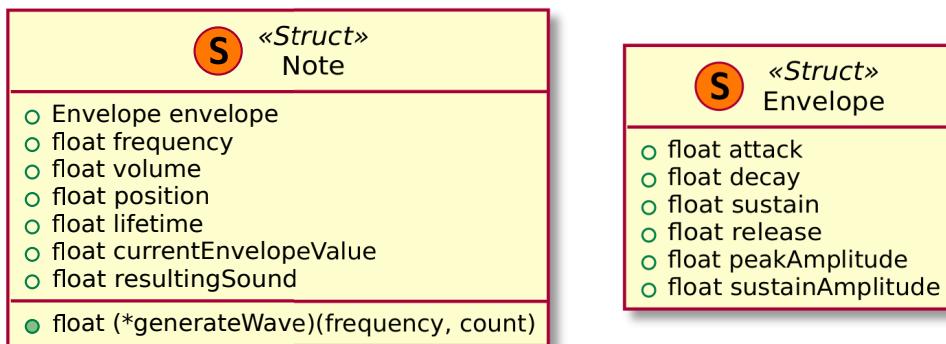


Figura 8.2: Estructura de una nota y su envolvente

Una vez que entendemos de forma simplificada como funciona nuestro sistema, llega el momento de definir qué es una nota y cómo se utiliza. Podemos ver su estructura en la figura [8.2].

Una nota musical se define principalmente por los siguientes parámetros: su forma de onda, su frecuencia, su volumen y su envolvente. La forma de onda en nuestro caso es generada por una función delegada, que a partir de la frecuencia deseada y el número de muestra, genera el valor correspondiente. De este modo, podemos asignar distintas formas de onda a distintas notas con gran facilidad, simplemente cambiando la función delegada de la misma. La frecuencia se corresponde con la frecuencia de la onda y el volumen con la amplitud, valor que oscilará entre 0 y 1. Por último, ya solo queda definir la envolvente, lo cual es algo más complejo.

La envolvente de una nota, o de un sonido en general, es algo así como "el ciclo de vida" de un sonido. Podemos ver la forma o estructura habitual de una envolvente en la figura [8.3]. A este tipo de envolvente se la denomina comúnmente envolvente *ADSR*, siendo estas siglas la denominación de las fases de la envolvente, en español, ataque, decaimiento, sostenimiento y relajación.

Vamos a clarificar qué es una envolvente con un ejemplo práctico: tocar una nota en un

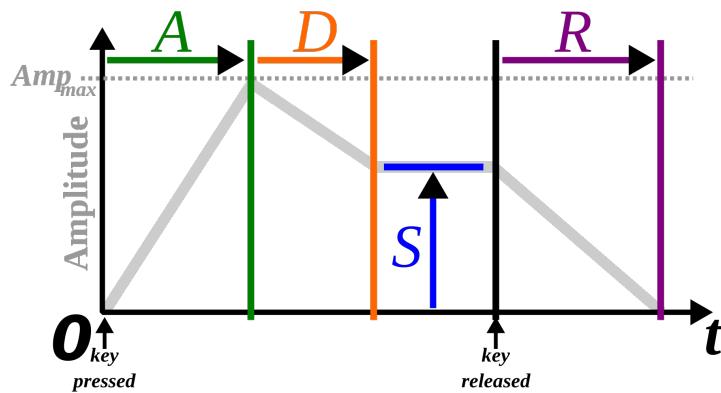


Figura 8.3: Envolvente de un sonido - Fuente: Wikipedia, por Abdull

piano real. Cuando tocamos una nota en un piano, hay un momento en el que se pasa del silencio a emitir un sonido. Esta es la fase de ataque, que se produce en el instante en que el martillo golpea las cuerdas. Tras esto, la amplitud (volumen) de la nota decae ligeramente, siendo esta la fase de decaimiento, pero, si mantenemos la nota pulsada, sigue sonando a una amplitud inferior, siendo esta la fase de sostenimiento. Ya por último, una vez dejamos de presionar la tecla, o si pasa el tiempo suficiente, el sonido empieza a decaer hasta que se desvanece, entrando por tanto en la fase de relajación. Prácticamente todos los instrumentos tienen estas fases, en mayor o menor medida. La envolvente de un sonido, por tanto, define la evolución del volumen del sonido a lo largo del tiempo, y define de forma drástica la forma en la que suena un instrumento. Por ejemplo, si pulsamos una tecla en un piano lentamente no sonará igual que si la pulsamos de golpe, porque tendrá una fase de ataque distinta, siendo la segunda mucho más breve. Además de definir estas cuatro fases, definimos dos variables más, la amplitud máxima y la amplitud de sostenimiento. La primera define el volumen que la nota alcanza tras la fase de ataque. La segunda define el volumen de la nota una vez que este decae y se sostiene, en la fase de sostenimiento.

Por tanto, pues, cada vez que actualicemos el valor de nuestra nota, también deberemos actualizar el valor de la envolvente en el ciclo de vida actual de la nota. Para ello, simplemente debemos crear una función que sea capaz de interpolar entre estos estados. Es decir, dado un tiempo de ataque, debe interpolar entre el reposo (0) y la amplitud máxima, una vez que se alcanza esta amplitud y acaba la fase de ataque, el volumen de la nota decae en la fase de decaimiento tanto tiempo como se especifique hasta estabilizarse en la amplitud de sostenimiento, que se mantendrá constante durante toda esta fase. Tras ello, entraremos en la última fase y en el final del ciclo de la vida de la nota, donde pasamos de la amplitud de sostenimiento al reposo de nuevo, el silencio. Es en este momento cuando termina el ciclo de vida de la nota.

Por tanto, en la estructura de la nota, que podemos ver en la figura [8.2], las primeras cuatro variables contienen información constante sobre la nota que se está reproduciendo, mientras que las tres últimas contienen estado: el tiempo de vida de la nota (cuando el tiempo de vida de la nota es igual a la duración de la envolvente, se considera que la nota ha terminado y por tanto se elimina de nuestro vector de notas), el valor de la envolvente para

el tiempo de vida actual, que dependerá de la fase de la envolvente en que nos encontramos y modificará el volumen de la nota y el sonido resultante, que se corresponde con el valor de retorno del método delegado que la nota contiene.

La única variable de la que aún no hemos hablado es en realidad una bastante interesante, la variable *position*. El valor de esta variable oscila entre 0 y 1, 0 representando el canal izquierdo y el 1 representando el canal derecho. El valor por defecto de esta variable es 0.5, que se corresponde al centro, o en otras palabras, nuestra nota sonando con la misma intensidad por el altavoz izquierdo y el derecho. Si asignamos a esta variable el valor 0, el sonido de nuestra nota se reproducirá solo por el canal izquierdo pero no por el derecho, y lo mismo se aplica a la inversa si aplicamos un valor de 1. Cualquier valor intermedio emitirá sonidos por ambos canales, tendiendo aquellos canales por debajo de 0.5 a la izquierda y aquellos por encima de 0.5 a la derecha.

Ahora que ya hemos explicado cómo funcionan las notas en nuestro sistema, podemos ver por fin el código para actualizarlas y reproducirlas por el canal izquierdo o derecho, como hacemos en el código [8.2].

Código 8.3: Actualización y obtención del valor de las notas

```

1 void Imp_Audio::UpdateNotes(long int currentCount)
2 {
3     for (auto &note : notes)
4     {
5         note.resultingSound = note.generateWave(note.frequency, currentCount) * note.currentEnvelopeValue * ↪
6             ↪ note.volume;
7     }
8
9 float Imp_Audio::GetLeftValue()
10 {
11     float sum = 0.f;
12
13     for (auto &note : notes)
14     {
15         float leftAmplitude = 1.f;
16         if (note.position > 0.5f)
17         {
18             leftAmplitude -= (note.position - 0.5f) * 2.f;
19         }
20
21         sum += note.resultingSound * leftAmplitude;
22     }
23
24     return sum;
25 }
```

En el código [8.3] se aporta el método para actualizar notas y el método para obtener el valor de muestra para el canal izquierdo. Se omite el del canal derecho ya que es prácticamente equivalente en funcionalidad al del izquierdo.

Para actualizar el valor de una nota, lo que hacemos es multiplicar el resultado que nos devuelve nuestro método delegado (en función de la frecuencia y el tiempo actual) por el

valor actual de la envolvente de la nota y el volumen general de la nota. Las envolventes no se actualizan en este bucle, si no que son actualizadas en la función *Update* de la propia clase, en lugar de actualizarse dentro de la función delegada que es gestionada por PortAudio. Esta decisión se ha tomado de modo que sea más fácil gestionar la actualización de la envolvente, que depende directamente del tiempo, por lo que es más fácil de actualizar con un intervalo de tiempo (*deltaTime*) que no con un número de muestra (*currentCount*) y también para aliviar la cantidad de cálculos, pues en lugar de tener que actualizar la envolvente por muestra (44100 veces por segundo) la actualizamos por fotograma (unas 60 veces por segundo). Esta decisión tiene sus ventajas e inconvenientes, pues por un lado implica no tener que estar constantemente actualizando la envolvente pero por otro, también facilita que se puedan producir pequeños cortes o cambios bruscos en la intensidad del sonido. En general, y bajo opinión personal, pienso que actualizando la envolvente una vez por fotograma el resultado es suficientemente satisfactorio, pero si se quisiera actualizar por muestra, sería tan sencillo como invocar a la función de actualizar envolvente en el bucle del código [8.2], pasándole por valor $1 \div 44100$, es decir, la cantidad de tiempo que transcurre de una muestra hasta la siguiente.

