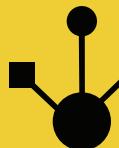




Escuela
Politécnica
Superior

Demo técnica para PC



Grado en Ingeniería Multimedia

Trabajo Fin de Grado

Autor:

Luis González Aracil

Tutor/es:

Francisco José Gallego Durán

Mayo 2019



Universitat d'Alacant
Universidad de Alicante

Demo técnica para PC

La demoscene

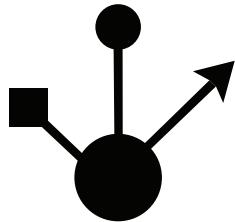
Autor

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	3
2.1	La demoscene	3
2.1.1	Qué es la demoscene	3
2.1.2	Orígenes de la demoscene	4
2.1.3	Composición y cultura de la demoscene	4
2.1.4	La demoscene en la actualidad	5
2.2	Eventos de demoscening	6
2.3	Grupos de demoscening	7
2.3.1	Farbrausch	7
2.3.2	Future Crew	9
2.3.3	PoPsY TeAm	10
2.3.4	Equinox	11
2.3.5	Fairlight	11
2.3.6	RGBA	12
2.3.7	Batman Group	13
2.4	Portales de demoscening	14
2.5	Demos destacables	15
2.6	Efectos gráficos más comunes	16
2.7	Influencia de la demoscene en la industria	19
3	Objetivos	21
4	Metodología	23
4.1	Software	23
4.2	Tests de rendimiento	24
4.3	Entorno: motor gráfico	24
4.4	Las demos	25
4.4.1	Búsqueda de información	25
4.4.2	Planteamiento formal	25
4.4.3	Implementación	25
4.4.4	Refinamiento	25
5	Tests de rendimiento	27
5.1	Implementación	27
5.2	Resultados	27
5.3	Posibles mejoras	27
5.4	Conclusión	27

6 El motor gráfico	29
6.1 Investigación inicial	29
6.2 Características	31
6.2.1 La textura de dibujado y el píxel	33
6.2.2 Detectar input	35
6.2.3 Dibujar texto	36
6.2.4 Dibujar puntos	39
6.2.5 Dibujar rectángulos	40
6.2.6 Dibujar líneas	41
7 Demos clásicas	43
7.1 Fuego	43
7.1.1 Investigación inicial	43
7.1.2 Planteamiento formal	43
7.1.3 Implementación	44
7.1.4 Refinamiento	47
7.1.5 Resultado	48
7.2 Túnel de puntos	49
7.2.1 Investigación inicial	49
7.2.2 Planteamiento formal	50
7.2.3 Implementación	50
7.2.4 Refinamiento	57
7.2.5 Resultado	60
7.3 RotoZoom	60
7.3.1 Investigación inicial	60
7.3.2 Planteamiento formal	61
7.3.3 Implementación	61
7.3.4 Refinamiento	61
7.3.5 Resultado	62
7.4 Deformaciones de imagen	62
7.4.1 Investigación inicial	62
7.4.2 Planteamiento formal	62
7.4.3 Implementación	62
7.4.4 Refinamiento	62
7.4.5 Resultado	63
7.5 Plasma	63
7.5.1 Investigación inicial	63
7.5.2 Planteamiento formal	63
7.5.3 Implementación	63
7.5.4 Refinamiento	63
7.5.5 Resultado	64
7.6 Planos infinitos	64
7.6.1 Investigación inicial	64
7.6.2 Planteamiento formal	64
7.6.3 Implementación	64

7.6.4	Refinamiento	64
7.6.5	Resultado	65
7.7	Geometría	65
7.7.1	Investigación inicial	65
7.7.2	Planteamiento formal	65
7.7.3	Implementación	65
7.7.4	Refinamiento	65
7.7.5	Resultado	66
8	Demo final	67
8.0.1	Investigación inicial	67
8.0.2	Planteamiento formal	67
8.0.3	Implementación	67
8.0.4	Refinamiento	67
8.0.5	Resultado	67
9	Conclusiones	69
Bibliografía		71

