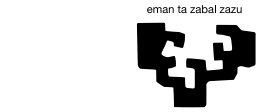


Grado en Ingeniería Informática
Computación

Trabajo de fin de grado

Simulación de nubes volumétricas

Eneko Alaminos Duran

Directora:
Carmen Hernández Gómez

Junio de 2020

Resumen

Simular una atmósfera realista supone tener en cuenta un fenómeno que podemos ver casi cada día en el cielo: las nubes. Este fenómeno compuesto de partículas de agua y/o hielo puede parecer fácil de representar, pero esconde una geometría fractal que lo complica. Por esto mismo, a lo largo de los años han surgido diferentes estudios que tratan de encontrar un modelo para ello. Además, otra parte importante de estas es su iluminación: cuando un rayo de luz atraviesa una nube, los fotones que componen la primera pueden ser dispersados o absorbidos por las partículas de la segunda. Simular este comportamiento es una tarea difícil.

En este trabajo de fin de grado se va a estudiar una de las técnicas que existen para la simulación de nubes volumétricas. En el modelo estudiado se utilizan varias texturas 2D y 3D creadas mediante ruido procedural (Perlin y Worley). Estas texturas se utilizan para definir dónde, con qué forma y detalle se van a mostrar las nubes. Para visualizarlas se utiliza un algoritmo llamado *ray marching* que calcula la densidad e iluminación de la nube por cada iteración. En el caso de la iluminación, se utilizan dos funciones para aproximarla: la ley de Beer y la función de fase de Henyey-Greenstein.

Además, se ha implementado una aplicación que permite visualizar nubes, permitiendo cambiar la cantidad que hay en el cielo, su densidad, la altura a la que se encuentran, su iluminación o la posición del sol. También es posible hacer que estas se muevan en una dirección y con una velocidad fijada.

Agradecimientos

Primero agradecer a mi directora de proyecto Carmen Hernández por toda la ayuda ofrecida, las dudas resueltas y los consejos dados en el desarrollo de este proyecto. Además, también me gustaría agradecerle por las clases que me ha impartido, gracias a ella mi interés por la computación gráfica ha crecido bastante y he aprendido mucho sobre este ámbito.

Después, quiero agradecer a mi familia por estar apoyándome y ayudándome durante la carrera y simplemente por estar ahí en cualquier momento. Por último, agradecer a todas las personas que he conocido durante estos años de carrera, aquellas que han estado apoyándome durante el desarrollo del proyecto y las que me han acompañado durante el confinamiento. Muchas gracias, se os quiere.

Dedicado a
mi familia

Índice

Lista de Figuras

Lista de Tablas

Capítulo 1

Introducción

Crear una atmósfera realista ha sido una tarea muy estudiada en la computación gráfica, pero lograr esto implica también simular un fenómeno que podemos ver casi cada día en el cielo: las nubes. Esto puede parecer una tarea sencilla, pero dada la forma fractal de las nubes y la importancia de la iluminación para que parezcan realistas, los cálculos se complican. Además, para la iluminación hay que tener en cuenta que las nubes son medios participantes, es decir, estas están formadas por partículas que pueden alterar los rayos de luz que atraviesan las mismas.

Debido a estas dificultades se han realizado muchos estudios en este área desde los inicios de la computación gráfica. A medida que la tecnología ha ido avanzando se han explorado otros métodos que, al principio, resultaban muy costosos computacionalmente. En este trabajo estudiaremos uno de estos métodos que son utilizados hoy en día para la simulación de nubes volumétricas. Al principio, hemos incluido una pequeña introducción a varias de las técnicas presentadas a lo largo de los años para, luego, centrarnos en la tesis de Häggström [12], la cual se basa en el trabajo realizado para el videojuego *Horizon Zero Dawn* [40] donde presenta un modelo para la generación e iluminación de nubes volumétricas.

En el renderizado volumétrico se suelen utilizar algoritmos como el *ray marching* que, por cada **píxel (*pixel*)**, lanza un rayo hacia un volumen y se evalúa en ciertos pasos predeterminados la densidad y la iluminación en cada muestra. Estos métodos son simples pero muy costosos, aunque dan buenos resultados visuales en el caso de volúmenes como las nubes. En todo caso, con el hardware de hoy en día es posible optimizar estos algoritmos para tener un rendimiento aceptable. De hecho, es lo que logran [12] o [40], teniendo una media de tiempo de renderizado de 2 ms con una resolución de 1920×1080.

1.1. Planificación

1.1.1. Objetivos del proyecto

El objetivo principal de este proyecto es el estudio de una de las técnicas para la simulación de nubes en un entorno 3D y de la implementación de la misma.

En concreto, los objetivos generales se pueden dividir en estos cinco:

- Estudiar los diferentes modelos para la representación de nubes que se han presentado hasta la fecha.
- Estudio de las técnicas matemáticas del modelo elegido.
- Estudio de las técnicas de iluminación de nubes.
- Comprender el uso de la librería *Three.js* y de las herramientas que sean necesarias para la implementación a realizar.
- Implementar el modelo usando las herramientas estudiadas y analizar los resultados.

Una vez estudiadas las diferentes técnicas presentadas hasta la fecha, se ha seleccionado una técnica de renderizado de nubes volumétricas en tiempo real desarrollada en la tesis [12]. Después de revisar este trabajo, hemos establecido dos objetivos específicos:

- Comprender las diferentes técnicas de generación de ruido procedural: ***fractal Brownian motion* o movimiento Browniano fractal** [24], ruido de Perlin [34] [35] y ruido de Worley [50].
- Comprender la técnica de renderizado: *Ray marching* [33] [29].

En cuanto a la implementación, el objetivo es lograr una representación con las siguientes características:

- Nubes que pueden variar en tamaño y densidad para poder generar tres tipos de nubes: cúmulos, estratocúmulos o estratos.
- Posible variación de la iluminación para representar diferentes momentos del día.
- Animación de las nubes.

1.1.2. Tareas

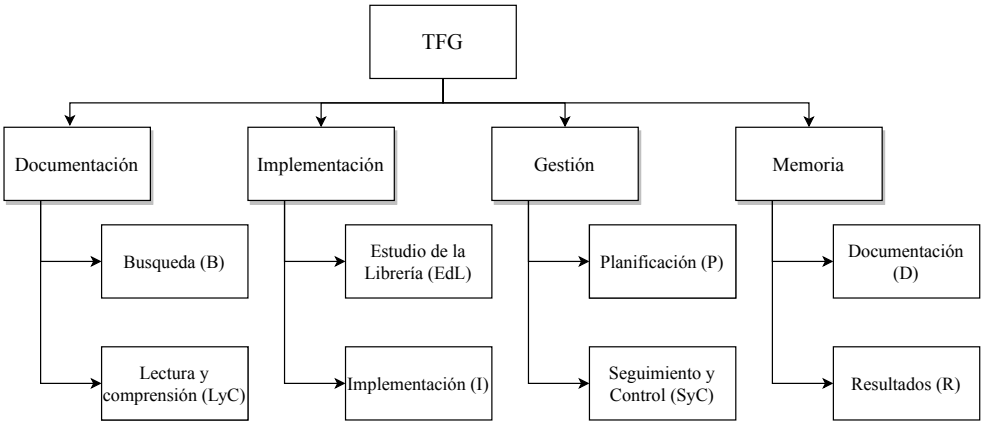


Figura 1.1: Distribución de las tareas del Proyecto de Fin de Grado.

Las tareas se han distribuido en cuatro bloques principales donde, cada uno de ellos se divide en otros dos bloques específicos como se puede ver en el despliegue recogido en la Figura 1.1. Cada bloque especifico agrupa diferentes tareas a realizar.

En la siguiente lista se muestran las tareas que contiene cada bloque:

- Bloque de Documentación.
  - Búsqueda (B).
    - **B1.** Crear un listado de los temas a buscar.
    - **B2.** Buscar artículos relacionados con el propósito del proyecto y del listado de temas.
    - **B3.** Leer, catalogar y/o desechar los artículos encontrados.
  - Lectura y comprensión (LyC).
    - **LyC1.** Repaso de conceptos necesarios para la realización del proyecto.
    - **LyC2.** Leer y comprender la idea general de cada artículo.
    - **LyC3.** Leer y comprender las técnicas necesarias para la implementación.

- Bloque de Implementación.
  - Estudio de la librería (EdL).
    - **EdL1.** Leer la documentación de *Three.js*.
    - **EdL2.** Realizar algunas pruebas con la librería.
  - Implementación (I).
    - **I1.** Implementación o búsqueda de librerías para la generación de ruido procedural, bien en *Three.js* o con otra herramienta.
    - **I2.** Implementación del modelado de nubes.
    - **I3.** Implementación de la iluminación.

- Bloque de Gestión.
  - Planificación (P).
    - **P1.** Identificación de las tareas a realizar, tiempos y fechas clave.
    - **P2.** Actualización de la planificación, en caso necesario.
  - Seguimiento y Control (SyC).
    - **SyC1.** Reuniones con la tutora y control del progreso del trabajo.
- Bloque de Memoria.
  - Documentación (D).
    - **D1.** Escribir el capítulo sobre el estado del arte.
    - **D2.** Escribir el capítulo de conceptos generales y diseño.
  - Resultados (R).
    - **R1.** Escribir el capítulo de implementación.
    - **R2.** Escribir el capítulo de resultados y conclusiones.
    - **R3.** Revisión y corrección de la memoria.

1.1.3. Tiempo de realización de las tareas

El periodo de realización de las tareas abarca 21 semanas, desde el día 27 de enero al 7 de junio, y se estiman unas 325 horas de dedicación total. En el siguiente diagrama se muestran la planificación de tareas semanal:

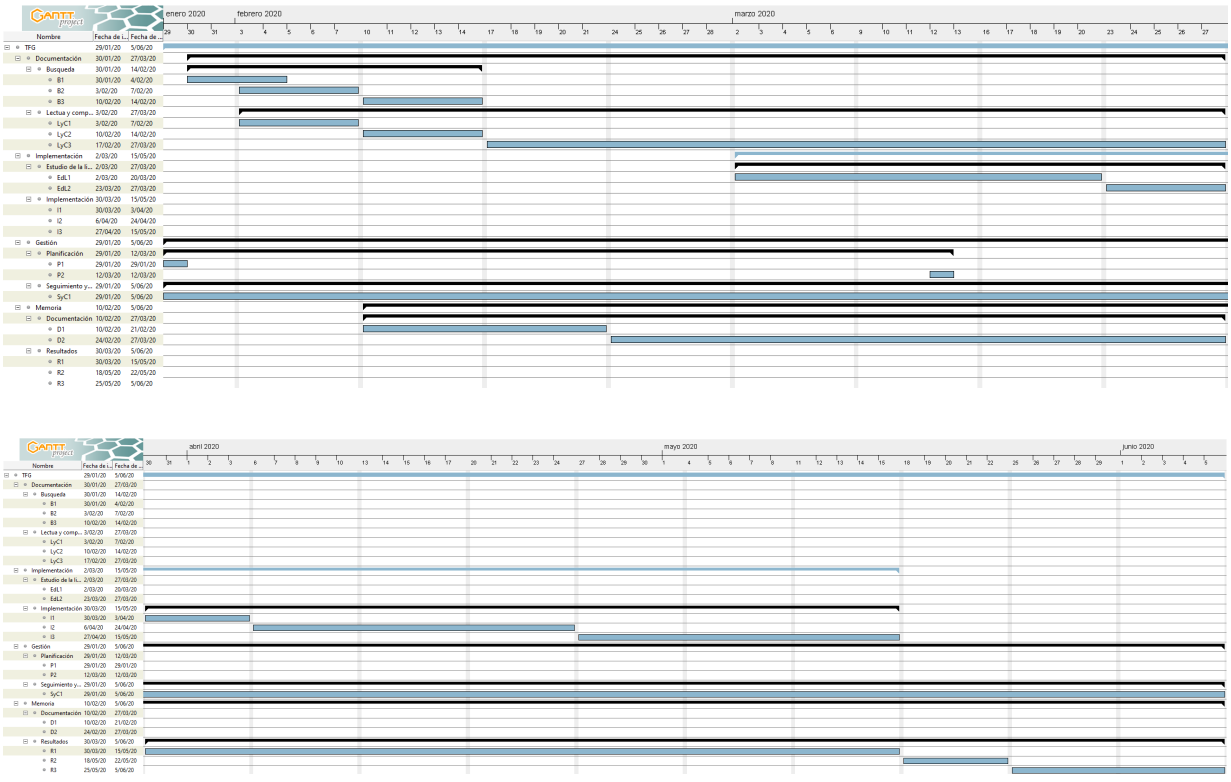


Figura 1.2: Diagrama de Gantt del Proyecto de Fin de Grado.

1.1.3.1. Dedicación estimada por cada bloque

En la siguiente tabla se muestra el tiempo estimado en horas para cada bloque de tareas:

Bloque	**Bloque especifico**	**Dedicación**	**Dedicación Bloque**
Documentación	Búsqueda	15h	65h
	Lectura y comprensión	50h	
Implementación	Estudio de la librería	25h	100h
	Implementación	75h	
Gestión	Planificación	10h	40h
	Seguimiento y Control	30h	
Memoria	Documentación	60h	120h
	Resultados	60h	
Total			325h

Tabla 1.1: Dedicación en horas por cada bloque de tareas.

1.1.4. Análisis de riesgos

Uno de los posibles riesgos puede residir en la realización de la tarea correspondiente al estudio de la herramienta *Three.js*, ya que es la primera vez que se utiliza, se desconocen las limitaciones que pueda tener y, por lo tanto, quizás algunas técnicas no se puedan implementar en la misma. Si eso ocurre, se buscará otra herramienta compatible para poder solventarlo.

Otro de los riegos es causado a raíz de la pandemia del COVID-19, siguiendo las pautas recomendadas se pueden reducir los riesgos que existen. Además, la mayor parte del proyecto se va a realizar en confinamiento, esta situación puede afectar negativamente por la monotonía y el distanciamiento social. Por lo tanto, junto con un horario para el proyecto, se han reservado varias horas para otras actividades.

1.2. Contenidos de la memoria

En el capítulo 2 se repasan los diferentes trabajos que se han realizado en este ámbito. En el siguiente capítulo 3 se explican varios de los conceptos que se necesitan para la comprensión de este trabajo. Después, en el capítulo 4 introducimos el diseño del modelo utilizado, explicando cómo se muestran las nubes junto con la iluminación de las mismas. El capítulo 5 detalla algunos aspectos de la implementación y en el capítulo 6 se muestran los resultados obtenidos. Finalmente, en el capítulo 7 se reflexiona sobre las conclusiones y las líneas futuras de este trabajo.

1.3. Herramientas utilizadas

1.3.1. Three.js

Three.js es una librería de JavaScript y ***Application programming interface* o Interfaz de programación de aplicaciones** (**API**) para crear y mostrar animaciones en gráficos 3D en un navegador web. Esta librería utiliza WebGL y maneja escenas, cámaras, luces, sombras, materiales (permitiendo añadir *shaders* GLSL), texturas y modelado 3D entre otros. Se ha elegido esta herramienta por su facilidad de uso y aprendizaje. La página oficial se encuentra en: `https://threejs.org/`.

1.3.2. GLSL

OpenGL Shading Language (**GLSL**) es un lenguaje de alto nivel de *shading* para OpenGL o WebGL. Tiene una sintaxis parecida a C aunque simplificada y con pequeñas diferencias. Están diseñados para ejecutarse en la GPU, dando mayor control del *pipeline* de renderizado y se envían dos tipos de programas: el *vertex shader* que se ejecuta por cada vértice de los modelos 3D y el *fragment shader* que se ejecuta por cada **píxel (*pixel*)**.

1.3.3. Overleaf

Overleaf es un editor de documentos online que utiliza LaTeX. Permite compartir, comentar y editar simultáneamente el documento y, por lo tanto, resulta muy útil por ser una herramienta apropiada para la creación de la memoria del proyecto. Su página web es la siguiente: https://www.overleaf.com.

1.3.4. Atom

Atom es un editor de textos de código abierto y gratuito, además de ser multiplataforma (Linux, Microsoft Windows y macOS). Permite utilizar varios *plug-ins* que permiten expandir sus funcionalidades, como poder resaltar el lenguaje HTLM, JavaScript o GLSL. Tiene un sistema de control de Git integrado, lo que permite utilizar dicha herramienta de forma rápida y sencilla.

1.3.5. Git

Git es un *software* libre y gratuito para el control de versiones. Se utiliza mucho para el seguimiento de los cambios que se hacen del código fuente durante el desarrollo del proyecto.

1.3.6. Gimp

GNU Image Manipulation Program (**GIMP**) es un programa de edición de imágenes libre y gratuito, además es multiplataforma (Unix, GNU/Linux, FreeBSD, Solaris, Microsoft Windows y macOS, entre otros). Se elegió por su sencillez de uso y para editar manualmente el mapa climático, ver sección 4.3.

Capítulo 2

Estado del arte

Como ya hemos comentado, la representación de las nubes ha sido un tema bastante estudiado desde el principio de la computación gráfica. Durante años se han investigado varias técnicas para el modelado de nubes entre las cuales podemos distinguir las siguientes categorías: modelos implícitos, sistemas de partículas, modelos de *voxels*, modelos procedurales y modelos basados en la física. A continuación, se explica brevemente cada tipo de modelado y se presentan uno o varios trabajos realizados hasta la fecha con cada uno de ellos.

