UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de la Imatge i la Tecnologia Multimèdia

Niagara particle system

Unai Diaz de Garayo

Director: Mónica Martín Mínguez

Grado: Diseño y Desarrollo de Videojuegos

Curso: 2022-23

Universidad: UPC

Índice

Resumen

Palabras clave 5

Enlaces 5

Índice de tablas 6

Índice de figuras 7

Glosario 8

1. Introducción 9

1.1 Motivación 9

1.2 Formulación del problema 9

1.3 Objetivos generales del TFG 10

1.4 Objetivos específicos del TFG 10

1.5 Alcance del proyecto 11

2. Estado del Arte/Marco Teórico/Contextualización/Estudio de Mercado 12

3. Gestión del proyecto 27

3.1. Herramientas y procediminetos 28

3.2. DAFO 28

3.3. Riesgos y plan de contingencias 29

3.4. Análisis inicial de costos 31

3.5 Gantt 31

4. Metodología 34

5. Desarrollo del proyecto 36

5.1 Desarrollo del proyecto 37

5.1.1 OpenGL 37

5.1.2 Devil 38

5.1.3 Imgui 39

5.2 Sistema de partículas complejo (simulación realista) 41

5.2.1 Formas de optimización 46

5.3 Sistema de partículas simple (alto rendimiento) 47

5.4 Visión general del desarrollo 57

6. Validación del proyecto 58

7. Conclusiones 59

8. Bibliografía 60

9. Anexos 61

Resumen

Los sistemas de partículas son un elemento muy importante de los videojuegos que aportan realismo e inmersión al jugador, estas, combinadas con efectos de postproducción, aportan detalles y completan la escena visualmente para que no parezca que falta algo o que la escena está vacía , no solo son importantes a nivel visual, también hay mecánicas que pueden basarse en este tipo de simulaciones.

El problema que presentan este tipo de simulaciones es que si se pretende conseguir resultados muy realistas, estas requieren de un gran cálculo de físicas y de renderizado que afectan al rendimiento general del juego, por lo tanto, es importante encontrar un sistema que consiga un punto medio que combine un buen resultado que sea creíble para el jugador, con una buena optimización que no afecte drásticamente al rendimiento del juego.

Para desarrollar este sistema se van a crear dos sistemas de partículas diferentes, una que simula de manera físicamente realista una simulación de humo con un renderizado de densidad realista, usando técnicas de *Ray Tracing* y de guardado en *caché* de la simulación para mayor optimización.

Por otra parte, el otro sistema estará más enfocado a la optimización, pensado para que afecte lo menos posible al rendimiento, pero también pensado para conseguir resultados que sean completamente creíbles para el jugador, este último sistema estará pensado para ser usado como los sistemas de partículas de Unity o Unreal, que dan más flexibilidad para crear todo tipo de sistemas.

El objetivo será comparar rendimientos y resultados de ambos sistemas para determinar cuál es más óptimo para ser usado en un contexto de *real time* y cuál es más apropiado para ser usado en un contexto donde los tiempos de cálculos no son tan relevantes como puede ser la industria cinematográfica.

Palabras clave

C++, Sistemas de partículas, OpenGL, Programación, Desarrollo, Análisis

Enlaces

https://github.com/unaidiaz/TFG_NIAGARA.git

Índice de Tablas

Tabla 1: DAFO...Pag. 30

Tabla 2: Estimado de costes según las horas trabajadas.......................................Pag. 34

Tabla 3: Estimado de costes de software y equipo complementario.....................Pag. 35

Tabla 4: Ventajas y desventajas de los sistemas..Pag. 70

Tabla 5: Técnicas utilizadas en cada uno de los
sistemas…………………….....................Pag. 71

Índice de Figuras

Figura 1: *Frames* de Star Trek ll........Pág. 14

Figura 2: Resultado visual del instanciado........Pág. 16

Figura 3: Resultados de distintos *shaders*........Pág. 17

Figura 4: Ejemplo de *fragment shader*........Pág. 18

Figura 5: Ejemplo de *vertex shader*........Pág. 18

Figura 6: Fórmulas de movimiento rectilíneo uniformemente acelerado........Pág. 20

Figura 7: Ejemplo de vórtice en Unity........Pág. 20

Figura 8: Figura 8 Fórmula de vórtice en código........Pág. 20

Figura 9: Fórmula de la gravitación universal........Pág. 21

Figura 10: Fórmulas de colisiones elásticas........Pág. 21

Figura 11: Ejemplos de los distintos casos con colisiones elásticas........Pág. 22

Figura 12: Cuadrícula 3D para la simulación de fluidos........Pág. 23

Figura 13: Ejemplo 2D del cálculo de simulación de fluidos........Pág. 24

Figura 14: Imagen del primer uso de la técnica de *Ray Tracing*........Pág. 24

Figura 15: Esquema del cálculo de rayos por pantalla........Pág. 25

Figura 16: Cálculos de intersección esfera - rayo........Pág. 26

Figura 17: Cálculos de intensidad de luz por cada punto de contacto........Pág. 27

Figura 18: Resultado de cálculos de *Ray Tracing* con 3 esferas y una luz........Pág. 27

Figura 19: Resultado de cálculos de *Ray Tracing* con humo........Pág. 28

Figura 20: Ejemplo de proyecto de Trello........Pág. 30

Figura 21: Lista de opciones de Unity Particle System........Pág. 32

Figura 22: Logo de la librería gráfica OpenGL........Pág. 38

Figura 23: Ventana de vinculación de librerías de Visual Studio........Pág. 39

Figura 24: Logo de la librería Devil........Pág. 39

Figura 25: Ejemplo de modificaciones en texturas de Devil........Pág. 40

Figura 26: Figura 26 Logo de la librería Imgui........Pág. 41

Figura 27: Ejemplo de interfaz de usuario con Imgui........Pág. 41

Figura 28: Resultado de renderizado realista........Pág. 42

Figura 29: Fórmula y explicación gráfica para pasar valores de entorno 3D a *array* 2D...Pág. 43

Figura 30: Código para cada paso de simulación del fluido..................................Pág. 44

Figura 31: Descripción gráfica de los cálculos de simulación de fluidos................Pág. 45

Figura 32: Explicación gráfica de rayo atravesando densidad para recoger valores…………………………………………………………………………… …………………………..Pág. 46

Figura 33: Código de *fragment shader* para el renderizado de densidad..............Pág. 46

Figura 34: Resultado temporal del renderizado de humo.....................................Pág. 47

Figura 35: *Vertex shader* del plano de renderizado final.....................................Pág. 47

Figura 36: Botón de cacheado del simulador...Pág. 48

Figura 37: Ventana de configuración del guardado...Pág. 48

Figura 38: *Start* del simulador..Pág. 49

Figura 39: Código principal del *PreUpdate* del simulador....................................Pág. 50

Figura 40: Código principal del *Update* del simulador..Pág. 51

Figura 41: Código principal del *PsotUpdate* del simulador..................................Pág. 51

Figura 42: UI de la configuración general del simulador......................................Pág. 52

Figura 43: UI de la configuración *transform* del simulador...................................Pág. 54

Figura 44: UI de la ventana de texturas del simulador...Pág. 55

Figura 45: UI de la ventana de configuración y edición de texturas del simulador.Pág. 55

Figura 46: UI de la configuración *textures* del simulador……………………………………Pág. 57

Figura 47: UI de la configuración *Physics* del simulador………………………….…………Pág. 58

Figura 48: Figura 48 Resultado visual de la prueba básica de rendimiento (sistema simple)...Pág. 61

Figura 49: Resultado técnico de la prueba básica de rendimiento (sistema simple)...Pág. 61

Figura 50: Resultado visual de la segunda prueba de rendimiento (sistema simple)...Pág. 62

Figura 51: Resultado técnico de la segunda prueba de rendimiento (sistema simple)..Pág. 62

Figura 52: Resultado técnico de la tercera prueba de rendimiento (sistema simple)..Pág. 63

Figura 53: Resultado visual del instanciado..Pág. 64

Figura 54: Archivos de guardado de la simulación...Pág. 65

Figura 55: Resultado visual de la primera prueba (sistema simple).......................Pág. 66

Figura 56: Recursos utilizados para la primera prueba visual (sistema simple).....Pág. 66

Figura 57: Resultado visual de la segunda prueba (sistema simple)......................Pág. 67

Figura 58: Recursos utilizados para la segunda prueba visual (sistema simple).....Pág. 68

Figura 59: Resultado visual 1 (sistema complejo)……………………………….…………….Pág. 69

Figura 60: Resultado visual 2 (sistema complejo)…………………………..….…………….…...Pág. 69

Glosario

Guardado en caché: Técnica que se basa en el guardado de geometría y otros datos de una simulación para no volverla a ejecutar y así ahorrar recursos.

Shaders: Programa que se ejecuta en la tarjeta gráfica usada para procesar y dibujar elementos gráficos de una escena 3D, usados para aplicar efectos visuales de todo tipo.

Opengl: Librería gráfica estándar en la industria de desarrollo de aplicaciones gráficas en tiempo real, ofrece un conjunto de funciones y herramientas para dibujar elementos en 3D y 2D.

GLSL: Lenguaje de programación desarrollado por Khronos usado para escribir *shaders* para el sistema gráfico OpenGL.

Instanciado: Técnica utilizada para mostrar y gestionar grandes cantidades de elementos gráficos de manera eficiente, de esta manera se crea una sola instancia de un elemento gráfico en lugar de crear una copia individual.

GPU: Es un elemento que contiene el ordenador especialmente diseñado para acelerar el procesamiento de gráficos y de esta manera mejorar el rendimiento de aplicaciones que usan gráficos en 3D o videos.

CPU: Este es el componente principal de un ordenador, encargado para ejecutar las instrucciones de los programas abiertos, consta de uno o varios núcleos de procesamiento capaces de ejecutar este tipo de instrucciones y cálculos.

Bus: Es el conjunto de líneas de comunicación que se utilizan para transmitir datos, señales de control y energía eléctrica entre los diferentes componentes de un ordenador.

Interpolación: Técnica que se basa en encontrar los puntos intermedios entre dos puntos ya dados para hacer una transición suave.

Depurar: En el desarrollo de software se refiere a la acción de identificar, analizar y corregir fallos en el código de un programa, estos pueden ser tanto de lógica, sintaxis, rendimiento, etc.

Deferred Shading: Es una técnica de programación gráfica que almacena en buffers especiales diferentes datos de cada objeto de forma individual para después juntarlo todo y renderizar por pantalla en un plano colocado delante de la pantalla.

1. Introducción

1.1 Motivación

La simulación de partículas en los videojuegos es un elemento muy importante porque permite a los desarrolladores crear una amplia gama de efectos visuales que añaden realismo e inmersión al mundo del juego.

Además de crear efectos visuales, las partículas también se pueden usar para crear o mejorar mecánicas de juego, por ejemplo, se pueden usar para simular movimientos de objetos del juego o para crear mecánicas completamente basadas en este tipo de simulaciones, además de poder simular interacciones físicas de distintos elementos como choques o explosiones.

En general, son una herramienta versátil para crear una amplia gama de elementos visuales.

Debido a la importancia de este elemento para la credibilidad y la estética general del juego o contenido audiovisual, este trabajo se centrará en estudiar la mejor manera de crear un sistema de partículas que combine un buen resultado con una correcta optimización.

Aparte de la investigación objetiva, este proyecto también me permitirá a nivel personal conocer, investigar y utilizar muchas técnicas desconocidas hasta el momento y que son realmente utilizadas en el mundo de los videojuegos y de esta manera ampliar mi conocimiento en este ámbito. También por la complejidad de estos sistemas, el desarrollo de este trabajo supondrá un reto que superar, además, al ser un área que está en constante movimiento y en busca de nuevas técnicas y tecnologías para realizar simulaciones más realistas, el estar interesado en este ámbito me permitirá estar siempre al tanto de las últimas tendencias tecnológicas.

1.2 Formulación del problema

Los sistemas de partículas que intentan simular la realidad tienen un problema principal que se debe abordar.

Cuanto más realista se pretende hacer un sistema, más cálculo de físicas y cálculos de renderizado se deben hacer para conseguir el resultado deseado, por supuesto esto para un proyecto cinematográfico no es problema, puesto que estos cálculos (tanto de físicas como el renderizado en sí) se pueden realizar con tiempo, pero en los proyectos *in real time,* como son en este caso los videojuegos esto no es posible.

Para este último caso se debe buscar un sistema que consiga crear buenos resultados (dependiendo el estilo que se busque para cada proyecto, realista, cartoon …), con un cierto uso de recursos que permita integrar este sistema con el resto de juego sin que el rendimiento se vea afectado en gran medida y que se pueda seguir disfrutando de la experiencia con total normalidad.

El problema consiste principalmente en encontrar el sistema que combine resultados creíbles como los que podemos encontrar en el cine, manteniendo una buena optimización.

1.3 Objetivos generales del TFG

El objetivo general de este proyecto es la creación de dos sistemas de partículas para realizar una comparación de los resultados que puede ofrecer cada una de ellas, tanto del acabado estético como del rendimiento para determinar cuál es la mejor opción para proyectos *in real time* y también investigar las diferentes técnicas que se utilizan en cada una de ellas, pudiendo llegar así a una conclusión de qué sistema utilizar (o combinación de los dos) para diferentes proyectos.

Como objetivo general, también podemos incluir la implementación de sistemas de optimización para tener la posibilidad de incluir el sistema de partículas más exigente de los dos en un proyecto *in real time*, sin que afecte drásticamente al rendimiento.

1.4 Objetivos específicos del TFG

Para la ejecución de este proyecto se deberán de abordar ciertos objetivos específicos, estos objetivos que se han ido presentando durante el desarrollo son los siguientes.

- Completar y ampliar los conocimientos de programación gráfica con OpenGL, incluyendo nuevas técnicas como *instancing* para mostrar muchos elementos por pantalla sin afectar al rendimiento.
- Aprendizaje del lenguaje GLSL para poder trabajar con *shaders* tanto para efectos de postproducción, crear la base de los propios sistemas y crear los sistemas de iluminación.
- Reunir conocimientos sobre distintas técnicas de renderizado, *Ray Tracin, Ray Maching*.
- Investigación e implementación de distintos sistemas de optimizado, como el de guardado en *caché*, para el guardado de la simulación.
- Investigación e implementación de las opciones que ofrecen los sistemas utilizados hoy en día (Unity, Unreal) para incluirlas e incluso ampliar su funcionalidad.
- Gestionar el volumen de trabajo que exige el proyecto.
- Poner a prueba los conocimientos adquiridos por el grado aplicándolos en este proyecto.

1.5 Alcance del proyecto

La finalidad de este proyecto es conseguir un sistema de partículas optimizado para el uso en proyectos *in real time*, con resultados interesantes, por lo tanto, este producto final podría estar destinado a la venta a estudios que estén desarrollando su propio motor, es decir, que no estén usando ni Unity ni Unreal, además de su uso para cualquier proyecto que requiera de un sistema de partículas parecido al de Unreal o Unity y que se esté desarrollando en otra plataforma.

2. Estado del arte/ Marco Teórico/ Contextualización/ Estudio de Mercado

Es difícil asegurar con certeza el uso del primer sistema de partículas en un videojuego, ya que han evolucionado, uno de los primeros usos que se le dio a estos sistemas más avanzados en la industria del entretenimiento fue en la película de 1982 "Star Trek II" en la escena en la que se usa el sistema trata de un torpedo que se lanza desde una nave, cuando este impacta contra un planeta estéril es capaz de reorganizar la materia y crear un mundo habitable, durante la secuencia en sí se puede ver un muro de fuego que atraviesa todo el planeta para crear vida.

Figura 1 Frames de Star Trek II.

En la industria de los videojuegos es probable que este tipo de sistemas de partículas más avanzados probablemente se empezarían a usar en la década de 1980, cuando los juegos comenzaron a tener gráficos más avanzados y más recursos de procesamiento para crear efectos visuales más impresionantes, al principio estos sistemas eran muy simples, pues eran un muy reducido número de partículas que siguen un conjunto de reglas muy básicas.

Esto con el paso del tiempo ha ido cambiando gracias a los avances tanto en software y hardware de los equipos, esto ha permitido, por una parte, mostrar más cantidad de partículas por pantalla, consiguiendo efectos visuales más potentes, combinados con sistemas de iluminación más realistas y, por otra parte, simular movimientos físicos más elaborados, esto no ha sido posible gracias únicamente a los avances en hardware, también ha sido posible gracias a técnicas como el instanciado, el uso de *shaders, Ray Tracing*…

Como ya se ha comentado antes, con el paso del tiempo y los avances en software y hardware se han podido desarrollar sistemas de partículas más complejos, los sistemas más populares actualmente en el desarrollo de videojuegos son los siguientes, Niagara Particle System de Unreal, Unity Particle System o Cry Engine Particle Editor, cada uno

de estos sistemas permite crear efectos visuales muy complejos para poder simular efectos de agua, fuego, humo, etc. cada uno de estos sistemas se componen de emisores y estos como su propio nombre indica emiten una gran cantidad de partículas que se mueven por un espacio siguiendo unas reglas físicamente correctas y simulando el comportamiento real, estos sistemas además permiten que las simulaciones no interactúen solo entre ellas mismas, sino también con el entorno, aportando de esta manera un plus en realismo.