A continuación, la función *GetLeftValue* calcula el valor para el canal izquierdo por nota y lo suma. Si el valor de la variable *position* de la nota se halla entre 0 y 0.5, entonces el factor de amplitud en el canal izquierdo para esa nota será de 1, pero el factor para esa nota en el canal derecho será menor que 1. Del mismo modo, si la posición de la nota es superior a 0.5, entonces el factor de amplitud en el canal izquierdo será menor, o en otras palabras, la nota sonará con menor intensidad por el canal izquierdo. Por tanto, situada en el centro (0.5), una nota sonará con su amplitud natural tanto por el canal izquierdo como por el derecho, mientras que si no se sitúa en el centro, su amplitud será menor en un canal o en otro. Una versión refinada de esta implementación sería una el que que el módulo del sonido total de la nota siempre fuera 1, por lo que en el centro, el factor para el canal izquierdo y derecho sería $\sqrt{\frac{1}{1+1}} = \sqrt{\frac{1}{2}} \approx 0.707$. No obstante, para evitar complejidad añadida y una mayor carga computacional, se ha optado por la solución que se muestra en código, más sencilla y con un resultado práctico similar.

Como podemos ver en la función *GetLeftValue* en el código [8.3], el valor de cada nota se suma acumulativamente y se devuelve como el valor total para el canal. Recordemos que este valor se corresponde al de la amplitud general de la salida de sonido para el canal, por lo que debe estar comprendido entre -1 y 1. De lo contrario, si sobrepasamos este límite, se producirán artefactos de sonido extraños y desagradables, que en el peor de los casos podrían llegar incluso a dañar nuestros altavoces (aunque en nuestro caso no corremos ese riesgo, dado que PortAudio se encargará de filtrar todos aquellos valores que se salgan de rango). No obstante, queda en manos de quien añada sonidos asegurarse de que la suma de los sonidos no sobrepease el umbral máximo. Por ejemplo, si hacemos sonar dos instrumentos a la vez, deberíamos hacer que cada uno sonase a la mitad de su amplitud, de modo que sumados, como máximo, sumasen la amplitud máxima. De este modo, dejamos al usuario la decisión de la masterización del sonido, que si bien implica una responsabilidad (de lo contrario se producirán artefactos de sonido desagradables) también otorga una mayor flexibilidad.

Así pues, recapitulando, hemos creado un sistema que nos permite gestionar y reproducir

notas de sonido complejas, con una envolvente asociada y en estéreo. No obstante, y antes de continuar a la siguiente sección, aun nos queda algo fundamental por definir, ¡una forma de generar ondas de sonido!

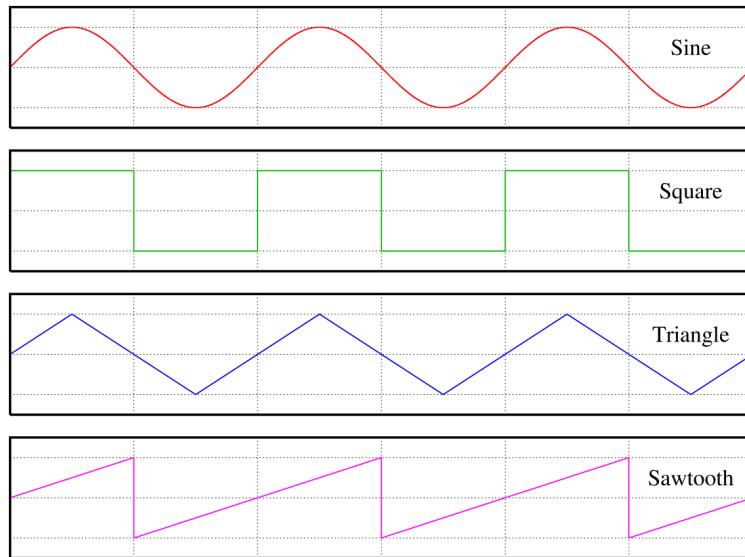


Figura 8.4: Distintas formas de onda - Fuente: Wikipedia, por Omegatron

Históricamente, encontramos cuatro tipos distintos de ondas generadas por ordenador, que podemos ver en la figura [8.4]. La onda cuadrada y la de diente de sierra son ondas que suenan muy robóticas o metálicas, por tanto muy lejanas al sonido natural. Sin embargo, son muy fáciles de calcular, ya que tienen un muy bajo coste computacional, por lo que los primeros ordenadores y videoconsolas que podían generar sonido, tenían capacidad para generar sólo este tipo de ondas. Al fin y al cabo, para generar un sonido se requiere una gran cantidad de muestras por segundo, por lo que en un ordenador de poca potencia, el cálculo para hallar el valor de una muestra debe ser lo más rápido posible, o de lo contrario, ralentizará la ejecución del programa. Es por ello, por tanto, que se generaban ondas cuadradas (que oscilan entre dos únicos valores, *+amplitud* y *-amplitud*) y las ondas de diente de sierra, cuyo valor de la amplitud se incrementa a lo largo de la longitud de la onda para decaer a 0 al final de cada iteración.

Las ondas triangulares son ondas que, siendo mucho más sencillas de calcular que una onda sinusoidal, resultan mucho más orgánicas que las dos anteriores, pues consiste en una interpolación continua entre la amplitud máxima y la máxima negativa. Tiene una carga de cómputo algo más elevada que la de las ondas cuadrada y de diente de sierra pero, sin embargo, ofrece un resultado sonoro bastante más orgánico a un coste relativamente bajo.

Por último tenemos las ondas sinusoidales. En la naturaleza, o en el mundo real, el sonido se propaga de manera natural con esta forma de onda. Sin embargo, como bien sabemos, el cálculo del seno es una operación matemática para nada trivial, por lo que este tipo de onda tardaría un tiempo en llegar a los ordenadores. Siempre se podría usar tablas precalculadas

en lugar de la operación matemática, pero recordemos que el uso de tablas precalculadas también introduce un cierto factor de complejidad y un error añadido, por su limitación en precisión. Nuevamente, la elección de una técnica u otra depende de la consideración personal.

Código 8.4: Cálculo de una onda sinusoidal con una frecuencia determinada

```

1 float Sounds::GetSineWaveValue(float frequency, long int currentCount)
2 {
3     int steps = SAMPLE_RATE / frequency;
4
5     float percentage = currentCount % steps / float(steps);
6
7     return sin(2 * Fast::PI * percentage);
8 }
```

En el código [8.4] vemos la implementación de la función para generar ondas sinusoidales. Como podemos ver, cumple la signatura del método delegado de una nota [8.2], de modo que podemos hacer fácilmente que una nota reproduzca sonido con forma de onda sinusoidal.

El sonido que generamos depende de la frecuencia, por lo que lo primero que hacemos es calcular, para la frecuencia dada, la cantidad de muestras necesarias para generar un único ciclo o iteración de la onda. Este valor se corresponde con la frecuencia de muestreo dividida por la frecuencia deseada. Por ejemplo, si tenemos una frecuencia de muestreo de 2000 muestras por segundo y queremos emitir un sonido a 200 Hercios, por tanto, 200 oscilaciones por segundo, entonces se deberá producir una oscilación a cada $\frac{2000}{200} = 10$ muestras.

Una vez hemos hallado la cantidad de muestras por oscilación, hallamos el punto o porcentaje en el que nos encontramos dentro de la oscilación. Para hacer esto, hallamos el módulo del número de muestra actual en función de la cantidad de muestras por oscilación y lo dividimos por la cantidad de muestras por oscilación. Por ejemplo, si estamos en la muestra 25 y tenemos 10 muestras por oscilación, entonces hallamos el módulo $25 \bmod 10 = 5$ y lo dividimos entre la cantidad de muestras por oscilación $\frac{5}{10} = 0.5$, hallando que nos encontramos a la mitad de la oscilación.

Una oscilación completa equivale a una circunferencia completa, es decir, 2π , por lo que sabiendo el punto de la oscilación en que nos encontramos, sólo tenemos que multiplicar por 2π y calcular el seno del valor obtenido para hallar la amplitud de nuestra onda a una frecuencia determinada en un instante de tiempo dado.

Podríamos pensar que ya hemos acabado con la generación de sonido, pero aún nos queda ser capaces de generar un sonido fundamental, ¡el ruido!

El ruido en esencia es la aleatoriedad, la desorganización, el caos. Un ruido normalmente tiene valores de amplitud para una gran cantidad de frecuencias en el espectro. Todo aquel sonido que no es armónico (no se repite periódicamente) puede ser potencialmente considerado un ruido.

Nos podemos preguntar para qué es necesaria la generación de ruido. La respuesta es que es

fundamental. La mayoría de instrumentos de percusión, por ejemplo, no son más que generadores de distintos tipos de ruido. El sonido de un tambor, por ejemplo, no es más que un ruido con una envolvente, de modo que tiene un ataque y decaimiento muy breve y, dependiendo del tambor, una relajación más o menos duradera. Adicionalmente, también podemos usar el ruido para generar sonidos de ambiente (agua, viento...) o para dar un toque más orgánico a un instrumento (ya que todos los instrumentos generan una pequeña cantidad de ruido. Por ejemplo, al tocar un piano, el martillo golpeando las cuerdas genera algo de ruido además de un sonido armónico, y si eliminamos este ruido de fondo del sonido del instrumento al generarlo por computador, el resultado se escuchará mucho más artificial).

Generar un ruido blanco (aquel que tiene amplitud en todas las frecuencia del espectro⁵), es extremadamente sencillo. Como ya hemos dicho, un ruido es aleatoriedad, por lo que para generar un ruido blanco, tan sólo necesitaremos generar un valor aleatorio comprendido entre -1 y 1 por muestra.