Índice de figuras

2.1	Assembly 2004 - Fuente: Wikipedia	8
2.2	Farbrausch 41: Debris - Fuente: YouTube	8
2.3	Videojuego de 96kB: .kkrieger - Fuente: YouTube	9
2.4	Farbrausch 63: Magellan - Fuente: YouTube	9
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>	10
2.6	VIP2 - Fuente: YouTube	11
2.8	Dead Ringer (por FairLight) - Fuente: YouTube - Demo 64k ganadora de Assembly 2006	12
2.9	Elevated - Fuente: YouTube - Intro 4K ganadora en Breakpoint 2009	13
2.10	Batman Forever - Fuente: YouTube	13
2.11	Heaven Seven (por Exceed) - Fuente: YouTube	16
2.12	State of the Art (por Spaceballs) - Fuente: YouTube	17
6.1	Diagrama simplificado de la estructura del motor gráfico	31
6.2	Estructura de un píxel en el motor gráfico	34
6.3	Funciones y estructuras del sistema de input	36
6.4	Pintado de líneas en pantalla (adaptación de una línea a una cuadrícula) - Fuente: Wikipedia	41
7.1	Matriz de convolución para generar efecto de fuego	44
7.3	Degrado en 32 celdas dadas 5 marcas de color	45
7.4	Fuego estático, usando el algoritmo en [7.3]	46
7.5	Fuego dinámico, con intensidad por píxel aleatorizada	48
7.8	Estructura básica de un círculo	51
7.9	Túnel de puntos básico	54
7.10	Órbitas de los planetas vistas desde la Tierra (por Giovanni Cassini) - Fuente: Wikipedia	55
7.11	Estructura de cada órbita que determina el camino	56
7.12	Tunel básico siguiendo un camino	57
7.13	Túnel final	61

Índice de tablas

Índice de Códigos

6.1	Método que renderiza un sólo carácter	37
6.2	Método que renderiza un sólo carácter	38
6.3	Método que renderiza una cadena de caracteres	38
6.4	Método para dibujar puntos	39
6.5	Métodos de repintado en pantalla	40
6.6	Versión simplificada y reducida del código para dibujar líneas en pantalla	41
7.1	Método para crear gradientes de color	45
7.2	Creación e inicialización del mapa de valores	45
7.3	Algoritmo básico de efecto de fuego	46
7.4	Algoritmo final de efecto de fuego	48
7.5	Algoritmo básico de dibujado de circunferencias	51
7.6	Inserción y eliminación de círculos	52
7.7	Actualización del túnel	52
7.8	Creación de un camino de turbulencia	55
7.9	Actualización de un camino de turbulencia	56
7.10	Generación de tablas precalculadas y uso en código	60

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.

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>

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/>

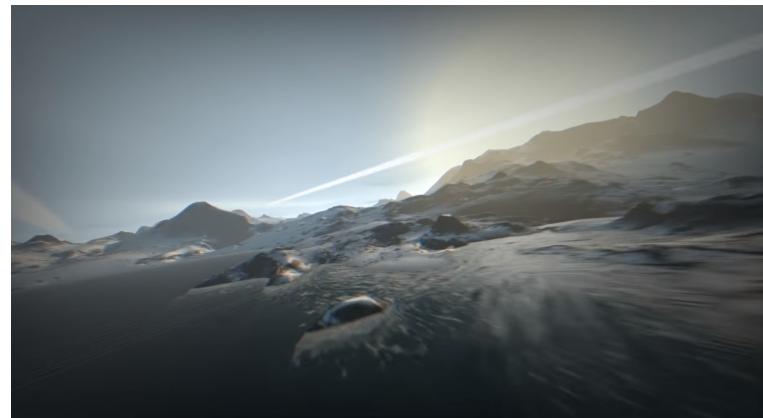


Figura 2.9: Elevated - Fuente: YouTube - Intro 4K ganadora en Breakpoint 2009

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.

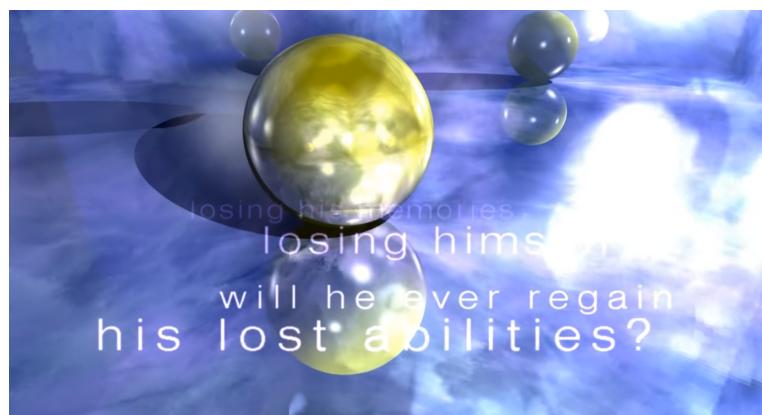


Figura 2.11: Heaven Seven (por Exceed) - Fuente: YouTube

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, tanto a nivel de coste temporal como espacial (y cantidad de instrucciones).