Cada uno de estos sistemas utilizan distintas técnicas avanzadas que ya se verán más adelante para conseguir resultados muy interesantes, no solo visualmente hablando, también en el aspecto del rendimiento, pero todas ellas utilizan de base las mismas reglas. Todas ellas asignan valores de velocidad, aceleración, escala, color, gracias a solo estos atributos básicos los sistemas pueden ser programados para simular una explosión con chispas o un efecto de lluvia simplemente ajustando estos valores básicos.

2.1 Técnica de instanciado

El instanciado es una técnica utilizada en programación de videojuegos para mostrar muchas copias de un mismo objeto por pantalla sin consumir demasiados recursos de procesamiento.

La idea base detrás de instanciado es muy sencilla, se basa en crear una sola copia del objeto que se quiere mostrar, llamado “prototipo” y luego crear copias de ese objeto llamadas “instancias” en tiempo de ejecución, esto permite mostrar muchas copias del mismo objeto sin tener que almacenar ni gestionar individualmente cada una de ellas, a cada una de estas instancias se les aplican diferentes transformaciones (posición, rotación, escala) para simular la individualidad de cada una de estas partículas.

Entrando más en profundidad sobre este tema al dibujar muchas copias de un mismo objeto sin la técnica de instanciado el programa llegará rápidamente a un cuello de botella debido a las muchas llamadas a dibujo, la librería gráfica (en este caso OpenGL) debe hacer ciertos preparativos antes de cada llamada a dibujado para poder dibujar vértices por pantalla, como decirle a la GPU de que buffer debe leer los datos, donde encontrar los diferentes atributos de cada vértice, etc. Todo esto además debe hacerse a través de un bus de la CPU a la GPU que es bastante lenta en comparación con los demás procesos.

Utilizar la técnica del instanciado es mucho más conveniente, pues, esta técnica se basa en enviarle todos los datos a la GPU una vez y luego decirle a OpenGL que dibuje varias copias de ese mismo objeto usando esos mismos datos, todo con una sola llamada a dibujado ahorrándonos muchas comunicaciones CPU -> GPU las cuales son las más lentas. Sin embargo, esta técnica no serviría de nada si todas esas copias se dibujaran

en la misma posición, rotación y escala, por lo tanto, a la GPU también se le pasa un *array* de las diferentes posiciones, rotaciones y escalas de cada una de esas copias.

Utilizando esta técnica, la tarjeta gráfica recibirá una sola vez la información del objeto a dibujar, la cantidad de copias que debe dibujar y un *array* de las posiciones, rotaciones y escalas de cada una de esas copias para que la tarjeta gráfica las dibuje sin tener que comunicarse más veces con la CPU.

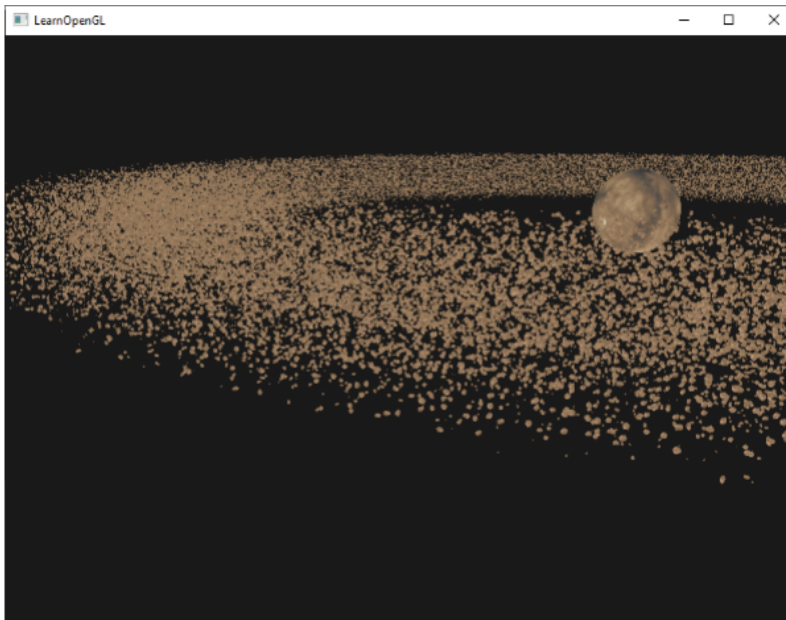


Figura 2 Resultado visual del instanciado.

El instanciado, como cualquier otra técnica, tiene ventajas y desventajas, como ventajas podemos destacar las siguientes.

- Eficiencia: Permite dibujar muchas veces un mismo objeto por pantalla a partir de una sola copia de este sin consumir demasiados recursos.
- Personalización: Esta técnica permite personalizar cada instancia de manera individual aplicando diferentes posiciones, escalas y rotaciones.
- Facilidad de uso: Esta técnica es bastante fácil de utilizar, pues las librerías gráficas ya ofrecen líneas de código para hacer esto de manera relativamente sencilla.

En cuanto a las desventajas, estas son algunas de las principales:

Mayor espacio de almacenamiento: El instanciado es bastante eficiente en términos de procesamiento, pero necesita almacenar la información del prototipo y de todas las instancias creadas a partir de él, por lo tanto, hará más uso de la memoria.

2.2 Uso de shaders

Los *shaders* son programas ejecutados en la GPU que son usados para dibujar gráficos por pantalla, principalmente hay dos tipos de *shaders, fragment shader* y *vertex shader*, el primero de todos es usado para aplicar texturas, colores, iluminación y sombreado entre otros muchos efectos, el último de ellos es usado principalmente para dibujar los modelos 3D en sí, además de aplicarles distintas transformaciones de posición, rotación, escala, etc.

Figura 3 Resultados de distintos *shaders.*

Como podemos ver en la anterior imagen modificando estos *shaders* podemos conseguir distintos aspectos de la misma escena, dotando de distintos estilos según el momento de la partida del lugar del mapa o del estilo generalizar del juego, estos se ejecutan a tiempo real en la tarjeta gráfica por lo que cualquier cambio que se realice en estos *shaders* se verán instantáneamente por pantalla, por lo que no solo se pueden usar para aspectos generales del entorno, también se pueden usar para distintos efectos como fundidos a negro, visión borrosa por el ataque de algún enemigo o la visión en blanco y negro entre muchas posibilidades.

Estos pequeños programas son escritos en el lenguaje de programación GLSL, como se puede ver a continuación.

```
#version 330 core
out vec4 FragColor;
in vec2 TexCoord;
uniform vec4 Color;
uniform sampler2D texture1;
layout(depth_less) out float gl_FragDepth;
void main()
{

  FragColor = texture(texture1, TexCoord) * Color;
}
```

Figura 4 Ejemplo de *fragment shader.*

La anterior imagen es un ejemplo muy sencillo de *fragment shader* donde podemos ver como se le aplica la textura al modelo según las coordenadas que le llegan de fuera, más tarde a ese resultado se le multiplica por un color para poder modificarlo, en caso de ser el color blanco no se modificara nada.

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoord;
out vec2 TexCoord;
out vec3 ourColor;
uniform mat4 transform;
uniform mat4 view;
uniform mat4 projection;
uniform bool isFaced;
mat4 myModelView;
void main()
{
   ourColor = vec3(0, 0, 0);
if (isFaced)
{

      myModelView=view* transform;

      myModelView[0][1] = 0.0;
      myModelView[0][2] = 0.0;

      myModelView[1][0] = 0.0;
      myModelView[1][2] = 0.0;

      myModelView[2][0] = 0.0;
      myModelView[2][1] = 0.0;
      vec4 P = myModelView * vec4(aPos, 1.0);


      gl_Position = projection  *P;

}
else
{

gl_Position = projection * view * transform * vec4(aPos, 1.0);

}



   TexCoord = vec2(aTexCoord.x, aTexCoord.y);
}
```

Figura 5 Ejemplo de *vertex shader.*

En este caso la última foto corresponde a un *vertex shader*, en este caso en el *main* tenemos un *if* que controla si la partícula debe estar mirando a cámara o no, si lo está debemos modificar su matriz de transformación para mantener la escala y la posición, pero modificando la rotación para que mire siempre a cámara, en el otro caso simplemente aplicamos las distintas transformaciones y las matrices de *view* y *projection*.

Los cálculos de las rotaciones de las partículas y las multiplicaciones de las matrices de *view*, *transform*, *projection* … Se hacen en la GPU en vez de la CPU, por una parte, para descongestionar la carga de trabajo de la CPU, por otra parte, porque la tarjeta gráfica cuenta con una gran cantidad de núcleos de procesamiento que les permite procesar grandes cantidades de datos, además de que están diseñadas para tener un gran rendimiento específicamente para tareas de cálculo.

2.3 Motores de físicas

Los motores de física en videojuegos se utilizan para simular de forma realista fenómenos físicos de la vida real en un entorno virtual, la cantidad de fenómenos físicos que estos motores pueden simular son muchos, los motores físicos pueden, como cabría esperar, simular fenómenos simples como gravedad, aceleración y movimientos simples, pero los motores más desarrollados y usados en el ámbito profesional pueden llegar a simular fenómenos más complicados como colisiones, dinámica de fluidos, fuerzas de empuje…

En el mercado existen muchos motores destinados a simular sucesos físicos en los videojuegos, como por ejemplo, PhysX, Havok, Bullet Physics entre muchos otros ejemplos, cada uno de ellos, sobre todo en fenómenos y cálculos muy precisos como simulación de tejidos o simulación de fluidos se obtienen resultados y acabados diferentes, por lo que dependiendo de las necesidades de cada proyecto tanto en el aspecto visual como en rendimiento es recomendable utilizar uno u otro.

En cuanto al cálculo de física de movimientos, aceleraciones, colisiones… Todos ellos lo realizan de una forma muy parecida, todos ellos generan variables básicas de velocidad, posición, rotación, aceleración… y todos ellos utilizan las mismas fórmulas que en la vida real para poder mostrar resultados, algunos de los ejemplos más comunes son, por ejemplo, de movimiento rectilíneo uniformemente acelerado que se muestran en la siguiente imagen.

```
Vf = Vo + a*Dt

Xf = Xo +(Vo*Dt)+(0.5*a*(Dt^2))
```

Figura 6 Fórmulas de movimiento rectilíneo uniformemente acelerado.

Todas estas fórmulas se aplican de forma independiente a cada eje del entorno virtual en el que se esté trabajando, por ejemplo, si se trabaja en un entorno 2D solo se aplicará en los ejes X e Y, en cambio, si se trabaja en un entorno 3D se aplicará en los 3 ejes, estos cálculos se calcularán cada *frame* por cada partícula que se esté simulando en ese momento de forma independiente para conseguir resultados distintos.

Gracias a que cada partícula posee sus propias variables de velocidad, posición y aceleración, se puede realizar cualquier efecto físico cambiando alguna de las variables para conseguir el efecto deseado, una de ellas es por ejemplo el efecto de vórtice que se vería de la siguiente manera.

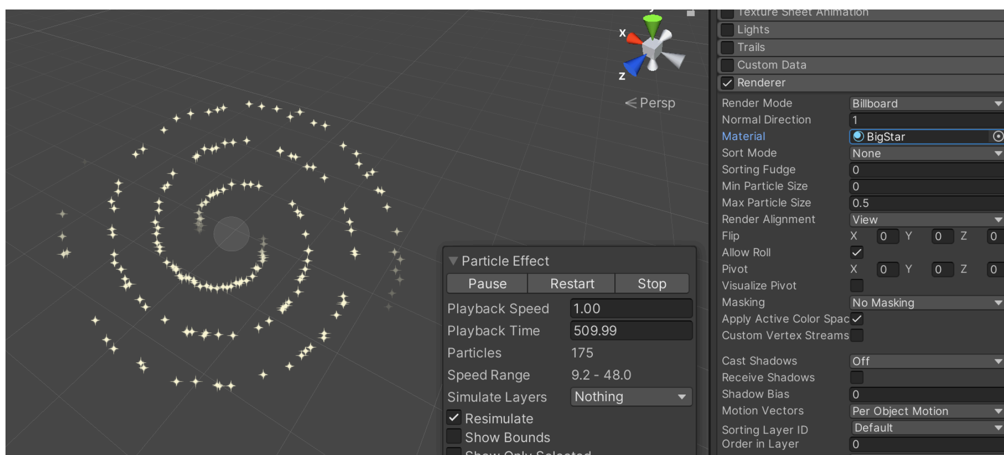


Figura 7 Ejemplo de vórtice en Unity.

Para este caso en concreto simplemente se debe ir modificando los valores de velocidad de cada eje para conseguir un movimiento circular, según la fórmula de abajo, combinando la función matemática de “sin” con la función matemática de “cos” se puede conseguir el movimiento cíclico usando el mismo valor, pues cada una de esas funciones dará como resultado el valor contrario.

```
actualSinVaule = auxiliar->GetParticleActualLifeTime() / ZVortexRadius;
newVelocity.x = (glm::sin((actualSinVaule * 180) / M_PI) * ZVortexVelocity) + oldVelocity.x;
newVelocity.y = (glm::cos((actualSinVaule * 180) / M_PI) * ZVortexVelocity) + oldVelocity.y;
newVelocity.z = oldVelocity.z;
```

Figura 8 Fórmula de vórtice en código.

Otro ejemplo destacable de fórmulas de la realidad que se aplican en un motor físico es la fórmula de la gravitación universal, que simplemente utiliza la distancia entre los dos objetos que se atraen y la masa de cada uno de los dos, ejerciendo más fuerza de atracción el que más peso tenga, en el caso de la fórmula real también se debe añadir la constante de gravitación universal representada abajo con la G.