Sin embargo, el ruido blanco, si bien sencillo de generar, no se encuentra presente en la naturaleza, y resulta por tanto un sonido poco orgánico, bastante artificial. Es un sonido, por ejemplo, como el que generaban los antiguos televisores cuando perdían la señal.

En la naturaleza, el ruido suele tener mayor amplitud en las frecuencias más bajas del espectro y menor amplitud o nula en las frecuencias altas. Es por ello que necesitaremos generar un filtro de pasa baja⁶. Un filtro de pasa baja nos permite filtrar las frecuencias altas, atenuándolas o eliminándolas, y dejando pasar solo las frecuencias bajas. De forma alternativa, generaremos también un filtro para ruido de pasa alta, que atenúa o elimina las frecuencias bajas. Si bien es posible que acabemos no haciendo uso de este filtro, es interesante disponer del mismo para, potencialmente, poder usarlo en algunos instrumentos.

Sin entrar en excesivo detalle, haremos uso de la fórmula para generar estos filtros con retroalimentación, que podemos ver en la figura [8.5].

$$y[n] = \alpha x[n] + (1 - \alpha)y[n - 1] \quad (8.1)$$

$$y[n] = \alpha y[n - 1] + \alpha(x[n] - x[n - 1]) \quad (8.2)$$

Figura 8.5: Filtros de pasa baja y de pasa alta

El valor de la amplitud de un sonido al que se le aplica un filtro de pasa baja [8.1], en un instante dado, equivale al valor de la muestra en ese instante multiplicado por un valor de intensidad definido por el usuario, sumado a la parte restante del valor de intensidad por la salida del filtro en el instante de tiempo anterior.

⁵https://es.wikipedia.org/wiki/Espectro_de_frecuencias

⁶https://en.wikipedia.org/wiki/Low-pass_filter

Hallar el resultado de un sonido al que se le aplica un filtro de pasa alta [8.2] es ligeramente más complejo, consistiendo en multiplicar el valor de la intensidad por el resultado anterior del filtro sumado a la intensidad multiplicada por el valor del sonido en el instante actual menos el valor del sonido en el instante anterior.

En el código [8.5] podemos ver la implementación de estos filtros para la generación de ruido. El valor de la intensidad se refiere a la severidad con la que el filtro se aplica, y oscila entre 0 y 1, de modo que con un valor de 1 deja pasar todas las frecuencias y con un valor cercano a 0, deja pasar sólo las frecuencias más graves o más agudas (dependiendo del filtro).

Código 8.5: Aplicación de un filtro de pasa baja y uno de pasa alta a la generación de ruido

```

1 float Sounds::GetLowPassNoiseValue(float intensity)
2 {
3     static float oldValue = 0;
4     float newValue = intensity * GetNoiseValue() + (1 - intensity) * oldValue;
5     oldValue = newValue;
6     return newValue;
7 }
8
9 float Sounds::GetHighPassNoiseValue(float intensity)
10 {
11     static float oldValueY = 0.f;
12     static float oldValueX = 0.f;
13
14     float newValueX = GetNoiseValue();
15     float newValueY = intensity * oldValueY + intensity * (newValueX - oldValueX);
16
17     oldValueX = newValueX;
18     oldValueY = newValueY;
19
20     return newValueY;
21 }
```

Ahora sí, ya disponemos de las herramientas y el marco de trabajo necesarios para poder generar prácticamente cualquier sonido en nuestra demo, de modo que podemos pasar a la creación de la misma.

8.4 Crear la demo

El proceso de creación de la demo, a decir verdad, es más un proceso de prueba y error que un proceso técnico. Al fin y al cabo, la parte técnica ya ha sido implementada en los efectos gráficos anteriores, por lo que ahora lo más relevante es probar y modificar valores para obtener los resultados que deseemos. Como ya se ha comentado anteriormente, buscamos un resultado lo más estético posible.

Desde un primer momento hubo una idea que me llamó mucho la atención. Dado que teníamos la capacidad de generar fuego y la capacidad de generar texto, parecía muy interesante tratar de implementar como inicio de la demo un texto de fuego. Tras iterar varias veces sobre la idea y algo de prueba y error, la mejor opción parecía la de crear un texto que apareciera con un *zoom* de entrada y se situase en el centro, con el título de la demo, y prendido en fuego.

Para ello, por tanto, era necesario tener un título. Este título debía ser capaz de expresar la esencia y contenido de la demo, y poner en valor sus puntos fuertes. ¿Cuál es el punto fuerte de la demo que quiero hacer? ¿Cuáles son sus limitaciones?

Bajo mi punto de vista, su mayor ventaja e inconveniente es uno solo: está generada usando únicamente la CPU del ordenador, sin operaciones con gráficos aceleradas por *hardware*, de modo que esto actúa como un limitante para la potencia o capacidad de la demo pero también como un punto fuerte y de interés, pues demuestra hasta qué punto puede ser potente una CPU en el manejo de operaciones gráficas. Es por ello que se optó por un nombre simpático y directo "CPU-T-U", que leído que inglés suena de forma similar a *CPU to you*, o *CPU para ti* en español. Además, decidí incluir como subtítulo el nombre del creador, pero siguiendo la moda tan común en el mundo de la *demoscene* de usar apodos en lugar de nombres reales, utilicé el apodo que uso en redes sociales, *donluispanis*.

La implementación fue bastante sencilla, recordemos que para el efecto de fuego [7.1], lo que hacíamos era convolucionar una matriz a lo largo de una matriz de valores, de forma que se podría un efecto de disipación que, al relacionarlo con un degradado de color, se asemejaba al fuego. Todo lo que necesitamos hacer ahora es, en nuestra matriz de valores, en lugar de asignar unos pocos valores aleatorios en la parte inferior de la misma, asignar valores de modo que formen letras, y a continuación, convolucionar la tabla para que parezca que las letras están en llamas.

Como ya habíamos implementado en nuestro motor gráfico una función para dibujar texto en pantalla, todo lo que necesitábamos hacer era copiar esta funcionalidad y modificarla ligeramente para permitir dibujar texto, en lugar de en una cadena de píxeles que se corresponden a la pantalla, en una cadena de números enteros que se corresponden a una textura. Una vez tenemos esto, tan sólo tenemos que usar el algoritmo para generar el efecto de fuego, que ya vimos en [7.4], modificando levemente valores hasta que el resultado obtenido sea de nuestro agrado.

Llegados a este punto, obtenemos el resultado que podemos ver en la figura [8.6].

No obstante, y aunque estamos tan sólo al inicio, ya nos encontramos con las primeras limitaciones. Queremos que nuestra demo se reproduzca a pantalla completa, no obstante, si usamos la resolución nativa de una pantalla común, en *Full HD*, nos vemos con que tenemos que iterar a través de nada más y nada menos que $1920 \times 1080 = 2073600$ píxeles por fotograma. Si a esto añadimos que tenemos que mantener también una matriz de valores del mismo tamaño, vemos que de pronto la cantidad de cálculos a realizar por fotograma parece excesiva y muy difícil de mantener con una tasa de refresco aceptable.

Es por ello que llega el momento de tomar una primera y dolorosa decisión: aunque reproducimos en pantalla completa, nuestra demo tendrá una resolución fija en HD, significando esto que tan sólo tendremos que manipular $1280 \times 720 = 921600$ píxeles por fotograma. En la práctica, este cambio de resolución no es demasiado evidente, ya que delegamos en el sistema el reescalado de nuestra imagen en HD a la resolución nativa de la pantalla. De este



Figura 8.6: Inicio de la demo

modo, obtenemos un gran mejora de rendimiento (reducimos a más de la mitad la cantidad de cálculos necesarios por fotograma) al precio de perder resolución.

Este es un sacrificio necesario, pues si ya calcular el efecto de fuego resulta costoso, cuando intentemos generar efectos más complejos, nos resultará imposible de no reducir la cantidad píxeles sobre la que iteramos.

Además, para el efecto de fuego, hay una optimización más que podemos incluir. Si nos fijamos en la imagen [8.6], hay mucho espacio vacío en la misma. Por tanto, si en lugar de iterar sobre toda la imagen, sobre todos los píxeles, iteraremos sólo sobre el rectángulo que ocupa el título y el rectángulo que ocupa el subtítulo, nos ahorraremos tener que iterar sobre todos aquellos píxeles de espacio "vacío" que sabemos que siempre estarán en negro, reduciendo así aún más la cantidad de píxeles sobre la que operamos y aliviando la carga de cómputo.

Una vez hecho esto, y creados nuestro fundido de entrada (el texto se hace cada vez más grande hasta ocupar el centro de la pantalla) y nuestro fundido de salida (un simple fundido a negro, consistente únicamente en multiplicar la intensidad del píxel por un valor de opacidad decreciente), tan sólo nos queda añadir música para la introducción.

La introducción tiene un potente aire *retro*, con el texto *pixelado* y un efecto de fuego con una gama de color limitada (definimos un degradado de color de 256 valores). Por tanto, parece adecuado crear un sonido que siga la misma línea de estilo. Para ello, creamos la función delegada que podemos ver en el código [8.6].