Modelos implícitos

Este tipo de modelado implícito fue presentado por primera vez por Blinn en 1982 [4]. Se utilizan superficies con valores constantes (isosuperficies), creadas a partir de la mezcla de primitivas, y vienen representadas por ecuaciones implícitas de la forma $f(x, y, z) = 0$. En el caso de las nubes, se modela y se representa su campo de densidad mediante funciones procedurales. La estructura principal del modelo son primitivas básicas, tales como esferas o elipsoides que vienen dadas por su formulación implícita correspondiente. Debemos, en todo caso, comentar que el muestreo de densidades definidas implícitamente puede resultar

costoso y repercutir en el rendimiento del renderizado. A continuación, presentamos varios trabajos en los que se han utilizado estos modelos.

En 2002 Kniss et al. [19] utilizaron funciones procedurales para crear una distorsión geométrica que añade una apariencia fractal a formas ordinarias. Este efecto se logra modificando las posiciones del vértice de la geometría que se va a visualizar. Aplicaron dicha distorsión mediante un *vertex shader*.

Schpok y otros [43] desarrollaron en 2003 un sistema completo para modelar, animar y renderizar nubes volumétricas basándose en los modelos propuestos por Ebert [10].

En 2005, Lipuŝ y Guid [20] presentaron algunos problemas encontrados en el uso de las técnicas de mezcla de superficies implícitas de la época, alegando que estas habían sido diseñadas principalmente para el uso de modelado de superficies. Además, introdujeron una técnica de mezcla implícita para primitivas apropiadas para el modelado volumétrico.

En 2008, Bouthors et al. [6], utilizaron un modelo basado en polígonos junto con una hipertextura [33] combinando volúmenes que limitaban la superficie.

Sistemas de partículas

En los sistemas de partículas, introducidos por Reeves en 1983 [39], los objetos se representan mediante un conjunto de geometrías simples con un volumen predefinido. Utilizan una gran cantidad de primitivas geométricas y se puede controlar la animación, el nacimiento y la muerte de dichas partículas mediante algoritmos. En el modelado de nubes se suelen utilizar partículas esféricas con una función radial de densidad. Podemos ver el uso de esta técnica en los siguientes artículos:

En 2002, Harris [13] describió un método para el renderizado en tiempo real de nubes en simuladores de vuelo y videojuegos. Debido al hardware de la época y al entorno complejo de los videojuegos, decidieron modelar las nubes utilizando partículas y representarlas utilizando impostores, una solución basada en texturas.

Bouthors, Neyret y Lefebvre [5], en 2006, utilizaron partículas para representar cúmulos. Su algoritmo genera partículas iterativamente en el volumen de la nube, colocando las partículas más pequeñas sobre las partículas más grandes. Esto crea una jerarquía de partículas con radio decreciente que especifica la superficie de la nube.

Xu et al. [51] propusieron en 2009 unos campos de probabilidad generados y gestionados por autómatas celulares. Para crear el campo de probabilidad adaptan el algoritmo de ***fractal Brownian motion* o movimiento Browniano fractal** (**fBm**) el cual describe el movimiento caótico natural de las partículas y dota a las nubes de características realistas. La simulación se ejecuta en la GPU y utilizan la técnica de *ray casting* para renderizar las nubes.

Yusov [52], en 2014, presentó un nuevo método para generar y visualizar cúmulos en tiempo real. Para crear la nube utilizó copias de una única partícula que se rotaba y se escalaba aleatoriamente.

Modelos con *voxel*

En cuanto a los modelos basados en **vóxels (*voxels*)**, debemos destacar que son una técnica sencilla pero que necesitan mucha memoria. Se caracterizan por almacenar los valores de densidad de la nube en una cuadrícula de **vóxels (*voxels*)** (Ver Figura 2.1). En el trabajo presentado por Dobashi et al. [8], a principios del año 2000, se describió un método basado en un autómata celular para la animación de nubes. Las dinámicas se expresaban mediante simples transiciones con las que se lograba simular movimientos complejos en pocos pasos de computación. Utilizaba la librería gráfica OpenGL en los sistemas hardware de la época, logrando movimientos de nubes, sombras sobre el terreno y haces de luz a través de las nubes.

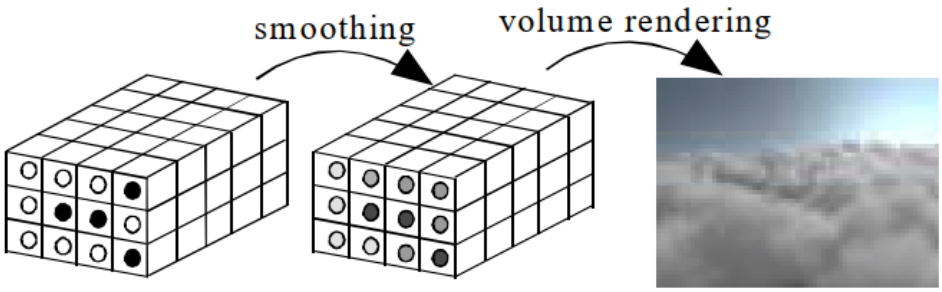


Figura 2.1: Proceso de renderizado utilizando *voxels* [8].

Modelos procedurales

En los modelos procedurales se suele definir una función de densidad junto con una combinación de ruido procedural; por ejemplo, ruido de Perlin [35] o ruido de Worley [50]. Muchas veces producen resultados que se pueden modificar más que los simuladores reales y en menor tiempo. Este tipo de modelado se ha presentado en los siguientes artículos:

Schneider [41] presentó en el congreso *SIGGRAPH 2015*, un sistema de nubes utilizado en el videojuego *Horizon Zero Dawn*, que usaba una combinación de ruido de Worley [50] y de Perlin [35] y utilizaba la técnica de *ray marching* para renderizar cúmulos realistas en tiempo real junto con una luz dinámica.

En 2018, Webanck et al. [49] propusieron un modelo procedural para representar diferentes tipos de nubes en un rango de altitudes. Su método permite animar nubes por ***morphing***, en vez de simular la evolución de las nubes mediante simulación basada en la física. Se calcula el movimiento de la nube usando interpolación por fotograma clave (*keyframe interpolation*) y se aborda el problema de ***morphing*** como un problema de transporte óptimo.

Además de estos artículos, también existen varias tesis que se basan en estas técnicas como las dos siguientes.

Figura 2.2: Ejemplos de nubes volumétricas en el videojuego *Horizon Zero Dawn* [41].

En 2016, Högfeldt [16] describió en su tesis cómo implementaron las nubes en tiempo real en el motor gráfico de la compañía *Electronic Arts (EA)*, *Frostbite*. Utilizaron la técnica de *ray marching* junto con una combinación del ruido de Perlin [35] y de Worley [50] para visualizar nubes volumétricas.

Dos años más tarde, Häggström [12] presentó en su tesis una implementación de nubes volumétricas con el objetivo de mejorar varios aspectos de las soluciones publicadas. Su aproximación utiliza diferentes texturas (de 2 y 3 dimensiones) junto con varios valores y funciones para crear la forma de la nube. Para la visualización utiliza un algoritmo de *ray marching*.

Modelos basados en la física

Por último, los modelos basados en la física intentan simular la creación de las nubes siguiendo las leyes dinámicas de fluidos y la termodinámica. Por ejemplo, en 2008, Dobashi et al. [9], propusieron un sistema para controlar la generación de cúmulos basado en la mecánica de fluidos computacional. El usuario especificaba la forma general de la nube y el método ajustaba los parámetros durante la simulación para crear la forma especificada.

Modelos combinados

Estos son algunos de los modelos que se suelen utilizar en el modelado de nubes. Generalmente no se utiliza un único modelo sino que se combinan varios como se puede ver en los siguientes trabajos.

Man [23] en 2006 presentó un método para la generación y renderizado en tiempo real de nubes estáticas. Utilizaba la función de ruido de Perlin [35] para generar el mapa 3D de la nube. El autor argumentaba que el almacenamiento del mapa en **vóxels (*voxels*)** no resultaba apropiado para el renderizado en tiempo real, por lo tanto, introducía una representación que aproximaba el mapa original utilizando ***metaballs*** y una red neuronal: ***Radial Basis Function* o función de base radial (RBF)**.

En 2017, Goswami y Neyret [11] presentaron un modelo procedural basado en la física para el renderizado en tiempo real y la animación de cúmulos en un paisaje. Para ello, combinaron un modelo Lagrangiano con amplificación procedural utilizando ruido volumétrico.

En el mismo año, Montenegro et al. [27] propusieron una técnica que combinaba modelos volumétricos implícitos y técnicas de ruido procedural para el modelado de nubes.

Por otra parte, y además de revisar trabajos más centrados en el modelado de nubes, también hemos consultado varios trabajos relacionados solamente con la visualización de las mismas.

En 2012, Dobashi et al. [7], aproximan el renderizado de las nubes con un problema inverso. Dada una imagen real de nubes, el algoritmo estima los parámetros de un modelo de densidad no-uniforme.

Hufnagel y Held [15] muestran un estudio sobre los diferentes tipos de técnicas de visualización de nubes y las clasifican según la representación del tipo de volumen. También muestran técnicas de iluminación global aplicables a la generación de efectos ópticos en un paisaje de la vida real con nubes.

En 2015, Mukhina y Bezgodov [28], presentaron un algoritmo de visualización en tiempo real de nubes planas. Utilizaron una semiesfera donde proyectaban las nubes creando un efecto de colocación de las mismas mucho más natural.

Finalmente, Kallweit y otros [18], presentaron una técnica para sintetizar eficientemente imágenes de nubes atmosféricas usando una combinación de la integración de Monte Carlo y redes neuronales.

En la Figura 2.3 podemos observar cada artículo mencionado ordenado por año y tipo de modelo.

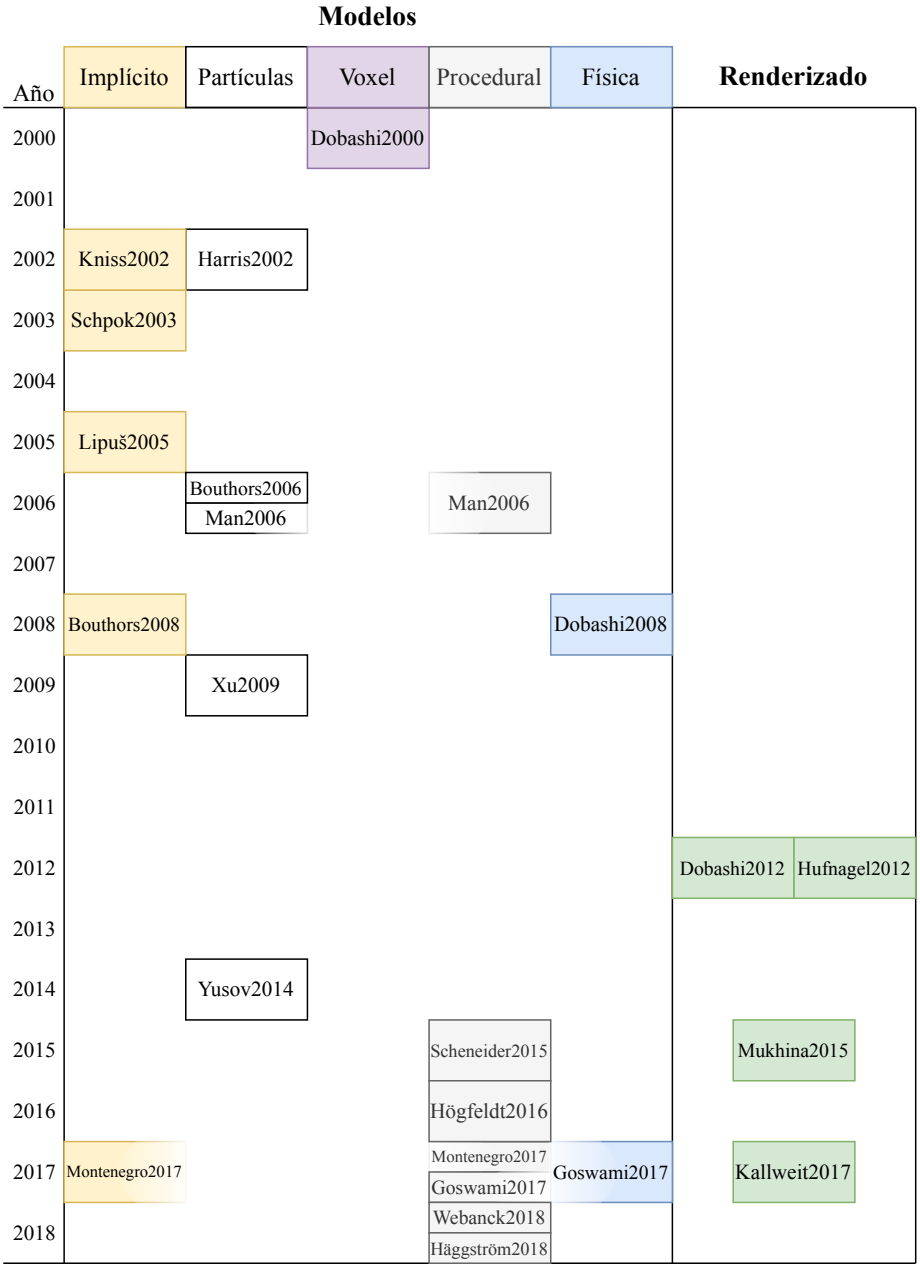


Figura 2.3: Relación de los artículos mencionados en este trabajo separados por año y categoría.

Capítulo 3

Conceptos generales

En esta sección vamos a introducir brevemente varios conceptos para comprender el diseño que posteriormente vamos a desarrollar en este trabajo. Empezamos explicando cómo y de qué están formadas las nubes, los tipos de nubes que hay y su iluminación. Terminamos con varias funciones para la generación de ruido procedural: **fBm**, ruido de Perlin y ruido de Worley.

3.1. Nubes

Antes de comenzar, tenemos que definir qué es lo que queremos representar. En este caso, es conveniente saber de qué está formada una nube, qué tipos de nubes hay y cómo afecta la luz al atravesar una nube.

Comenzando desde la definición dada por la ***World Meteorological Organization*** **u Organización Meteorológica Mundial, SIG** (**WMO**) que describe una nube de la siguiente forma [32]: «*Una nube es un hidrometeoro formado por partículas diminutas de agua líquida o hielo, o los dos, suspendida en la atmósfera y, normalmente, sin tocar el suelo. También puede incluir partículas más grandes de agua líquida o hielo, así como*

líquido no acuoso o partículas sólidas tales como las presentes en el humo o polvo».

3.1.1. Formación de nubes

En cuanto a la formación, las nubes se crean cuando el aire se eleva calentado por la irradiación terrestre. Mientras se va calentando, el aire sube hasta que llega al punto de rocío, que es cuando el vapor se condensa en partículas diminutas de agua líquida o hielo. Estas, al encontrarse suspendidas en el aire, están en continuo movimiento y, al verse sometidas a corrientes ascendentes, chocan una con otra y se agrupan.

Estas partículas tienen un tamaño de entre 0.004 y 0.1 mm. Las nubes están formadas por una gran cantidad de partículas de este tipo. Además, si la nube está lo suficientemente alta y el aire es lo suficientemente frío, las nubes están también formadas por partículas de hielo, dando una apariencia de nube delgada y tenue.

3.1.2. Tipos de nubes

Aunque las nubes están continuamente cambiando y puedan aparecer en diferentes formas, estas tienen unas determinadas características que nos permiten realizar una clasificación de las mismas.

Se suelen clasificar por tipos, especies y variación. En nuestro caso, solo vamos a tener en cuenta la primera clasificación. Como podemos observar en la Figura 3.1 existen 10 tipos de nubes según su altura. Vamos a ver cada una de ellas teniendo en cuenta solamente tres altitudes.

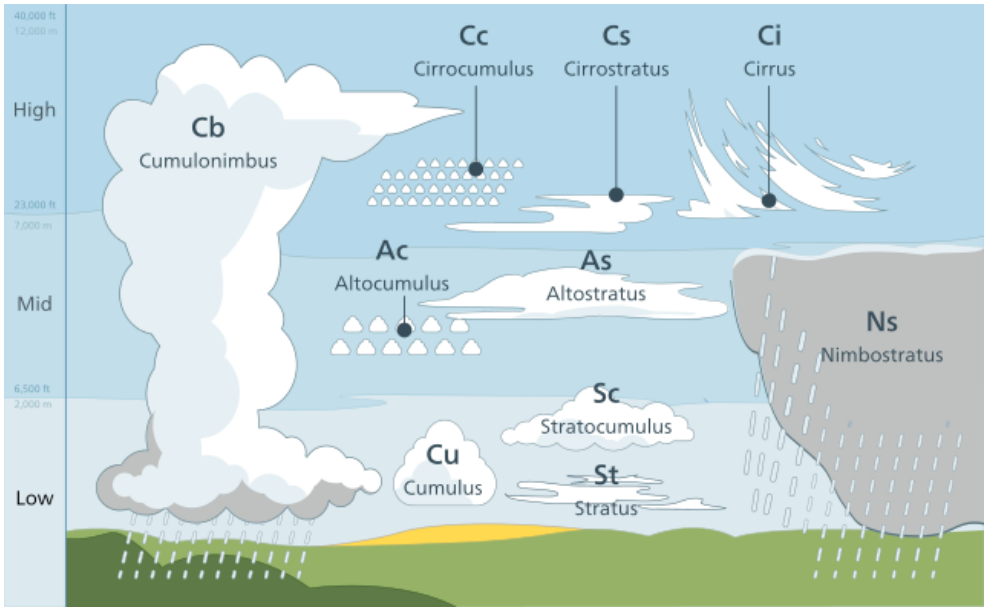


Figura 3.1: Tipos de nubes.