```
F = G((M1 * M2)/(R^2))
```

Figura 9 Fórmula de la gravitación universal.

Esta fórmula se utiliza principalmente para crear efectos de atracción o repulsión, muy utilizados en videojuegos para diferentes efectos visuales, como la gravitación de planetas, estrellas y todo tipo de objetos.

Otro cálculo esencial para cualquier motor físico es el de cálculo de colisiones entre objetos, hay varios tipos de colisiones según si pierden energía o no, colisiones elásticas o inelásticas.

Primero de todo se ha de comprobar si dos objetos están colisionando o no, en el caso de dos partículas se puede simplemente adjudicar un radio a cada una de ellas para facilitar el cálculo, inmediatamente después utilizando las fórmulas de la imagen de abajo se toma en cuenta la velocidad y la masa de cada una de ellas para sustituir la velocidad final de las dos, simulando así una colisión real entre dos objetos.

$$v'_2 = \frac{2m_1}{m_1 + m_2} v_1 - \frac{m_1 - m_2}{m_1 + m_2} v_2$$

$$v'_1 = \frac{m_1 - m_2}{m_1 + m_2} v_1 + \frac{2m_2}{m_1 + m_2} v_2$$

Figura 10 Fórmulas de colisiones elásticas.

Dependiendo de la masa y de la velocidad de cada una de las partículas que interactúan, se pueden llegar a dar diferentes casos, dando como resultado diferentes velocidades finales (teniendo en cuenta que la masa de ambas partículas en principio no cambia).

En el caso de dos partículas colisionando se da por hecho que ambas dos van a cambiar su estado, pero se puede dar el caso de que el objeto contra el que se está colisionando no cambie su estado (una pared no cambiará su velocidad ni su masa), en este caso solo cambiará el estado de una de las dos partes facilitando el cálculo.

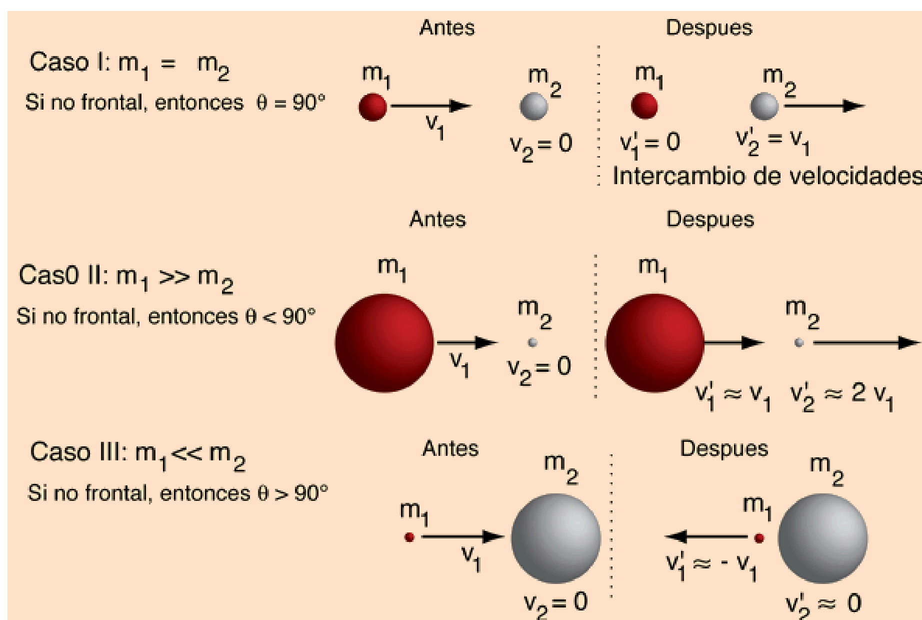


Figura 11 Ejemplos de los distintos casos con colisiones elásticas.

Realizar todos estos cálculos, cada *frame* para cada partícula puede llegar a ser bastante costoso dependiendo del hardware y la situación que se esté intentando simular, por ello según los requerimientos de cada proyecto se pueden realizar técnicas de interpolación para poder ahorrar cálculos y no hacer el cálculo de todas las partículas cada *frame*, sino una vez cada X *frames*, otra opción sería realizar los cálculos de física en otro hilo de procesamiento para minimizar el efecto en el rendimiento general, esta técnica es muy utilizada en los juegos AAA.

Hasta ahora se han mostrado los cálculos de los efectos más comunes en los sistemas de partículas en videojuegos, pero los motores de física también son capaces de hacer simulación de fluidos realistas.

Esta simulación se basa en dividir un espacio 3D en múltiples cuadrados y a cada uno de ellos les asigna un valor de densidad (que al principio es cero) y un valor de velocidad.

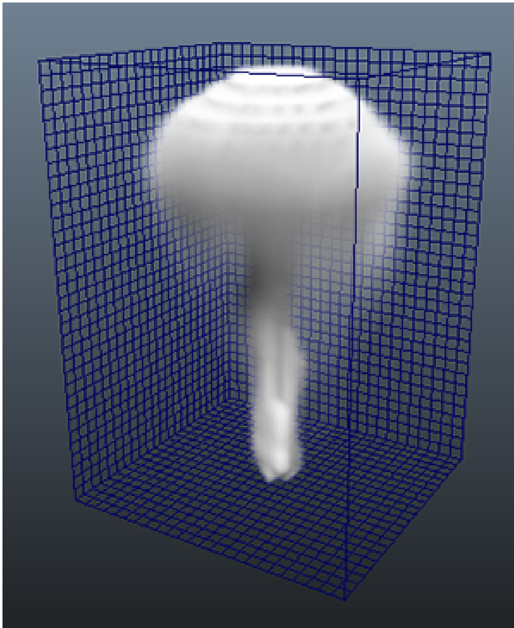


Figura 12 Cuadrícula 3D para la simulación de fluidos.

La simulación comienza cuando se agrega valores de densidad a ciertos cuadrados y un valor de velocidad, la simulación consiste en ir transmitiendo esos valores de densidad y velocidad a los cuadrados que hay alrededor como lo haría un gas en la vida real, de esta manera mediante interacciones se van calculando los valores de densidad de los distintos cubos que componen el espacio.

En la imagen inferior se puede ver con un ejemplo 2D el método de ir calculando los valores de densidad de los distintos cuadrados del *grid* y la forma en la que estos se van propagando por los distintos cuadrados según los valores de velocidad, una vez hechos todos estos cálculos, se obtiene un *array* con valores de densidad de todo el espacio que se está calculando, con estos datos posteriormente se pueden realizar cálculos de *Ray Tracing*.

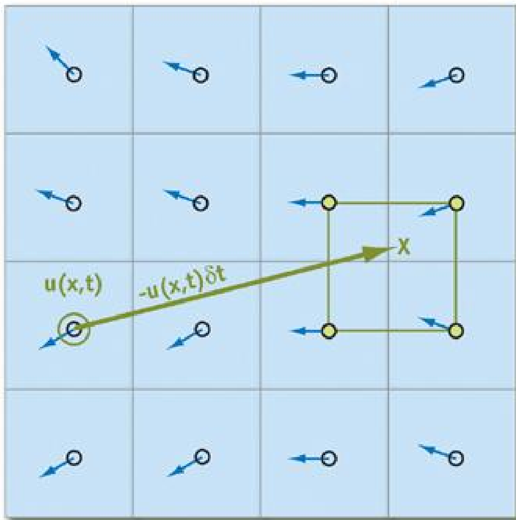


Figura 13 Ejemplo 2D del cálculo de simulación de fluidos.

2.4 Tecnica de Ray Tracing

El *Ray Tracing* es una técnica de renderizado que se basa en simular de manera físicamente correcta el funcionamiento de los rayos de luz y cómo interactúan con todos los objetos de una escena, con esta técnica conseguimos resultados muy parecidos a la realidad, pues estamos replicando de manera exacta el funcionamiento de la luz en la realidad.

El primer uso conocido de esta técnica data de 1972 en un corto animado llamado “A Computer Animated Hand” para un proyecto de postgrado de la universidad de Utah, donde se creó un modelo 3D de una mano izquierda y se usó la técnica de *Ray Tracing* para crear una iluminación realista con sombras suaves.

Figura 14 Imagen del primer uso de la técnica de Ray Tracing.

Hasta hace un tiempo esta técnica era únicamente utilizada en cine o en proyectos que no fueran *in real time* por la complejidad de los cálculos, pero gracias a los avances en hardware las tarjetas gráficas de hoy en día son capaces de hacer estos cálculos de manera muy eficiente y desde hace tiempo ya se llevan usando en videojuegos.

Como ya se ha comentado antes y el propio nombre indica, la técnica se basa en lanzar un rayo por cada pixel de pantalla para saber contra qué objeto colisiona, una vez tenemos los puntos de colisión de cada objeto y tenemos las posiciones de cada punto de luz podemos calcular la cantidad exposición que tiene cada punto de superficie a las diferentes luces y de esta manera adjudicar un color u otro dependiendo de la intensidad de cada luz.

A continuación se muestra el proceso matemático para conseguir una iluminación realista utilizando 3 esferas.

Lo primero es lanzar un rayo por cada píxel de pantalla, para lograr una línea en un espacio 3D se necesitan mínimo dos puntos, uno de ellos será la posición de la cámara y otra será la posición normalizada de cada píxel de la pantalla.

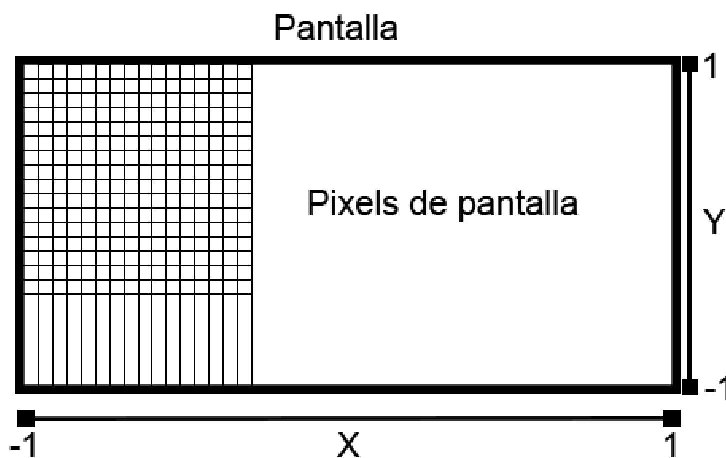
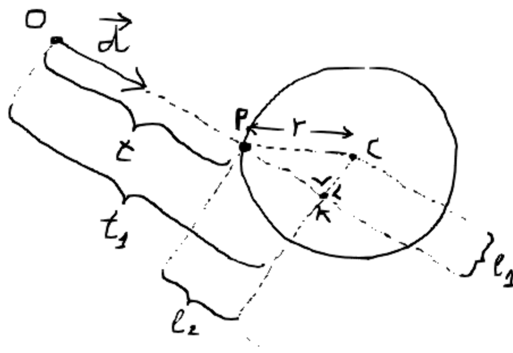


Figura 15 Esquema del cálculo de rayos por pantalla.

Una vez tengamos todas las direcciones de todos los rayos lanzados solo necesitaremos las posiciones de cada esfera y su radio, con esta información podemos calcular si un rayo ha colisionado con una esfera y en ese caso podemos calcular el punto de contacto.

Ray-sphere intersection



Inputs:

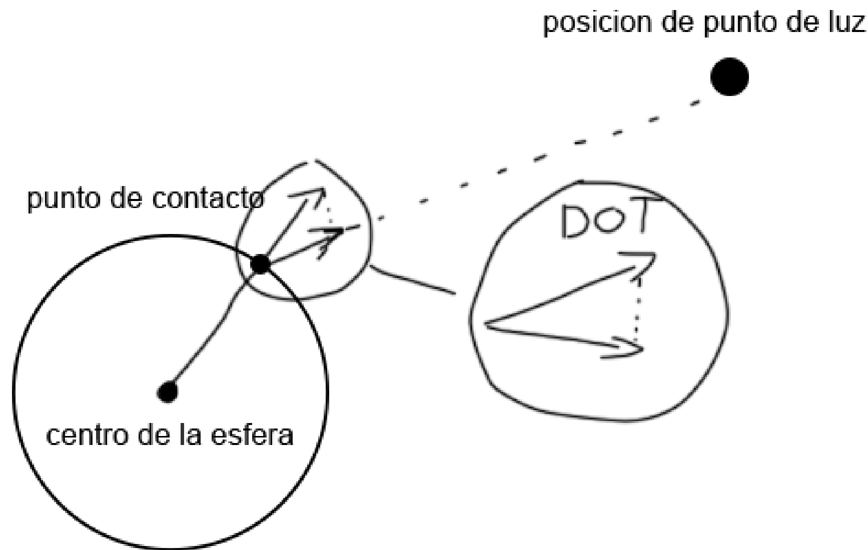
O – eye origin
$\vec{d}$ – ray direction
C – sphere center
r – sphere radius

$$t_1 = dot(\vec{OC}, \vec{d})$$
$$K = O + t_1 * \vec{d}$$
$$l_1 = length(\vec{CK})$$
$$l_2 = \sqrt{r^2 - l_1^2} \leftarrow \text{Pythagoras}$$
$$t = t_1 - l_2$$
$$P = O + t * \vec{d}$$

Figura 16 Cálculos de intersección esfera - rayo.

Mediante estos simples cálculos podemos determinar si un rayo intercepta con una esfera, si el valor de L1 es menor o igual que el radio de la esfera, el rayo estará colisionando y entonces podremos calcular el punto de intersección en este ejemplo llamado P.

Con el punto en 3D de P podemos saber la normal de este punto con relación a la esfera en total, haciendo un vector entre ese punto y el centro de la esfera.
Podremos también hacer un vector con el punto de contacto y el punto de luz y con esos dos vectores (normalizados) podemos hacer el "DOT" para saber la proyección de un vector sobre el otro, este valor irá de (-1 a 1), haciendo este proceso por cada punto de la esfera podemos asignar los valores de intensidad de luz.



normal de la esfera = punto de contacto - centro de la esfera

normal de la luz = posicion de punto de luz - punto de contacto

valor de la intensidad en ese punto = DOT(normal de la esfera,normal de la luz)

Figura 17 Cálculos de intensidad de luz por cada punto de contacto.

Si los vectores son exactamente los mismos, quiere decir que ese punto está de cara al punto de luz y tendrá un valor de 1, pues, la proyección de uno de los vectores sobre el otro es total, repitiendo el proceso conseguiremos diferentes vectores y, por lo tanto, diferentes proyecciones que se traducen en un degradado suave en la esfera.

Replicando estos cálculos en un *shader* podremos conseguir resultados como este.

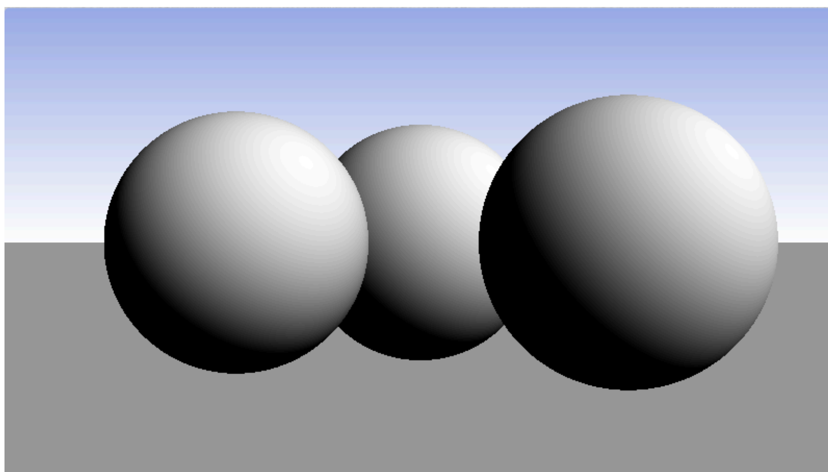


Figura 18 Resultado de cálculos de *Ray Tracing* con 3 esferas y una luz.

Este es un ejemplo muy simple de *Ray Tracing*, pero siguiendo esta idea se pueden crear imágenes muy realistas e incluso renderizar humo, teniendo un *array* de valores de densidad (un cubo con valores internos de densidad) se pueden lanzar rayos que vayan del final hasta el principio del cubo acumulando los valores de densidad y mostrando finalmente una especie de volumetría.

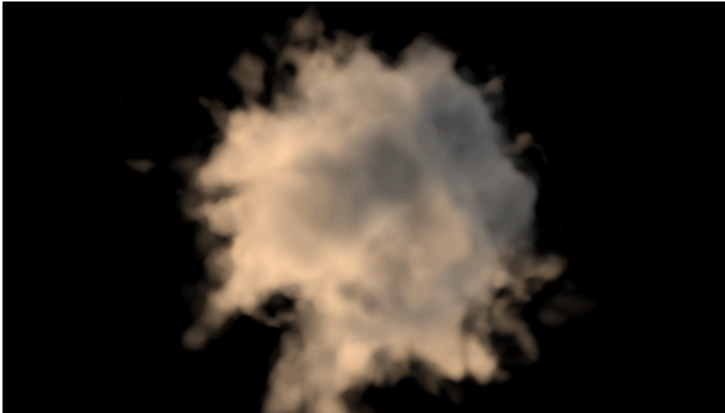


Figura 19 Resultado de cálculos de *Ray Tracing* con humo.

3. Gestión del proyecto

3.1. Herramientas y procedimientos de organización.

En este apartado se mostrarán las herramientas que se han utilizado para poder asegurar la correcta organización de todos los apartados del TFG

3.1.1 Trello

Trello es un software de organización de tareas, este programa permite crear diferentes tableros para distintas áreas, es decir, permite crear un tablero para las tareas del trabajo, otro tablero para las tareas de casa, etc.

Dentro de cada tablero el programa permite crear diferentes listas, con un título diferente cada una, y dentro de cada una de esas listas se pueden ir añadiendo distintas tarjetas de las tareas específicas, cada una de estas tarjetas se pueden modificar y personalizar de muchas maneras, colocando fechas, colores, etiquetas, descripciones detalladas, adjuntar archivos, etc.

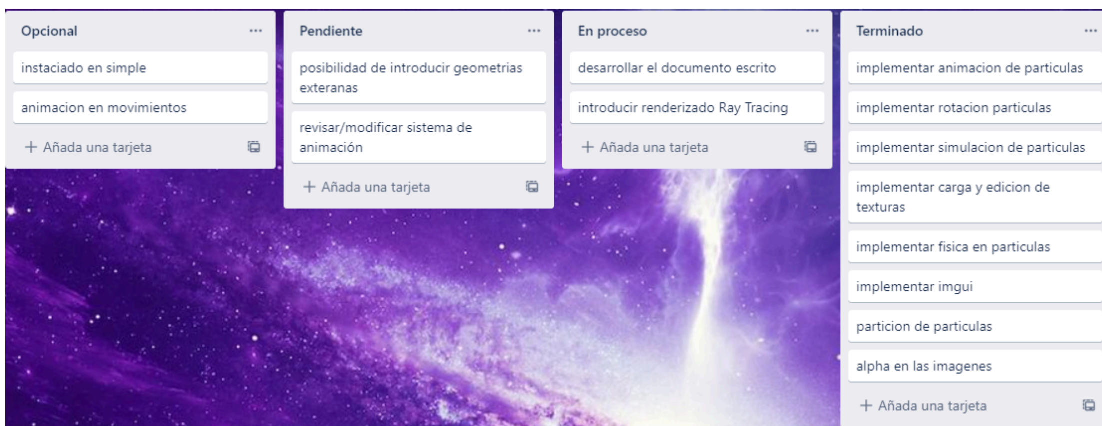


Figura 20 Ejemplo de proyecto de Trello.

Cada usuario puede organizar la tabla y las listas como desee, para este proyecto se ha hecho de la siguiente manera.

Se han añadido 4 listas principales que son “Opcional”, “Pendiente”,”En Proceso” y “Terminado”. Como sus propios nombres indican en la lista “Opcional” se incluirán las tareas que son opcionales para el desarrollo del proyecto, pero que nos son esenciales, en “Pendiente” se añadirán las tareas que están pendientes por realizar, en la lista”En Proceso” las que están en proceso y “Terminado” las que ya se han realizado.

Esta es una forma visual y muy efectiva de organizar y ser consciente de la cantidad de tareas que se han de realizar o de las que ya se han terminado, y de esta manera ser más eficiente a la hora de organizarse en cuanto a tiempo.

3.1.2 GitHub

GitHub es una plataforma que permite a los desarrolladores de software guardar sus proyectos en repositorios en línea, facilitando de esta manera la colaboración entre varias personas a la vez, en esta plataforma también se pueden encontrar todo tipo de proyectos que se pueden descargar de manera gratuita y modificarlos para cualquier uso.

Para este proyecto se ha utilizado GitHub para alojar el proyecto en un repositorio en línea y de esta manera tener un acceso más rápido y cómodo a él.

3.2. DAFO

	Positivos	**Negativos**
Origen Interno	**Fortalezas**	**Debilidades**
Origen Externo	**Oportunidades**	**Amenazas**

Tabla 1: DAFO

- **Debilidades:**
    - Código poco optimizado.
    - Posibilidad de mejores resultados con más experiencia en programación gráfica.

- **Amenazas:**
    - Sistemas de partículas más potentes y mejor optimizados ya en el mercado.

   - Tiempos de entrega.

- **Fortalezas:**
   - Interesante comparación de dos sistemas de partículas diferentes.
   - Investigación y desarrollo de diferentes técnicas y conocimientos para cada uno de los sistemas.

- **Oportunidades:**
   - Posible uso en motores “indies” por su modularidad.
   - Gran capacidad de evolución y uso en distintos ámbitos.

3.3. Riesgos y plan de contingencia

- **Implementaciones innecesarias**

Los sistemas de partículas que se han analizado para este proyecto (Niagara, Unity) contienen muchas funcionalidades y opciones para poder personalizar y de esta manera conseguir resultados increíbles.

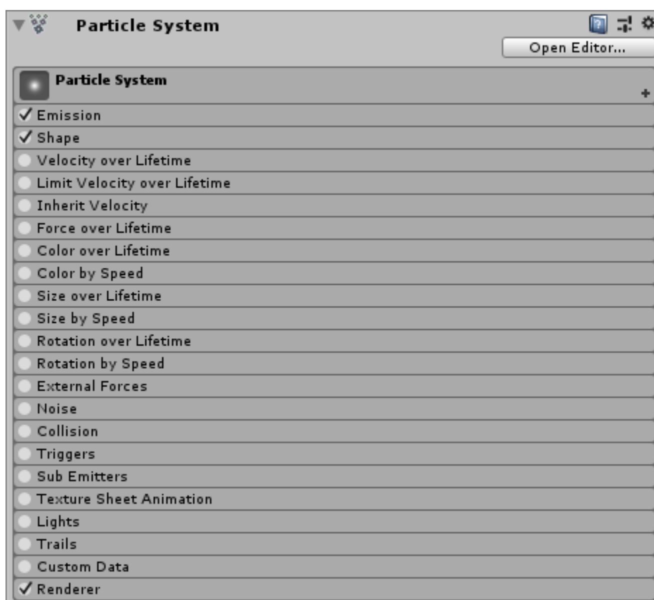


Figura 21 Lista de opciones de Unity Particle System.

El riesgo que tiene este aspecto para el proyecto en general es perder tiempo en intentar implementar funcionalidades innecesarias o que realmente no añaden ningún valor a los sistemas y en consecuencia no dedicar lo suficiente a las funcionalidades y opciones que realmente son básicas y que aportan un valor.

La solución a este posible problema es previamente hacer un análisis de todas las opciones que contienen cada uno de los sistemas, así posteriormente seleccionar cuáles son las funcionalidades que realmente son interesantes de implementar y cuáles no aportan nada, la clave es saber seleccionar las funciones más importantes de cada uno de los sistemas.

- **Falta de tiempo**

Uno de los riesgos evidentes y principales a la hora de asumir un proyecto que se debe entregar en un periodo limitado de tiempo es el hecho de saber administrarse el tiempo y saber dedicarle el tiempo indicado a cada punto.

Este problema está incluido en el punto anterior, pero es un problema también a nivel general, el hecho de no saber cuánto tiempo dedicarle a cada aspecto del proyecto, no solo en el aspecto práctico del proyecto sino también en el apartado escrito.

Una posible solución a este problema es utilizar como ya se ha comentado antes el software Trello, este programa permite fragmentar todo el proyecto en tareas individuales y de esta manera tener una visión más general del proyecto y ser capaz de invertir el tiempo indicado a cada tarea, una acción que se podría tomar para intentar mitigar este problema a lo largo de todo el desarrollo es mantener una comunicación y monitoreo constante con el profesorado para recibir feedback y saber que se le está dedicando el tiempo correcto a los puntos y tareas correctas.

- **Falta de conocimiento en el área**

Un riesgo evidente en este tipo de trabajos es enfrentarse a métodos de trabajo, software, entornos… desconocidos, esto puede provocar frustración y sobre todo una gran pérdida de tiempo intentando entender programas a los que no se está acostumbrado, además de esto también se puede dar el caso de encontrarse con una brecha de conocimiento sobre algún tema o punto en específico del proyecto y dedicarle demasiado tiempo a intentar entender ese concepto para luego dedicarle más tiempo a intentar implementarlo en el código.

Como solución a este problema se ha decidido trabajar en Visual Studio sin programas secundarios, trabajar en un entorno conocido y con el que sentirse cómodo hace que el desarrollo del trabajo sea mucho más ágil y efectivo que si se hiciera en programas o entornos desconocidos, respecto a los posibles conocimientos que puedan ser complicados de entender y que haya una brecha de conocimiento clara, se valorará en cada caso si es conveniente dedicarle el tiempo que requiere y en ese caso ayudarse

del profesorado de la universidad para intentar avanzar más rápido y de manera segura.

- **No alcanzar los objetivos visuales y técnicos deseados**