Código 8.6: Generación de un sonido retro

```
1 float Sounds::CreateRetroSound(float frequency, long int currentCount)
2 {
3     return GetSawtoothWaveValue(frequency * 0.5f, currentCount) * 0.4f +
4         GetSquaredWaveValue(frequency, currentCount) * 0.4 +
```

```

5     GetSquaredWaveValue(frequency * 2.f, currentCount) * 0.15 +
6     GetNoiseValue() * 0.05;
7 }
```

Como podemos ver, el sonido que generamos es la combinación de una onda de diente de sierra como base sumada a dos ondas cuadradas, la segunda con el doble de amplitud que la primera, y añadiendo un pequeño ruido de fondo. La suma total de las ondas será siempre un valor comprendido entre -1 y 1. Nos aseguramos de esto multiplicando cada onda por un factor distinto, siendo la suma de todos estos factores 1. Al usar únicamente ondas de diente de sierra y cuadradas, que eran las únicas que los primeros ordenadores con sonido podían generar, obtenemos una sensación de sonido muy *retro*.

Usamos esta función delegada en combinación con una envolvente con un ataque lento para generar un sonido de intensidad creciente mientras que el texto aumenta de tamaño en pantalla. El sonido se estabiliza y desvanece conforme el texto desaparece.

Con esto, ya tenemos una introducción digna para nuestra demo, tanto a nivel visual como sonoro. Llega el momento de seguir, y entrar ya en el cuerpo de la demo. La forma de proceder me pareció clara, ya que en el proceso de creación de la intro, una idea arraigó fuertemente en mi cabeza.

Hemos estado hablando y tratando recientemente con ondas, especialmente a nivel sonoro, aunque también las hemos usado para generar efectos de deformación de imagen [7.4]. Además, hemos explicado la importancia de la generación de ruido, y cómo se puede usar un ruido filtrado, con un filtro de pasa baja, para generar sonido ambiente, como por ejemplo el del agua. Y es esto lo que me parece especialmente interesante, el agua en movimiento forma ondas, y a su vez, produce ondas sonoras. Usando el efecto de geometría [7.7], parecía bastante factible la posibilidad de generar una malla de puntos interconectados y actualizarla periódicamente de modo que ondulase. Con un par de retoques extra y efectos de sonido, se podría obtener un resultado de lo más satisfactorio.

Para ello, lo primero que tenemos que hacer es, como ya hemos comentado, crear un objeto 3D [7.32] que contenga una malla de puntos. Esto es, generar un rejilla de puntos, con dos simples bucles encadenados, e interconectar los puntos entre ellos. Para generar índices, lo que hacemos es unir cada punto con el punto a su derecha y el punto bajo él, como podemos ver en la figura [8.7]. De este modo, creamos una malla de puntos de forma sencilla.

Una vez tenemos nuestra malla de puntos creada y la posicionamos en escena, llega el momento de animarla, de modo que tenga un movimiento ondulatorio. Para ello, implementamos el código que podemos ver en [8.7].

Código 8.7: Aplicación de una deformación de onda a nuestra malla de puntos

```

1 void Imp_Geometry::ApplyWaveTransformation(Object3D &object, float amplitude, float wavelength)
2 {
3     grid.colours.clear();
4     for (float j = 0; j < vertexPerDepth; j++)
5     {
```

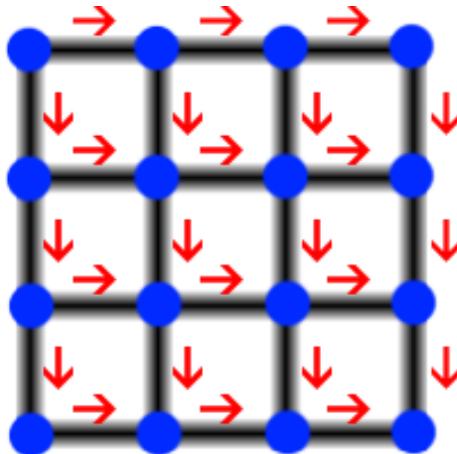


Figura 8.7: Generación de una malla de vértices

```

6     float wave = sin((j + phase) / wavelength);
7
8     for (float i = 0; i < vertexPerWidth; i++)
9     {
10        grid.points[j * vertexPerWidth + i].Y += amplitude * wave;
11
12        grid.colours.push_back((Pixel(0, 0, 125) + Pixel(255, 255, 125) * (1 - (wave + 1) * 0.5f)));
13    }
14  }
15 }
16 }
```

Como vemos, todo lo que hacemos es modificar la altura de cada vértice en función de una onda sinusoidal que depende de la posición en la que nos encontramos y la fase, que se incrementa en función del tiempo. Además, también cambiamos el valor del color de cada vértice en función de la onda. Esto es algo que no podíamos hacer previamente, pues en la demo de geometría [7.7], definíamos un solo color para todo el objeto. Ahora, nuestro objeto tiene la posibilidad de tener un color por vértice, y añadimos en la función del motor gráfico para dibujar un línea la posibilidad de definir un color inicial y un color final. Si proveemos esta información, entonces la función hace una simple interpolación lineal en la que el color de la línea cambia con su avance.

Por tanto, en la línea [13], lo que hacemos es definir un color base de valor (0, 0, 125), que se corresponde con azul oscuro y sumarle un valor (255, 255, 255) que se corresponde con blanco, de modo que cuando la onda se encuentre en su amplitud mínima (-1), el vértice se dibuje de color azul y cuando la onda se encuentre en su amplitud máxima (+1) el vértice se dibuje de color blanco.

Esta nueva posibilidad de coloración refuerza la sensación de estar observando un "mar digital". Podemos ver el resultado en la figura [8.8]. Como observamos, los vértices situados en la "cresta de la ola" son de color blanco, mientras que los situados en su parte más baja, son de color azul oscuro. El color se interpola a lo largo de las líneas y la malla se muestra completamente ondulada.

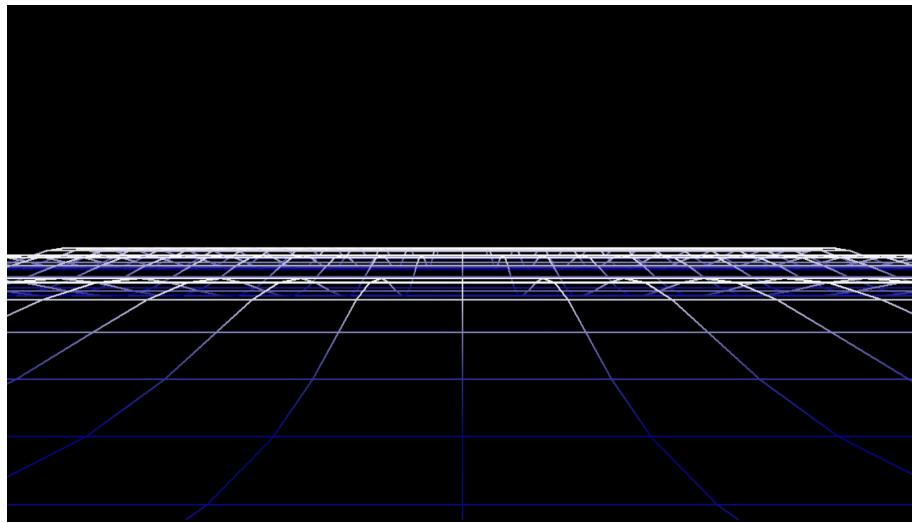


Figura 8.8: La malla de vértices ondulada y coloreada

Ahora, llega el momento de añadir un sonido que recuerde al mar. Esto lo hacemos con el código [8.8]. Realmente, todo lo que estamos haciendo aquí es devolver un ruido filtrado en pasa baja cuya amplitud o volumen es modulada por la frecuencia que pasamos a la función delegada. De este modo, si definimos una frecuencia de 0.5, por ejemplo, entonces se escuchará como una ola "aparece y desaparece" cada 2 segundos. Todo lo que queda hacer es que el sonido se genere de la forma más coordinada posible con respecto al movimiento de las olas para conseguir un resultado de lo más completo.

Código 8.8: Generación de un sonido de olas

```

1 float Sounds::CreateSeaWavesSound(float frequency, long int currentCount)
2 {
3     return GetLowPassNoiseValue(0.1f) *
4         GetSineWaveValue(frequency, currentCount) * 0.9 +
5         GetLowPassNoiseValue(0.1f) * 0.1;
6 }
```

Una vez llegados a este punto, no obstante, tenía claro que aún no había llegado el momento de abandonar la geometría. Es una lástima, teniendo un efecto tan potente, no explotarlo un poco más. La primera idea que visualicé fue el mar digital que había creado descomponiéndose en cubos individuales. No obstante, si bien esta idea me fascinaba, también planteaba muchos inconvenientes, dada la complejidad que suponía convertir un solo objeto 3D en múltiples instancias separadas pero interconectadas y con su propio ciclo de vida y animación. Parecía una idea excesivamente ambiciosa y arriesgada, y teniendo en cuenta la posibilidad de que el resultado final acabase distando de aquél que tenía en mente, decidí optar por otras opciones.

La siguiente idea que se me ocurrió me fascinó del mismo modo que la anterior, y sin embargo, tras reflexionar sobre la misma, parecía bastante más factible: hacer que nuestro

mar de puntos se transformase en una esfera perfecta.

Una de las formas en las que podemos generar una esfera en el espacio 3D es mediante el uso de *anillos*. Esto consiste, básicamente, en generar una esfera a partir de circunferencias. En la imagen [8.9] podemos ver un ejemplo de una esfera formada por anillos interconectados.