Nubes de altitud baja

Desde la superficie de la Tierra hasta una altitud de 2 km podemos encontrar cuatro tipos de nubes:

Los **estratos** (**St**) se caracterizan por ser capas horizontales con una base uniforme y pueden generar llovizna, nieve o gránulos de nieve. Cuando el Sol es visible a través de las nubes, su contorno también lo es. Aunque no producen el efecto de halo, excepto a temperaturas muy bajas.

A las nubes abundantes y densas, con una extensión vertical que puede llegar hasta los 12 km de altura, se les denomina **cumulonimbos** (**Cb**). Parte de su porción superior es lisa, fibrosa o estriada y, normalmente, aplanada. Esta parte casi siempre se expande en forma de yunque. Frecuentemente, en la base de esta nube, la cual suele ser muy oscura, hay nubes bajas o irregulares que, a veces, pueden estar fusionadas con ella.

Por JACLOU-DL en Pixabay

Figura 3.2: Estrato.

Por marcelkessler en Pixabay

Figura 3.3: Cumulonimbos.

Los **estratocúmulos** (**Sc**) son nubes de color gris y/o blanquecino que, generalmente, se pueden ver en grupos, líneas u ondas y casi siempre tienen partes oscuras. Están compuestas por mosaicos o masas redondeadas y tienen un aspecto no fibroso, además de poder estar o no fusionadas entre sí.

Un **cúmulo** (**Cu**) es una nube densa y con contornos marcados, que se desarrolla verticalmente en forma de montículos, cúpulas o torres. Las partes iluminadas son casi de un blanco brillante, mientras que su base es oscura y casi horizontal.

Por PIX1861 en Pixabay

Figura 3.4: Estratocúmulo.

CC BY-SA 2.0 de wikimedia

Figura 3.5: Cúmulo.

Nubes de altitud media

Entre los 2 y 7 km podemos encontrar tres tipos de nubes:

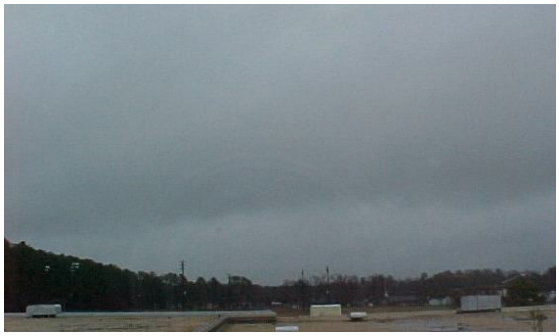
Los **altocúmulos** (**Ac**) son nubes de color gris y/o blanquecino caracterizadas en capas o parches, generalmente con sombreado y compuestas por mosaicos o masas redondas. A veces tienen aspecto parcialmente fibroso o difuso y pueden o no estar fusionadas.

En cuanto al **altostrato** (**As**) podemos decir que son nubes o capas grisáceas o azuladas de apariencia estriada, fibrosa o uniforme. Pueden cubrir parcial o totalmente el cielo, y tienen partes lo suficientemente delgadas para dejar ver el Sol como a través de un vidrio esmerilado. No producen el efecto de halo.

Figura 3.6: Altocúmulo.

Figura 3.7: Altostrato.

Las nubes **nimboestratos** (**Ns**) son una capa grisácea, muchas veces oscura, cuya apariencia es difusa por la lluvia o nieve que les acompaña. Además, son lo suficiente gruesas como para tapar el Sol. Puede haber nubes bajas e irregulares debajo de la capa y estar fusionadas con ella o no.



Dominio público wikimedia

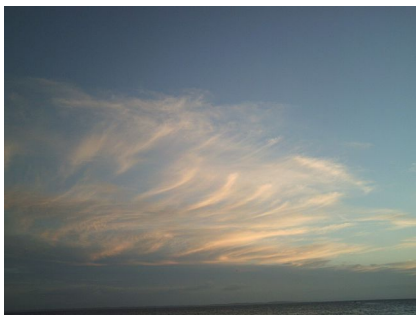
Figura 3.8: Nimboestrato.

Nubes de altitud alta

Desde los 5 hasta los 13 km podemos encontrar tres tipos diferentes de nubes:

Las nubes denominadas **cirros** (**Ci**) son nubes separadas en forma de filamentos blancos delicados o, principalmente, manchas blancas o bandas estrechas. Estas nubes tienen una apariencia fibrosa (como el pelo) y/o un brillo sedoso.

Los **cirrocúmulos** (**Cc**) son parches delgados blancos o capas de nubes sin sombreado, compuestas de elementos muy pequeños en forma de granos, ondas, etc. Pueden estar juntas o separadas y más o menos organizadas.



CC0 wikimedia

Figura 3.9: Cirros.

CC BY-SA 3.0 wikimedia

Figura 3.10: Cirrocúmulo.

Por último, tenemos los **cirrostratos** (**Cs**) que son velos de nubes transparentes y blanquecinas de aspecto fibroso (similar al cabello) o liso. Suelen cubrir total o parcialmente el cielo y, a menudo, producen el efecto de halo.

Dominio público, wikimedia

Figura 3.11: Cirrostrato.

3.1.3. Iluminación de las nubes

Cuando un rayo de luz proveniente del Sol entra en una nube ocurren diversos efectos que derivan en un cambio de radiancia en el mismo. Los fotones del rayo interactúan con las partículas que forman la nube y en esta interacción pueden:

1. Ser absorbidos por las partículas.
2. Salir de la nube e ir hacia nuestros ojos. Esto hace que destaquen los bordes de la nube, como se puede ver en la Figura 3.13.
3. Salir de la nube pero sin llegar a nuestra vista, esparciéndose fuera de ella. Esto crea bordes oscuros en la nubes dejando ver la forma esponjosa que tienen como los de la Figura 3.14.

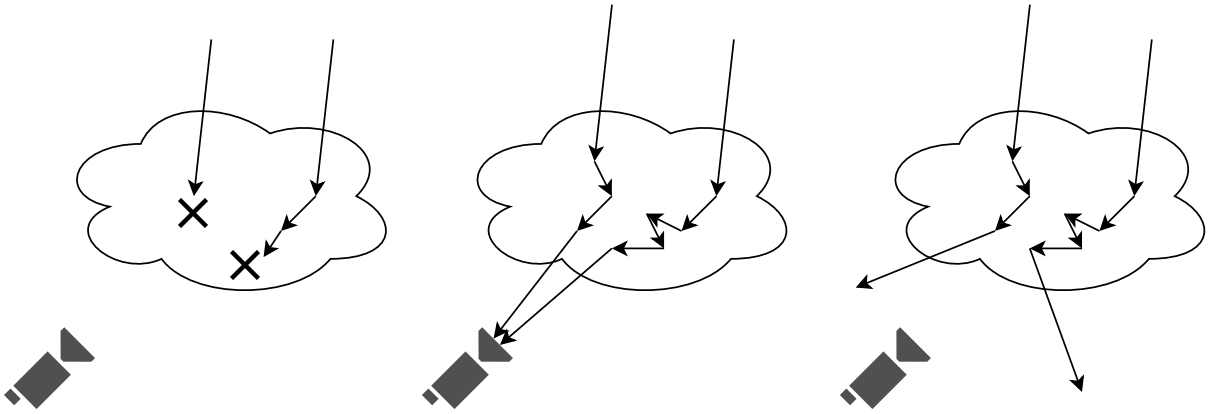


Figura 3.12: Ejemplos de las tres interacciones que le puede ocurrir a un rayo de luz al atravesar una nube: absorción (izquierda), dispersión hacia al punto de vista (centro), dispersión fuera del punto de vista (derecha).

PublicDomainPictures en Pixabay

Figura 3.13: Nubes destacando su borde.

zsoravecz en Pixabay

Figura 3.14: Nubes con bordes oscuros dejando ver su forma esponjosa.

En la Figura 3.12 se pueden ver las tres posibilidades descritas anteriormente.

Para más información sobre iluminación consúltese el Anexo D.

3.2. Ruido

En la generación de nubes vamos a utilizar tres texturas procedurales. Estas texturas han sido generadas mediante varias funciones de ruido y, a continuación, vamos a explicar brevemente cada una de ellas.

3.2.1. Fractal Brownian Motion

El movimiento Browniano es un movimiento donde la posición de un objeto cambia a lo largo del tiempo en incrementos aleatorios. Estos movimientos definen un camino aleatorio pero autosimilar. En cuanto al **fBm**, este es una generalización del movimiento Browniano. Es un proceso similar con la diferencia de que los incrementos no son completamente aleatorios entre sí, sino que tienen un tipo de memoria en el proceso.

En la geometría fractal, la autosimilitud es muy útil para modelar cualquier fenómeno natural, ya sean desde nubes o hasta montañas. De hecho, podemos observar que las formas en la naturaleza se pueden descomponer en varias partes similares que aportan la forma global, otras partes más pequeñas que distorsionan esta forma y partes similares más y más pequeñas que aportan más detalle a la forma original. Esta es la idea principal del **fBm**: comienza con una señal de ruido básica utilizando cualquier tipo de método que la genere (p. ej. Perlin, Worley, Voronoi o Simplex) y va añadiendo continuamente más y más detalle. Lo podemos ver en la siguiente formula descrita en [38][1]:

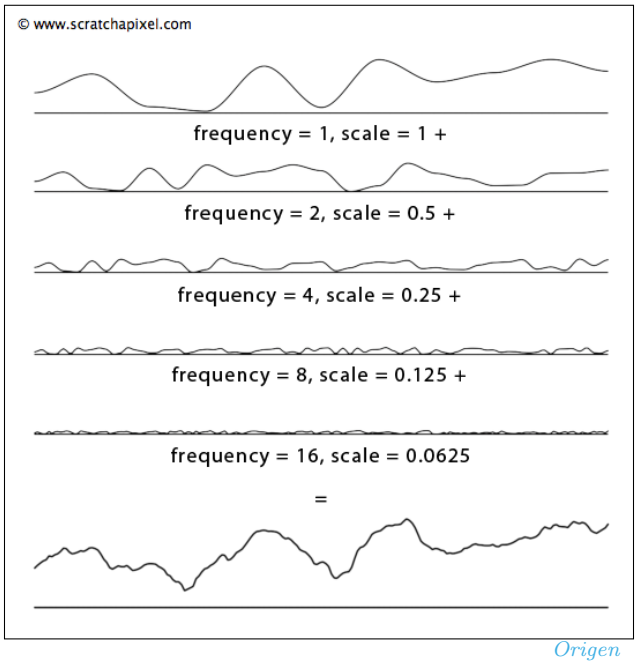
$$\boxed{fBm(x,H) = \sum_{i=0}^{n} (2^i)^{-H} noise(2^i x)} \tag{3.1}$$

Una formula equivalente pero más eficiente a la hora de implementarla es la siguiente:

[1] Origen: https://www.iquilezles.org/www/articles/fbm/fbm.htm.

$$fBm(x,H) = \sum_{i=0}^{n}(a_i)noise(f_i x)$$
$$siendo$$
$$a_0 = 1, a_i = Ga_{i-1},$$
$$f_0 = 1, f_i = 2f_{i-1},$$
$$G = 2^{-H} \tag{3.2}$$

La suma del ruido se le denomina **octava**. Como podemos ver en la ecuación 3.2, se duplica la **frecuencia** (f) por cada iteración, a esto se le suele denominar **lacunaridad**, y se multiplica la amplitud (a) por el parámetro G, denominado **ganancia** y cuyo valor suele ser 1/2. Es decir, por cada octava duplicamos la frecuencia y dividimos entre dos la amplitud. En la Figura 3.15 se puede ver un ejemplo de este proceso.



Origen

Figura 3.15: Ejemplo de **fBm** en 1D.

3.2.2. Ruido de Perlin

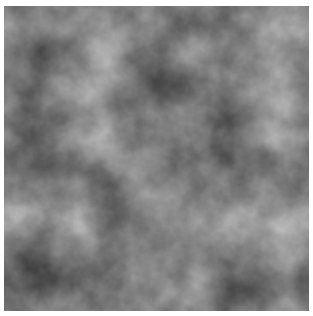


Figura 3.16: Ejemplo de ruido de Perlin en 2D de 5 octavas.

El ruido de Perlin es una función de ruido **basado en celdas** presentado en 1985 por Ken Perlin [34]. Desde entonces se ha utilizado esta técnica para la generación de todo tipo de fenómenos naturales, incluidas las nubes. Para generarlo, por ejemplo en 2D, se divide una cuadrícula en varias celdas. Por cada intersección de las celdas se crea un gradiente. Estos gradientes son vectores normalizados (la dimensión corresponde al de la entrada) y se generan aleatoriamente. No obstante, generar los gradientes de esta forma daba lugar a varios problemas, por lo que en 2002 se sugirió otra alternativa [35].

Cuando a la función le llega una entrada tiene que devolver un número real. Para ello, se suele interpolar entre el punto de entrada y los cuatro valores de las intersecciones de las celdas. El problema es que no tenemos valores sino vectores. Por lo tanto, Perlin sugirió calcular la distancia entre el punto de entrada y los gradientes y, de esta forma, obtener varios vectores. En la Figura 3.17 podemos ver un ejemplo de cuatro gradientes y de cuatro vectores de distancia en una celda. Después, se realiza un producto escalar entre los gradientes y los vectores para obtener un número real por cada uno, quedando cuatro valores (en 2D). Por último, se interpolan dichos valores para obtener el resultado.

Figura 3.17: Ejemplo de gradientes (izquierda) y de vectores de distancia (derecha) en una celda. *Origen*.

3.2.3. Ruido de Worley

El ruido de Worley (también llamado ruido celular) es un método para generar ruido basado en puntos que fue introducido en 1996 por Steven Worley [50]. En este caso, también se utiliza una cuadrícula separada en varias celdas. La idea principal es seleccionar un punto aleatorio por cada celda, haciendo que ese punto sea el destacado de la celda en la que está contenido. Después, por cada posición se calcula la distancia al punto destacado más cercano.

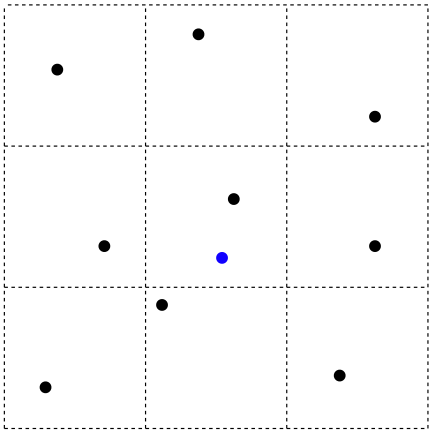


Figura 3.18: Ejemplo de una cuadricula del ruido de Worley. Los puntos negros son los puntos destacados, uno por cada celda, y el punto azul es la muestra actual. Solamente se va a calcular la distancia hacia estos puntos.

Para que el algoritmo sea más eficiente solo se calcula la distancia de las celdas que rodea a la celda en la que está contenido el punto[2]; es decir, 8 celdas en el caso de 2D y 26 celdas en el caso de 3D. Una vez obtenida la distancia al punto más cercano, se normaliza al rango $[0, 1]$. De esta forma podemos obtener un resultado como el de la Figura 3.19.

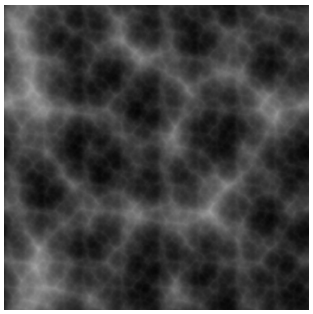


Figura 3.19: Ejemplo del ruido de Worley en 2D de 5 octavas.

En nuestro proyecto, para poder utilizar este tipo de ruido, necesitamos que se asemeje más a las nubes, por lo que tenemos que invertir el ruido, a fin de obtener un patrón ondulante similar al que podemos ver en la Figura 3.20.

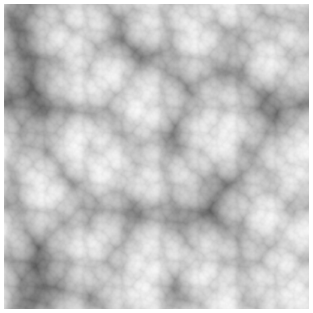


Figura 3.20: Ejemplo de ruido de Worley inverso en 2D de 5 octavas.

[2]Para que el algoritmo sea periódico, siempre calcularemos la distancia como si le rodeasen 8 ó 26 celdas.

Capítulo 4

Diseño

En este capítulo se presenta el modelo que se va a usar para la generación de nubes. El modelo se basa en el presentado en la tesis de Häggström [12], el cual se basó en [41], [40] y [42] entre otros.

Empezamos explicando brevemente las diferentes partes de nuestro diseño. Después se explican cada parte por separado, comenzando por el modelado de las nubes y terminando con el modelo de iluminación.

4.1. Partes del diseño

Nuestro diseño se puede dividir en cuatro partes como podemos observar en la Figura 4.1: el **mapa climático** que describe donde van a estar colocadas las nubes, con cuanta altura y con qué densidad; el **modelo de la nube** que modifica la forma, la altura y la densidad de las nubes; el **modelo de iluminación** que, como su nombre indica, trata sobre la iluminación de las nubes y, finalmente, el algoritmo de ***ray marching*** que utilizamos para visualizar las nubes.