Uno de los mayores riesgos que se pueden esperar de este tipo de trabajos que combinan resultados visualmente atractivos con una base técnica sólida es no llegar a cumplir las expectativas tanto de un lado como del otro.

Una solución a este problema es establecer claramente y de manera realista los resultados a los que se espera llegar y asegurarse de que finalmente el resultado se ajuste a lo previamente establecido, si finalmente el resultado tanto técnico como visual no llega a las expectativas habrá que hacer un análisis de cuál es la raíz del problema, si es un fallo de diseño o un problema de programación y actuar en consecuencia.

3.4. Análisis inicial de costos

En este apartado se analizarán los gastos económicos que supondría este proyecto, suponiendo que se hiciera de manera profesional.

Este aparato se ha dividido en dos apartados principales, gastos de software y gastos de recursos humanos, cada uno de ellos tiene una tabla explicativa de los gastos aproximados que supondría enfrentarse a un proyecto de estas características, el gasto total del proyecto será la suma del total de cada una de las tablas.

En el apartado de recursos humanos tenemos una tabla que se dividirá por las diferentes etapas del proyecto, dentro de cada etapa se definirán las diferentes tareas que se deberán realizar, a cada una de estas tareas se les asignará una cantidad aproximada de horas que tomara realizarlas, estableciendo una cantidad de dinero por hora de trabajo tendremos un aproximado del sueldo que se debería pagar.

Euros por hora trabajada	8	
Preproduccion	Estimado de horas invertidas	Coste total
investigacion de la tecnica del instanciado	15	120
Investigacion de la tecnica de ray tracing	35	280
Investigacion de los diferentes sistemas de particulas	5	40
Seleccion de las caracteristicas mas importantes que debe tener un sistema	2	16
Desarrollo del documento escrito	15	120
Investigacion de como implementar la fisica en cada uno de los sistemas	25	200
Total	97	776
Produccion	Estimado de horas invertidas	Coste total
Implementacion de OpenGL	2	16
Implementacion de Devil	3	24
Implementacion de Imgui	2	16
Implementacion del instaciado	30	240
Implementacion de ray tracing	40	320
Desarrollo de la base de los dos sistemas	20	160
Implementacion de todas las funcionalidades de los dos sistemas	25	200
Desarrollo del documento escrito	20	160
Total	142	1136
Postproduccion	Estimado de horas invertidas	Coste total
Desarrollo del documento escrito	10	80
Comparacion de rendimiento de los dos sistemas	3	24
Analisis de resultados y conclusiones	5	40
Total	18	144
Total del Proyecto	257	2056

Tabla 2: Estimado de costes según las horas trabajadas.

En el apartado de software se incluirá el costo de cualquier programa externo que se deba pagar, también se incluirá el costo del equipo utilizado para realizar el proyecto, suponiendo que no se contaba con él al comenzar el proyecto y que se ha tenido que adquirir, estos costes totales son suponiendo que la duración del proyecto es de un aproximado de 5 meses.

Equipo	Costo
Silla	120
Portatil	2000
Mesa	90
Raton	30
Otros	10
Total	2250
Software	Costo
Trello	0
Github	0
Visual Studio	0
OpenGL	0
Devil	0
Imgui	0
Total	0
Gastos indirectos	Costo
Comida	300
Agua	30
Electricidad	40
Total	370
Coste Total	2620

Tabla 3: Estimado de costes de software y equipo complementario.

Teniendo en cuenta el resultado de cada una de las tablas, podemos afirmar que el proyecto completo tendrá un costo aproximado de **4.676 €.**

3.5. Gantt

Para este proyecto se van a dividir los objetivos del desarrollo por las mismas entregas de la universidad que están establecidas en el campus, en total la universidad contempla tres entregas totales, utilizaremos las fechas de esas tres entregas para determinar los objetivos.

- Entrega 1:

  Para esta entrega el objetivo se basa en desarrollar el documento escrito para determinar los objetivos generales del proyecto y organizar toda la carga de trabajo, durante este periodo también se comenzará con el desarrollo implementando librerías.

- Entrega 2:

  Los objetivos para esta entrega serán establecer las bases de los dos simuladores, para esta entrega se espera tener los primeros resultados aunque

no sean los finales, el objetivo principal es lograr una buena base con la que poder continuar más adelante.

- Entrega 3:

  Para este punto del desarrollo sé esperan los resultados finales de ambos sistemas, con resultados visuales pulidos y un análisis de la comparación de los dos sistemas para determinar las fortalezas y debilidades de cada uno de ellos.

4. Metodología

Para el desarrollo de este trabajo se ha decidido dividir la metodología en tres bloques principales y bien diferenciados, cabe destacar que el documento escrito que se debe entregar se irá desarrollando a lo largo de todas las etapas debido a las diferentes entregas que están establecidas, este documento también podrá ir cambiando a lo largo del tiempo por diferentes cambios que puedan suceder durante todo el desarrollo.

- **Fase 1**

La fase 1 se podría también denominar como preproducción, en esta fase el objetivo será hacer un trabajo de investigación de los diferentes sistemas de partículas que existen en el mercado y ver las características principales que comparten todas ellas para implementarlas en el proyecto y de esta manera también ver en qué características o funcionalidades se encuentran las diferencias principales entre ellas, de esta manera hacer un análisis de qué es lo que funciona mejor para poder implementarlo en el proyecto.

Una vez hecho este análisis de los principales sistemas de partículas también se investigará el funcionamiento exacto de cada una de las funcionalidades que contienen los sistemas, por poner un ejemplo, Los sistemas de partículas utilizan el instanciado para poder mostrar más objetos por pantalla, esa es una característica que comparten todos ellos, ahora hay que investigar cómo se implementa esta técnica en cuanto a código y cómo se gestiona esto realmente con OpenGL.

En resumen esta fase se trata de investigar las características de cada uno de los sistemas a nivel general comparándolos entre ellos y también investigar esas mismas características de manera más profunda (código, OpenGL) para conocer cómo implementarlas, esto se hará para los dos sistemas de partículas que se pretenden desarrollar.

- **Fase 2**

La fase 2 entraría en el campo de la producción en sí, aquí comienza el desarrollo del proyecto, en esta se creará el propio proyecto en Visual Studio y se comenzarán a introducir todas las librerías externas necesarias para el desarrollo del proyecto, entre las que se encuentran OpenGL para la creación de gráficos en 3D y desarrollo de *shaders*, Devil para la carga y gestión de imágenes o ImGui para la creación de la UI para el propio uso de los sistemas.

En esta etapa se hará uso de los conocimientos que se han adquirido en la primera fase, se irán implementando todas las funciones, fórmulas y algoritmos que sean necesarios para el desarrollo de los dos sistemas.

Al final de esta fase se deberían tener dos sistemas de partículas diferentes y completamente funcionales para su posterior análisis en rendimiento y acabado visual.

- **Fase 3**

La fase 3 será la de postproducción, esta fase se centrará en analizar y comparar el rendimiento de cada uno de los sistemas, de esta manera se podrá determinar cuál es el mejor para proyectos *in real time* y cuál es más indicado para otro tipo de proyectos.

El análisis no se basa únicamente en el rendimiento de los sistemas, también se basa en el resultado visual de cada uno de ellos y de esta manera saber cuál es más apto para cada tipo de proyecto o resultado que se espera.

5. Desarrollo del proyecto

En el siguiente apartado se mostrarán y explicarán las diferentes decisiones que se han tomado para el desarrollo del proyecto, con el objetivo de desarrollar dos sistemas de partículas para su posterior comparación y estudio, para cumplir esta meta se han tenido que desarrollar diferentes técnicas y métodos para cada uno de los sistemas, por lo tanto, este apartado se ha dividido en dos secciones, uno por cada sistema y en cada apartado se ha explicado las diferentes decisiones y metodologías seguidas en cada uno de ellos.

5.1 Herramientas externas

Para el desarrollo del proyecto ha sido necesaria la utilización de diferentes librerías externas como OpenGL, Devil e Imgui, a continuación se explicara el motivo por el que se ha incluido en el proyecto, además de una breve explicación de las funciones que se han utilizado de cada una de ellas y las ventajas que nos ofrecen.

5.1.1 OpenGL

OpenGL es una librería gráfica que nos permite crear gráficos tanto en 2D como en 3D para el desarrollo de videojuegos y aplicaciones, esta es multiplataforma, por lo que puede usarse para el desarrollo en diferentes sistemas operativos y dispositivos.

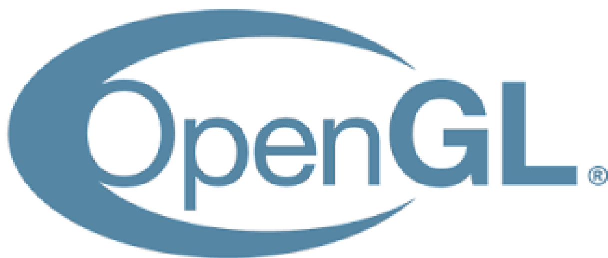


Figura 22 Logo de la librería gráfica OpenGL.

Existen varios motivos por los que se ha seleccionado esta librería gráfica para este proyecto, a continuación se muestran algunos de ellos.

- Amplio conocimiento previo de su uso gracias a las asignaturas de motores y proyecto 3 de la universidad.
- Renderización fácil y eficiente gracias al uso de la tarjeta gráfica que hace por defecto OpenGL.
- Uso e integración fácil de shaders en los programas que nos permite tener control y flexibilidad sobre el renderizado, también conseguimos una mejora de rendimiento gracias a que el proceso de datos se ejecuta en la GPU paralela a la CPU, también conseguimos efectos visuales avanzados como sombras suaves o iluminación realista que de otra manera sería difícil de conseguir.

- Gran cantidad de documentación y de usuarios para solucionar problemas o dudas que puedan ir surgiendo.
- Gran capacidad de adaptación y de flexibilidad de la librería que permite adaptarse perfectamente a cada proyecto y a sus necesidades.

Esta librería solamente contiene unos archivos .lib y unos .h que simplemente se deben incluir en el proyecto y posteriormente enlazar la librería y el proyecto, una vez hecho esto ya se podrá empezar a utilizar.

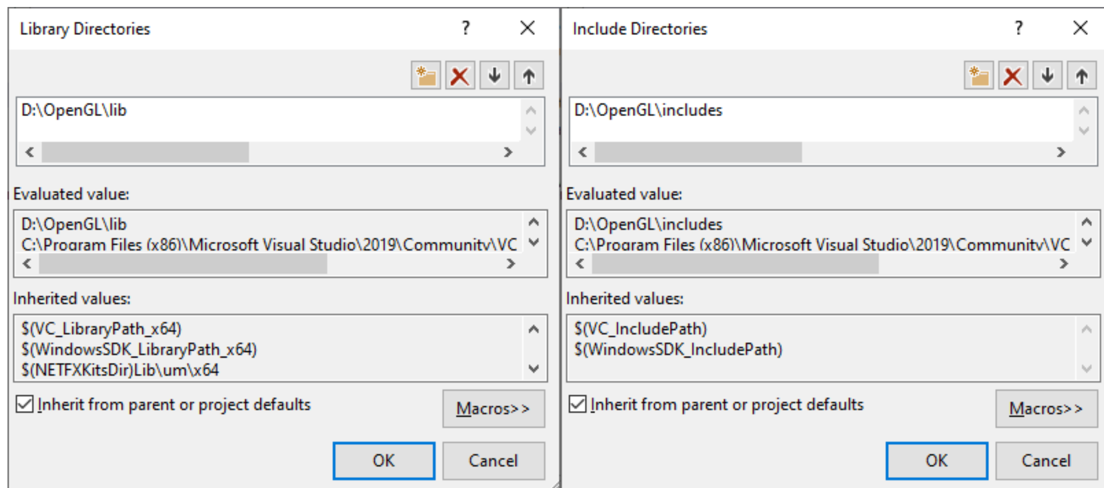


Figura 23 Ventana de vinculación de librerías de Visual Studio.

5.1.2 Devil

Devil es una librería que ofrece distintas funciones que hacen muy sencilla la carga, manipulación y el guardado de imágenes en el desarrollo de videojuegos o de diferentes aplicaciones, esta librería es compatible con una gran variedad de formatos de imágenes y es muy fácil de integrar y utilizar junto a librerías gráficas como OpenGL.

Figura 24 Logo de la librería Devil.

Hay varias razones por las que se ha utilizado esta librería para gestionar el uso de imágenes en el proyecto.

- Es una librería de código abierto que se puede modificar y adaptar para las necesidades de cada proyecto.
- Ofrece funciones de carga de imágenes muy sencillas de utilizar y con soporte para una gran variedad de formatos, y como ya se ha comentado antes, la integración con OpenGL es muy sencilla.
- Ofrece también funciones básicas de edición de imágenes como corrección de gamma, ruido o ecualización de imagen, entre otras.

Gamma Correction

iluGammaCorrect applies gamma correction to an image using an exponential curve. The single parameter **iluGammaCorrect** accepts is the gamma correction factor you wish to use. A gamma correction factor of 1.0 leaves the image unmodified. Values in the range 0.0 - 1.0 darken the image. 0.0 leaves a totally black image. Anything above 1.0 brightens the image, but values too large may saturate the image.

Figure 4-9. *Result of gamma correction of 0.5*

Figure 4-10. *Result of gamma correction of 1.9*

Negativity

iluNegative is a very basic function that inverts every pixel's colour in an image. For example, pure white becomes pure black, and vice-versa. The resulting colour of a pixel can be determined by this formula: new_colour = ~old_colour (where the tilde is the negation of the set of bits). **iluNegative** does not accept any parameters and is reversible by calling it again.

Figure 4-11. *iluNegative example*

Figura 25 Ejemplo de modificaciones en texturas de Devil.

El uso de esta librería he permitido integrar en poco tiempo y de manera efectiva la función de carga y edición de imágenes que es vital para el desarrollo del proyecto.

5.1.3 Imgui

Imgui es una librería que permite crear interfaces de usuario de una forma muy sencilla, la librería ofrece mostrar botones, texto, despegables… entre muchas otras opciones que permite a los desarrolladores crear interfaces de usuario personalizadas.

Figura 26 Logo de la librería Imgui.

Algunas de las razones por las que se ha decidido utilizar Imgui para este proyecto son.

- Integración y uso sencillo con apis como OpenGL.
- Gran variedad de widgets gráficos como botones, casillas de verificación, barras de progreso, listas desplegables y campos de texto.
- Gran capacidad de personalización de la interfaz para cada proyecto.

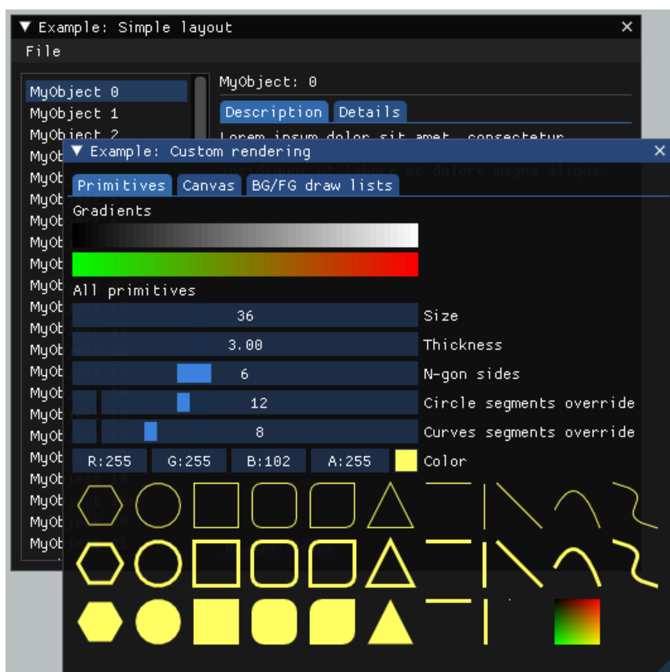


Figura 27 Ejemplo de interfaz de usuario con Imgui.

El uso de esta librería en el proyecto ha permitido crear de manera muy sencilla una interfaz de usuario que no solo es útil para el uso de la aplicación en sí, sino también es una herramienta muy potente que permite depurar la aplicación de manera muy sencilla y rápida, ahorrando mucho tiempo en el desarrollo de la misma.

5.2 Sistema de partículas complejo (simulación realista)

Para este sistema de partículas se prioriza sobre todo la velocidad del cálculo de las físicas, para este caso el método de guardar los datos y procesarlos es completamente diferente, pues se va a utilizar un método de renderizado completamente diferente.

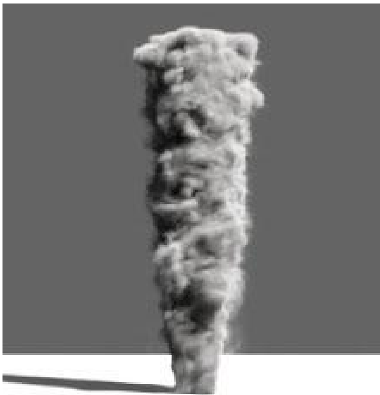
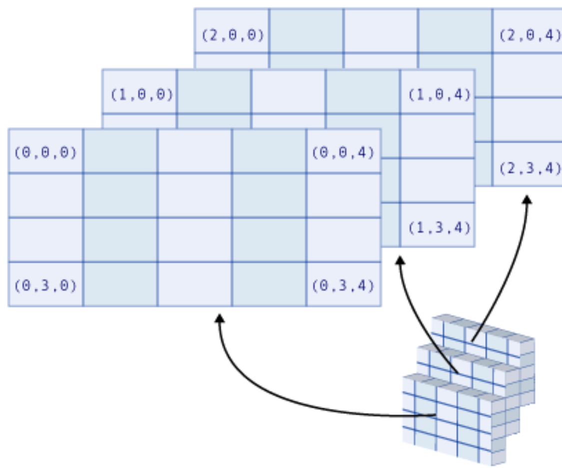


Figura 28 Resultado de renderizado realista.

Como primer paso debemos limitar el espacio a un cubo, es decir, todas las simulaciones tendrán lugar únicamente en ese espacio previamente definido, una vez definido ese espacio debemos dividir ese cubo en cubos más pequeños dentro de él, simplemente, debemos dividir el largo, ancho y alto por un valor, cuanto más alto sea ese valor, más cubos tendremos dentro de ese espacio y, por tanto, tendremos más resolución y resultados con más calidad, pero eso también significa muchos más cubos internos de los que posteriormente tendremos que calcular la física y por último el renderizado final.

Todos los valores de densidad los tendremos guardados en un simple *array* unidimensional, pero los cálculos tanto de física, como para el renderizado los tendremos que calcular como si fuera un cubo 3D, es decir, debemos buscar una manera de pasar posiciones 3D a un índice que pasarle al *array* unidimensional para acceder a la posición deseada.