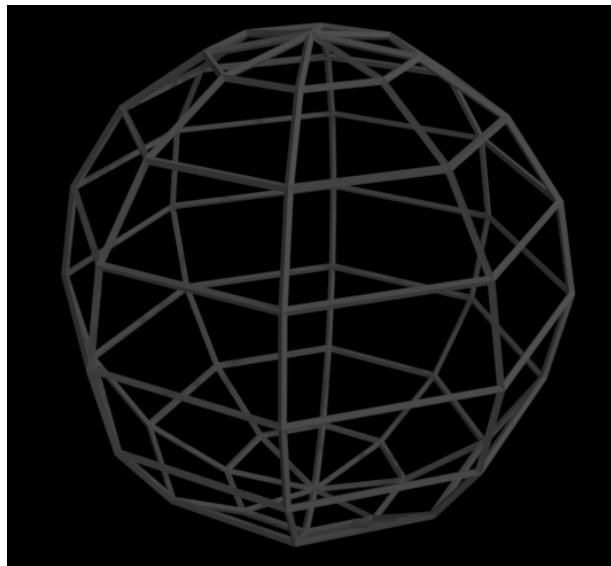


Figura 8.9: Esfera formada por anillos

Tras un tiempo de desarrollo, conseguí establecer una conexión entre los puntos de la malla que habíamos generado y los puntos que forman una esfera. Podemos ver el resultado visual en la figura [8.10]. Como podemos observar en la malla de este ejemplo, tenemos 7 grupos de color. Cada uno de estos grupos se correspondería con un anillo de la esfera. De este modo, los 4 vértices centrales, de color rojo, se corresponderían con el punto inferior de la esfera y los cuatro vértices de los extremos, en color naranja, convergerían en el punto superior de la esfera. Todos los vértices intermedios convergerían en distintos anillos, correspondiéndose, en el ejemplo, el color magenta con el anillo central de la esfera. Los colores azul claro y azul oscuro se corresponderían con el anillo inmediatamente superior e inferior al anillo central y así sucesivamente.

A continuación se adjunta el código que realiza esta asociación entre puntos de la malla y puntos de la esfera [8.9]. Es bastante denso y requiere de una explicación detallada, teniendo en mente y como referencia las figuras [8.9] y [8.10].

Código 8.9: Método que establece una relación entre los vértices de un plano y una esfera

```
1 Point3D Imp_Geometry::GetPointInSphereFromPlane(const int posX, const int posY, const int gridSize, ↵
    ↵ const float radius)
2 {
3     const int halfGrid = gridSize / 2;
```

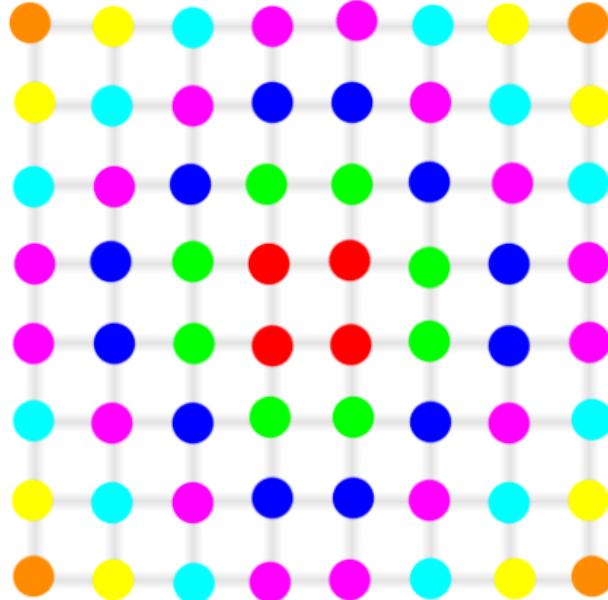


Figura 8.10: Relación entre una malla de puntos y una esfera

```

4
5     const int firstQuarterPosX = (posX < gridSize / 2) ? posX : gridSize - posX - 1;
6     const int firstQuarterPosY = (posY < gridSize / 2) ? posY : gridSize - posY - 1;
7
8     const int ring = gridSize - 1 - firstQuarterPosX - firstQuarterPosY; //The numbers of rings goes from ↪
9         ↪ 1 to gridSize - 1
10    const float radiusSign = (ring < halfGrid) ? -1.f : 1.f;
11
12    //top && bottom
13    if (ring == 1 || ring == gridSize - 1)
14    {
15        return Point3D(0.f, radius * radiusSign, 0.f);
16    }
17    //middle ones
18    else
19    {
20        const int positiveRing = (ring < halfGrid) ? ring : gridSize - ring;
21        const float perRingHeight = radius / float(halfGrid - 1);
22        float height = perRingHeight * (halfGrid - positiveRing) * radiusSign;
23
24        const float sine = height / radius;
25        const float circleRadius = radius * sqrt(1 - sine * sine); //circleRadius = radius * cos()
26
27        const float quarterCircle = Fast::PI / 2.f;
28
29        float quadrant = CalculateQuadrant(posX, posY, halfGrid);
30
31        const int positiveFirstQuarterPosX = (ring <= halfGrid) ? firstQuarterPosX - (halfGrid - ring) : ↪
32            ↪ firstQuarterPosX;
33        const float angle = ((positiveFirstQuarterPosX + 1) / float(positiveRing)) * quarterCircle + ↪
34            ↪ quadrant;
35
36        Point3D p = Point3D(circleRadius * cos(angle), height, circleRadius * sin(angle));
37        return p;
38    }

```

Empezamos estableciendo la siguiente relación: la cantidad de anillos de una esfera generada a partir de una malla es igual a la cantidad de vértices que forman la malla en una sola línea menos uno. En otras palabras, si tenemos una malla de 8x8, como la de la figura [8.10], entonces la esfera resultante tendrá 7 anillos, dos de los cuales se corresponderán con el punto superior e inferior de la esfera.

Además, para facilitar los cálculos, trasladamos cualquier punto de la malla al primer cuadrante de la misma, como vemos en las líneas [5] y [6]. De este modo, de cara a los cálculos, empezando en cero, los puntos (1, 6), (6, 6) y (6, 1) se asociarán al punto (1, 1) del primer cuadrante en la malla [8.10]. De este modo, podemos tratar todos los puntos como si estuvieran en el mismo cuadrante, simplificando así nuestra tarea.

Una vez calculado esto, hallamos el número de anillo en el que nos hallamos, valor que va de 1 a el tamaño de una línea de la malla menos 1, en nuestro ejemplo, de 1 a 7. Podemos ver este cálculo en la línea [8]. Como podemos ver, el anillo en que nos encontramos en la esfera se corresponde con el número de anillos menos la posición x e y que ocupamos en la malla (una vez trasladados al primer cuadrante). De este modo, ponemos de ejemplo el punto (7,6) que se corresponde con el punto (0,1) del primer cuadrante. Por tanto, hacemos el cálculo (*anillos* - 0 - 1) y obtenemos que nos hallamos en el anillo 6. Siendo el color rojo el primer anillo y el color naranja el último (7), vemos que efectivamente el color amarillo se corresponde con el anillo 6.

Una vez sabemos en qué anillo se encuentra nuestro punto, calculamos también su signo. Esto es, cualquier anillo por encima del anillo central tendrá un signo positivo y cualquier anillo por debajo, un signo negativo. Hacemos esto porque para formar nuestra esfera, la situamos en el origen de coordenadas, por lo que sabemos que el anillo central se situará en el plano $y = 0$, y cualquier anillo que no sea el central tendrá una altura positiva o negativa, por lo que necesitaremos hallar su signo.

Con esto hecho, comprobamos si nuestro punto se encuentra en el primer o último anillo y de ser así, lo situamos en el origen y especificamos su altura, que será el radio positivo o negativo de la esfera, dependiendo del anillo.

Si nuestro anillo a calcular es uno de los centrales, el cálculo se vuelve, sin embargo, algo más complejo, como podemos ver a partir de la línea [17]. Empezamos por calcular el valor absoluto del anillo en el que nos encontramos. De este modo, el anillo 2 y 6 son equivalentes, y el 3 y el 5 también, ya que realmente se trata del mismo anillo pero con alturas opuestas. Además, calculamos la altura a la que se situará nuestro anillo en la esfera. Para ello, empezamos por calcular la distancia que hay entre dos anillos, como vemos en la línea [20], que se corresponde con el radio dividido por la mitad del tamaño de la malla menos uno. De este modo, si tenemos una esfera con 8 vértices por línea y radio 90, la mitad de su tamaño menos 1 será 3 y por tanto el tamaño de cada incremento será de 30 unidades. Como vemos, esto concuerda con nuestro modelo, pues en nuestra esfera de siete anillos, con el anillo central situado en el origen, tenemos tres anillos por encima y tres anillos por debajo del central.

Si vamos en incrementos de 30 en 30 unidades, entonces nuestro anillo superior e inferior se corresponderán con el radio definido, como debería ser.

A continuación, pasamos a calcular la altura a la que se encuentra nuestro anillo, como vemos en [21]. Esta altura se corresponde con el valor positivo del anillo en el que nos encontramos restado a la mitad del anillo por el incremento por anillo por el signo. De este modo, si tenemos un punto que pertenece al anillo 6, este se corresponderá con el anillo 2. El anillo central en este ejemplo es el 4° , por lo que, $4 - 2 = 2$, el signo será positivo, pues el anillo 6 se encuentra en la parte positiva de la esfera y el incremento será 30, por lo que el anillo 6 se situará a una altura de $2 * 30 = 60$.

Una vez sabemos la altura a la que se sitúa nuestro anillo, llega el momento de emplear un poco de trigonometría para hallar el radio del anillo a la altura calculada. Usaremos como referencia la figura [8.11].