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 de la reserva y liberación de memoria, coste del acceso a memoria y coste de los bucles y las operaciones condicionales.

Tras analizar los resultados, se elaborarán una serie de directrices para escribir código que sea generalmente más rápido y/o más fácilmente optimizable por el compilador. 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.

⁹<http://valgrind.org>

¹⁰<https://godbolt.org>

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)

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 Implementación

5.2 Resultados

5.3 Posibles mejoras

5.4 Conclusión

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

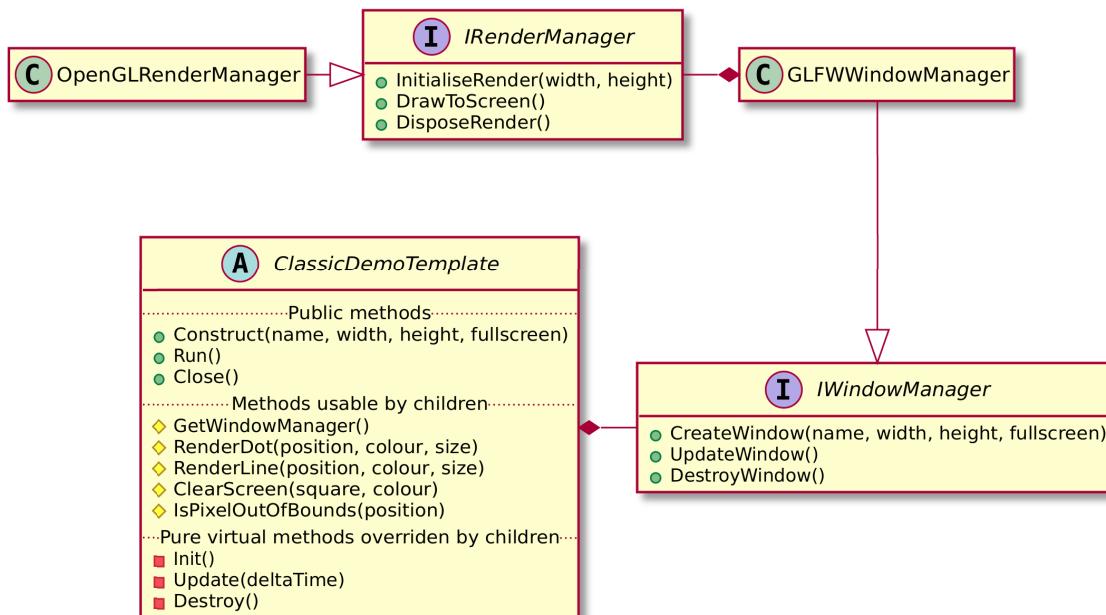


Figura 6.1: Diagrama simplificado de la estructura del motor gráfico

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

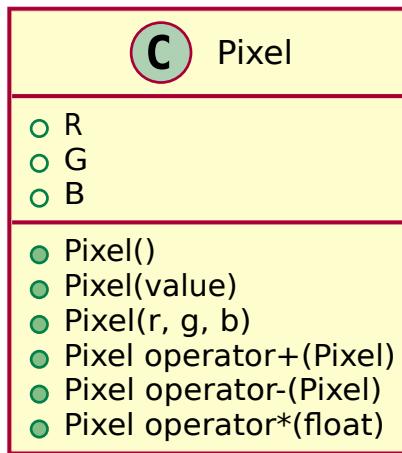


Figura 6.2: Estructura de un píxel en el motor gráfico

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.

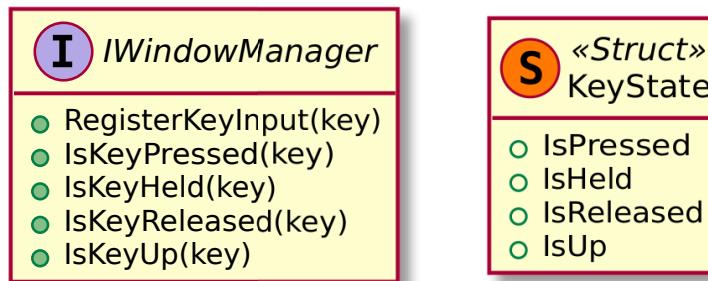


Figura 6.3: Funciones y estructuras del sistema de input

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     for (auto c : txt)
9     {
10        RenderCharacter(c, posX, posY, scale, colour);
11        posX += 6 * scale;
12    }
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.