```
indice = x + ancho * y + ancho * alto * z
```

Figura 29 Fórmula y explicación gráfica para pasar valores de entorno 3D a array 2D

Como se puede ver en la imagen superior, el cubo en sí podemos separarlo por capas, tendiendo en cuanta que tenemos los valores de alto y ancho del cubo, podemos utilizar la posición 3D para multiplicarlo por esos valores y de esta manera conseguir una posición hipotética en un *array* unidimensional de manera fácil y rápida, esta fórmula será muy útil durante todo el código, tanto para el cálculo de físicas como para hacer los calculo del renderizado en el *shader*.

Una vez definido como vamos a almacenar los datos densidad y como vamos a acceder a ellos, toca hacer la simulación de fluidos para calcular como interactúan las partículas

```
void FluidCube::FluidCubeStep()
{
    int N = size;
    float visc = this->visc;
    float diff = this->diff;
    float dt = this->dt;
    float* Vx = this->Vx;
    float* Vy = this->Vy;
    float* Vz = this->Vz;
    float* Vx0 = this->Vx0;
    float* Vy0 = this->Vy0;
    float* Vz0 = this->Vz0;
    float* s = this->s;
    float* density = this->density;

    if (state == simulationState::CACHING) {
        dt = realFramerate;
    }
    diffuse(1, Vx0, Vx, visc, dt, 1, N);
    diffuse(2, Vy0, Vy, visc, dt, 1, N);
    diffuse(3, Vz0, Vz, visc, dt, 1, N);

    project(Vx0, Vy0, Vz0, Vx, Vy, 1, N);

    advect(1, Vx, Vx0, Vx0, Vy0, Vz0, dt, N);
    advect(2, Vy, Vy0, Vx0, Vy0, Vz0, dt, N);
    advect(3, Vz, Vz0, Vx0, Vy0, Vz0, dt, N);

    project(Vx, Vy, Vz, Vx0, Vy0, 1, N);

    diffuse(0, s, density, diff, dt, 1, N);
    advect(0, density, s, Vx, Vy, Vz, dt, N);
}
```

Figura 30 Codigo para cada paso de simulación del fluido.

A cada *frame* se llamará al *update* que a su vez llamara a la función “FluidCubeStep()”, aquí se llamaran a todas las funciones necesarias para realizar la simulación, estas funciones simplemente requieren del *delta Time*, de las velocidades en los distintos ejes, la cantidad de interacciones que se desean para más o menos realista y el tamaño del cubo en el que se realizara la simulación, representado con la variable “N”

A continuación se mostrará con más detalle el proceso en sí de una simulación de fluidos realista.

Como ya se ha comentado antes, para hacer la simulación el espacio se debe dividir en pequeñas celdas o espacios, y a cada una de ellas se le asigna un valor de velocidad, densidad, velocidad y presión. A partir de estos valores iniciales se aplican las ecuaciones de “Navier-Strokes”, gracias a estas ecuaciones se pueden calcular en comportamiento de cada celda en un intervalo de tiempo, encadenando estos cálculos a lo largo del tiempo se pueden conseguir comportamientos realistas de un fluido, para realizar estos cálculos en este trabajo se han utilizado las ecuaciones de “Mike Ash”, que se basan en principalmente en las ecuaciones previamente mencionadas.

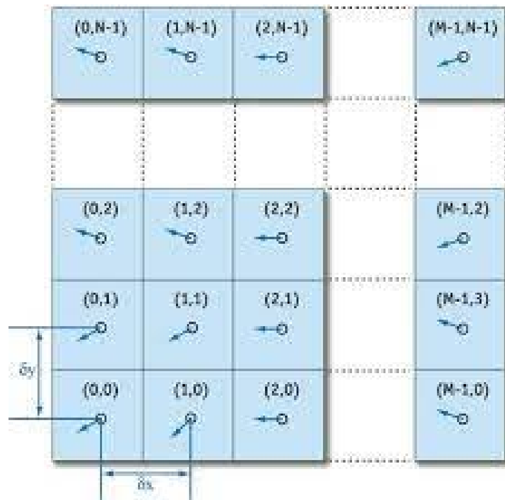


Figura 31 Descripción gráfica de los cálculos de simulación de fluidos.

Las ecuaciones de Mike Ash incluyen, entre otros, el algoritmo de proyección de presión y el método de suavizado de partículas (SPH). El algoritmo de proyección de presión es un enfoque común que utiliza la ecuación de Navier-Stokes para calcular la velocidad y la presión del fluido en cada celda. El método SPH, por otro lado, utiliza partículas individuales para representar el fluido y calcula la interacción entre ellas para simular el movimiento del fluido.

Una vez tenemos los valores de densidad final guardados en el *array*, debemos renderizar, para ello podemos pasar el *array* de valores de densidad como textura 3D al *shader*. La técnica de renderizado utilizada para este caso es el *Ray Tracing*, que consiste en lanzar rayos y recibir información de como interactúa con el entorno.

En los casos en los que los objetos de la escena son completamente sólidos y opacos, es sencillo determinar el rebote del rayo y obtener la información, en el caso del humo no es posible hacer eso, en su lugar debemos afrontar el problema de otra manera, la manera de proceder en este caso es lanzar un rayo que atraviese el volumen al completo e ir acumulando los valores de densidad a lo largo de todo el rayo, es decir recorrer el rayo de atrás hacia adelante y mediante las posiciones ir extrayendo la información de densidad en cada punto que se analice, finalmente con cada rayo se obtendrá un valor de densidad, debajo se adjunta una imágen que muestran como se realiza este proceso.

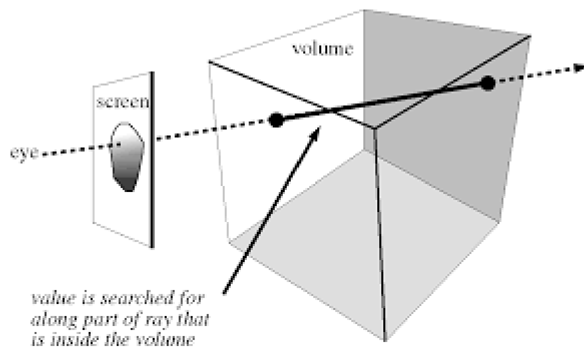


Figura 32 Explicación gráfica de rayo atravesando densidad para recoger valores.

A continuación se explicará brevemente el *shader* utilizado para renderizar la densidad por pantalla, este shader implementa la función básica de ir lanzando rayos, e ir recogiendo los valores de densidad para mostrarlos por pantalla explicada anteriormente, en la imagen inferior se puede observar el bucle que hace esa acción.

```
vec3 shading(vec3 cameraPosition, vec3 rd) {

     vec3 ld = normalize(vec3(-1, -1, 0));

    const float nbStep = 30., diam = 3., rayLength = diam / nbStep;
    float start = length(cameraPosition) - diam / 2., end = start + diam;
    float sumDen = 0., sumDif = 0.;

    for(float d = end; d > start; d -= rayLength) { // raymarching
        vec3 p = cameraPosition + d * rd;
      if(dot(p,p) > diam * diam) break;
        float den = texture(texture1,p).r;
        sumDen += den;
        sumDif += max(0., den - texture(texture1,p + ld * .17).r);
    }

    const vec3 lightCol = vec3(.95, .75, .3);
    float light = 10. * pow(max(0., dot(rd, ld)), 5.);
    vec3 col = .01 * light * lightCol;
    col +=  .4 * sumDen * rayLength * vec3(.8, .9, 1.); // ambient
    col += 1.3 * sumDif * rayLength * lightCol;         // diffuse
      return col;
}
```

Figura 33 Código de *fragment shader* para el renderizado de densidad.

Esta implementación tan simple con unos cálculos de luz muy sencillos podemos obtener los primeros resultados que se ven en la imagen de abajo, de momento quedan muchos aspectos a mejorar en el *shader* para conseguir resultados mejor logrados, pero la base del *shader* y su funcionamiento está implementado.

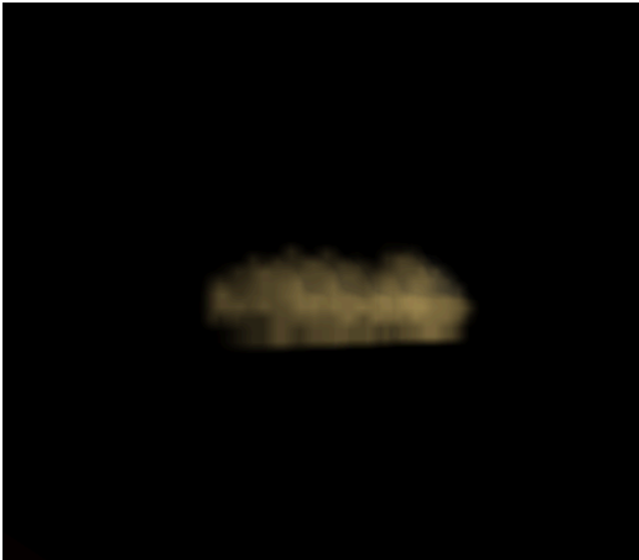


Figura 34 Resultado temporal del renderizado de humo.

Una vez tengamos toda la información debemos dibujarlo como textura en un plano que colocaremos justo delante de la cámara, para ello no debemos multiplicar la posición por la matriz de proyección ni por la matriz de *view*, pues nos interesa que el plano este constantemente delante de la cámara, así que simplemente usaremos la posición de los vértices como se muestra en el *vertex shader* de abajo, la técnica de colocar un plano delante de la cámara donde se renderice toda la escena es conocida como *deferred shading*.

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoord;

out vec2 TexCoord;
uniform mat4 view;
uniform mat4 projection;
uniform mat4 transform;


void main()
{
      gl_Position = vec4(aPos, 1.0);
}
```

Figura 35 Vertex shader del plano de renderizado final.

5.2.1 Formas de optimización

Para este simulador se ha utilizado principalmente una técnica de optimización, esta técnica permite ahorrar mucho tiempo de cálculo de física logrando los mismos resultados visuales, consiguiendo así una mejor nota en el rendimiento en general.

La técnica se basa principalmente en hacer la simulación previamente y guardar los resultados de la simulación en archivos secundarios, de esta manera en la ejecución del juego solo se deben leer los archivos con la información de la simulación, mediante este sistema en el juego en sí no se debe calcular nada, solo mostrarlo por pantalla.

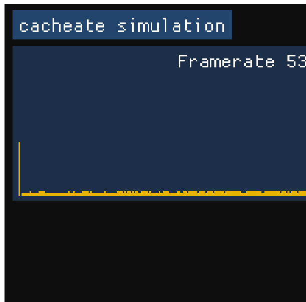


Figura 36 Botón de cacheado del simulador

En el simulador se ha ubicado un botón para cachear la simulación como se puede ver en la imagen de arriba, el sistema ofrece diferentes opciones para guardar la simulación como se puede ver en la imagen inferior.