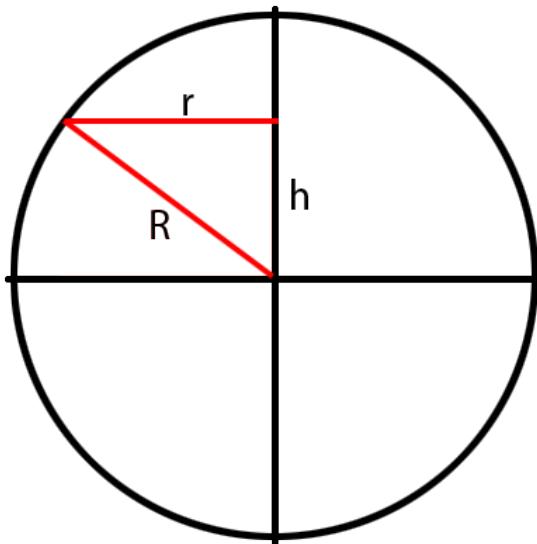


Figura 8.11: Hallar r a partir de R y h

Conocemos el valor del cateto que se corresponde con la altura h , a la que se sitúa nuestro anillo, y la hipotenusa que se corresponde con el radio R de nuestra esfera, y debemos hallar el radio r de nuestro anillo. Para ello, calculamos $\frac{R}{h}$ para hallar el valor del seno del cateto opuesto. Sin embargo, para poder calcular r necesitamos el valor del coseno. Por suerte, hay una fórmula matemática que nos resuelve este problema, pues el valor del coseno de un ángulo es equivalente a la raíz cuadrada de 1 menos el valor cuadrado del seno del ángulo, es decir $\cos(\alpha) = \sqrt{1 - \sin(\alpha)^2}$. De este modo, hemos hallado el valor del coseno entre r y R , por lo que basta con multiplicar R por el valor obtenido para hallar el radio r de nuestro anillo.

Tras ello, en la línea [28], invocamos una función que nos devuelve, dependiendo del cu-

drante en el que nos situemos, un ángulo distinto. Así pues, el primer cuadrante empezará en 0, el segundo en $\frac{\pi}{2}$, el tercero en π y el cuarto en $\frac{3\pi}{2}$. Así, una vez teniendo este ángulo como base, calculamos la posición en el anillo en que se encuentra nuestro punto, que se corresponderá a la posición relativa del punto en el anillo para el primer cuadrante más el valor del ángulo para el cuadrante en el que nos encontramos. Para hallar la posición relativa de nuestro punto en la circunferencia, lo que hacemos es tomar el valor de la x en el primer cuadrante. Si nos fijamos, el primer anillo de la circunferencia tendrá 1 vértice por cuadrante, el segundo, 2, el tercero, 3 y así sucesivamente. Así pues, sabiendo el anillo en el que nos encontramos y el desplazamiento en x , podemos hallar el ángulo que corresponde al punto. Por ejemplo, un punto situado en $(1, 1)$ está en el 5 anillo, que se corresponde con el tercero. Por tanto, sabemos que el punto $(1, 1)$ tendrá tres vértices en el primer cuadrante. Como estamos en el vértice 1, empezando por 0, de los tres vértices que tiene el tercer anillo por cuadrante, estamos pues a $\frac{2}{3}$ de progreso en el cuadrante, por lo que multiplicamos este valor por el valor de un cuadrante ($\frac{\pi}{2}$) y le sumamos el desplazamiento calculado en función del cuadrante en que nos hallamos para saber con qué ángulo de la circunferencia se corresponde nuestro vértice.

Tras todo este largo proceso, calculamos y almacenamos el valor del punto y creamos una función que nos permita interpolar entre la posición de los puntos en el plano y la posición de los vértices equivalentes en la esfera. En la imagen [8.12] podemos ver el resultado final. El resultado que se incluye en la demo es un poco distinto al que se muestra en esta figura, ya que se ha iterado y refinado para darle un toque más interesante.

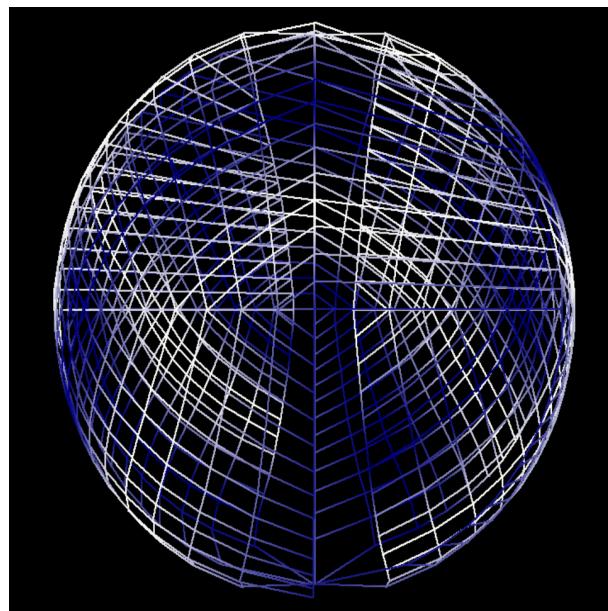


Figura 8.12: Esfera a partir del plano

Tras ello, pasamos a jugar un poco con el resultado obtenido, refinándolo, animando el tamaño y fase de la esfera y añadiendo música que suene acorde. Llegamos incluso a dejar

de dibujar la esfera línea a línea para dibujarla como un mar de puntos, lo cual genera un efecto visualmente atractivo.

Una vez tenemos esto hecho, podemos decir que hemos dedicado suficiente tiempo al efecto de geometría, y llega el momento de pasar a implementar otros efectos. En cuestión, de los efectos que nos quedan por implementar, hay varios que son similares entre ellos, pues todo lo que hacen es operar sobre una textura. Estos son el efecto de plasma [7.5], el efecto de RotoZoom [7.3] y el efecto de deformaciones [7.4]. ¿Y si los combinamos todos en uno? Podríamos crear una textura de plasma animada a la que aplicamos una deformación y rotamos y escalamos al estilo del más clásico RotoZoom.

Aplicar este efecto es muy sencillo, y sin embargo, el resultado visual es de lo más satisfactorio. Empezamos por escribir en una textura la frase "*did you ask for plasma?*", en español, "¿has pedido plasma?". Dibujamos esta textura inicialmente en blanco, sobre fondo negro, y la hacemos aparecer con un *zoom in* consistente en un escalado básico.

A continuación, pintamos nuestro plasma sobre la textura, pero de forma selectiva: coloreamos con el valor correspondiente sólo aquellos píxeles que son blancos, de forma que el texto se colorea con el efecto de plasma pero el fondo sigue siendo negro.

Una vez hecho esto, aplicamos el efecto de deformación de bandera que ya vimos en el apartado de deformaciones [7.4], de modo que nuestro texto empieza a ondear. A nivel personal, el resultado me parece de lo más satisfactorio, ya que ver el plasma animado, de naturaleza ya por sí ondulante sobre un texto animado ondulado produce un efecto visual de lo más atractivo. Para acabar de coronar esta transformación, aplicamos el efecto de RotoZoom, de modo que toda la escena que hemos construido, con el plasma animado sobre el texto ondulado, culmina con su desaparición rotando cada vez más rápido y haciéndose cada vez más pequeño, hasta que se produce un fundido de salida a blanco.

Podemos ver distintas capturas de lo descrito en la figura [8.13]. No obstante, si nos damos cuenta, se aplica una gran cantidad de operaciones por píxel en esta parte de la demo. Si bien es cierto que manejamos una resolución HD (1280x720), aun así tantas transformaciones pueden resultar excesivas, y afectar de forma considerable a la tasa de fotogramas de la demo. Es por ello que, nuevamente, tenemos que tomar una decisión crítica: sacrificar resolución por rendimiento. No vamos a bajar de calidad HD, sin embargo, es sólo que la textura que vamos a usar y sobre la que aplicaremos todos los cálculos y transformaciones tendrá una resolución menor, concretamente, de la mitad de la altura y anchura de la pantalla, quedándonos en $640 \times 360 = 230400$ píxeles, cuatro veces menos que la cantidad que manejamos en pantalla. De este modo, iteraremos sobre una textura relativamente pequeña que a continuación escalaremos a la resolución de nuestra demo y que a su vez el sistema escalará a la resolución de la pantalla.

Esta vez, sin embargo, la pérdida de resolución resulta mucho más notoria. Por suerte, no obstante, debido a que esta parte de la demo es especialmente dinámica, la bajada en resolución resulta menos aparente, con lo cual conseguimos mantener una tasa de fotogramas

adecuada sin impactar excesivamente el resultado de nuestra demo.



Figura 8.13: Distintas capturas del efecto de plasma + deformaciones + RotoZoom

A lo largo de la parte final del efecto de geometría y durante la primera parte del efecto de plasma, utilizamos varios sonidos que no hemos visto hasta ahora. Se trata del sonido de láser y de los sonidos de percusión.

Código 8.10: Otros sonidos empleados en la demo

```

1 float Sounds::CreateLaserSound(float frequency, long int currentCount)
2 {
3     return (GetSawtoothWaveValue(frequency, currentCount) * 0.5f +
4            GetSawtoothWaveValue(frequency, currentCount) * 0.5f * GetHighPassNoiseValue(0.05)) *
5            0.5 +
6            GetHighPassNoiseValue(0.05) * 0.5;
7 }
8
9
10 Envelope laserEnv = {0.0f, 0.f, 0.0f, 0.2f, 1.f, 1.0f};
11
12 Envelope drumEnv = {0.f, 0.f, 0.f, 0.1f, 1.f, 1.f};
13 Envelope snareEnv = {0.f, 0.f, 0.f, 0.3f, 0.5f, 0.5f};

```

Como podemos ver en el código [8.10], añadimos un nuevo sonido al que hemos bautizado como sonido de láser. Es un efecto muy sencillo en el que generamos el sonido a partir de una onda de diente de sierra más una segunda onda modulada en función de un filtro pasa alta. Esto crea un efecto de distorsión en el sonido, que sumado al sonido metálico de la onda de diente de sierra, resulta en una sensación con un cierto aire de película de ciencia ficción. Envolvemos este sonido en una envolvente muy breve que entra en fase de relajación desde el inicio.