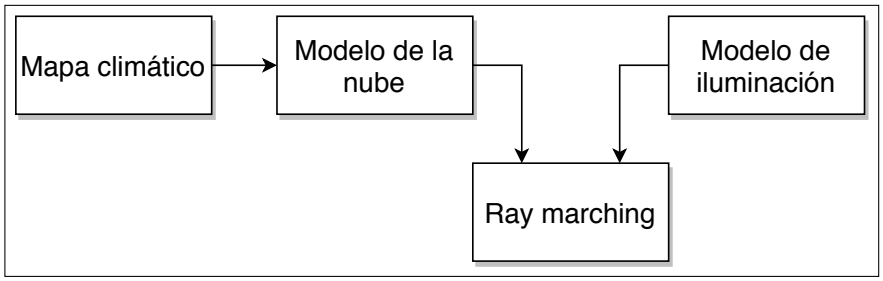


Figura 4.1: Estructura general del diseño.

4.2. Parámetros y métodos generales

En este apartado presentamos varios parámetros y funciones que se van a utilizar a lo largo de este capítulo. Primero explicaremos los parámetros que se utilizan para controlar la altura, la nubosidad y la densidad de las nubes y después las funciones que se van a utilizar más a menudo.

4.2.1. Parámetros generales

Tenemos cuatro parámetros para controlar dónde, cómo y con qué densidad se va a visualizar la nube:

- Altura base (al_b): Es la distancia mínima donde van a empezar a generarse las nubes.
- Altura máxima (al_m): Es la altura máxima hasta donde pueden generarse las nubes.
- Nubosidad global ($n_g \in [0, 1]$): Este parámetro controla cuánto cubren el cielo donde 0 significa que no cubre nada y 1 lo cubre todo.
- Densidad global ($d_g \in [0, \infty]$): Este parámetro controla la opacidad global de las nubes.

4.2.2. Métodos generales

A lo largo del diseño se van a utilizar muy a menudo varias funciones. En este apartado vamos a definir brevemente cada una de ellas.

La primera función es la función de remapeo, *Remapear*(), que convierte un valor de un rango a otro rango distinto:

$$Remapear(v, i_a, f_a, i_n, f_n) = i_n + \frac{(v - i_a)(f_n - i_n)}{f_a - i_a} \tag{4.1}$$

siendo $v \in \mathbb{R}$ el valor que queremos transformar, $i_a \in \mathbb{R}$ el valor inicial del rango anterior, $f_a \in \mathbb{R}$ el valor final del rango anterior, $i_n \in \mathbb{R}$ el valor inicial del rango nuevo y $f_n \in \mathbb{R}$ el valor final del rango nuevo.

La función de saturación, *Sat*(), que limita un valor ($v \in \mathbb{R}$) al rango $[0, 1]$.

$$Sat(v) = \begin{cases} 0 & \text{si} \quad v < 0 \\ 1 & \text{si} \quad v > 1 \\ v & \text{c.c.} \end{cases} \tag{4.2}$$

La función *Lerp*() devuelve el valor de una interpolación lineal entre dos valores $v_0 \in \mathbb{R}$ y $v_1 \in \mathbb{R}$ para un valor del parámetro $i_{val} \in \mathbb{R}$.

$$Lerp(v_0, v_1, i_{val}) = (1 - i_{val})v_0 + i_{val}v_1 \tag{4.3}$$

Para obtener en qué porcentaje de la altura se encuentra la muestra actual se utiliza la siguiente función, *MuestraAltura*(), siendo $p \in \mathbb{R}^3$ la posición en 3D de la muestra:

$$MuestraAltura(p) = Sat\left(\frac{p.z - al_b}{al_m - al_b}\right) \tag{4.4}$$

Para abreviar la notación, en nuestro diseño utilizaremos el parámetro pa_m que define a qué altura se encuentra la muestra actual (p. ej. 0 % si está en la base, 50 % si está a mitad de altura o 100 % si ha llegado al máximo).

4.3. Mapa climático

El mapa climático es una textura de dos dimensiones que define en qué zonas de la escena van a aparecer las nubes. Aunque se pueden utilizar mapas de diferentes tamaños, en nuestro diseño vamos a utilizar un mapa de 512×512 **píxeles (*píxels*)** con información en los cuatro canales de color RGBA. Cada canal contiene diferentes parámetros de las nubes: el canal rojo (R) y el canal verde (G) contienen la información de dónde pueden aparecer las nubes, siendo R ($mc_{n0} \in [0, 1]$) el valor que indica cuándo hay baja nubosidad y G ($mc_{n1} \in [0, 1]$) el valor que indica cuándo hay alta nubosidad. El canal azul (B) contiene la información sobre la altura máxima que pueden alcanzar las nubes ($mc_a \in [0, 1]$) y el canal alfa (A) contiene los valores de densidad ($mc_d \in [0, 1]$).

Los dos primeros canales se puede generar mediante funciones de ruido o manualmente con un programa de edición de imágenes para poder dar la apariencia deseada. Podemos tener varios tipos de mapas que definen diferentes tipos de nubes. En el capítulo 6 se muestra un ejemplo de una textura de este tipo así como el resultado obtenido.

La probabilidad de dónde pueden aparecer las nubes en la muestra actual se calcula mediante la siguiente función:

$$Nubosidad() = Lerp(mc_{n0}, mc_{n1}, n_g) \tag{4.5}$$

4.4. Modelo de la nube

Como ya hemos explicado, con el uso del mapa climático obtenemos dónde pueden estar colocadas las nubes. Mediante las siguientes funciones determinamos la altura, densidad, forma y detalle de las nubes.

4.4.1. Altura

Para que las nubes no tengan bordes cortantes se utilizan dos funciones que dependen de la altura. La primera se utiliza para redondear la nube hacia abajo, *RedondearAbajo*(). Para ello, se remapea el valor pa_m para que, cuando esté entre el 0 % y 7 % de altura, estos valores se conviertan al rango entre 0 % y 100 %.

$$RedondearAbajo() = Sat(Remapear(pa_m, 0, 0.07, 0.0, 1.0)) \tag{4.6}$$

La segunda función redondea la nube hacia arriba, *RedondearArriba*(). En este caso, se remapea el valor pa_m para que, cuando esté entre $mc_a \cdot 0.2$ y mc_a, estos valores se conviertan al rango 100 % y 0 %.

$$RedondearArriba() = Sat(Remapear(pa_m, mc_a \cdot 0.2, mc_a, 1.0, 0.0)) \tag{4.7}$$

Finalmente, la función *Altura*() se obtiene a partir de la multiplicación de los resultados de las anteriores funciones.

$$Altura() = RedondearAbajo() \cdot RedondearArriba() \tag{4.8}$$

Cuando incluimos el valor de la altura máxima del mapa climático, conseguimos que cuando el valor esté cerca de 1 tengamos nubes de mucha altura y, al revés, cuando el valor se acerque al 0 tengamos nubes de poca altura.

4.4.2. Densidad

Las siguientes dos funciones están relacionadas con la densidad y se utilizan para que las nubes sean más esponjosas en la parte baja y tengan una forma más definida en la parte superior.

La primera función, $DensidadAbajo()$, se usa para reducir la densidad hacia la parte de abajo de la siguiente forma: cuando pa_m se encuentra entre 0 % y 15 %, el valor crece lentamente hasta alcanzar el 15 % y, a partir de ese valor, crece linealmente según pa_m hasta llegar al 100 %.

$$DensidadAbajo() = pa_m \cdot Sat(Remapear(pa_m, 0.0, 0.15, 0.0, 1.0)) \tag{4.9}$$

La segunda función, $DensidadArriba()$, reduce la densidad cuando pa_m está entre el 90 % y 100 %, y convierte los valores de ese rango al rango 100 % y 0 %. De esta forma, se consigue una transición más definida en la parte superior de la nube.

$$DensidadArriba() = Sat(Remapear(pa_m, 0.9, 1.0, 1.0, 0.0)) \tag{4.10}$$

Para obtener la función $Densidad()$ se combinan los resultados de las dos funciones anteriores, la densidad global y la densidad del mapa climático como se muestra a continuación:

$$Densidad() = d_g \cdot DensidadAbajo() \cdot DensidadArriba() \cdot mc_d \cdot 2.0 \tag{4.11}$$

Nótese que se multiplica por dos la ecuación anterior a fin de crear nubes con más densidad cuando $mc_d > 0.5$.

4.4.3. Forma y detalle

Por ahora hemos definido dónde y con qué altura y densidad se genera la nube. No obstante, solamente con la utilización del mapa climático, la densidad ha sido uniforme sin añadir ningún tipo de características típicas de las nubes. Para lograr el modelo de nube básico se utilizan dos texturas 3D: una para dar forma a la nube y otra para dar detalle. La primera textura tiene valores en los cuatro canales de color RGBA con una resolución de $128 \times 128 \times 128$ **píxeles (*píxels*)**. En los cuatro canales se va incrementando la cantidad de celdas, comenzando con unas pocas desde el canal rojo. El canal R ($f_r \in [0, 1]$) contiene la información de un ruido de Perlin-Worley; esto es, una mezcla del ruido de Perlin y del ruido de Worley. Para los siguientes tres canales GBA ($f_g, f_b, f_a \in [0, 1]$), se utilizan tres ruidos de Worley cada uno con más celdas que en el anterior.

El primer canal se utiliza para dar la forma base a la nube. No obstante, esto no es suficiente para darle una apariencia detallada y, por lo tanto, se utiliza una combinación de los otros tres canales y una función de remapeo para añadir dicha combinación al ruido de Perlin-Worley. De esta forma, se previene que el interior de la nube sea no homogéneo y se añade detalle solo en las zonas que se van a ver (Ver Figura 4.2).

$$FormaBase() = Remapear(f_r, (f_g \cdot 0.625 + f_b \cdot 0.25 + f_a \cdot 0.125) - 1.0, 1.0, 0.0, 1.0) \quad (4.12)$$

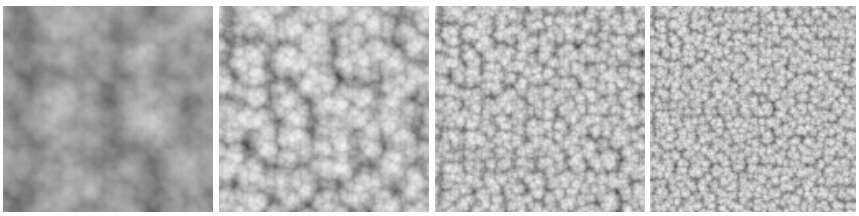


Figura 4.2: Ejemplo de los cuatro canales de la textura, en 2D, para la generación de la forma básica de la nube.

Para crear la forma básica de la nube combinamos la función de nubosidad de la sección 4.3 y la función de altura 4.4.1 junto con el parámetro global n_g para que afecte a la nubosidad si se modifica este parámetro.

$$Forma() = Sat(Remapear(FormaBase() \cdot Altura(), 1.0 - n_g \cdot Nubosidad(), 1.0, 0.0, 1.0)) \tag{4.13}$$

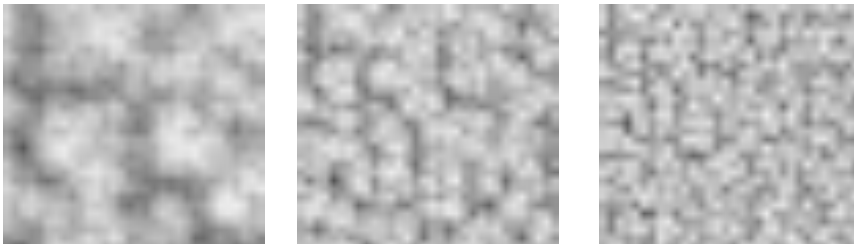


Figura 4.3: Ejemplo de los tres canales de la textura de detalle en 2D.

Para dar más detalle a la nube se utilizan los tres canales RGB de otra textura 3D de una resolución de $32 \times 32 \times 32$. Esta textura contiene tres ruidos de Worley con frecuencia incremental (Ver Figura 4.3). Los valores RGB ($d_r, d_g, d_b \in [0, 1]$) se combinan mediante la siguiente función:

$$DetalleBase() = d_r \cdot 0.625 + d_g \cdot 0.25 + d_b \cdot 0.125 \tag{4.14}$$

La función 4.15 se utiliza para modificar el detalle base. Se reduce la influencia que pueda tener a un máximo de 0.35 y se combina con $e^{n_g * 0.75}$ para tener en cuenta la nubosidad global. Con la interpolación lineal se logra que la nube sea más esponjosa en la base y más ondulante en la parte de arriba.

$$Detalle() = 0.35 \cdot e^{n_g \cdot 0.75} \cdot Lerp(DetalleBase(), 1.0 - DetalleBase, Sat(pa_m \cdot 5.0)) \tag{4.15}$$

Finalmente, se utiliza la siguiente función para obtener la densidad total. Para ello se combinan las funciones vistas anteriormente: la función de forma, la de detalle y la de densidad (véase la Sección 4.4.2).

$$DensidadTotal() = Sat(Remapear(Forma(), Detalle(), 1.0, 0.0, 1.0)) \cdot Densidad() \quad (4.16)$$

4.5. Renderizado de las nubes

Una vez presentados los métodos para calcular la forma, densidad y detalle de la nube, vamos a describir cómo vamos a visualizarlas. En general, la técnica más utilizada es el renderizado de polígonos mediante *shaders*. Pero esta técnica no es muy apropiada cuando queremos simular fenómenos con densidad variable y con formas fractales, ya que este tipo de objetos no son tan fáciles de representar mediante modelos poligonales. Por lo tanto, en estos casos se suelen utilizar otras técnicas de renderizado.

Una de estas técnicas es el método conocido como *ray marching*, el cual lanza rayos desde la perspectiva de la cámara y calcula a qué distancia se encuentra el objeto a renderizar en cada instante de tiempo. Si el rayo llega a una distancia mínima, se toma como que el rayo ha impactado con el objeto. Por cada rayo se calcula el color del **píxel (*pixel*)** correspondiente.

Para nuestro objetivo, necesitamos definir tres elementos en la escena: el área donde se van a mostrar las nubes, definido por un cubo o dos esferas, la cámara y la información de la fuente de iluminación (en nuestro caso, la posición y el color del Sol). Dicha información, junto con las texturas, es la que recibe el *shader* el cual también utiliza las funciones definidas en las secciones anteriores. Además, utilizaremos la técnica de *ray marching* para calcular la densidad en cada **píxel (*pixel*)**.

4.5.1. Ray Marching

Para el caso de las nubes, el algoritmo de *ray marching* lanza un rayo desde la perspectiva de la cámara por cada **píxel (*pixel*)** y, en cada punto, se calcula la densidad mediante las funciones definidas en las secciones anteriores. La densidad es acumulativa y el proceso termina cuando se realiza para un número fijo de muestras. Además, por cada densidad que no sea nula se lanza otro rayo hacia la posición del Sol para obtener la densidad entre el punto y la luz. Al final, el valor de densidad total acumulado en el rayo se utiliza para el canal alfa del **píxel (*pixel*)**, mientras que los canales RGB dependen del color de la luz y de la densidad acumulada de los rayos entre las muestras y la luz (este último cálculo se explica en la sección 4.6).

4.5.2. Optimizaciones del algoritmo

Este algoritmo, aunque sencillo, es muy costoso como ya se menciona en [12]. La parte más importante y más delicada de este algoritmo se corresponde con el proceso de optimización a fin de aumentar el rendimiento sin perjudicar a la calidad final de las nubes. En este trabajo se van a desarrollar algunas de las optimizaciones que se plantean en la literatura y que se explican a continuación.

4.5.2.1. Intervalo del rayo

Sabemos que las nubes se van a renderizar es un espacio limitado, ya sea en el interior de un cubo o entre dos esferas. Por lo tanto, podemos predecir la trayectoria del rayo (de donde a donde va) en vez de calcularlo desde la cámara. Esto es, si calculamos la intersección[1] entre rayo-cubo o rayo-esfera, según la geometría escogida, podemos determinar el inicio y final del rayo.

[1] Para más información consúltese el Anexo C.

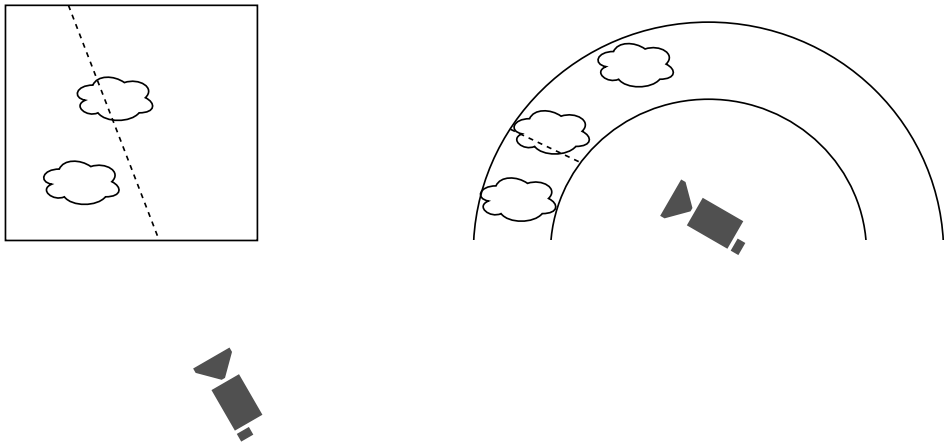


Figura 4.4: Optimización del proceso de *ray marching*: calculamos el punto de inicio y el punto final del área donde se van a mostrar las nubes; a la izquierda, para un cubo y a la derecha, para dos esferas.