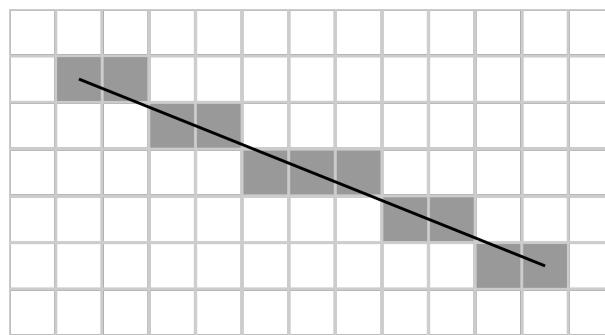


Figura 6.4: Pintado de líneas en pantalla (adaptación de una línea a una cuadrícula) - Fuente: Wikipedia

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 Demos clásicas

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>

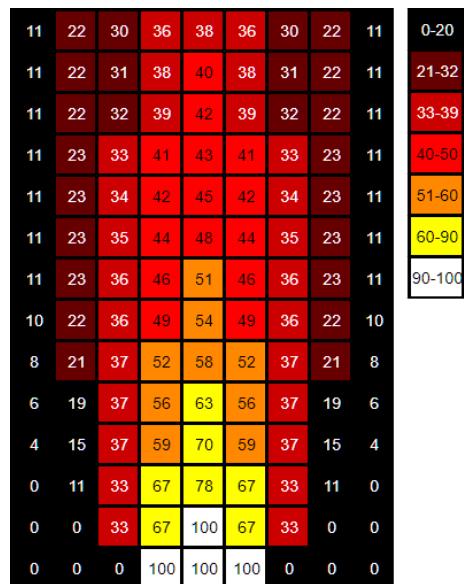
$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix} \quad (7.1)$$

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.

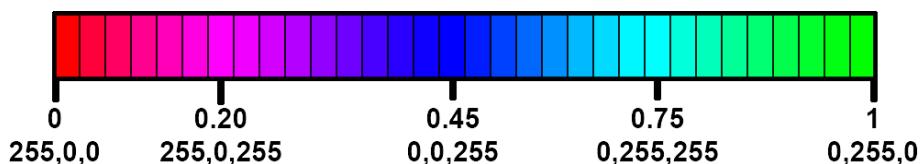


Figura 7.3: Degradado en 32 celdas dadas 5 marcas de color

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.

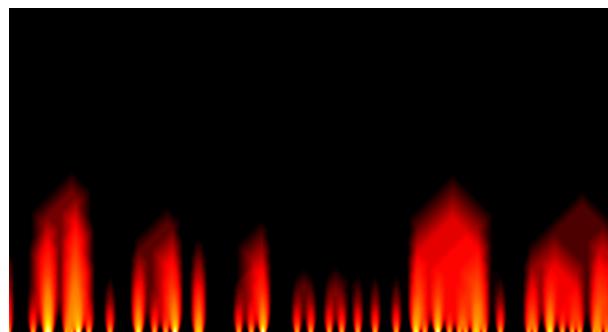


Figura 7.4: Fuego estático, usando el algoritmo en [7.3]