Para los instrumentos percutivos, todo lo que hacemos es aplicar una envolvente distinta a un ruido blanco. Siendo la primera envolvente similar a la de un tambor (*drumEnv*), y siendo la segunda, con un mayor tiempo de relajación y un volumen de base menor, más similar al ruido de una caja (*snareEnv*).

El sonido que suena, sin embargo, mientras que nuestra textura de plasma empieza a rotar y disminuir en tamaño hasta hacerse irreconocible es bastante más interesante. Quería generar una sensación de sonido que resultase agobiante y causase la ilusión de un sonido constantemente ascendente. Para ello, intenté generar la ilusión o escala de Shepard. Podemos ver la implementación en el código [8.11].

Código 8.11: Generación de la escala de Shepard

```

1 const float mask[12] = {0.01f, 0.025f, 0.05f, 0.075f, 0.14f, 0.2f, 0.2f, 0.14f, 0.075f, 0.05f, 0.025f, 0.01f};
2
3 static float baseFrequency = 261.63 * 0.125;
4 const static float incrementer = 1.059463f;
5
6 //Generate Sephard illusion of ascending sound
7 notes.push_back({Sounds::GetSineWaveValue, laserEnv, baseFrequency * 1, mask[1] * volume, 0.3f});
8 notes.push_back({Sounds::GetSineWaveValue, laserEnv, baseFrequency * 2, mask[3] * volume, 0.7f});
9 notes.push_back({Sounds::GetSineWaveValue, laserEnv, baseFrequency * 4, mask[5] * volume, 0.3f});
10 notes.push_back({Sounds::GetSineWaveValue, laserEnv, baseFrequency * 6, mask[7] * volume, 0.7f});
11 notes.push_back({Sounds::GetSineWaveValue, laserEnv, baseFrequency * 8, mask[9] * volume, 0.3f});
12 notes.push_back({Sounds::GetSineWaveValue, laserEnv, baseFrequency * 10, mask[11] * volume, 0.7f});
13
14 baseFrequency *= incrementer;
15 if (baseFrequency >= 523.f)
16 {
17     baseFrequency = 261.63f;
18 }

```

La ilusión de Shepard es una ilusión auditiva que produce la sensación de ascenso infinito cuando en realidad reproducimos un sonido en bucle. Este efecto se consigue, dada una frecuencia, generando varios armónicos de la misma (6 en nuestro código) y aplicando un volumen distinto a cada uno (como vemos, el volumen de cada armónico queda determinado por un conjunto de valor constante que definimos). De este modo, cada vez que reproducimos una nueva nota, aumentamos su frecuencia en un semitono ($2^{\frac{1}{12}} = 1.059463$), hasta que llegamos a la octava (el doble de la frecuencia) y en ese momento volvemos a la frecuencia base.

¿Por qué se produce esta sensación de circularidad, sin embargo? Es realmente difícil de explicar con palabras o imágenes, por lo que recomiendo el siguiente [vídeo](#). No obstante, trataré de explicarlo: esto se debe a que, como aplicamos una máscara que atenúa el volumen de los armónicos más graves y más agudos, aquellos que destacan son los armónicos centrales. A su vez, la frecuencia de estos armónicos aumenta por iteración. Como cada vez que completamos una octava, empezamos el bucle de nuevo, el sonido de la frecuencia fundamental, que inicialmente era 261.63, en la interacción anterior a reiniciar el bucle tiene un valor de 493.89, de modo que cuando reiniciamos el bucle, el valor de la frecuencia fundamental vuelve a ser 261.63 pero el valor del siguiente armónico ($baseFrequency * 2$) pasa a ser 523.26, siendo este el equivalente de aumentar 493.89 en un semitono. Por tanto, se produce una sensación de circularidad, ya que cuando se repite el bucle, el último valor de la iteración mantiene continuidad con respecto al primer valor de la próxima iteración del siguiente armónico. Además, el hecho de atenuar las frecuencias más bajas y altas contribuye aún más a reforzar esta sensación de continuidad y disimular la repetición del bucle (pues las frecuencias que rompen su continuidad al repetirse el bucle, que están a los extremos, suenan con mucha menor intensidad).

Gracias a este efecto, logramos causar la sensación que buscábamos. Además, para reforzar esta misma impresión aun más, recortamos la distancia temporal que pasa entre una nota y la siguiente, de modo que los sonidos se reproducen cada vez más juntos, reforzando la sensación de agobio y crecimiento.

Este sonido culmina con una fundido de salida al silencio que se coordina con el fundido

de salida visual del efecto, quedando por unos instantes una pantalla vacía, en blanco, y sin sonido. Llega el momento de pasar al próximo efecto.

En esta nueva escena pondremos en práctica el efecto de planos infinitos de un modo que no habíamos empleado hasta ahora. Generaremos una textura de gran tamaño sobre la que escribiremos mensajes de texto, e iremos recorriendo la escena como si la estuviésemos viendo en primera persona, de modo que dé tiempo a leer cada uno de los mensajes.

En el mundo de la *demoscene*, los mensajes de texto, retos o incluso mofas entre *demosceners*, a veces ocultos y a veces en primer plano, son algo de lo más común a encontrar dentro de una demo. Es por tanto una oportunidad que no vamos a dejar pasar. Esto es lo que escribiremos en nuestra textura ”*A CPU is so slow... A CPU is not for graphics... Wait... Am I dreaming? Is this heaven? Can't be true...*”, es español, ”*Una CPU es tan lenta... Una CPU no es para gráficos... Espera... ¿Estoy soñando? ¿Es esto el cielo? No puede ser cierto...*”. Dejamos así nuestra pequeña reivindicación personal.

Reforzamos este efecto reflejando la mitad inferior de la pantalla en la mitad superior, de modo que todo sea vea espejado, y añadimos una línea negra en el centro a modo de horizonte, tanto por cuestión estética como para facilitar la orientación en la escena.

Una vez hecho esto, definimos un mapa de posiciones e interpolamos entre ellas, de modo que podamos recorrer de forma automática nuestra escena. El resultado se puede apreciar en la figura [??].

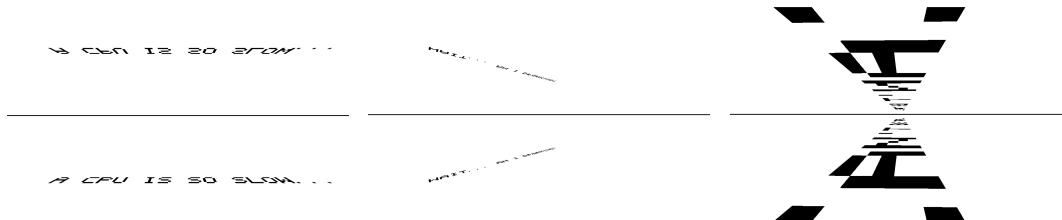


Figura 8.14: Distintas capturas del efecto de planos infinitos

Para reforzar la sensación etérea que causa este efecto, buscamos generar un sonido que suene etéreo. Para ello, buscamos sonidos y referencias que nos gusten y las analizamos, tanto desde el punto de vista del tipo de envolvente que tienen como analizamos también su espectro de frecuencias, para ver en cuántas frecuencias suenan y de qué forma lo hacen. El resultado tras el análisis es el siguiente código [8.12].

Código 8.12: Generación de un sonido etéreo

```

1 float Sounds::CreateSynthSound(float frequency, long int currentCount)
2 {
3     const float LFO = GetSineWaveValue(2, currentCount);
4     return GetTriangleWaveValue(frequency * 1.0, currentCount) * (0.45 + 0.1 * LFO) * 0.6 +
5         GetTriangleWaveValue(frequency * 1.3333, currentCount) * (0.45 + 0.1 * LFO) * 0.1 +

```

```

6     GetTriangleWaveValue(frequency * 1.6666, currentCount) * (0.45 + 0.1 * LFO) * 0.1 +
7     GetTriangleWaveValue(frequency * 2.0, currentCount) * (0.45 + 0.1 * LFO * 2.f) * 0.1 +
8     GetTriangleWaveValue(frequency * 2.6666, currentCount) * (0.45 + 0.1 * LFO * -1.f) * 0.1 +
9     GetTriangleWaveValue(frequency * 4.0, currentCount) * (0.45 + 0.1 * LFO) * 0.1 +
10    GetTriangleWaveValue(frequency * 5.3333, currentCount) * (0.45 + 0.1 * LFO * 2.f) * 0.1 +
11    GetTriangleWaveValue(frequency * 6.6666, currentCount) * (0.45 + 0.1 * LFO * -1.f) * 0.1;
12}
13
14 Envelope synthEnv = {1.f, 1.f, 1.f, 2.f, 1.f, 0.7f};

```

Para que nuestro sonido suene etéreo, entre otras cosas necesitamos que tanto su fase de ataque como de relajación sean lentas, de modo que entre y se desvanezca de forma suave y poco agresiva. Del mismo modo, un sonido etéreo debe sonar armónico y organizado pero debe causar una cierta sensación de fragilidad o inestabilidad. Esta sensación la logramos con el uso de un oscilador en baja frecuencia (*low frequency oscillator*, LFO⁷) que aplicamos de forma individual sobre el volumen de cada uno de los armónicos que conforman nuestra frecuencia, aplicando además distintos valores de intensidad. De este modo se produce una sensación de irregularidad o inestabilidad dentro de un sonido que suena de forma armónica, pero al mismo tiempo, titilante, logrando ese difícil equilibrio entre armonía y fragilidad.