4.5.2.2. Número de pasos por cada muestra

Otra de las optimizaciones que planteamos, consiste en calcular cuánta distancia se va a dar entre muestra y muestra. En el caso de realizar pasos pequeños conseguiremos nubes bien definidas pero con un coste computacional muy alto y, al contrario, si utilizamos una distancia grande conseguiremos un mejor rendimiento a costa de la calidad, además de producir bandeado y ruido. Por todo ello, trataremos de que las nubes que estén más cerca de la cámara se rendericen con mayor calidad y que las más lejanas, se visualicen con menor calidad. La función 4.17 implementa esta mejora donde $paso_{longit}$ es el número de pasos constante, $incr_{dist}$ es una constante del incremento de la distancia y $dist_{inicio}$ son los pasos entre muestra y muestra que dependen de la distancia a la que se haya iniciado el rayo.

$$paso = paso_{longit} + dist_{inicio} \cdot incr_{dist} \tag{4.17}$$

De igual forma, y siguiendo con otras optimizaciones similares, cuando la muestra está fuera de una nube se toman pasos más largos. Por el contrario, si nos encontramos muestreando

dentro de la nube se toman pasos más pequeños. Después de salir de una nube, se esperan ciertos pasos hasta volver a dar pasos más grandes.

Por otra parte, en la escena, solo se va a ver la parte que rodea la nube, por lo tanto, cuando se den ciertos pasos dentro de la nube, solo se va a calcular la forma, pero no el detalle. Además, cuando lleguemos al valor de densidad máximo, el rayo se detiene en ese punto.

4.6. Modelo de iluminación

Durante el proceso de *ray marching* por cada densidad no nula, se lanza un rayo hacia el Sol. Para ese rayo se calcula la densidad entre el punto de la muestra y el Sol ya que se necesita esta información para los cálculos de la iluminación. Por ello, en este caso, calculamos los puntos de muestra en un cono en vez de a través del rayo, es decir, compensamos el punto de la muestra con un vector de ruido entre $(-1, -1, -1)$ y $(1, 1, 1)$. De esta forma, consideramos que la iluminación de ese punto se ve afectada por un área cercana a la dirección de la luz.

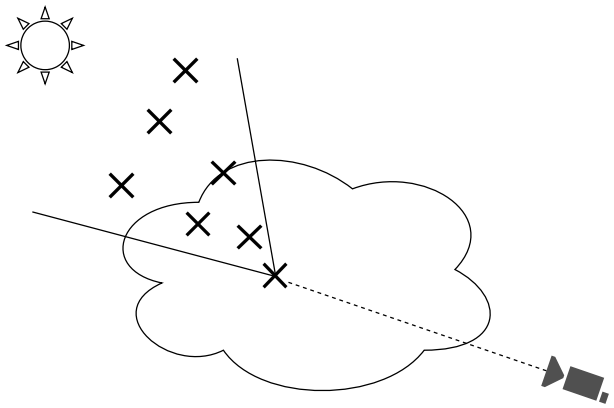


Figura 4.5: Un ejemplo de una muestra de iluminación en el área de un cono.

Aunque esto no es más que una aproximación a la iluminación real presentada en la sección 3.1.3, la dispersión de la luz dentro de la nube se pierde completamente. Para aproximar la

iluminación utilizamos varias funciones. La primera y principal es una versión simplificada de la ley de Beer [2] que se utiliza para el cálculo de la atenuación de la luz. Para obtener el efecto de los bordes brillantes de las nubes, se utiliza la función de Henyey-Greentein [14]. Y, por último, usamos la inversa de la ley de Beer para obtener los bordes oscuros que se aprecian en la superficie de la nube.

4.6.1. Ley de Beer

Como ya hemos comentado previamente, la componente principal del modelo de iluminación es una versión simplificada de la ley de Beer que, originalmente, se utilizaba para la química analítica y que modela la atenuación de la luz cuando esta pasa a través de un material.

La función simplificada se define de la siguiente manera:

$$A(b, d_s) = e^{-b \cdot d_s} \tag{4.18}$$

siendo $d_s \in [0, \infty)$ la densidad acumulada hacia el Sol y $b \in [0, \infty)$ la cantidad de absorción de la luz.

4.6.2. Función de fase de Henyey-Greenstein

En las nubes hay más probabilidad de que haya dispersión frontal, esto es lo que crea el efecto de los bordes brillantes comentado en la sección 3.1.3 y que se puede observar en la Figura 3.14.

En 1941, Henyey y Greenstein [14] desarrollaron una función para imitar la dependencia angular de la luz dispersada por pequeñas partículas. Fue utilizada para describir la dispersión de luz por nubes de polvo interestelar, aunque dada su simplicidad se puede utilizar

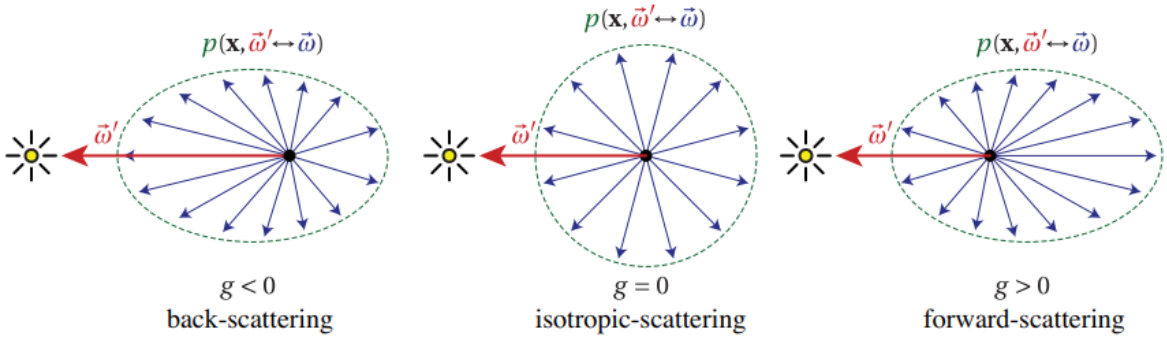


Figura 4.6: Tres tipos de dispersión: en el caso simple, la luz se dispersa equitativamente en todas las direcciones (*isotropic-scattering*). Aunque otros materiales naturales se dispersan bien delante (*forward-scattering*) o atrás (*back-scattering*) [17].

para simular la dispersión de otros medios, incluyendo las nubes. La función solo depende del ángulo θ (el ángulo entre la dirección del Sol y la dirección del rayo de la cámara) y se define como:

$$p_{HG}(\theta, g) = \frac{1 - g^2}{4\pi(1 + g^2 - 2g\,cos\theta)^{1.5}} \tag{4.19}$$

El término $g \in [-1, 1]$ representa la dispersión posterior (*back-scattering*) para $g < 0$, la dispersión isotropica (*isotropic-scattering*) para $g = 0$ y la dispersión frontal (*forward-scattering*) para $g > 0$.

Se puede añadir un término extra para controlar la intensidad alrededor del Sol:

$$IS_{extra}(\theta) = cs_i \cdot Saturar(\theta)^{cs_e} \tag{4.20}$$

donde cs_i es la cantidad de intensidad extra a añadir alrededor del Sol y cs_e es el exponente que controla como de centrada va a estar esa intensidad.

Finalmente, combinamos la función y el término de la siguiente forma:

$$IOS(\theta) = Lerp(max(p_{HG}(\theta, in_s), IS_{extra}(\theta)), p_{HG}(\theta, -out_s), ivo) \tag{4.21}$$

donde $in_s \in [0, 1]$ es la cantidad de luz que se dispersa hacia la cámara y $out_s \in [0, 1]$ es la cantidad que se dispersa fuera de la cámara. Además, se añade el término $ivo \in [0, 1]$ para poder controlar entre el primer tipo de dispersión y el segundo.

4.6.3. Función de probabilidad In-Scattering

La ley de Beer es un modelo de extinción, es decir que tiene en cuenta la atenuación de la luz basándose en la profundidad y, por lo tanto, no tiene en cuenta el efecto que se puede producir cuando el rayo de la cámara esté cerca de la dirección del rayo del Sol. Este efecto crea unos bordes negros sobre las nubes y suele aparecer en regiones densas y redondas de las mismas. Este es el resultado contrario a lo que la ley de Beer modela, por lo tanto se puede obtener este efecto combinando la ley de Beer y su inversa:

$$L(b, d_s) = 2 \cdot A(b, d_s) \cdot (1 - A(b, d_s \cdot 2)) \tag{4.22}$$

4.6.4. Función final

Finalmente, obtenemos la función final de iluminación combinando estas dos funciones:

$$E = L(b, d_s) \cdot IOS(\theta) \tag{4.23}$$

Capítulo 5

Implementación

Este capítulo trata todos los aspectos relacionados con la implementación. Como ya hemos mencionado en la introducción, utilizamos la librería *Three.js*.

5.1. Fases de la implementación

La implementación se ha dividido en tres fases: generación del algoritmo de *ray marching* y aplicación de las texturas de ruido, modelado de la nube e iluminación de la nube.

5.1.1. Fase 1: Ray marching y texturas de ruido

Los objetivos de esta primera fase son implementar un algoritmo de *ray marching* que muestre un cubo o una esfera y generar las texturas de ruido, además de aplicar estas texturas al volumen mostrado. En la siguiente fase se modificará el algoritmo para poder calcular la densidad de cada **píxel (*pixel*)** como ya hemos explicado en el capítulo 4.

El algoritmo implementado es el siguiente:

Datos:

- *origenRayo*. Un punto 3D desde donde se lanza el rayo.
- *direccionRayo*. Un vector 3D que apunta hacia donde se dirige el rayo.
- *maximoPasos*. El máximo de puntos que se pueden evaluar.
- *maximoDistancia*. La longitud máxima que puede tener el rayo.
- ε. Distancia mínima a la que tiene que estar el punto del volumen para considerarse que han colisionado.

$longitudRayo \leftarrow 0$
$punto \leftarrow (0, 0, 0)$
para $i \leftarrow 0$ **a** $maximoPasos$ **hacer**
  $punto \leftarrow origenRayo + longitudRayo \cdot direccionRayo$
  $distancia \leftarrow FuncionDistancia(punto)$
  $longitudRayo \leftarrow longitudRayo + distancia$
  si $distancia < \varepsilon$ **entonces**
    Devuelve el color del píxel, en nuestro caso se obtiene desde la textura.
  fin
  si $longitudRayo < maximoDistancia$ **entonces**
    Termina y devuelve un color con el canal alfa a 0, es decir, un vector 4D con el último valor a 0.
  fin
fin
Devuelve un color con el canal alfa a 0, es decir, un vector 4D con el último valor a 0.

Este algoritmo se incluye dentro del *fragment shader*. La función $FuncionDistancia()$ devuelve la distancia a la que se encuentra de colisionar el rayo con el volumen a mostrar desde el último punto evaluado.

En el siguiente paso, generamos las texturas de ruido necesarias para dar la forma y el

detalle a la nube, tal y como se explica en la sección 4.4.3. Inicialmente se pensó en utilizar alguna librería de *Three.js* para realizar estas tareas. Pero, ya que no se encontró ninguna librería que incluyera una gran parte de funciones útiles para tal fin, se buscaron otras alternativas tales como:

- FastNoise. Una librería de C++ para la generación de ruido.
- Ambient. Un paquete de R que utiliza la librería FastNoise.
- WebGL-Noise. Generación de varios tipos de ruido escrito en el lenguaje GLSL.

Inicialmente se utilizó el paquete de R, ya que parecía que era la manera más rápida de obtener las texturas. Se consiguió generar una textura de ruido de Worley en 3D y se almacenó en un formato específico para poder ser utilizada posteriormente por el *shader*. Los valores de cada **píxel (*pixel*)** de la textura son, por tanto, una lista de números donde cada uno representa un canal especifico.

Esto es, la primera parte del fichero es una cabecera donde, por cada línea, se definen las características de la textura en este orden: dimensión de la textura, número de canales de color (3 para RGB ó 4 para RGBA), anchura, altura y profundidad. Los datos de la textura se encuentran dispuestos en el orden en el que se van a enviar al *shader*.

Cuando se utilizó una de estas texturas, se detectó un problema. La librería que utilizaba el paquete de R, *FastNoise*, generaba un ruido que no era periódico y, por lo tanto, se descartó dicho paquete.

La siguiente opción era utilizar *WebGL-Noise*, pero no se encontraron buenos ejemplos de cómo generar ruido en 3D, por lo que se descartó también.

La última opción fue modificar la librería *FastNoise* para que generase ruido periódico. Por suerte, indagando en su repositorio, se encontró una modificación hecha por otra persona que resolvía este problema (link). Por lo tanto, utilizando dicha modificación, se generaron las texturas necesarias. En la Figura 5.1 se pueden ver algunos ejemplos en 2D y en la Figura 5.2 podemos apreciar un ejemplo de la textura de forma aplicada a una esfera.

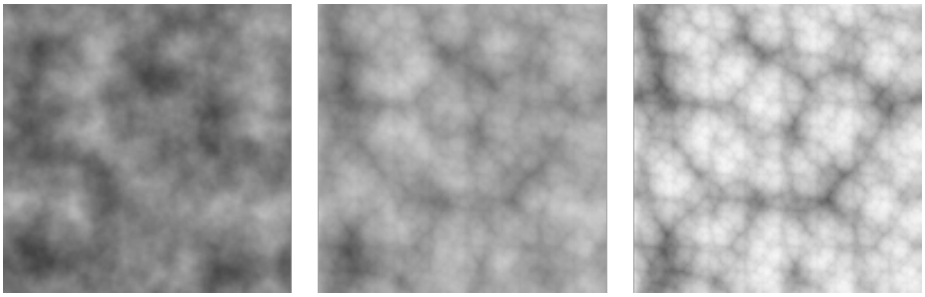


Figura 5.1: Dos tipos de ruido generados por *FastNoise* y una combinación de los dos, de izquierda a derecha: ruido de Perlin, ruido de Perlin-Worley y ruido de Worley.

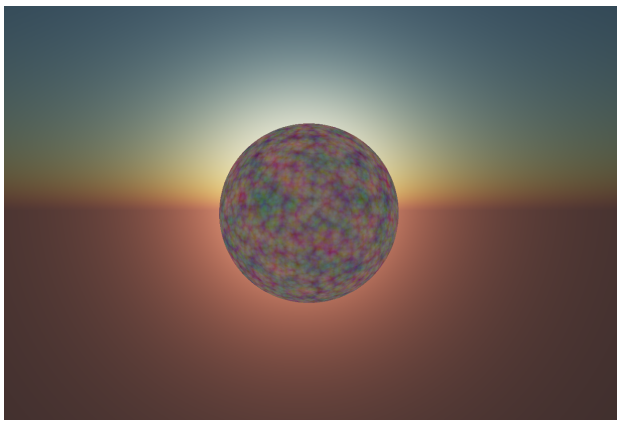


Figura 5.2: Ejemplo de la textura de forma aplicada a una esfera.

5.1.1.1. Carga local de los ficheros

Debido a la **política del mismo origen** de los navegadores, no se permite la carga de texturas u objetos cuando se utiliza de forma local un fichero .html. No obstante, existen diversas soluciones en *Three.js*: una de las opciones es desactivar la seguridad de ficheros locales en el navegador y la otra es crear un servidor web local. El problema de la primera opción es que nos ponemos en riesgo al navegar por la web. Por lo tanto, se ha optado por la segunda opción: utilizar un servidor web local.

5.1.2. Fase 2: Modelado de la nube

En la fase 1 se han generado las texturas de ruido y se han transferido al *shader*, por lo cual, ya podemos empezar a generar las nubes. El objetivo de esta fase es la implementación de las funciones descritas en las secciones 4.3, 4.4.1 y 4.4.3 del capítulo 4, además de añadir la posibilidad de animar las nubes.

En esta fase tenemos que cambiar algunos aspectos del algoritmo de *ray marching* ya que, en nuestra solución no obtenemos la distancia hasta un cierto objeto, sino que, por cada muestra obtenemos la densidad en ese punto. Además, se comienza a calcular desde el primer punto de la intersección[1] del área donde se van a mostrar y se termina en el segundo punto. El algoritmo de *ray marching* resultante se puede observar en 2.

Para comprobar que la modificación del algoritmo del *ray marching* y los algoritmos de intersección funcionaban se mostraron el cubo y la esfera con un color fijo. Después se creó el mapa climática utilizando una herramienta externa (*Gimp*) y se proyectó el primer canal. A continuación se puede observar el mapa climático y la proyección:

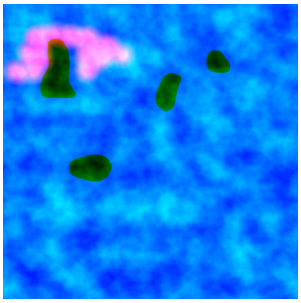


Figura 5.3: Mapa climático.

Figura 5.4: Resultado al mostrar el primer canal del mapa climático.

[1] Las funciones de intersección se implementan como están descritas en el Anexo C.

Datos:

- *direccionRayo*. Un vector 3D que apunta hacia donde se dirige el rayo.
- *inicio*. El punto desde donde se comienzan a calcular las muestras.
- $final$. El punto donde terminan.