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 }
```

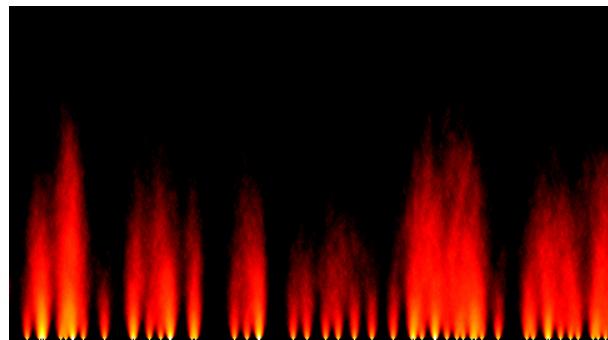
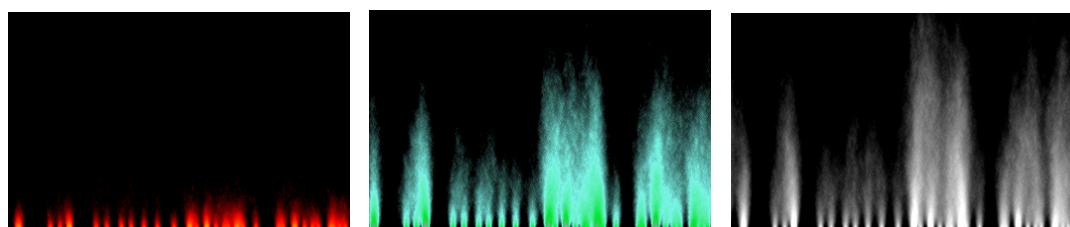


Figura 7.5: Fuego dinámico, con intensidad por píxel aleatorizada

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

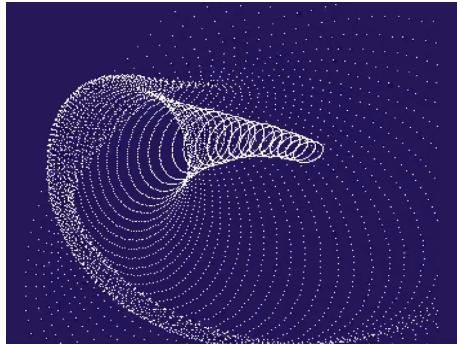
7.2 Túnel de puntos

7.2.1 Investigación inicial

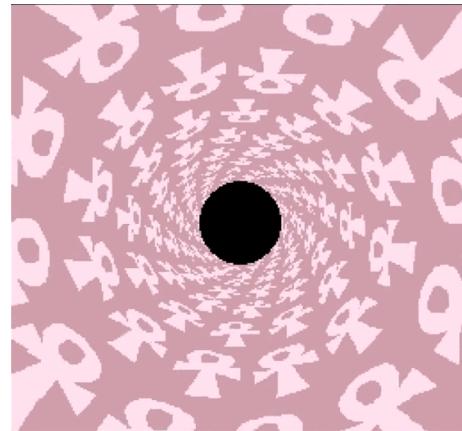
Un efecto también muy común en el mundo de la *demoscene* es el efecto de túnel o vórtice, entre otras causas por su relativa sencillez sumada a su espectacularidad (por la sensación de profundidad y de dinamismo, evocando a escenas futuristas o situadas en el espacio).

Es por ello que el efecto de túnel parecía un candidato perfecto para ser la segunda demo 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 Virtual 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

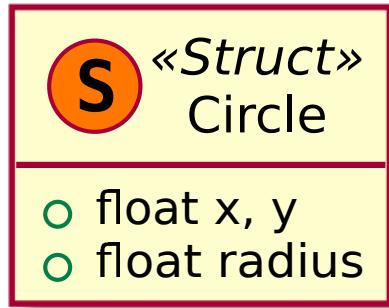


Figura 7.8: Estructura básica de un círculo

$(2 \times \pi) \div 4 = \frac{\pi}{2}$, 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