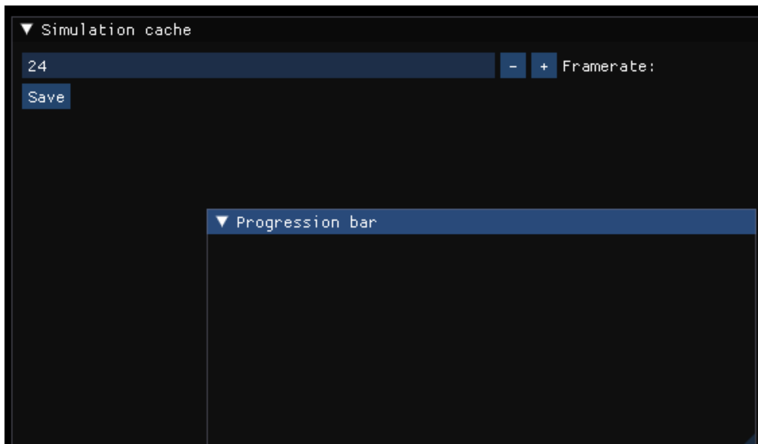


Figura 37 Ventana de configuración del guardado

5.3 Sistema de partículas simple (alto rendimiento)

Este sistema de partículas está más enfocado a poder utilizarse como los sistemas de partículas de Unity, que pueden usarse para poder realizar todo tipo de efectos, en este caso nos interesa incluir una gran cantidad de opciones para que el usuario pueda personalizarlo y conseguir el efecto que desea.

Para este caso, en comparación al sistema de partículas complejo, se tratará cada partícula de forma individual, la forma de afrontar este sistema es diferente al anterior,

al principio de todo se generarán y cargarán todas las partículas en memoria y se iniciarán todas sus variables, esto se hace para no crear y destruir partículas en plena ejecución para evitar problemas de velocidad y de accesos a memoria y de esta manera hacer el programa más seguro y rápido.

```
position = { 0,0,0 };
maxParticles = lastMaxParticles = 200;
maxParticleLifeTime = 5;
spawnPosition = { 0,0,0 };
spawnAceleration = { 0,0,0 };
spawnVelocity = { 0,1,0 };

textureAnimation = new GeneralTextureAnimation();

physicsModule = new Physics();
physicsModule->SetParticlesInfo(&activeParticles);
for (int a = 0; a < maxParticles; a++) {

    addParticle();
}
```

Figura 38 Start del simulador.

En este caso el sistema se divide en tres grandes funciones: PreUpdate, *Update* y PostUpdate. Cada una de estas funciones se centra en una función específica,

El *PreUpdate* se encarga de sacar de la lista las partículas que ya no se estén utilizando, es decir, por ejemplo cuando una de las partículas ya ha superado el tiempo de vida se marca que no está activa y es en *PreUpdate* donde se detecta cada una de estas partículas y se elimina de la lista, para que de esta manera no se procese ni se renderice posteriormente.

En esta función también se hace el proceso inverso, en las opciones del usuario se encuentra una frecuencia de aparición, es decir que el usuario puede indicar que por ejemplo aparezcan 3 partículas por segundo, esta función comprueba si ya es la hora de activar nuevas partículas, para hacerlo lee las especificaciones que el usuario ha determinado como velocidad inicial, color, tamaño…(veremos todas las opciones más adelante) la función inicializa la partícula con esas especificaciones e indica que está activa y se incluye en la lista, de esta manera en los siguientes procesos se podrá calcular y renderizar

```
auxiliar->GetParticleTexture()->SetActualTextureID(fixedIDToSet);

//Position
auxiliar->SetParticleInitialPosition(spawnPosition + position);
auxiliar->SetParticlePosition(spawnPosition + position);

//Rotation
auxiliar->SetParticleAngleRotation(auxiliarRotationAngle);
auxiliar->SetParticleAxisRotation(auxiliarRotationAxis);
auxiliar->SetParticleFrontVector({ 0,0,1 });

//velocity
auxiliar->SetParticleInitialVelocity(spawnVelocity);
auxiliar->SetParticleVelocity(spawnVelocity);

//aceleration
auxiliar->SetParticleAceleration(spawnAceleration);
//scale
auxiliar->SetParticleScale(spawnScale);
//lifeTime
auxiliar->setParticleMaxLifeTime(maxParticleLifeTime);
auxiliar->setParticleActualLifeTime(0);
//Physics
auxiliar->SetParticleMass(fixedMass);
auxiliar->SetParticleRadius(fixedRadius);
```

Figura 39 Código principal del *PreUpdate* del simulador.

En el caso del *Update* se centra en actualizar todos los parámetros posibles de la partícula, es decir, actualiza la física de esta, el color, el tamaño, la textura que debe mostrar, etc. Esta función recorre la lista de partículas activas y actualiza cada apartado por orden, primero comprueba si la partícula ha superado el tiempo de vida que se le ha asignado, si es así se marca como no activa, y si no, se actualiza de forma normal, primero de todo se actualiza la física, es decir, teniendo en cuenta la aceleración se calcula la velocidad y en función de esto la nueva posición de la partícula, posteriormente se calculan todos los demás parámetros de escala, color y textura en función de la configuración del usuario.

```
void Emitter::Update(float dt)
{
    if (actualPartition != 0) {
        for (std::list<Emitter*>::iterator it = SecondEmitters.begin(); it != SecondEmitters.end(); it++)
        {
            it._Ptr->_Myval->Update(dt);
        }
    }
    for (std::list<particle*>::iterator it = activeParticles.begin(); it != activeParticles.end(); it++)
    {
        UpdateSecondariEmitters(0, it._Ptr->_Myval);
        it._Ptr->_Myval->addParticleActualLifeTime(dt);
        UpdateSecondariEmitters(1, it._Ptr->_Myval);
        if (it._Ptr->_Myval->GetParticleMaxLifeTime() > it._Ptr->_Myval->GetParticleActualLifeTime())
        {

            physicsModule->Update(it._Ptr->_Myval,dt, it);
            UpdateRotation(it._Ptr->_Myval, dt);


            UpdatePhysics(it._Ptr->_Myval, dt);
            UpdateTransformEfects(it._Ptr->_Myval, dt);


            UpdateScale(it._Ptr->_Myval);
            UpdateAlphaAndColor(it._Ptr->_Myval);

            UpdateTextures(it._Ptr->_Myval,dt);

        }
        else
        {
            it._Ptr->_Myval->setIsActive(false);
        }
    }
}
```

Figura 40 Código principal del *Update* del simulador.

Otros ejemplos de apartados para actualizar son la escala, el color y las texturas, en el caso de las texturas existen diferentes opciones que el usuario pude escoger, entre ellas se encuentra el poder hacer crear una animación con diferentes imágenes, esta función se encargaría de actualizar las imágenes para cada partícula en función de la animación que se haya especificado, para esto simplemente se debe cambiar el ID de la imagen que se desea renderizar para más tarde en el PostUpdate bindear la imagen correcta para cada imagen.

Por último tenemos el *PostUpdate* esta función únicamente se centra en renderizar todas y cada una de las partículas, en esta función se pasarán al *shader* todas las opciones de color, tamaño, posición para ser renderizado.

```
void Emitter::PostUpdate(float dt) {
    if (actualPartition != 0) {
        for (std::list<Emitter*>::iterator it = SecondEmitters.begin(); it != SecondEmitters.end(); it++)
        {
            it._Ptr->_Myval->PostUpdate(dt);
        }
    }
    for (std::list<particle*>::iterator it = activeParticles.begin(); it != activeParticles.end(); it++)
    {
        Draw(it._Ptr->_Myval);
    }
    physicsModule->PostUpdate();
}
```

Figura 41 Código principal del *PostUpdate* del simulador.

A continuación se mostrarán todas las opciones que el sistema ofrece para poder personalizar cada efecto.

La primera pestaña principal se trata de la configuración general del sistema, este apartado se compone de las siguientes opciones:

- 3 opciones de comenzar, parar y resetear el sistema.
- Texto informativo de la cantidad de partículas que están activas.
- Opción para que el sistema se esté ejecutando en bucle o no.
- Cantidad de partículas que deben aparecer por cada segundo.
- Opción para que las partículas estén constantemente de cara a la cámara o no.
- Cantidad de partículas que se han cargado en memoria.
- Distintas opciones para establecer el tiempo de vida de cada una de las partículas.
    - Opción para establecer un tiempo de vida fijo para todas las partículas.
    - Opción para establecer un tiempo aleatorio a cada partícula siempre dentro de dos valores.
- Distintas opciones para la aparición de las partículas.
    - Opción de aparición "custom", el usuario puede personalizarlo a su gusto.
    - Opción de aparición "cone", las partículas aparecerán subiendo hacia arriba de forma cónica.
    - Opción de aparición "sphere", las partículas aparecerán de forma esférica, dirigiéndose hacia todas las direcciones.
- Distintas opciones para poder dividir las partículas llegado un punto.
    - Opción para que el sistema no parta las partículas.
    - Opción para que llegado a un punto las partículas se dividan en otras, generando así un emisor nuevo completamente configurable.

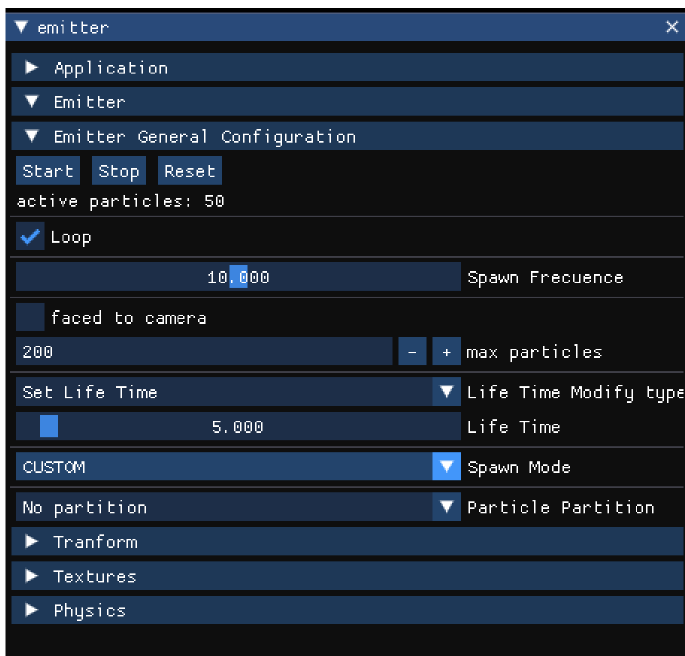


Figura 42 UI de la configuración general del simulador.

La segunda pestaña se trata del “trasnform” de las partículas, esta sección se encarga de establecer la velocidad, aceleración y posición de aparición de esta en el mundo, junto a otras opciones.

- Distintas para establecer la posición de aparición de las partículas.
    - Opción para establecer una posición fija en el entorno 3D.
    - Opción para establecer posiciones aleatorias en los 3 ejes.
- Distintas para establecer la velocidad de aparición de las partículas.
    - Opción para establecer una velocidad fija en el entorno 3D.
    - Opción para establecer velocidades aleatorias en los 3 ejes.
- Distintas para establecer la aceleración de aparición de las partículas.
    - Opción para establecer una aceleración fija en el entorno 3D.
    - Opción para establecer aceleraciones aleatorias en los 3 ejes.
- Tres opciones para establecer la rotación de las partículas.
- Distintas opciones para establecer la masa de las partículas.
    - Opción para establecer una masa fija a todas las partículas.
    - Opción para establecer aleatoriamente la masa de las partículas entre dos valores.
- Distintas opciones para establecer el radio de colisión de las partículas.
    - Opción para establecer un radio de colisión fija a todas las partículas.
    - Opción para establecer aleatoriamente un radio de colisión de las partículas entre dos valores.
- Una opción para poder crear un efecto de vórtice configurable para cada eje
- Distintas opciones para establecer la escala de las partículas.
    - Opción para establecer una escala fija a todas las partículas.
    - Opción para establecer aleatoriamente una escala a las partículas entre dos valores.
    - Opción para personalizar la escala de la partícula a lo largo del tiempo mediante un gráfico y una escala máxima.

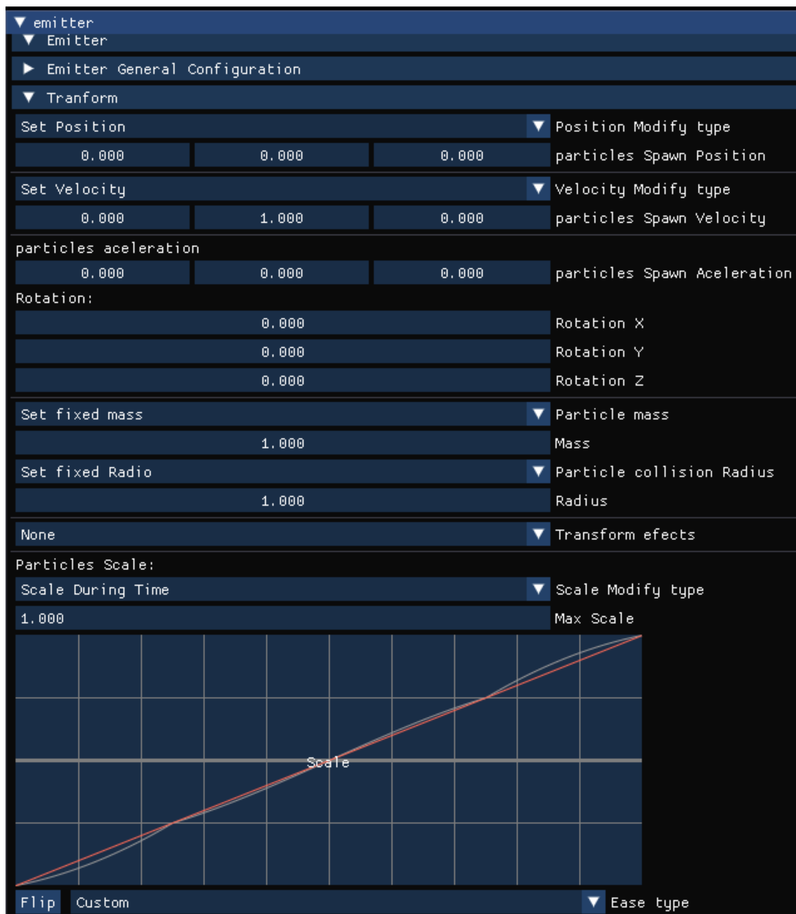


Figura 43 UI de la configuración *transform* del simulador.

La tercera pestaña se trata del apartado de las texturas y del apartado artístico en general.

- Opción para cargar las texturas, el programa accederá a los archivos del ordenador para poder cargar imágenes.
- Opción para ver la información de cada una de las texturas que están cargadas en memoria.
    - Al acceder a la información aparece una ventana con una lista de todas las imágenes cargadas por el programa

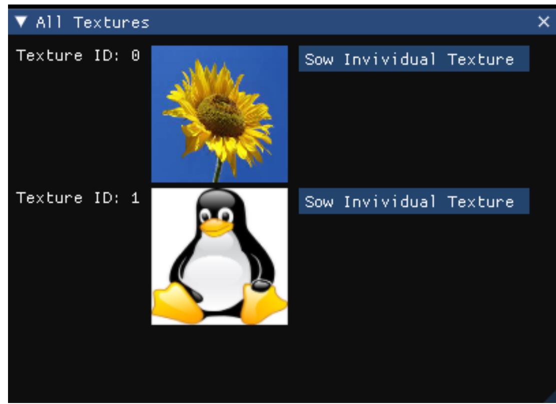


Figura 44 UI de la ventana de texturas del simulador.

- Sí se accede a la opción de mostrar información detallada de la textura, sé mostrará otra ventana independiente con la información detallada de la imagen.

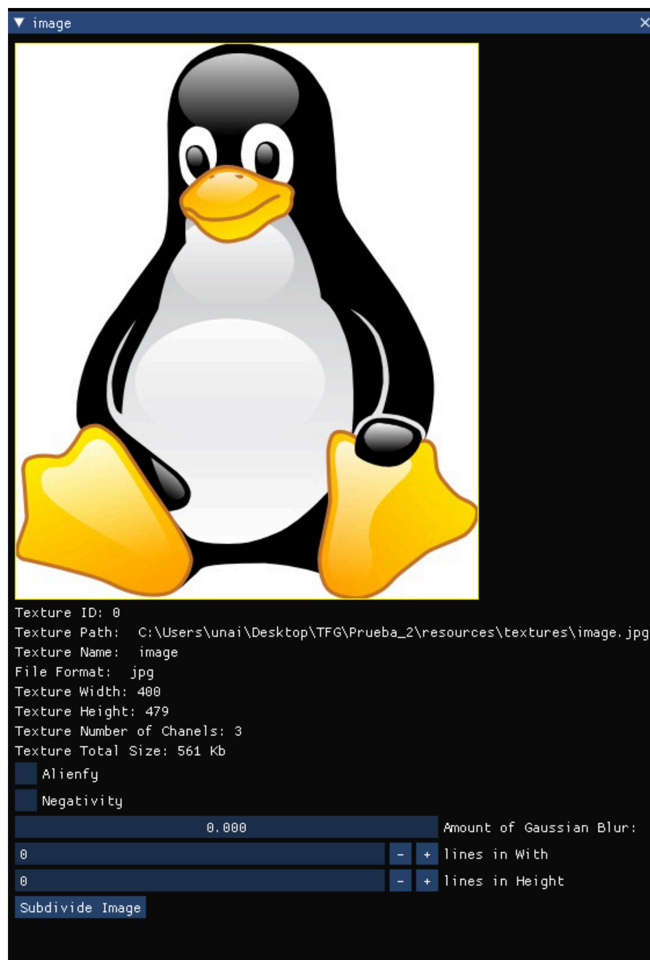


Figura 45 UI de la ventana de configuración y edición de texturas del simulador.