$dir \leftarrow final - inicio$
$grueso \leftarrow \|dir\|$
$dir \leftarrow \dfrac{dir}{\|dir\|}$
$muestras \leftarrow 64$
$punto \leftarrow inicio$
$densidadAcumulada \leftarrow 0$
para $i \leftarrow 0$ **a** $muestras$ **hacer**
  $densidad \leftarrow CalcularDdensidad(punto)$[a]
  si $densidadAcumulada > 1$ **entonces**
    Hemos llegado a la densidad máxima, por lo tanto, se terminan las iteraciones
  fin
  $punto \leftarrow punto + \dfrac{grueso}{muestras} \cdot dir$
fin
Devuelve un color con el canal alfa siendo la densidad acumulada

Algoritmo 1: Algoritmo de *ray marching*

[a] La función $CalcularDensidad(punto)$ devuelve la densidad de la nube en ese punto concreto.

El siguiente paso fue implementar las funciones de nubosidad, altura, densidad y de forma, con el siguiente resultado:

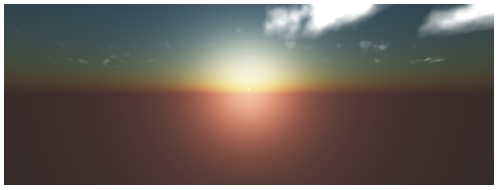


Figura 5.5: Resultado con las funciones de nubosidad, altura, densidad y forma.

Para finalizar la función de densidad, se añadió el detalle:

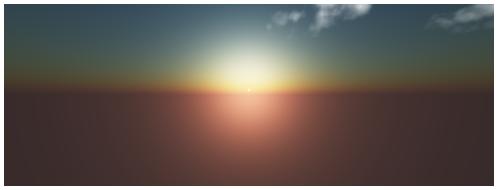


Figura 5.6: Resultado después de incluir la función de detalle.

Con unas cuantas modificaciones se lograron los siguientes resultados:

Figura 5.7: Ejemplo con nubosidad media alta y densidad alta.

Figura 5.8: Ejemplo con nubosidad media y densidad media.

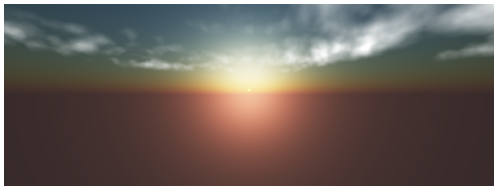


Figura 5.9: Ejemplo con nubosidad media y densidad baja.

Las características de las nubes se pueden modificar en cualquier momento en la aplicación mediante una interfaz (ver anexo E.3 para más información). Por último, se ha incluido la posibilidad de que las nubes se muevan en una cierta dirección y con una cierta velocidad.

5.1.3. Fase 3: Iluminación de la nube

El objetivo de esta fase es implementar las funciones que hemos visto en la sección 4.6. Para ello, tenemos que calcular la densidad entre el punto de la muestra y el Sol, como ya hemos comentado anteriormente. Por lo tanto, implementamos un nuevo algoritmo de *ray marching*, pero con el cálculo de las muestras en el área de un cono. Además, calcularemos un número fijo de muestras para que no afecte al rendimiento y por cada iteración bajamos el nivel de detalle de la textura a uno. El algoritmo resultante se puede observar en 3.

Una vez calculada la densidad de la luz, podemos realizar los cálculos descritos en la sección 4.6. Se realiza una suma por cada cálculo de la luz y se remapea[2] si se supera el rango $[0, 1]$. Existe un problema visual denominado *banding*, aunque se ha solucionado mediante el uso de ruido azul[3].

Para representar el cielo, se utiliza un módulo de *Three.js* llamado *Sky.js*. Primero se realizan los cálculos del cielo y se obtiene una textura que se le pasa al *shader* que realiza los cálculos necesarios para la generación de nubes. A la hora de obtener el color de la nube, se mezcla el color obtenido mediante la iluminación y el color del cielo.

[2] El rango del color del **píxel (*pixel*)** es $[0, 1]$.
[3] Utilizamos las texturas que se proveen en el siguiente link: Moments in Graphics.

Datos:

- *direccionRayo*. Un vector 3D que apunta hacia el Sol.
- *inicio*. El punto desde donde se comienza a calcular la muestra.
- *densidad*. La densidad de la muestra actual.
- *grueso*. Distancia entre el punto inicial y el punto final.

$muestras \leftarrow 6$
$distMuestras \leftarrow \frac{(0.5 * grueso)}{muestras}$
$distancia \leftarrow 0.0$
para $i \leftarrow 0$ **a** $muestras$ **hacer**
  $punto \leftarrow inicio + distancia * dir$
  $distancia+ = distMustras$
  si $densidad < 0.3$ **entonces**
    Calculamos la densidad calculando el detalle.
    $densidad \leftarrow calcularDensidad(punto + vectorAleatorio)$
  fin
  en otro caso
    Calculamos la densidad sin calcular el detalle.
    $densidad \leftarrow calcularDensidadSinDetalle(punto + vectorAleatorio)$
  fin
fin
Devuelve la densidad

Algoritmo 2: Algoritmo de *ray marching* de iluminación

5.2. Programa principal

En esta sección explicamos de forma general cómo funciona la aplicación. En la Figura 5.10 se muestra un esquema general de las dos partes fundamentales de la aplicación: inicialización y animación.

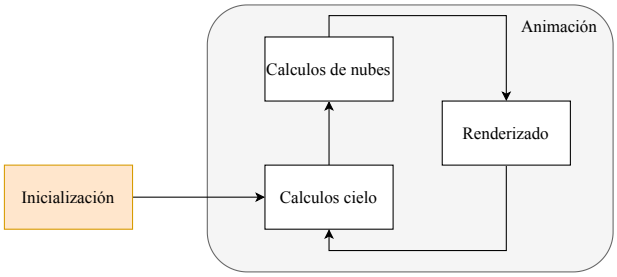


Figura 5.10: Esquema de la aplicación.

Esto es, las dos partes más importantes de la aplicación son:

- Inicialización: Carga las texturas, creación de los *shaders* e inicialización de las variables necesarias.
- Animación: Es el bucle principal del renderizado. Primero se calcula la textura del cielo y después se realizan todos los cálculos de las nubes para finalmente mostrar el resultado por pantalla.

Para más información consúltese el Anexo E.

Capítulo 6

Resultados

En este capítulo se muestran algunos de los resultados obtenidos y se discuten algunos aspectos del rendimiento de la aplicación.

Se puede consultar el código de la aplicación en el siguiente repositorio: https://gitlab.com/Ealaminos/nubes-volumetricas-publico

6.1. Resultados obtenidos

6.1.1. Según el tipo de nube

La configuración del mapa climático para la creación de diferentes tipos de nubes es difícil, por lo tanto, en esta sección mostramos tres tipos de nubes con sus respectivos mapas climáticos y sus parámetros.

Cúmulo

Para crear nubes de tipo cúmulo usamos el siguiente mapa climático y los valores 0.4 y 1.0 para los parámetros de nubosidad y densidad respectivamente.

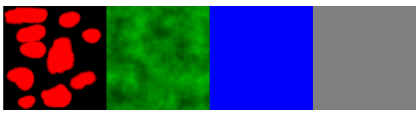


Figura 6.1: Mapa climático usado para nubes tipo cúmulo y estratocúmulo.

Figura 6.2: Nubes tipo cúmulo con nubosidad 0.4 y densidad 1.0.

Estratocúmulo

Para crear nubes de tipo estratocúmulo usamos el mismo mapa climático que el utilizado en la generación de cúmulos y los parámetros de nubosidad y de densidad con valores 0.8 y 0.5 respectivamente.

Figura 6.3: Nubes tipo estratocúmulo con nubosidad 0.8 y densidad 0.5.

Estrato

Para crear nubes de tipo estrato usamos el siguiente mapa climático. En este caso, la configuración del canal de altura tiene en cuenta que las nubes deben ser más pequeñas. Los parámetros nubosidad y densidad se configuran a 1.0 y 0.2 respectivamente.

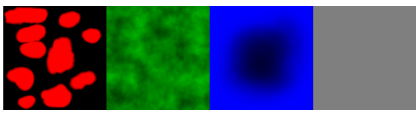


Figura 6.4: Mapa climático usado para nubes tipo estrato.

Figura 6.5: Nubes tipo estrato con nubosidad 1.0 y densidad 0.2.

6.1.2. Iluminación

En esta sección mostramos tres tipos de iluminación con el mapa climático de 6.1.1 y los parámetros de nubosidad y de densidad con valor 0.8. Los parámetros de iluminación para las tres imágenes son las siguientes:

- ***In scatter***. $in_s = 0.2$.
- ***Out scatter***. $out_s = 0.1$.
- ***In o Out Scatter***. $ivo = 0.5$.
- ***Silver intensity***. $cs_i = 3.0$.
- ***Silver exponent***. $cs_e = 2.0$.
- **Absorción de luz**. $b = 6.0$.

Figura 6.6: Iluminación mediodía.

Figura 6.7: Iluminación tarde.

Figura 6.8: Iluminación atardecer.

6.1.3. Animación

En el siguiente vídeo se puede observar un ejemplo de la animación: https://youtu.be/A5XeOjqtLjk .

6.2. Rendimiento

Como ya hemos mencionado a lo largo de esta memoria, los algoritmos estudiados en este trabajo son muy costosos computacionalmente hablando. Por lo tanto, es necesario el uso de varias técnicas de optimización. En nuestro caso, aunque se han implementado varias de ellas, no han sido suficientes para que el renderizado se llevara a cabo en un número de fotogramas aceptable como puede observarse en la Tabla 6.1 en la que se muestran los **frames por segundo** (**FPS**) por cada tipo de nube y para cuatro resoluciones diferentes.

	Resolución			
Tipo de nube	$256 \cdot 256$	$512 \cdot 512$	$800 \cdot 800$	$1920 \cdot 1080$
Cúmulos $Nub = 0.4$ $Den = 1.0$	40 FPS	12 FPS	5-7 FPS	1-2 FPS
Estratocúmulos $Nub = 0.8$ $Den = 0.5$	22 FPS	8 FPS	3-5 FPS	1-2 FPS
Estrato $Nub = 1.0$ $Den = 0.2$	14 FPS	4 FPS	2-3 FPS	1-2 FPS

Tabla 6.1: Fotogramas por segundo por cada tipo de nube y en cuatro resoluciones diferentes.

Si observamos la tabla, podemos ver que cuánto menor es la densidad, peor es el rendimiento. Por lo que una de las posibles técnicas de optimización a incorporar debería orientarse a la generación de nubes poco densas. No obstante, debemos aclarar que la optimización del algoritmo no era uno de los objetivos establecidos en este trabajo.

Capítulo 7

Conclusiones

Una vez se ha completado la aplicación, podemos concluir que se han cumplido los objetivos propuestos. Hemos estudiado una de las técnicas que se usa hoy en día para el modelado de las nubes. Además, hemos creado una implementación de esta misma y, para ello, hemos aprendido a utilizar la librería *Three.js*. No solo eso, sino que también hemos aprendido sobre fractales, funciones de ruido procedural (Perlin y Worley), la función de *ray marching* e iluminación de medios participantes.

En cuanto a la implementación, no se han tenido en cuenta todas las optimizaciones que podrían hacerse. Pero, se ha logrado crear una aplicación que permite simular nubes y cambiar varios parámetros de las mismas: la nubosidad, su densidad, la iluminación, la posición del sol, entre otros. Además de poder animarlas en una dirección y velocidad.

7.1. Lineas futuras

Varias de las posibles mejoras de la aplicación se refieren al renderizado y al rendimiento de la aplicación. En cuanto a la visualización de las nubes, para solucionar el problema de *banding* se ha utilizado ruido azul lo cual genera ruido en varias partes de las nubes. Este

problema se puede percibir más con ciertos valores en los parámetros que con otros. Por lo tanto, se debe estudiar otra manera de solucionar este problema o de conseguir eliminar el ruido que genera.

El rendimiento es una de las mejoras más prioritarias. Para ello, se deben estudiar varias técnicas de optimización para el algoritmo de *ray marching*. Una de las mejoras que se menciona en [12] es el de la reproyección. En este caso, se trata de reducir los **píxeles (*píxels*)** que se renderizan por cada fotograma.

Bibliografía

[1] Sassi Abdessamed, NourEddine DJEDI, and Sassi Amina. Real-time realistic illumination and rendering of cumulus clouds. *International Journal of Multimedia & Its Applications*, 5, 10 2013.

[2] August Beer and PA Beer. Determination of the absorption of red light in colored liquids. 1852.

[3] Philippe Blasi, Bertrand Le Saec, and Christophe Schlick. A rendering algorithm for discrete volume density objects. In *Computer Graphics Forum*, volume 12, pages 201–210. Wiley Online Library, 1993.

[4] James F Blinn. A generalization of algebraic surface drawing. *ACM transactions on graphics (TOG)*, 1(3):235–256, 1982.

[5] Antoine Bouthors, Fabrice Neyret, and Sylvain Lefebvre. Real-time realistic illumination and shading of stratiform clouds. In *Eurographics Workshop on Natural Phenomena*, pages 41–50, 09 2006.

[6] Antoine Bouthors, Fabrice Neyret, Nelson Max, Eric Bruneton, and Cyril Crassin. Interactive multiple anisotropic scattering in clouds. Proceedings of the Symposium on Interactive 3D Graphics and Games, I3D 2008, 02 2008.

[7] Yoshinori Dobashi, Wataru Iwasaki, Ayumi Ono, Tsuyoshi Yamamoto, Yonghao Yue, and Tomoyuki Nishita. An inverse problem approach for automatically adjusting the parameters for rendering clouds using photographs. *ACM Transactions on Graphics (TOG)*, 31, 11 2012.

[8] Yoshinori Dobashi, Kazufumi Kaneda, Hideo Yamashita, Tsuyoshi Okita, and Tomoyuki Nishita. A simple, efficient method for realistic animation of clouds. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, page 19–28, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[9] Yoshinori Dobashi, Katsutoshi Kusumoto, Tomoyuki Nishita, and Tsuyoshi Yamamoto. Feedback control of cumuliform cloud formation based on computational fluid dynamics. *ACM Trans. Graph.*, 27, 08 2008.

[10] David Ebert, F.K. Musgrave, D. Peachey, Ken Perlin, Steve Worley, W.R. Mark, and John Hart. *Texturing and Modeling: A Procedural Approach: Third Edition*. 12 2002.

[11] Prashant Goswami and Fabrice Neyret. Real-time Landscape-size Convective Clouds Simulation and Rendering. In Fabrice Jaillet and Florence Zara, editors, *Workshop on Virtual Reality Interaction and Physical Simulation*. The Eurographics Association, 2017.

[12] Fredrik Häggström. Real-time rendering of volumetric clouds. Master's thesis, Umeå University, Department of Computing Science, 2018.

[13] Mark Harris. Real-time cloud rendering for games. volume 1. Game Developers Conference, 01 2002.

[14] Louis G Henyey and Jesse L Greenstein. Diffuse radiation in the galaxy. *The Astrophysical Journal*, 93:70–83, 1941.

[15] Roland Hufnagel and Martin Held. A survey of cloud lighting and rendering techniques. *Journal of WSCG*, 20, 01 2012.

[16] Rurik Högfeldt. Convincing cloud rendering-an implementation of real-time dynamic volumetric clouds in frostbite. Master's thesis, University of Gotherburg, 2016.

[17] Wojciech Jarosz. *Efficient Monte Carlo Methods for Light Transport in Scattering Media*. PhD thesis, UC San Diego, September 2008.

[18] Simon Kallweit, Thomas Müller, Brian Mcwilliams, Markus Gross, and Jan Novák. Deep scattering: Rendering atmospheric clouds with radiance-predicting neural networks. *ACM Transactions on Graphics*, 36, 09 2017.

[19] Joe Kniss, S. Premoze, C. Hansen, and David Ebert. Interactive translucent volume rendering and procedural modeling. In *Proceedings of the IEEE Visualization Conference*, pages 109–116, 12 2002.

[20] Bogdan Lipuš and Nikola Guid. A new implicit blending technique for volumetric modelling. *The Visual Computer*, 21:83–91, 02 2005.

[21] Rayleigh Lord. On the light from the sky, its polarization and colour. *Phil Mag*, 41:274, 1871.

[22] Louis Lorenz. *Lysbevægelsen i og uden for en af plane Lysbølger belyst Kugle*. na, 1890.

[23] Petr Man. Generating and real-time rendering of clouds. *Central European seminar on computer graphics*, 01 2006.

[24] Benoit B Mandelbrot and John W Van Ness. Fractional brownian motions, fractional noises and applications. *SIAM review*, 10(4):422–437, 1968.

[25] Monica Sanchez Meteorología En Red. Cómo se forman las nubes. https://www.meteorologiaenred.com/como-se-forman-las-nubes.html.

[26] Gustav Mie. Beiträge zur optik trüber medien, speziell kolloidaler metallösungen. *Annalen der physik*, 330(3):377–445, 1908.

[27] Anselmo Montenegro, Icaro Baptista, Bruno Dembogurski, and Esteban Clua. A new method for modeling clouds combining procedural and implicit models. In *2017 16th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, pages 173–182, 11 2017.

[28] Ksenia Mukhina and Alexey Bezgodov. The method for real-time cloud rendering. *Procedia Computer Science*, 66:697–704, 12 2015.

[29] Adolfo Muñoz. Higher order ray marching. In *Computer Graphics Forum*, volume 33, pages 167–176. Wiley Online Library, 2014.

[30] Met Office. What are clouds and how they form? https://www.metoffice.gov.uk/weather/learn-about/weather/types-of-weather/clouds/what-are-clouds-and-how-do-they-form.