11         RenderDot(x, y, Pixel(255), 1);
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?

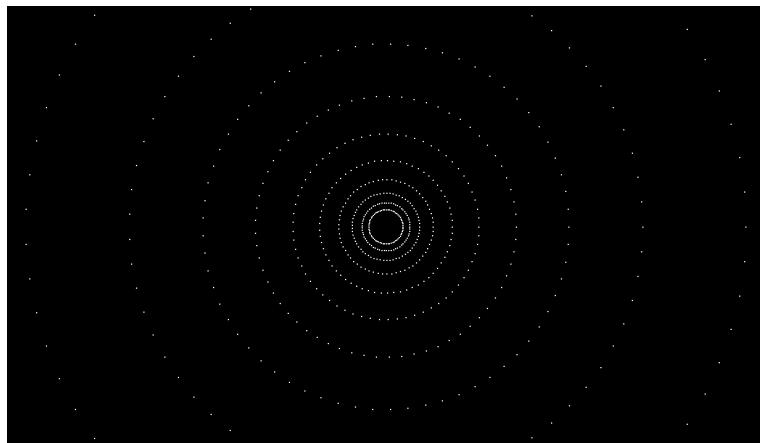


Figura 7.9: Túnel de puntos básico

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.

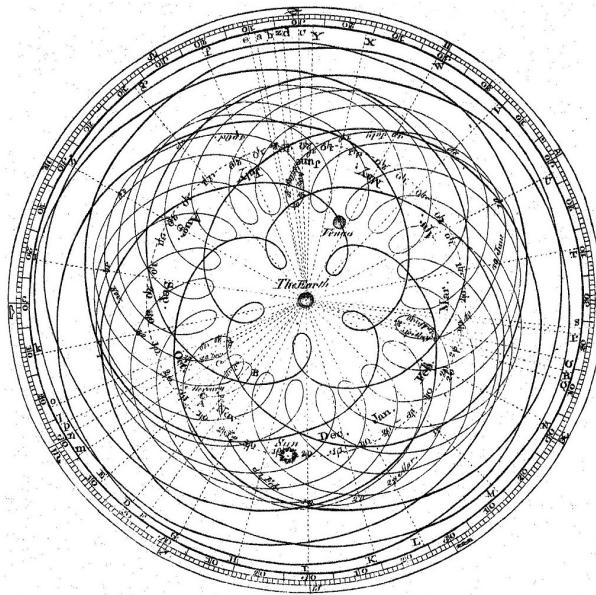


Figura 7.10: Órbitas de los planetas vistas desde la Tierra (por Giovanni Cassini) - Fuente: Wikipedia

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

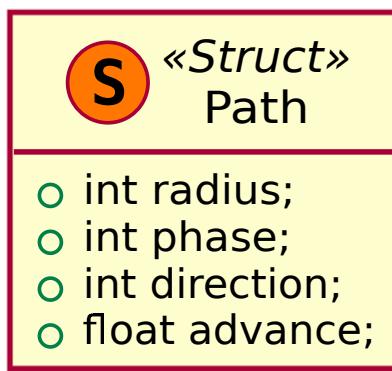


Figura 7.11: Estructura de cada órbita que determina el camino

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].

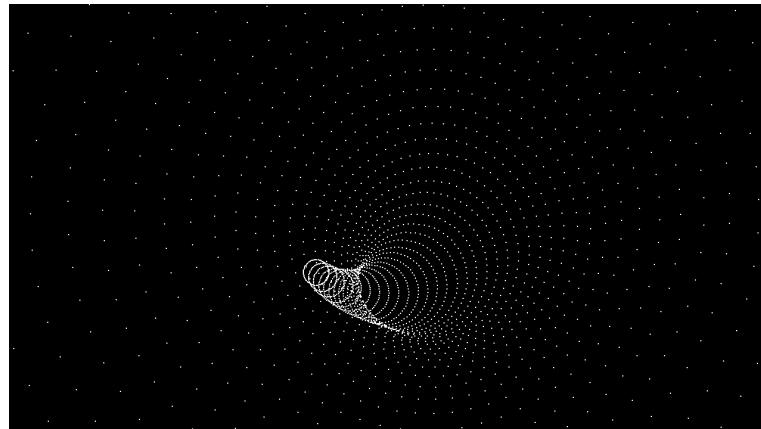


Figura 7.12: Tunel básico siguiendo un camino

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 esta demo 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 futuras demos, 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 esta demo 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).

La demo 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