    - En esta ventana se muestra información como el ID, el nombre, el Path, el formato, alto y ancho de la imagen, número de canales y espacio total que ocupa, como opciones extra está la opción de dividir la imagen en varias otras para extraer, por ejemplo, las imágenes independientes de una animación de una sola imagen donde estén todas guardadas.

- Distintas opciones para mostrar las texturas cargadas.
    - Opción para crear animación entre todas las animaciones cargadas o solo entre algunas pocas seleccionadas
    - Opción para mostrar únicamente una textura durante toda la vida de la partícula
    - Opción para mostrar aleatoriamente diferentes texturas.
- Distintas opciones para establecer el Alpha de las texturas.
    - Opción para establecer un Alpha fija durante todo el rato.
    - Opción para establecer un Alpha random de entre dos valores.
    - Opción para personalizar el Alpha de la partícula a lo largo del tiempo mediante un gráfico.
- Distintas opciones para establecer el color de las texturas.
    - Opción para establecer un color fijo durante todo el rato.
    - Opción para establecer un color random.
    - Opción para personalizar el color de la partícula a lo largo del tiempo mediante un gráfico de colores ordenado por tiempo.

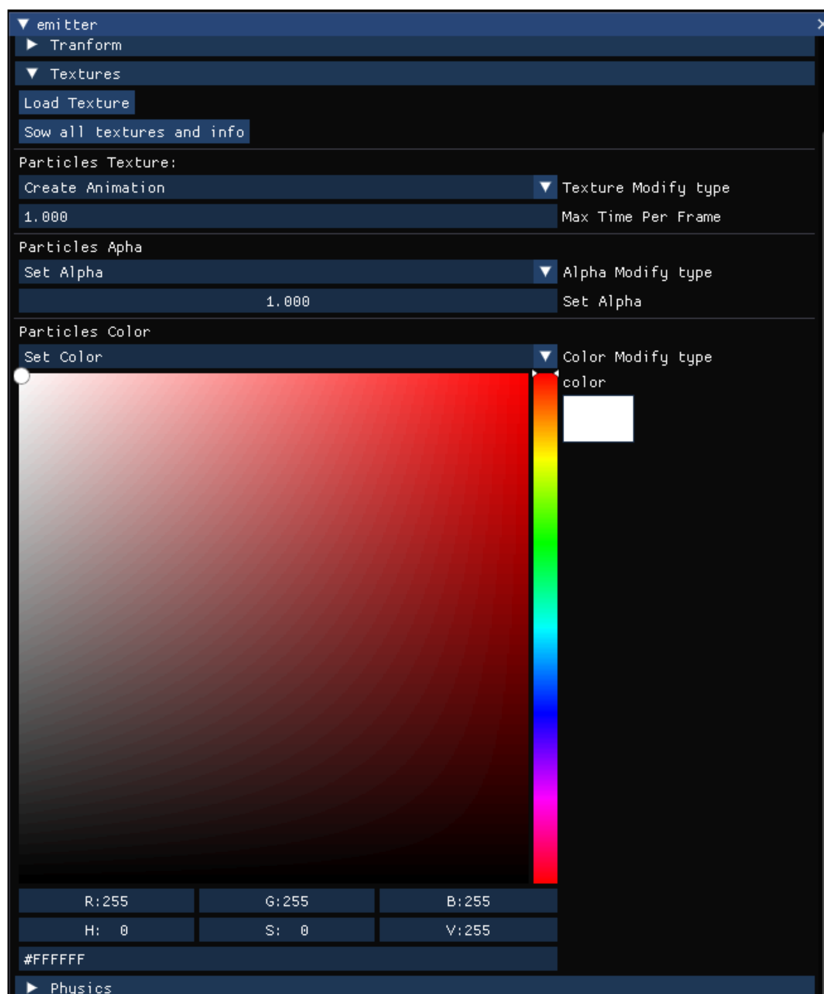


Figura 46 UI de la configuración *textures* del simulador.

La última opción disponible es la de "Physics", esta ventana se encarga de crear y de gestionar todo el sistema de físicas del sistema, tanto las colisiones que se producen entre las mismas partículas como de las colisiones contra agentes externos.

- Seleccionando la primera a opción se activarán las colisiones entre todas las partículas del sistema, si la opción está desactivada las colisiones entre ellas se ignorarán.
- Pulsando el botón "create force" se creará una esfera que dependiendo la configuración de abajo emitirá una fuerza repulsara o una fuerza absorbente.
  - La primera opción es modificar la posición de la esfera.
  - Seguido tendremos la opción de modificar el radio de la esfera, toda partícula que se encuentre dentro de esta esfera será afectada por la fuerza.
  - Por último tendremos la opción de la fuerza en sí, en el caso de que el valor de la fuerza sé a positivo se ejerce una fuerza repulsora, en caso de que el valor sea negativo la fuerza será atrayente.

- Pulsando en el botón de “create wall” crearemos una pared contra la que las partículas podrán chocar, dependiendo de como esté configurada dicha pared el resultado de la colisión será diferente.
    - La primera opción es la posición de la pared en sí.
    - La segunda opción es el tipo de pared que queremos crear, en este caso tendremos dos opciones principales.
        - La opción de “wall” se trata de una pared normal contra la que las partículas colisionaran rebotando de manera normal.
        - La opción “Elim”, esta opción permite que cuando una partícula colisione contra la pared con esta configuración se elimine completamente, esto permite delimitar zonas para que las partículas no se salgan de ciertas zonas.

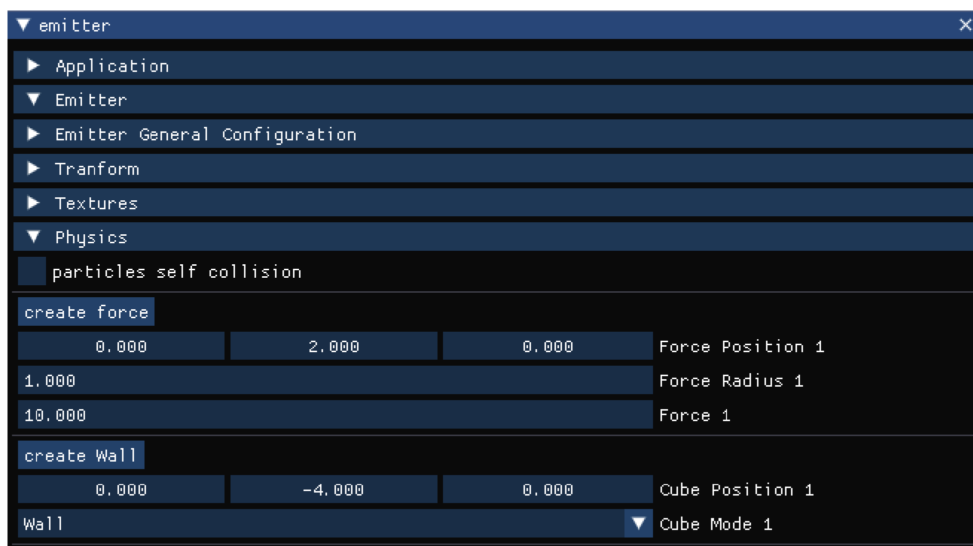


Figura 47 UI de la configuración *Physics* del simulador.

5.4 Visión general del desarrollo

Hasta ahora se han cumplido con los objetivos previstos de desarrollo, como se comentó en el apartado de gestión del proyecto en el apartado 3.5 de Gantt.

El objetivo para esta entrega se trataba de tener una base sólida para los dos sistemas y lograr los primeros resultados de ambos, aunque no fueran los definitivos, en este punto podemos decir que hemos logrado definir las bases principales de los dos sistemas y que es un buen punto de partida para continuar el desarrollo.

En cuanto a la tercera entrega, el objetivo era tener dos sistemas de partículas y realizar los análisis de los resultados de ambos dos, habiendo investigado las diferentes técnicas utilizadas para cada uno de los sistemas. El reto más destacable en este proceso y que más tiempo ha tomado, ha sido la creación del *shader* que utiliza *ray tracing*, en esta entrega también se puede asegurar que se han cumplido los objetivos marcados al principio.

6. Validación del proyecto

Para el apartado de validación del proyecto se ha contado con la participación del profesor de informática y matemáticas de la universidad del país vasco, Jesús Villadangos.

Con el objetivo de completar este apartado se le entregó este documento escrito y se le pidio su opinión general sobre el proyecto, el siguiente texto es la opinión recibida de parte del experto.

“**Validación del Trabajo Fin de Grado**

El trabajo desarrollado plantea una comparativa interesante de combinación de funcionalidades para desarrollar sistemas de partículas. En un caso, se pretende desarrollar un sistema orientado a un mayor realismo sacrificando el rendimiento. En otro caso se desarrolla un sistema orientado a una visualización en tiempo real, optimizando el rendimiento, pero sacrificando la visualización realista.

Este es un ejercicio muy interesante para comprender como influyen los distintos componentes a la hora de integrarlos en un juego y con ello, como desarrollador de videojuegos, tener un criterio claro de cuándo pueden implementarse cada uno de los componentes en función del momento del juego. Este proyecto consideró que supone un paso importante para el estudiante que lo realiza, ya que le fuerza a aprender a controlar algunos de los elementos que afectan críticamente al rendimiento, pero que hay que aprovechar al máximo su uso para mejorar el realismo de la visualización.

Creo que el trabajo hace un planteamiento interesante para el alumno y se ha elaborado un trabajo que permite comprobar que el alumno ha comprendido los parámetros que entran en juego a la hora de valorar entre rendimiento y realismo.”

Lo que se puede sacar en claro de esta opinión es que el experto comprende que este proyecto es una buena manera de comprender los aspectos que afectan al rendimiento general de una aplicación y destaca que es un buen ejercicio para comprender las técnicas y procedimientos a la hora de desarrollar este tipo de sistemas.

Por tanto, podemos concluir que esta opinión respalda y confirma los objetivos que han sido planteados al comienzo del desarrollo.

7. Conclusiones

Para finalmente sacar conclusiones sobre este proyecto debemos analizar cada uno de los sistemas tanto técnicamente como visualmente, primero analizaremos técnicamente ambos casos.

7.1 Análisis técnico

Para realizar este análisis técnico primero estableceremos el hardware que se ha utilizado para llevar las pruebas.

CPU: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz   2.59 GHz

GPU: NVIDIA GeForce GTX 1660 Ti

RAM: 16 GB

7.1.1 Análisis técnico sistema simple

7.1.1.1 Análisis sin instanciado

Para el apartado técnico dividiremos el análisis en diferentes etapas de la simulación y someteremos al sistema a diferentes situaciones para ver el rendimiento según la situación.

Como ya se ha comentado anteriormente, para este sistema se han implementado técnicas de optimización como el instanciado por esto el análisis se dividía en dos bloques principales, el análisis del sistema sin este método de optimización y el análisis con el instanciado, esto puede ser interesante principalmente para ver el efecto real que puede tener esta técnica en la performance

El primer escenario consiste en simplemente añadir una textura a las partículas y añadir una fuerza que las empuje hacia arriba, este es el escenario por defecto que ocurre al iniciar el simulador y es el más sencillo, a continuación se muestra una tabla con los valores de las especificaciones del emisor en este caso.

Particles max live time	5 sec
Particles spawn frecuence	10 per sec
Active particles	50
Max particles	200

Figura 48 Resultado visual de la prueba básica de rendimiento (sistema simple).

Con las especificaciones técnicas mencionadas anteriormente tenemos un resultado de entre 500 - 600 FPS de media.

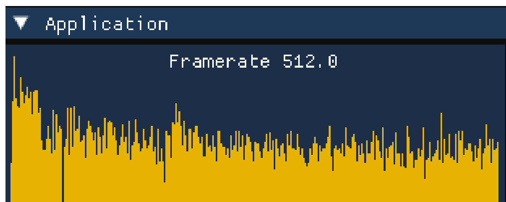


Figura 49 Resultado técnico de la prueba básica de rendimiento (sistema simple).

El segundo caso consiste en añadir dos texturas con una animación que cambia cada segundo, se añade una posición random en los tres ejes y también se añade una velocidad random en los tres ejes, en este caso el Alpha no es constante como en el caso anterior, en este caso tenemos un Alpha que va aumentando con el paso del tiempo, para este caso también se han modificado las especificaciones del emisor.

Particles max live time	10.54 sec
Particles spawn frecuence	19.5 per sec
Active particles	201
Max particles	400

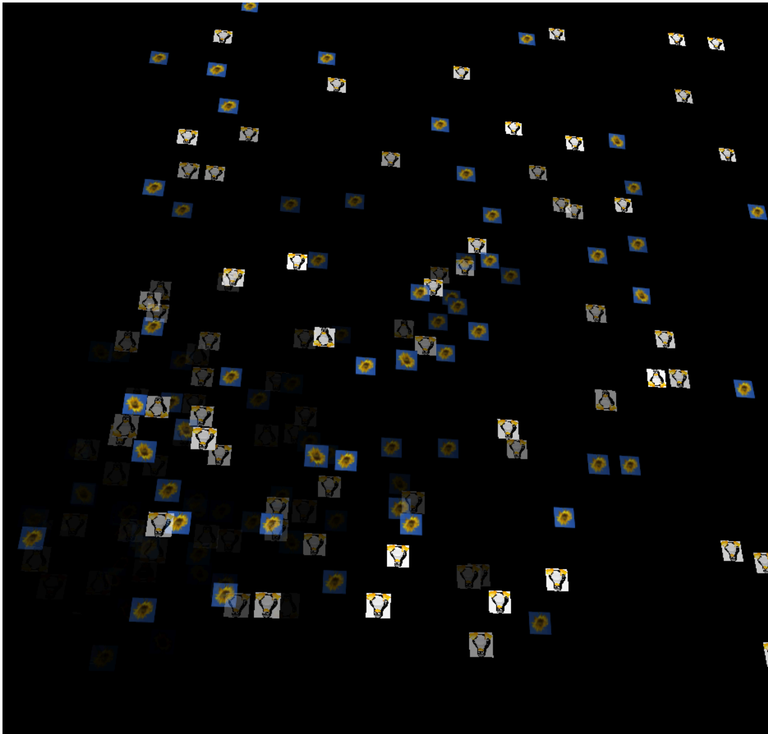


Figura 50 Resultado visual de la segunda prueba de rendimiento (sistema simple).

En este caso se han conseguido unos 200 - 300 FPS de media.

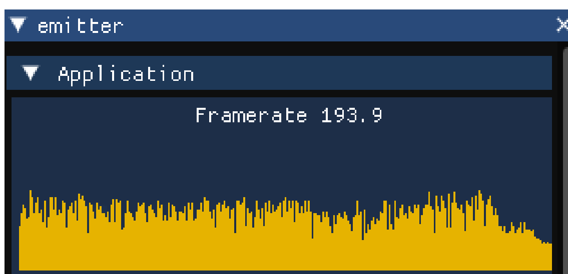


Figura 51 Resultado técnico de la segunda prueba de rendimiento (sistema simple).

El último caso que analizaremos será descubrir el máximo de partículas que podríamos tener por pantalla con una tasa de *frame rate* estables

Particles max live time	93.87 sec
Particles spawn frecuence	20.0 per sec

Active particles	1046
Max particles	2000

En este caso hemos conseguido unos 50 -100 FPS de media

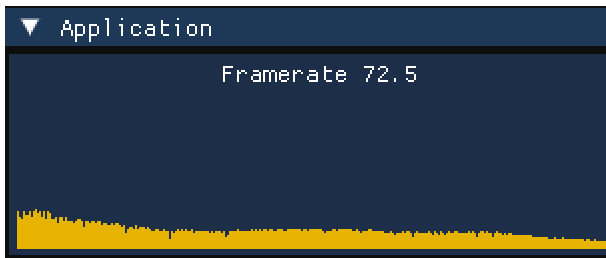


Figura 52 Resultado técnico de la tercera prueba de rendimiento (sistema simple).

7.1.1.2 Análisis con instanciado

Usando la técnica del instanciado es obvia la diferencia en cuanto a rendimiento que se obtiene.

En la siguiente fotografía a modo de ejemplo se han instanciado 50.000 cubos por pantalla, en este caso el programa ha ido a unos 300- 400 FPS, demostrando la gran utilidad y poder que tiene el instanciado.