[31] Rikard Olajos. Real-time rendering of volumetric clouds. Master's thesis, Lund University, Department of Computer Science, Faculty of Engineering LTH, 2016.

[32] World Meteorological Organization. Clouds. https://cloudatlas.wmo.int/clouds.html.

[33] K. Perlin and E. M. Hoffert. Hypertexture. *SIGGRAPH Comput. Graph.*, 23(3):253–262, July 1989.

[34] Ken Perlin. An image synthesizer. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, page 287–296, New York, NY, USA, 1985. Association for Computing Machinery.

[35] Ken Perlin. Improving noise. *ACM Transactions on Graphics*, 21:681–682, 07 2002.

[36] Arcot J Preetham, Peter Shirley, and Brian Smits. A practical analytic model for daylight. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 91–100, 1999.

[37] Juraj Pálenik. Real-time rendering of volumetric clouds. Master's thesis, Masaryk University, Faculty of Informatics, Brno, 2016.

[38] Inigo Quilez. https://www.iquilezles.org.

[39] William T Reeves. Particle systems—a technique for modeling a class of fuzzy objects. *ACM Transactions On Graphics (TOG)*, 2(2):91–108, 1983.

[40] Andrew Schneider. Real-time volumetric cloudscapes. In Wolfgang Engel, editor, *GPU Pro 7*, chapter 4, pages 97–127. CRC Press, 2016.

[41] Andrew Schneider. The real-time volumetric cloudscapes of horizon: Zero dawn. SIGGRAPH Advances in Real-Time Rendering in Games course, Aug. 2015. https://www.guerrilla-games.com/read/the-real-time-volumetric-cloudscapes-of-horizon-zero-dawn.

[42] Andrew Schneider. Nubis - authoring realtime volumetric cloudscapes with the decima engine. SIGGRAPH Advances in Real-Time Rendering in Games course, Aug. 2017. https://drive.google.com/open?id=0B-D275g6LH7LOE1RcVFERGpkS28.

[43] Joshua Schpok, Joseph Simons, David Ebert, and Charles Hansen. A real-time cloud modeling, rendering, and animation system. *Proceedings of Symposium on Computer Animation*, 2005, 01 2003.

[44] Scratchapixel. https://www.scratchapixel.com/.

[45] Edwin Vane. Cloud rendering using 3D textures. Jan. 2004.

[46] Patricio Gonzalez Vivo and Jen Lowe. The book of shaders. https://thebookofshaders.com.

[47] Gang Wang, Zhenzhou Ji, and Zexu Zhang. Using instance for large scale volumetric clouds rendering in real-time. *International Journal of Innovative Computing, Information and Control*, 8:4407–4420, 06 2012.

[48] Niniane Wang. Realistic and fast cloud rendering. *Journal of Graphics Tools*, 9, 01 2004.

[49] Antoine Webanck, Yann Cortial, Eric Guérin, and Eric Galin. Procedural cloudscapes. *Computer Graphics Forum*, 37, 04 2018.

[50] Steven Worley. A cellular texture basis function. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, page 291–294, New York, NY, USA, 01 1996. Association for Computing Machinery.

[51] Jiangbin Xu, Chao Yang, Jian Zhao, and Lingda Wu. Fast modeling of realistic clouds. In *2009 International Symposium on Computer Network and Multimedia Technology*, pages 1–4, 12 2009.

[52] E. Yusov. High-performance rendering of realistic cumulus clouds using pre-computed lighting. *High-Performance Graphics 2014, HPG 2014 - Proceedings*, pages 127–136, 01 2014.

Glosario y Acrónimos

metaballs En computación gráfica, las *metaball* son una técnica para simular interacción orgánica entre diferentes objetos n-dimensionales. A menudo ha sido utilizado para modelar objetos orgánicos o crear una malla base para la escultura. 15

morphing Es un efecto especial donde una imagen u objeto se transforma en otro mediante una transición. 13

AABB *axis-aligned bounding box*. 76

Ac altocúmulos. 22

API *Application programming interface* o Interfaz de programación de aplicaciones. 8

As altostrato. 22

Cb cumulonimbos. 20

Cc cirrocúmulos. 23

Ci cirros. 23

Cs cirrostratos. 24

Cu cúmulo. 21

fBm *fractal Brownian motion* o movimiento Browniano fractal. XI, 3, 12, 18, 26, 27

FPS frames por segundo. 60

GIMP *GNU Image Manipulation Program*. 9

GLSL *OpenGL Shading Language*. 8

medio participante (*participating media*) Termino usado para describir volúmenes que están compuestos por partículas. 83

Ns nimboestratos. 22

píxel (*pixel*) Unidad básica en un mapa de bits o el elemento más pequeño presentado en una pantalla. 2, 8, 34, 37, 39, 40, 46, 48, 53, 62, 83

RBF *Radial Basis Function* o función de base radial. 15

RGN *random number generators* o generador de números aleatorios. 74

Sc estratocúmulos. 21

St estratos. 20

vóxel (*voxel*) Unidad cubica de un objeto tridimensional. Presenta la unidad básica en una matriz tridimensional, es el equivalente de píxel en un mapa de bits. 12, 15

WMO *World Meteorological Organization* u Organización Meteorológica Mundial, SIG. 18

Anexos

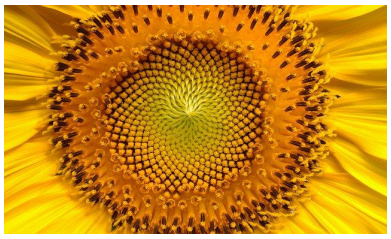
Anexo A

Anexo: Fractales

La geometría fractal empezó a ser estudiada gracias, entre otros, al matemático Benoit Mandelbrot. Este tipo de geometría describe formas complejas que nos podemos encontrar en la naturaleza, como pueden ser montañas, nubes o plantas (por ejemplo podemos ver la forma fractal en un brócoli (A.1) o en un girasol (A.2)) y logra convertir esta formas caóticas en simples ecuaciones que encapsulan mucha complejidad. Además, los gráficos por computador son muy interesantes para la representación de estas ecuaciones, ya que los objetos fractales son difíciles de entender sin una representación gráfica. Aparte de poder simular formas complejas de la naturaleza, también sirven para crear imágenes artísticas como las de las Figuras A.3 y A.4.

Por Shutterbug75 en Pixabay

Figura A.1: Brócoli.



Por wifeypoo en Pixabay

Figura A.2: Girasol.

Por realworkhard en Pixabay

Figura A.3: Ejemplo de arte fractal 1.

Por BarbaraALane en Pixabay

Figura A.4: Ejemplo de arte fractal 2.

En [10] se definen los fractales de esta forma: «*Un objeto geométricamente complejo, cuya complejidad surge a través de la repetición de una forma dada en un rango de escalas*». Es decir, cuando se aumenta o se disminuye la escala de un objeto fractal, su forma no varia, a esto se le suele denominar **autosimilitud**. Otra de las propiedades de los fractales es la llamada dimensión fractal. En la geometría euclidiana, la dimensión cero representa a un punto, una línea tiene una dimensión igual a uno, un plano igual a dos y el espacio tiene una dimensión igual a 3. Por el contrario, la dimensión fractal puede ser un número real como 2.3, es decir, es una generalización de la dimensión ya que se añade una parte fraccional, denominada incremento fractal.

Anexo B

Anexo: Ruido

En el ámbito de gráficos por computador la generación de ruido procedural se comenzó a desarrollar a mediados de los 80 como una alternativa al de usar imágenes para texturizar objetos, ya que, con los ordenadores de la época no tenían suficiente memoria como para almacenar dichas texturas. Por lo tanto, las personas que trabajaban en este ámbito comenzaron a buscar alternativas. Una idea es la de utilizar ***random number generators* o generador de números aleatorios** (**RGN**), pero no es suficiente para añadir variación a objetos 3D o simular cualquier fenómeno de la naturaleza.

La razón es simple, el patrón de la naturaleza suele cambiar suavemente, cosa que con **RGN** no podemos conseguir. Por ejemplo, un ruido que podemos obtener a partir de ello es el ruido blanco (*white noise*), como podemos observar en la figura B.1 este ruido tiene una distribución uniforme, cada valor es aleatorio y no tienen ninguna correlación entre sí. Con esto podemos ver que generar ruido que sirva para simular formas fractales no es tan sencillo, de hecho, idealmente la función tendría que tener las siguientes propiedades:

1. **Pseudoaleatoriedad.** El ruido que estamos intentando generar puede parecer aleatorio, pero en realidad es determinista. Es decir, si le damos una entrada tiene que devolver siempre el mismo valor.

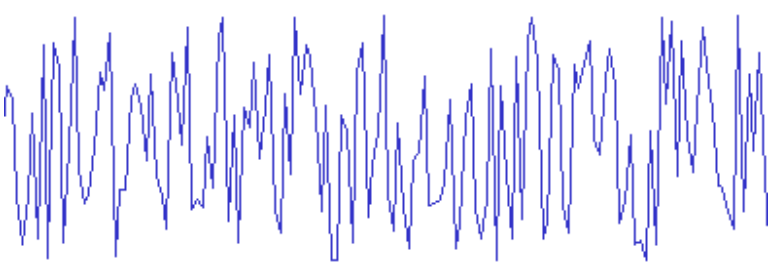
2. **Varias dimensiones.** La función debe devolver un valor sin importar la dimensión de la entrada. Matemáticamente, queremos una función tal que $\mathbb{R}^n \to \mathbb{R}$.

3. **Frecuencia limitada.** Su frecuencia no puede ser más alta que el valor 1.

4. **Continuidad y diferenciabilidad.** Se suele utilizar una función de alisado para difuminar los números aleatorios. Está función tiene que ser continua y diferenciable.

5. **Rango conocido.** El valor devuelto se encuentra en cierto rango, normalmente entre $[-1, 1]$ o $[0, 1]$.

6. **Periódica**. Para que no se note la transición al repetirse, la función debe ser periódica en cualquier dirección. Aunque esto puede influir en que se note su repetición, por lo que se utilizan valores de entradas grandes para que no se perciba.



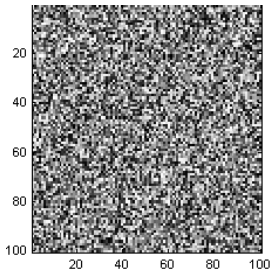


Figura B.1: Ruido blanco en 1D (izquierda) y 2D (derecha).

Anexo C

Anexo: Intersecciones

Información obtenida de [44]. Concretamente:

- *Ray-Box Intersection*
- *Ray-Sphere Intersection*.

C.1. Intersección rayo-cubo

Asumimos que el cubo está alineado con los ejes del sistema de coordenadas (comúnmente llamado ***axis-aligned bounding box*** (**AABB**)). Sabemos que el rayo se puede expresar de forma paramétrica como:

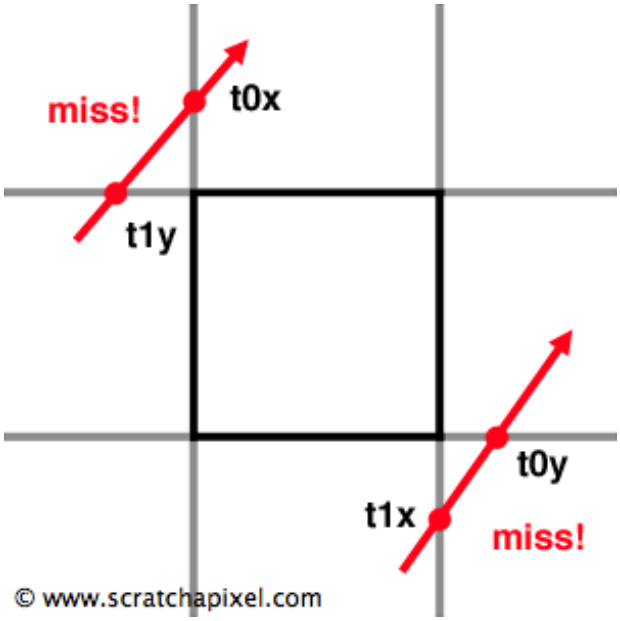
$$O + tD \tag{C.1}$$

Siendo O el origen del rayo y D la dirección. Cambiando el valor a t podemos obtener diferentes puntos del rayo. Para representar el cubo, necesitamos un punto mínimo (p_{min})

y un punto máximo (p_{max}) para definir los limites del mismo. Para encontrar el punto en donde interseca el rayo con el cubo, basta con resolver la siguiente ecuación:

$$O + tD = p \longrightarrow t = (p - O)/D \tag{C.2}$$

Con esto, podemos calcular seis valores que indican donde interseca el rayo en los planos del cubo paralelos a los ejes x, y y z. Por lo tanto, el siguiente paso es encontrar cuales de estos valores corresponden con una intersección con el cubo, si es que la hay. Para ello, calculamos el valor máximo para los tres valores del punto mínimo y el mínimo para los tres del máximo. Pero es posible que el rayo no interseque con el cubo, como podemos ver en el ejemplo de la Figura C.1.



Origen

Figura C.1: Ejemplo de dos rayos que no intersecan en un cubo

Por lo tanto, para calcular los puntos, teniendo en cuenta si interseca o no, podemos hacer lo siguiente:

1. Una vez obtenido los puntos utilizando C.2 calculamos el mínimo y máximo de los dos.
2. Calculamos los dos valores utilizando el mínimo y máximo del paso anterior.
3. Si la distancia entre el segundo valor y el primero es negativa, entonces los valores están fuera del cubo, es decir, no intersecan.

C.2. Intersección rayo-esfera

En el trazado de rayos, la intersección entre un rayo y una esfera suele ser la más sencilla de calcular. Además, dada su simplicidad, es rápido de calcular. Existen dos formas de realizar el cálculo: resolviéndolo mediante geometría o de forma algebraica. Es nuestro caso vamos a utilizar el segundo método.

El rayo se puede expresar de forma paramétrica como:

$$O + tD \tag{C.3}$$

Siendo O el origen del rayo y D la dirección. Cambiando el valor a t podemos obtener diferentes puntos del rayo. La esfera también se puede representar paramétricamente:

$$x^2 + y^2 + z^2 = R^2 \tag{C.4}$$

Siendo x, y y z las coordenadas de un punto cartesiano y R el radio de la esfera centrada en el origen. Si consideramos x, y y z como un punto P, podemos reescribir la ecuación de arriba como:

$$P^2 - R^2 = 0 \tag{C.5}$$

Sabemos que C.3 define todos los puntos a lo largo del rayo, por lo tanto, si reemplazamos P de C.5 con dicha ecuación:

$$|O + tD|^2 - R^2 = 0 \tag{C.6}$$

Desarrollando la función:

$$O^2 + (tD)^2 + 2ODt - R^2 = D^2t^2 + 2ODt + (O^2 - R^2) \tag{C.7}$$

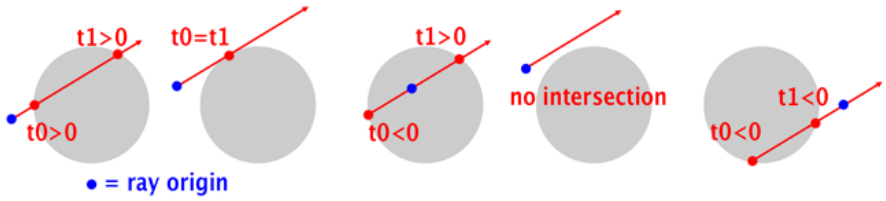
Podemos observar que hemos obtenido una función cuadrática. Por lo tanto, con $a = D^2$, $b = 2OD$ y $c = O^2 - R^2$ podemos calcular las raíces de esta función para obtener los puntos donde interseca el rayo, es decir, t. Como es una función cuadrática, podemos calcularlo sus raíces con la siguiente ecuación:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$
$$\Delta = \sqrt{b^2 - 4ac}$$

Δ es el discriminante y su signo indica si hay dos, una o ninguna raíz en la función:

1. Si $\Delta > 0$ existen dos raíces en la función, es decir, el rayo interseca en dos puntos de la esfera.
2. Si $\Delta = 0$ existe una raíz en la función, es decir, el rayo interseca una sola vez la esfera.
3. Si $\Delta < 0$ no existen raíces en la función, es decir, el rayo no interseca la esfera.

Puede ocurrir que una o las dos raíces sea negativa, esto es, que el rayo interseca la esfera, pero detrás de su origen. En la siguiente figura se pueden ver todos los posibles casos:



Origen

Figura C.2: Posibles casos de intersección de rayo-esfera, de izquierda a derecha: el rayo interseca dos veces (delante del origen), el rayo interseca una vez, el rayo interseca dos veces (uno delante del origen y el otro detrás), no hay intersección y el rayo interseca dos veces (detrás del origen).

Si queremos comprobar la intersección con esferas que no están en el origen, tenemos que modificar la función C.5 añadiendo un parámetro C que define a que distancia esta trasladada la esfera del origen:

$$|P - C|^2 - R^2 \tag{C.8}$$

Por lo tanto:

$$|O + tD - C|^2 - R^2 = 0 \tag{C.9}$$

Esto nos da $a = D^2$ (o $a = 1$ si D es un vector normalizado), $b = 2D(O - C)$ y $c = (O - C)^2 - R^2$.

Anexo D

Anexo: Iluminación

En esta sección vamos a dar una introducción a la iluminación en el área de gráficos por computador concentrándonos en la iluminación de las nubes. Se utiliza la información y terminología de [17].