Con esto hecho, pasamos a implementar la que será la última parte de nuestra demo, el túnel de puntos [7.2]. Tras una pequeña transición en la que rotamos la línea que pintábamos sobre el horizonte de modo que se sitúe en vertical, aplicando la misma transformación que ya hemos visto tantas veces [7.15], pasamos a mostrar nuestro túnel de puntos.

Para ello, empezamos con una transición inicial en la que vamos pintando la pantalla de negro desde el centro hacia ambos extremos de la misma. Al mismo tiempo, empezamos a dibujar el túnel, aunque sólo pintaremos aquellos puntos del mismo que se hallen dentro de la zona negra. Tras ello, nos limitamos a seguir actualizando nuestro túnel durante unos instantes, mientras incorporamos música muy sencilla a la escena. Simplemente reproducimos notas con el sonido del láser mientras que suenan instrumentos percutivos. Sincronizamos, no obstante, la percusión con el túnel, de modo que cada vez que suena un golpe, el túnel acelera brevemente.

Tras ello, con un fundido de entrada pasamos a mostrar los créditos de la demo, dibujándolos encima del túnel. Para dar un pequeño efecto de dinamismo, no obstante, se modifica levemente la posición del texto en función de la trayectoria del túnel, creando así la sensación de que el texto se mueve al compás que el túnel avanza. Podemos ver los efectos descritos anteriormente en la figura[8.15].

Ya por último, con un fundido a negro y un desvanecimiento progresivo de la música, la demo finaliza.

⁷https://en.wikipedia.org/wiki/Low-frequency_oscillation

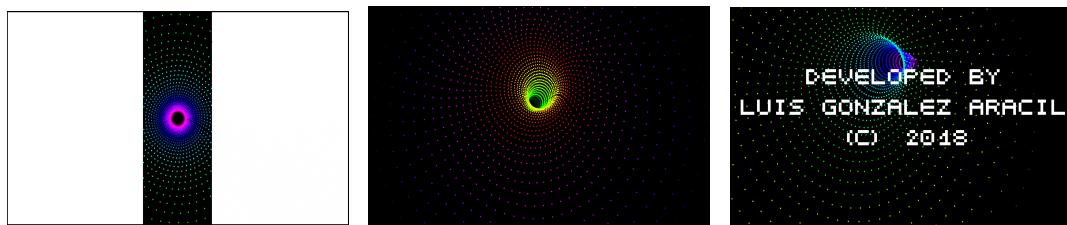


Figura 8.15: Distintas capturas del efecto de túnel de puntos

8.5 Conclusiones de la demo

Y hasta aquí llega el desarrollo de la demo final. Lo cierto es que ha sido un desarrollo de lo más interesante. Durante la realización de los efectos gráficos por separado no era consciente de hasta qué punto se podían estirar o combinar estos efectos para producir resultados que anteriormente hubieran parecido imposibles.

Lo cierto es que con el set de técnicas aprendido, de pronto aparece una gran cantidad de posibilidades y una infinidad de combinaciones.

Es así como esta demo supone la culminación de todo lo aprendido anteriormente, y si cada uno de los efectos gráficos estudiados era un pequeño ladrillo, ahora se nos abre la posibilidad de construir una gran estructura, donde la creatividad y el ingenio son primordiales, ya que las barreras de aquello que podemos lograr se encuentran más en los límites de nuestra imaginación que en los límites del ordenador.

Esta demo está ejecutada únicamente en CPU, en un sólo hilo (aunque la librería de sonido genera un hilo separado para la reproducción de audio) y cuenta con una combinación de efectos gráficos en dos dimensiones y algunos que simulan la tridimensionalidad. Todo esto ejecutado en tiempo real.

Además, en la demo hemos abierto la posibilidad de generar sonido, lo cual ha creado un inmenso campo de posibilidades que sin embargo ha habido que limitar para no salir del ámbito de este trabajo. Aun así, hemos conseguido generar una gran cantidad de sonidos en tiempo real, coordinados con la acción en pantalla y que intentan reforzar aquello que sucede en la misma. Además, la demo se reproduce en estéreo, y algunos sonidos tienen un posicionamiento definido, por lo que se recomienda escucharla con auriculares.

Por tanto, esta demo representa no solo una aplicación de todo lo aprendido, si no también la apertura ante un amplio abanico de posibilidades, tanto desde el punto de visto técnico como creativo, y que me gustaría poder seguir ampliando y refinando a lo largo del tiempo, una vez finalizado este trabajo.

9 Conclusiones

Este trabajo empezó por la investigación de la subcultura informática de la *demoscene*, una cultura estrechamente relacionada con los gráficos por computador y cuyo mayor objetivo era la competición y superarse a sí mismo. Si bien en la actualidad se está perdiendo, en parte debido a aquellas opiniones que abogan que el bajo nivel debería ser olvidado hoy en día y en parte debido al hecho de que la *demoscene* siempre ha sido una cultura muy cerrada que no ha logrado adaptarse a la masificación de la informática, lo cierto es que hay mucho que aprender de la misma. Al fin y al cabo, la esencia de la *demoscene* es la pasión por el conocimiento, por superarse, por explorar los límites de un sistema y por causar un impacto en la comunidad.

Otro de los grandes objetivos de este trabajo era intentar implementar cuanto más efectos posibles desde cero, partiendo de un razonamiento inicial y formalizándolo hasta lograr un código de producción propia. Esto a veces podría ser considerado como reinventar la rueda. Al fin y al cabo, lo que muchas personas se preguntan es "¿para qué resolver un problema que ya ha resuelto otra persona?". Para mí, después de este trabajo, la respuesta es clara: para aprender y para descubrir nuevos puntos de vista. **Para avanzar.**

Y es que creo que este es un punto muy importante, **avanzar no significa olvidar el pasado**. Ante la tendencia creciente hacia las nuevas tecnologías y la abstracción, parece que hemos olvidado que todas esas tecnologías y capas de abstracción están construidas sobre los mismos principios que ya asentó Alan Turing¹ hace más de 80 años.

Por ello, pienso que es importante desarrollar un pensamiento crítico. No siempre va a resultar útil "reinventar la rueda" y habrá ocasiones en que ahondar en los detalles de implementación de ciertas librerías o programas informáticos resulte impráctico o inviable. Pero en muchos otros casos, ir a los detalles de implementación permite una mejor comprensión del funcionamiento del software y del hardware con el que se interactúa. Y conocer en profundidad el medio con el que se interactúa es vital en el transcurso de la vida profesional del programador. Del mismo modo que un médico debe conocer en profundidad el cuerpo humano, para tener capacidad de resolución ante distintos tipos de pacientes y enfermedades, lo mismo se aplica al programador en su ámbito. Conocer cómo funciona el sistema con el que se interactúa permite programar de un modo que sea más amigable de cara al sistema, permite identificar con mayor facilidad posibles *bugs* y errores, permite desarrollar con mayor facilidad mejoras o refinamientos en el software y dota de una comprensión de alto nivel basada en el conocimiento del bajo nivel.

Ante todas las voces que abogan hoy en día que el programador debería preocuparse únicamente por resolver problemas, con independencia de la plataforma, vuelvo al paralelismo

¹https://en.wikipedia.org/wiki/Alan_Turing

con el médico hecho anteriormente. ¿Es mejor un médico que resuelve problemas en función del *modelo general* de paciente que aquel que se ciñe a los específicos del paciente que está tratando?

Es por ello que este trabajo ha habilitado una mejor comprensión del bajo nivel y de los gráficos por computador que se reflejan a su vez en una mayor comprensión de los gráficos por computador y del funcionamiento de un ordenador desde el alto nivel. Muchas de las demos que se han realizado en este trabajo se han hecho sin tener en cuenta los ejemplos de implementación que se podían encontrar en la red, y si bien en algunos casos las implementaciones son similares, en otros las implementaciones varían, mostrando modelos de pensamiento distintos que dan un resultado muy similar. Se resuelve por tanto el mismo problema desde un punto de vista muy distinto. Un ejemplo claro de esto es la demo del túnel de puntos [7.2], que en el mundo de la *demoscene* se solía implementar proyectando un túnel de puntos en el espacio tridimensional a la pantalla, mientras que en este trabajo se ha optado por imitar el efecto de la proyección, construyendo un túnel en todo momento bidimensional, a base de círculos, pero cuyo efecto final es muy similar.

Además, de cara a muchas demos, había más de un camino de partida inicial, y tomar una decisión u otra se ha basado a partes iguales en buscar el mejor rendimiento y el mejor resultado visual.

Por tanto, este trabajo refleja la importancia del pensamiento crítico a la hora de resolver un problema de programación y cómo se puede tener en cuenta el conocimiento del software y el hardware para optimizar o mejorar los resultados obtenidos.

No siempre será necesario tener un conocimiento en profundidad del hardware, no siempre será necesario tener un conocimiento en profundidad del software o de cada capa de abstracción. Esto sería impráctico en la mayor parte de los casos. Pero es necesario estar preparados y tener la capacidad de poder entender y profundizar en el comportamiento de un ordenador, pues esto amplía nuestro conocimiento sobre el problema a resolver y nos permite llegar a puntos de vista y conclusiones de otro modo imposibles de alcanzar.

Al fin y al cabo, **es muy difícil construir una buena casa sin tener en cuenta los cimientos.**