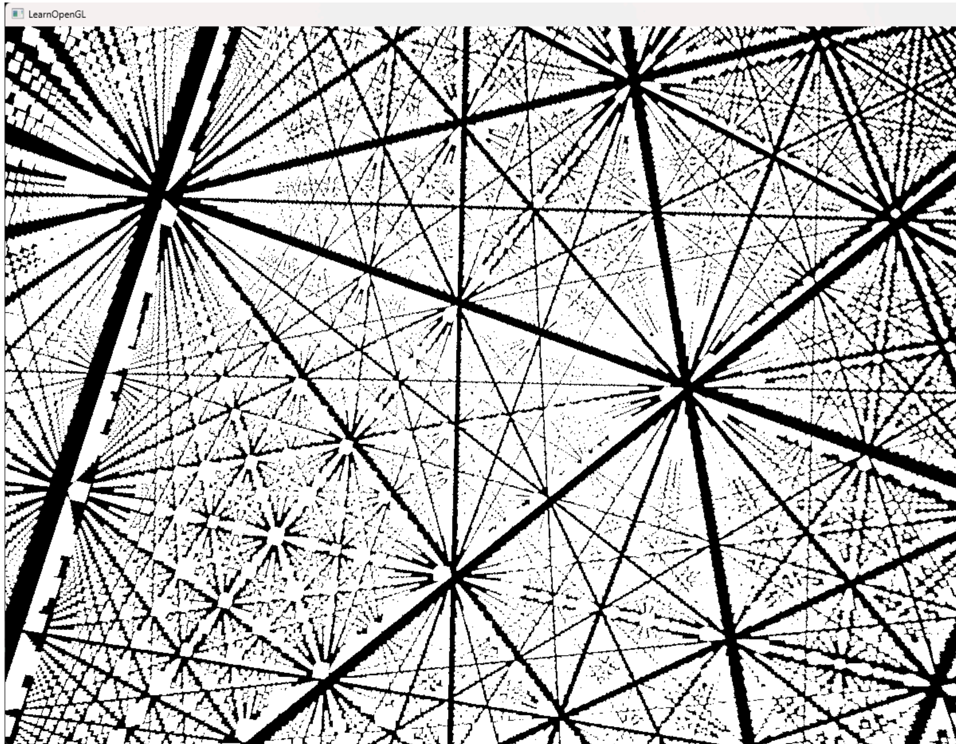


Figura 53 Resultado visual del instanciado.

En otras pruebas se han obtenido resultados también muy positivos, por ejemplo, teniendo 343.000 partículas en pantalla se obtienen una media de 60 - 80 FPS.

7.1.2 Análisis técnico sistema complejo

En este caso, el sistema debe simular un movimiento de humo realista de la que se encarga la CPU, más tarde debe renderizar ese humo mediante *ray tracing*, esto obviamente requiere de más recursos.

Este sistema con una simulación de un cubo de 40 * 40 *40, es decir simulando 64.000 partículas y posteriormente renderizándolas mediante *ray tracing* conseguimos una media de 40 - 50 FPS, en este caso hemos utilizado el sistema con dos procesos principales ejecutándose, por una parte, el cálculo de la simulación y, por otra parte, el renderizado con ray tracing.

Ahora veremos si utilizando el guardado de datos, habiendo hecho la simulación con anterioridad y habiéndola guardado en archivos externos, permite aumentar los *frames* del sistema, pues de esta manera el sistema solo debería renderizar la escena y ahorrarse la simulación.

De esta manera, ejecutando la animación desde archivos externos, con la misma densidad de cubos del ejemplo anterior, unos 64.000, se consiguen entre 80 y 100 FPS, aportando una notable mejora en el rendimiento, esta técnica aunque ayuda sin duda en el rendimiento, genera un problema de memoria, el sistema debe guardar en archivos independientes cada valor de densidad de todos los cubos, esto por supuesto por cada *frame* de simulación.

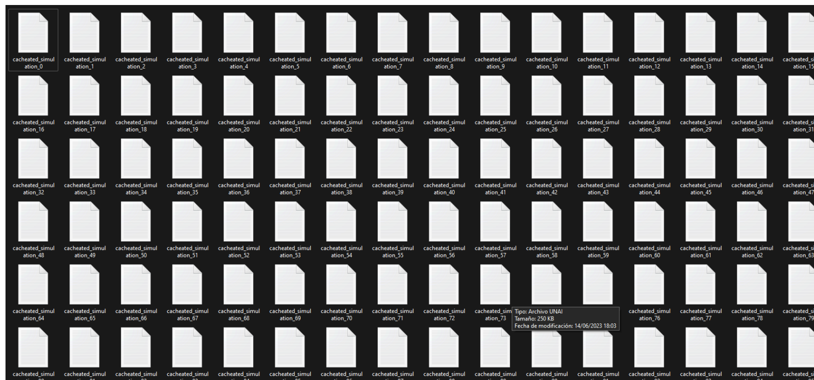


Figura 54 archivos de guardado de la simulación.

Si como en el caso anterior, la densidad de cubos guardado es de 64.000 cubos con un valor de densidad cada uno, cada archivo debería guardar 64.000 valores de densidad, haciendo así que cada *frame* de simulación pese un aproximado de 250KB, gracias a esta técnica ganamos en rendimiento, pero perdemos en memoria.

7.2 Análisis visual

Para empezar con el análisis visual debemos tener en cuenta que los dos sistemas no están pensados para el mismo fin ni se pueden visualmente juzgar de la misma manera.

El primer sistema es muy similar al sistema de Unity, este está pensado para hacer todo tipo de efectos en un juego, no está pensado para conseguir resultados extremadamente realistas, está pensado para que sea versátil y eficiente para poder realizar cualquier tipo de efecto en el juego.

El segundo, sin embargo, está más limitado en cuanto a variedad de posibles resultados, pues está pensado únicamente para mostrar un resultado de humo realista, este sistema está enfocado a un propósito muy concreto.

7.2.1 Análisis visual sistema simple

Como ya se ha comentado antes, este sistema tiene una gran capacidad de personalización y la cantidad suficiente de opciones para conseguir cualquier resultado, los siguientes son unos simples ejemplos de los resultados que se pueden obtener tocando unos simples parámetros.

En la siguiente imagen se muestra un punto de emisión emitiendo partículas en forma cónica, el color se indica de forma aleatoria al instanciar la partícula, al igual que la velocidad en los tres ejes y la textura de estrella que se usara.

Figura 55 Resultado visual de la primera prueba (sistema simple).

Todo esto se consigue utilizando tres simples imágenes de estrellas blancas.

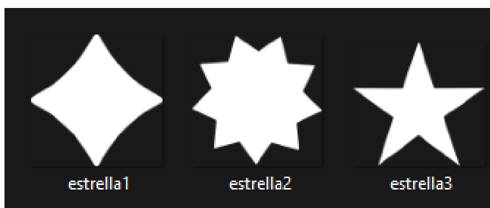


Figura 56 Recursos utilizados para la primera prueba visual (sistema simple)

Al igual que se pueden crear efectos no realistas como en el caso anterior, también se pueden simular efectos más realistas con este sistema, como en el siguiente caso.

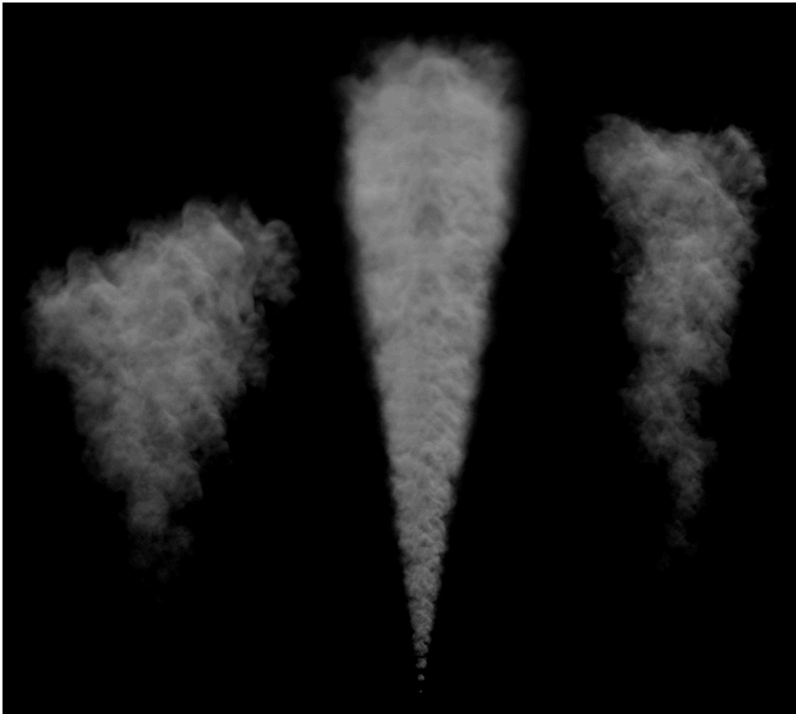


Figura 57 Resultado visual de la segunda prueba (sistema simple).

Tocando simplemente los ajustes de velocidad, posición y rotación se pueden conseguir resultados tan realistas como este, estas imágenes están extraídas de una textura que alberga todas ellas y que el sistema es capaz de separar en texturas más pequeñas.

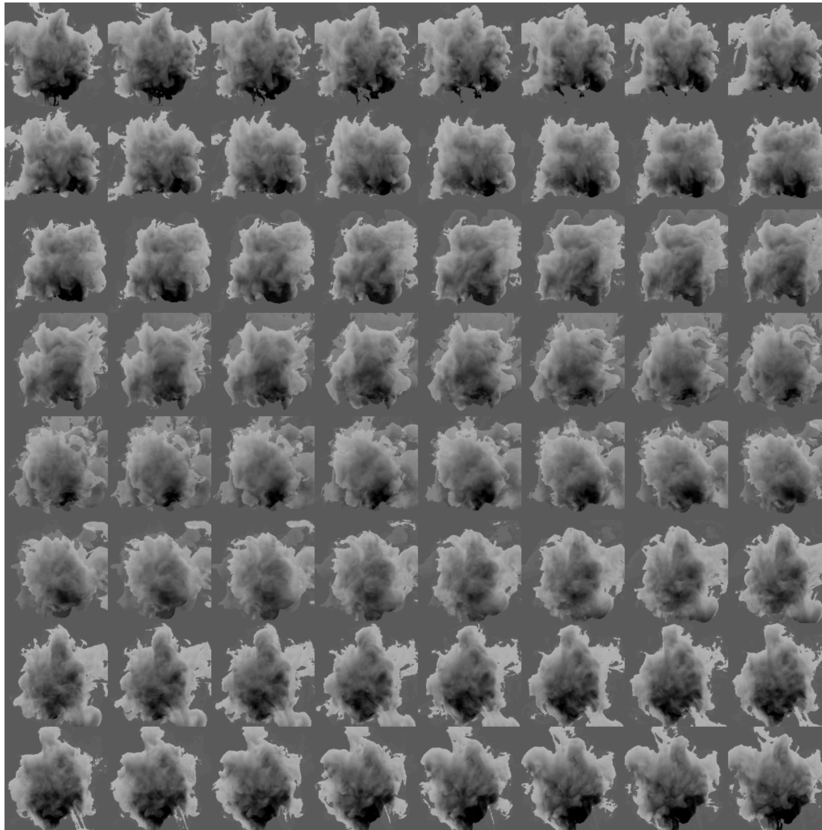


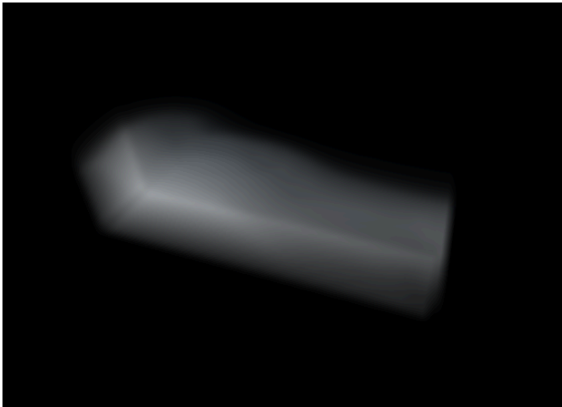
Figura 58 Recursos utilizados para la segunda prueba visual (sistema simple).

Estos solo son dos pequeños ejemplos de todos los resultados que se pueden conseguir con el sistema, desgraciadamente mostrando los resultados mediante imágenes no se puede apreciar la simulación y el movimiento, ni tampoco el efecto que causan las colisiones entre las propias partículas.

7.2.2 Análisis visual sistema complejo

Para este sistema se ha implementado un *shader* de *ray tracing* con un sistema de iluminación básico.

Figura 59 Resultado visual 1 (sistema complejo)



El humo está como se puede ver en la imagen anterior contenido en un cubo, el sistema ofrece diferentes opciones al usuario, por ejemplo, pulsando el teclado, el usuario puede agregar humo (densidad) para ver como se va expandiendo, también puede cambiar los valores de viscosidad y de difusión consiguiendo distintos efectos de expansión y de movimiento del humo.

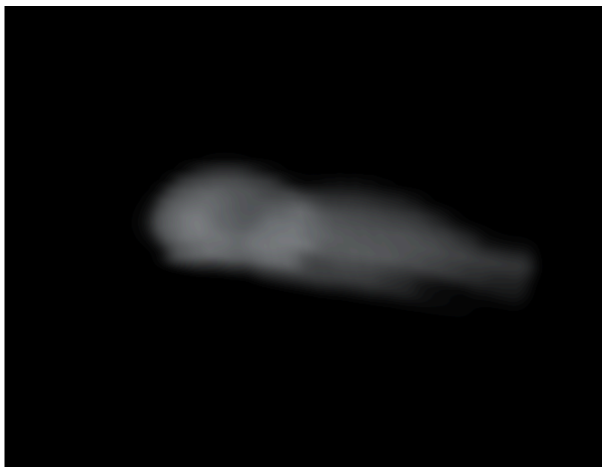


Figura 60 Resultado visual 2 (sistema complejo)

7.3 Conclusiones finales

Este proyecto estaba enfocado a ver las principales diferencias entre estos dos sistemas, tanto técnicamente como visualmente, para de esta manera además saber en qué tipo de proyectos estaría recomendado utilizar cada uno de ellos, pero también está enfocado a ver que tipos de sistemas de partículas se pueden crear y que tipo de diferentes técnicas se utilizan para cada una de ellas, además de investigar y entender cada una de estas técnicas tanto de renderizado como optimización de una manera más profunda, aplicando estas técnicas a un proyecto real.

En definitiva, y como conclusión final, podemos decir que tanto las técnicas y procesos utilizados en el primer sistema como el sistema en general estaría más enfocado a proyectos en los que se requiera un mayor rendimiento técnico y proyectos donde se requiera de una gran versatilidad para poder crear con un solo sistema una gran variedad de efectos con una gran capacidad de personalización.

El segundo sistema, sin embargo, estaría más enfocado a conseguir resultados realistas tanto visualmente como técnicamente en cuanto al comportamiento del humo en la realidad, sin embargo, no está enfocado a proyectos que requieran de una gran optimización, ni para proyectos que requieran de un sistema versátil, pues está creado para crear un resultado en específico (renderizado realista de volumen).

A continuación se mostrarán varias tablas de ventajas, desventajas y de técnicas utilizadas de cada uno de los sistemas de ambos sistemas para ver de una manera más visual la diferencia entre ambos.

	Ventajas	**Desventajas**
Sistema simple	• Gran eficiencia técnica. • Gran versatilidad para conseguir variedad de resultados. • Muy personalizable. • Portable y fácil de implementar en cualquier proyecto	• Complejos si se pretende conseguir resultados realistas. • Limitaciones en algunos aspectos técnicos.

Sistema complejo	• Resultados fieles a la realidad. • Se utilizan las técnicas más avanzadas en el ámbito. • Portable y fácil de implementar en cualquier proyecto	• Requiere de muchos recursos técnicos. • Creado solo para conseguir simulación de humo.

Tabla 4 Ventajas y desventajas de los sistemas

Sistema simple	• Instanciado de partículas.
Sistema complejo	• Ray tracing (renderizado) • Deferred Rendering • Guardado de datos

Tabla 5 Técnicas utilizadas en cada uno de los sistemas

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de la Imatge i la Tecnologia Multimèdia

8. Bibliografía

1. learn Opengl.

   https://learnopengl.com/Getting-started

2. Fluid Simulation for dummies by Mike Ash

   https://mikeash.com/pyblog/fluid-simulation-for-dummies.html

3. How to create instanced Particle System

   https://levelup.gitconnected.com/how-to-create-instanced-particles-in-opengl-24cb089911e2

4. Intro to particle systems

   https://www.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-particle-systems/a/intro-to-particle-systems

5. Shadertoy

   https://www.shadertoy.com/

6. Advanced Graphics_Ray Tracing_All the math

   https://www.cl.cam.ac.uk/teaching/1718/AdvGraph/Printable%20(1-up).pdf

7. Fluid Dynamics Simulation in C++ and OpenGL

   https://blog.seanholloway.com/2021/09/09/fluid-dynamics-simulation-in-c-and-opengl/

8. Gentle Introduction to Realtime Fluid Simulation for Programmers and Technical Artists

   https://shahriyarshahrabi.medium.com/gentle-introduction-to-fluid-simulation-for-programmers-and-technical-artists-7c0045c40bac

9. Level5_Particle-System

   https://github.com/KieranBond/Level5_Particle-System

10. The book of shaders

    https://thebookofshaders.com/

11. Real time rendering

    https://www.realtimerendering.com/