Para empezar, los algoritmos de iluminación global tratan de imitar el comportamiento físico de la luz que son emitidos por una fuente de luz, esta luz se dispersa por los elementos de una escena hasta que es detectado por una cámara virtual.

En la teoría de la luz existen diversos modelos que son completos pero complicados, estos son: óptica geométrica, óptica física, óptica electromagnética y óptica cuántica. En la computación gráfica se suele basar en el modelo más simple de estos, la óptica geométrica. Este modelo simplifica el comportamiento de la luz, solo teniendo en cuenta que la luz puede ser emitida, reflejada y transmitida. Además, se asume que la luz viaja en líneas rectas y a velocidad infinita. Todo esto hace que no se pueda o sea más difícil simular los efectos descritos por los otros modelos.

Para poder entender los algoritmos de iluminación es necesario saber algunos de los términos definidos por la radiometría. Esta es una ciencia que se ocupa del estudio de la medida física de la radiación electromagnética, incluyendo la luz visible.

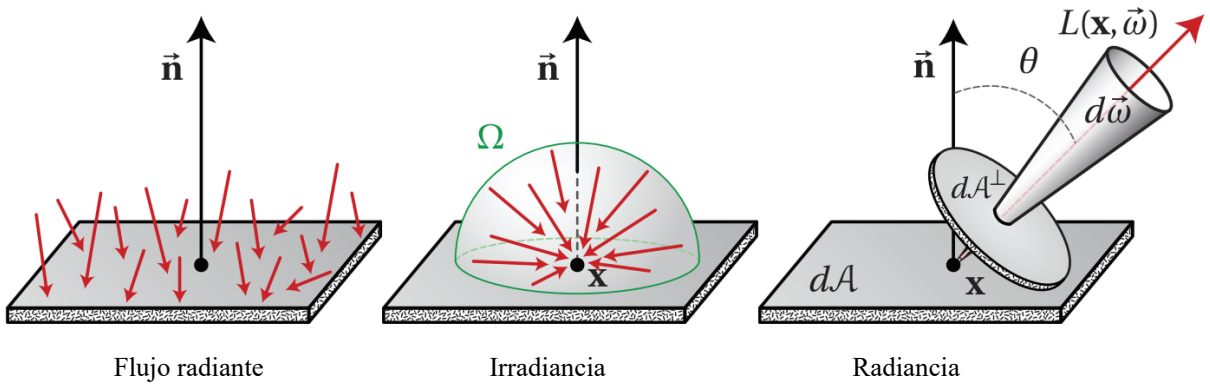


Figura D.1: Ejemplo de las tres medidas de la radiometría de [17].

Flujo radiante

El flujo radiante, denotado por Φ, expresa la cantidad de energía transportada en una superficie con relación al tiempo. Se utiliza la misma medida al calcular la energía de una bombilla y se usa la unidad de vatios ($W = J \cdot s^{-1}$).

Irradiancia

La irradiancia es la magnitud que expresa la cantidad incidente de potencia sobre una superficie por unidad de área. Su unidad es $W \cdot m^{-2}$. La irradiancia se calcula en una posición x de una superficie junto con la normal de la superficie $\overrightarrow{n}$. Se puede expresar en función al flujo radiante como:

$$E(x) = \frac{d\Phi(x)}{dA(x)} \tag{D.1}$$

Radiancia

La radiancia expresa cuanta luz llega a través de una dirección $d\overrightarrow{w}$ sobre un área perpendicular a esa dirección ($dA^{\perp}$). Tiene $W \cdot sr \cdot m^{-2}$[1] como unidad y se expresa como:

$$L(x, \overrightarrow{w}) = \frac{d^2\Phi(x, \overrightarrow{w})}{d\overrightarrow{w} dA^{\perp}(x)} \tag{D.2}$$

Normalmente queremos obtener la radiancia en una superficie más que en una superficie hipotética, para ello se puede utilizar la relación $dA^{\perp} = (\overrightarrow{n} \cdot \overrightarrow{w})dA$, $(\overrightarrow{n} \cdot \overrightarrow{w})$ nos devuelve el coseno del ángulo entre $\overrightarrow{w}$ y la normal de la superficie $\overrightarrow{n}$.

$$L(x, \overrightarrow{w}) = \frac{d^2\Phi(x, \overrightarrow{w})}{(\overrightarrow{n} \cdot \overrightarrow{w})d\overrightarrow{w} dA(x)} \tag{D.3}$$

La radiancia está relacionada con la percepción del brillo en los colores en el ojo humano y la cantidad que se necesita calcular por cada **píxel (*pixel*)** en un renderizado.

D.1. Iluminación de las nubes

En el apartado anterior hemos asumido que la interacción de la luz solo ocurre en una superficie, pero cuando tratamos de medios como las nubes no podemos definirlos como tal. Estos están compuestos de partículas y cuando los rayos de luz las atraviesan los fotones interactúan con ellas. Por ello, un medio como las nubes se le denomina como un **medio participante (*participating media*)**. Cuando los fotones viajan entre estas partículas pueden ocurrir varias situaciones. El fotón puede continuar sin interactuar con las partículas o interactuando con una o varias de ellas. En esta interacción pueden ocurrir dos cosas: la partícula absorbe el fotón o el fotón se dispersa en otra dirección. En cualquiera de los

[1] Vatios por estereorradián por metro cuadrado

dos casos esto crea un cambio en la radiancia a lo largo del rayo. Aunque estos no son los únicos efectos que pueden ocurrir. Vamos a ver cada caso por separado según lo describen en [15], en la Tabla D.1 se muestran los símbolos que se utilizan junto con su descripción.

Símbolo	Descripción
κ_a	Coeficiente de absorción
κ_s	Coeficiente de dispersión
κ_k	Coeficiente de extinción
ρ	Densidad de una partícula
$p(\overrightarrow{\omega}, \overrightarrow{\omega'})$	Función de fase, describe la distribución de luz después de ser dispersada
$L(x, \overrightarrow{\omega})$	Radiancia de un rayo de luz en una dirección $\overrightarrow{\omega}$ y en una posición x
L_e	Radiancia emitida
T	Transmitancia

Tabla D.1: Símbolos utilizados para la descripción de la iluminación de las nubes

Absorción

Este es el proceso donde la radiación se convierte en calor. La atenuación de un rayo está descrito por:

$$(\overrightarrow{\omega} \cdot \nabla^2) L(x, \overrightarrow{\omega}) = -\kappa_a(x) L(x, \overrightarrow{\omega}) \tag{D.4}$$

Las partículas de las nubes no tienen mucha absorción, por lo que en la interacción más común entre fotón y partícula en una nube es la de la dispersión.

Dispersión

La radiancia del rayo puede ser absorbida y reemitida a otras direcciones. Podemos definir la atenuación de la radiancia a través de una dirección $\overrightarrow{w}$ dado a la dispersión a otras

[2] ∇ es el operador gradiente y esta definido por $\nabla = (\partial/\partial x, \partial/\partial y, \partial/\partial z)$

direcciones como:

$$(\overrightarrow{\omega} \cdot \nabla)L(x, \overrightarrow{\omega}) = -\kappa_s(x)L(x, \overrightarrow{\omega}) \tag{D.5}$$

Si la dispersión es hacia la dirección $\overrightarrow{w}$ desde todas las direcciones en un punto x:

$$(\overrightarrow{\omega} \cdot \nabla)L(x, \overrightarrow{\omega}) = \frac{\kappa_s(x)}{4\pi} \int_{4\pi} p(\overrightarrow{\omega'}, \overrightarrow{\omega})L(x, \overrightarrow{\omega'})d\omega' \tag{D.6}$$

Extinción

El coeficiente de extinción es la probabilidad de que un fotón interactué con una partícula. Esta probabilidad de que interactué y, por lo tanto, reduzca la radiancia es la suma de la probabilidad de que sea absorbido o dispersado hacia otras direcciones:

$$\kappa_t(x) = \kappa_a(x) + \kappa_s(x) \tag{D.7}$$

Emisión

La emisión incrementa la radiancia debido a que otras formas de energía se han transformado en luz. Se puede definir de la siguiente forma:

$$(\overrightarrow{\omega} \cdot \nabla)L(x, \overrightarrow{\omega}) = \kappa_a(x)L_e(x, \overrightarrow{\omega}) \tag{D.8}$$

Esto no suele tener relevancia para el renderizado de las nubes, ya que estás no emiten luz a no ser que se coloque una fuente de luz dentro de la misma.

Transmitancia

Denota la cantidad de fotones que han viajado a través de una línea recta sin interacciones. La transmitancia se puede calcular mediante la ley de Beer-Lambert.

D.1.1. Función de fase

Una función de fase es una descripción probabilística de la distribución direccional de la luz dispersada. Normalmente depende de la longitud del rayo y de la forma y tamaño de las partículas del medio. Existen diversas funciones de fase, vamos a describir brevemente las más usadas en la computación gráfica:

Función de fase isotropica

La dispersión isotrópica es el equivalente al reflejo difuso[3]. Esta función es constante:

$$p(\overrightarrow{\omega}, \overrightarrow{\omega'}) = \frac{1}{4\pi} \tag{D.9}$$

Los medios isotrópicos dispersan la luz uniformemente a todas las direcciones sin importar la dirección entrante.

Función de fase Henyey-Greenstein

Una de las funciones de fase no isotrópicas es la presentada por Henyey y Greenstein en 1941 [14] como ya hemos comentado en la sección 4.6.2 del capítulo 4.

[3]El término difuso se utiliza para representar la luz que ha sido transmitida, absorbida y dispersada en una superficie.

Función de fase de Schlick

En la función de Henyey-Greenstein el cálculo del exponente fraccional en el denominador es relativamente costoso. Además, la forma exacta de la función no suele ser esencial en la computación gráfica. Por lo tanto, Blasi y sus compañeros desarrollaron en 1993 [3] una función similar, la cual aproxima la forma de la función de Henyey-Greenstein mediante un elipsoide:

$$p_S(\theta, k) = \frac{1 - k^2}{4\pi(1 + k\,cos\theta)^2} \tag{D.10}$$

Donde el parámetro $k \in [-1, 1]$ es similar al parámetro g. Como en g, $k < 0$ es la la dispersión hacia atrás (*back-scattering*), $k = 0$ la dispersión isotrópica (*isotropic-scattering*) y $k > 0$ la dispersión hacia delante (*forward-scattering*).

Dispersión de Rayleigh

En 1871, Rayleigh [21] presentó varias expresiones para la dispersión de luz fuera de las moléculas de aire. Esta dispersión es una aproximación para el comportamiento de la luz, ya que dispersa en medios compuestos de partículas que son un una décima parte de la longitud del rayo de luz. En el modelo se asume una distribución aleatoria de esferas especulares muy pequeñas, quedando la función muy simple:

$$p_R(\theta) = \frac{3}{16\pi}(1 + cos^2\theta) \tag{D.11}$$

Además, es muy dependiente de la longitud del rayo de luz. Esto se puede ver en el coeficiente de dispersión:

$$\kappa_s = \frac{2\pi^5}{3}\frac{d^6}{\lambda^4}\left(\frac{n^2-1}{n^2+2}\right)^2 \tag{D.12}$$

Siendo λ la longitud del rayo, d el diámetro de la partícula y n su índice de refracción.

Teoría de Lorenz-Mie

La dispersión de Rayleigh es una aproximación que funciona cuando la partícula es pequeña relativo a la longitud del rayo. Cuando las partículas son comparables a la longitud del rayo, como las gotas de agua en niebla, es más conveniente usar la más complicada teoría de Lorenz-Mie [22, 26]. Esta teoría puede ser usada para derivar funciones de fase para una colección homogénea de partículas esféricas con cualquier radio y longitud del rayo. Está esta basada en una teoría más general derivada de las ecuaciones de Maxwell.

Anexo E

Anexo: Programa principal

En este anexo explicamos brevemente las dos partes fundamentales de la aplicación, la interfaz de usuario de la misma y las librerías que se han utilizado.

E.1. Inicialización

Una vez importadas las librerías (sección E.4), comprueba si el navegador admite *WebGL2* y si es así, define las variables globales y llama a la función que inicializa todo el entorno: *init()*. Esta función llama a su vez a otras funciones de inicialización, como podemos observar en el siguiente código:

```
Comienzo init()
    initScene();
    initShader();
    initSky();
    createGui();
Fin init()
```

Listing E.1: Función de inicialización

La función *initScene()* inicializa todos los objetos relacionados con el entorno: se obtiene un contexto (WebGL2 en nuestro caso) y se crea el renderizador. Después, se generan dos escenas: la principal donde se muestran las nubes y una secundaria donde se genera la textura del cielo. Por último, se crea el nodo principal del grafo de la escena que contiene varios objetos, entre ellos la cámara con los controles orbitales.

En cuanto a la función *initShader()* inicializa el *shader* creando un plano transparente[1] y el material que contiene el *shader*, generando todas las variables uniformes. Además, en esta función es donde se cargan las texturas que recibirá después el *shader*. La carga de texturas se ha hecho de tal forma que, hasta que no estén todas cargadas, no se renderice la escena.

La siguiente función *initSky()* inicializa todas las variables que tengan que ver con el cielo, es decir, crea el objeto *sky* mediante el módulo *Sky.js* y lo añade tanto a la escena principal como a la escena que se va a usar para generar la textura del cielo. Además, añade una esfera para representar el Sol.

Por último, la función *createGui()* crea la interfaz donde se pueden cambiar los parámetros de la aplicación, además de crear la función *guiChanged()* que es llamada cada vez que se cambia alguno de los parámetros y actualiza cada uno de estos por el valor que tengan en la interfaz.

E.2. Animación

Una vez se han cargado todas las texturas comienza el bucle de renderizado, de esto se encarga la función *render()* que a su vez llama a *animate()*. Para empezar, activamos los *stats* del módulo *stats.module.js* para que calcule el tiempo de renderizado hasta que le digamos que pare. Después, actualizamos las variables que controlan la animación de las nubes y generamos la textura del cielo. Para ello, tenemos que asignar al visualizador un objetivo con la función *setRenderTarget()*. Finalmente, enviamos esta textura al *shader*

[1] Necesitamos que el material este asociado a un objeto.

y renderizamos la escena principal. Como las nubes pueden ser animadas, tenemos que repetir el renderizado de nuevo, mediante la función *requestAnimationFrame(animate)*. De esta forma, la escena se actualiza cuando se active la animación, se cambie la resolución o el tamaño de la ventana.

E.3. Interfaz Usuario

Como ya hemos mencionado, existe una interfaz para poder cambiar varios parámetros de la aplicación y, de esta forma, cambiar la apariencia de las nubes. En esta sección vamos a explicar cada parámetro de cada sección de la interfaz:

E.3.1. General

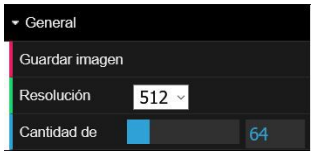


Figura E.1: Interfaz con parámetros generales.

- **Guardar imagen**. Crea una imagen del renderizado actual.
- **Resolución**. Resolución de la ventana del renderizado.
- **Muestras**. Número de muestras en el algoritmo de *ray marching*.

E.3.2. Cielo

Estos parámetros sirven para cambiar el cielo y mover el Sol de posición. Se utilizan en el módulo *Sky.js*, y se han obtenido de [36].

Figura E.2: Interfaz donde se puede editar el cielo.

E.3.3. Nubes

Figura E.3: Interfaz donde se puede editar las nubes.

- **Altura Base/Máxima**. La altura entre la que pueden encontrarse las nubes.
- **Nubosidad**. Cuánto cubren en cielo las nubes.
- **Densidad**. Cuán densas son las nubes.
- **Tam. climático/forma/detalle**. El tamaño de las tres texturas que forman las nubes.
- **Cubo o esfera**. El tipo de área donde se muestran las nubes: un cubo o dos esferas.

E.3.4. Animación

Figura E.4: Interfaz para la animación de las nubes.

- **Velocidad**. Velocidad de la animación.
- **Dirección eje X/Y**. El eje en el que se van a mover las nubes.

E.3.5. Iluminación

Figura E.5: Interfaz donde se puede editar la iluminación.

- **In scatter**. Parámetro in_s de 4.21.
- **Out scatter**. Parámetro out_s de 4.21.
- **In o Out Scatter**. Parámetro ivo de 4.21.
- **Silver intensity**. Parámetro cs_i de 4.20.
- **Silver exponent**. Parámetro cs_e de 4.20.
- **Absorción de luz**. Parámetro b de 4.18.

E.4. Librerías utilizadas

Para la implementación de la aplicación se han utilizado las siguientes librerías:

- *FastNoise*. *FastNoise* es una librería de código abierto de generación de ruido procedural con una gran colección de algoritmos de ruido.
- *three.module.js*. Módulo *Javascript* con el núcleo principal de *three.js*.
- *dat.gui.module.js*. Módulo *Javascript* para la creación de interfaces en *three.js*. Permite crear un conjunto de parámetros los cuales pueden ser cambiados por el usuario.
- *stats.module.js*. Módulo *Javascript* con el cual se pueden mostrar varia información, así cómo mostrar los frames por segundo o los milisegundos que tarda en renderizar un frame.
- *OrbitControls.js*. Permite crear una cámara que orbita alrededor de un objetivo, en nuestro caso, el centro de la escena.
- *WebGL.js*. Se utiliza para comprobar si el navegador es compatible con WebGL2 o no mediante la función *isWebGL2Available()*.
- *Sky.js*. Permite crear un cielo con un Sol proveyéndole de varios parámetros.