```

rgerg sgsdgsfdh
sg
drga
rg
ergerearg
```

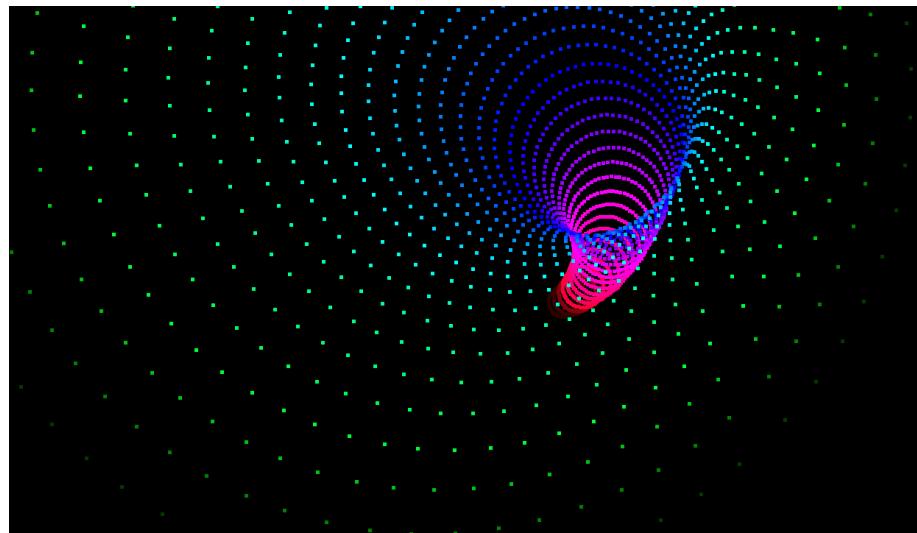


Figura 7.13: Túnel final

rgerg

7.3.2 Planteamiento formal

rgerg sgsdgsfdh
sg
drga
rg
ergregearg
rgerg

7.3.3 Implementación

rgerg sgsdgsfdh
sg
drga
rg
ergregearg
rgerg

7.3.4 Refinamiento

rgerg sgsdgsfdh
sg
drga
rg
ergregearg
rgerg

7.3.5 Resultado

rgerg sgsdgsfdh
sg
driga
rg
ergeregearg
rgerg

7.4 Deformaciones de imagen

7.4.1 Investigación inicial

rgerg sgsdgsfdh
sg
driga
rg
ergeregearg
rgerg

7.4.2 Planteamiento formal

rgerg sgsdgsfdh
sg
driga
rg
ergeregearg
rgerg

7.4.3 Implementación

rgerg sgsdgsfdh
sg
driga
rg
ergeregearg
rgerg

7.4.4 Refinamiento

rgerg sgsdgsfdh
sg
driga
rg
ergeregearg
rgerg

7.4.5 Resultado

rgerg sgsdgsfdh
sg
driga
rg
ergregearg
rgerg

7.5 Plasma

7.5.1 Investigación inicial

rgerg sgsdgsfdh
sg
driga
rg
ergregearg
rgerg

7.5.2 Planteamiento formal

rgerg sgsdgsfdh
sg
driga
rg
ergregearg
rgerg

7.5.3 Implementación

rgerg sgsdgsfdh
sg
driga
rg
ergregearg
rgerg

7.5.4 Refinamiento

rgerg sgsdgsfdh
sg
driga
rg
ergregearg
rgerg

7.5.5 Resultado

rgerg sgsdgsfdh
sg
driga
rg
ergeregearg
rgerg

7.6 Planos infinitos

7.6.1 Investigación inicial

rgerg sgsdgsfdh
sg
driga
rg
ergeregearg
rgerg

7.6.2 Planteamiento formal

rgerg sgsdgsfdh
sg
driga
rg
ergeregearg
rgerg

7.6.3 Implementación

rgerg sgsdgsfdh
sg
driga
rg
ergeregearg
rgerg

7.6.4 Refinamiento

rgerg sgsdgsfdh
sg
driga
rg
ergeregearg
rgerg

7.6.5 Resultado

rgerg sgsdgsfdh
sg
driga
rg
ergregearg
rgerg

7.7 Geometría

7.7.1 Investigación inicial

rgerg sgsdgsfdh
sg
driga
rg
ergregearg
rgerg

7.7.2 Planteamiento formal

rgerg sgsdgsfdh
sg
driga
rg
ergregearg
rgerg

7.7.3 Implementación

rgerg sgsdgsfdh
sg
driga
rg
ergregearg
rgerg

7.7.4 Refinamiento

rgerg sgsdgsfdh
sg
driga
rg
ergregearg
rgerg

7.7.5 Resultado

rgerg sgsdgsfdh
sg
driga
rg
ergregearg
rgerg

8 Demo final

- 8.0.1 Investigación inicial**
- 8.0.2 Planteamiento formal**
- 8.0.3 Implementación**
- 8.0.4 Refinamiento**
- 8.0.5 Resultado**

9 Conclusiones

El bajo nivel es importante, la demoscene no debe perderse, las matemáticas son fundamentales, avanzar hacia el futuro teniendo muy presente el pasado, a veces para avanzar hay que mirar atrás, entender lo que hicieron los que iban por detrás de nosotros y saber aplicarlo

A test for bibliography with Xe^LA_TE_X.[1]

Bibliografía

M. Alfonso, B. Bernardo, C. Carlos, and D. Domingo. El problema de los gatos y los perros. *Mascotas*, 50:112–115, 2010.

M. Alfonso, M. Marta, and N. Nuria. Mi viaje a EEUU. *Revista de viajes*, 14:50–56, 2010.