UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Generador de fenómenos atmosféricos para three.js

Moisés Huedo Gayo
Tutor: Carlos Aguirre Maeso

Julio 2016

Generador de fenómenos atmosféricos para three.js

AUTOR: Moisés Huedo Gayo
TUTOR: Carlos Aguirre Maeso

Dpto. de ingeniería informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
29 de Junio de 2016

Resumen

En la actualidad, gracias a lenguajes como javascript y estándares como HTML5, el desarrollo web está en su momento de mayor auge. Por otro lado, gracias al incremento de la capacidad de computo de los ordenadores y a los lenguajes orientados a gráficos de bajo nivel cada vez los desarrollos web incluyen más habitualmente aplicaciones 3D integradas.

Estas dos circunstancias se dan la mano en el desarrollo de aplicaciones web de nueva generación que , por un lado, permiten la independencia de la plataforma o sistema operativo, ya que estos se ejecutan sobre el navegador y por otro, permite aprovechar la potencia de la librería gráfica WebGL o la versatilidad de javascript.

En este contexto nos encontramos con three.js un framework javascript que permite trabajar con WebGL de una manera muy sencilla, abstrayendo al programador de la complejidad de WebGL.

En este trabajo fin de grado se pretende crear una librería de apoyo para three.js que permita añadir efectos climatológicos de una manera sencilla. Para ello dividiremos el trabajo en dos grandes bloques. Por un lado la generación de la textura para el cielo y por otro un motor de partículas. Para el cielo utilizaremos un algoritmo de ruido de Perlin que nos permitirá generar texturas de forma procedural de gran calidad y de una forma rápida. Para las partículas usaremos un motor de partículas que nos permitirá, además de crearlas, darles un movimiento realista.

Tal y como se ha planteado el desarrollo, la librería es bastante versátil y se dan unas pautas para el uso básico y para un uso avanzado. Esto es debido a que se ha separado la generación de los datos y el renderizado de los mismos y también se ha realizado el motor de partículas que puede funcionar junto a los datos o de forma independiente. De esta forma, el código es mucho más claro y legible permitiendo además, a los programadores más experimentados, adaptar los datos a una renderización propia, adaptar el renderizado a los datos generados por el mismo y utilizar las partículas de una forma personalizada.

El resultado final es un prototipo de la librería que, de forma transparente al programador, permite crear distintos tipos de cielo y varios fenómenos atmosféricos como lluvia, nieve, tormentas de arena, ventiscas y viento. En la parte de conclusiones se explora el camino que podría seguir la librería de cara a aumentar su funcionalidad.

Palabras clave

Three.js, WebGL, ruido de Perlin, renderizado, motor de partículas, librería.

Abstract

Today, thanks to languages like javascript and standards like HTML5, web development is at its peak moment. On the other hand , thanks to increased computing capacity and low level graphics oriented languages, web developments include most commonly integrated 3D applications.

These two circumstances come together at the new generation web developments, that allows the independence of the platform or operating system , as these run on the browser and can harness the power of the WebGL graphics library or the versatility of javascript.

In this context we find Three.js a javascript library that allows working with WebGL in a very simple way , abstracting the programmer from the complexity of WebGL.

In this bachellor tesis we create a library of support for allowing three.js add weather effects in a simple way . To do so we divide the work into two blocks . On the one hand the generation of texture to the sky and the other a particle engine. For the sky we will use a Perlin noise algorithm that will allow us to generate procedural textures of high quality in a quick way . For particles we will develop a particle engine that allows create and giving them a realistic movement.

The final library is quite versatile and we provide guidelines for basic and advanced use. This is because we have separated data generation and rendering and has also performed the particle engine operable with data or independently . Thus , the code is much clearer and readable also allowing more experienced programmers , adapt the data to a personal rendering, adapt the rendering of an own data and use the particles in a custom way.

The final result is a prototype library that, transparently to the programmer, allows to create different types of sky and various atmospheric phenomena such as rain, snow, sand storms, blizzards and wind. In the conclusions part we explore the way that could follow the library in order to increase its functionality.

Keywords

Three.js, WebGL, Perlin noise, render, particles engine, library.

Agradecimientos

Quería dedicar este trabajo de fin de grado a Javi. La alegría de terminar el grado contrasta con la tristeza por tu pérdida. Sin tu apoyo y sin tu ayuda nunca habría llegado hasta aquí. Gracias de corazón.

También quería dedicarle el trabajo a mi madre, que fue quien me convenció para embarcarme en esta aventura universitaria. Gracias por tu confianza.

Gracias Xenia por soportar estos cuatro años en los que hemos vivido separados en distancia pero más unidos que nunca.

Por supuesto quiero dar las gracias a Carlos Aguirre por entender mi complicada situación personal y permitirme trabajar sin presiones.

Por último quería agradecer a mis compañeros de grado Alber, Chache, Tavi, Fran, Borja, Sara y muchos más que han hecho este camino mucho más divertido, a todos los profesores que tanto me han enseñado y a mis amigos, que todavía no se creen que lo haya conseguido.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

1.1 Motivación

Los lenguajes web han ido evolucionando de una manera notable en los últimos años. Javascript se ha convertido en el lenguaje más utilizado en el entorno web y WebGL en el lenguaje orientado a gráficos de bajo nivel estándar para la web.

De este hecho, nace la necesidad que cubre Three.js. Three.js es una librería Javascript para la explotación de recursos WebGL pero sin la complicación de este lenguaje. Además, permite acelerar mucho los desarrollos y facilitar su posterior mantenimiento.

Ya existen multitud de webs desarrolladas con three.js. En su propia página web, podemos ver enlaces a páginas como la de la película "The Hobbit"[1] o la página de experimentos de realidad virtual de Google[2]. Lo que no se ha desarrollado aún de manera comercial, son videojuegos hechos con Three.js. La facilidad de crear una aplicación a partir de un desarrollo web, ya sea para PC o para dispositivos móviles, hace que en un futuro esta sea una vía de desarrollo que puede llegar a ser importante.

Una de las ventajas de Javascript es la gran cantidad de librerías que existen y la facilidad de integrarlas en los desarrollos. Esto facilita enormemente las tareas del programador web simplificando el código y evitando reinventar la rueda. Sirva como ejemplo la librería dat.gui.js usada en los entornos de prueba de este proyecto que nos permite hacer una interfaz de usuario conocida, ya que se usa en muchos entornos de prueba y, sobre todo, de forma extremadamente sencilla.

Three.js, a pesar de ser una librería en sí misma, tiene otras librerías desarrolladas en paralelo para facilitar aún más su uso. En nuestro entorno de pruebas 3D utilizamos la librería OrbitControls.js que permite controlar la cámara con el ratón añadiendo una sola línea de código.

Debido a estas razones, se ha decidido desarrollar una librería centrada en la creación de fenómenos atmosféricos para Three.js al no existir nada parecido aún.

1.2 Objetivos

El objetivo final de este trabajo es crear una librería que permita al programador añadir efectos climatológicos de manera sencilla. Se ha buscado que la librería sea lo más versátil posible y que sus partes sean lo más independientes que el desarrollo permita.

Tras la fase de aprendizaje y compresión de la librería Three.js, se llegó a la conclusión de la necesidad de centrar el desarrollo de dos subsistemas, un generador de ruido de Perlin para la creación de la textura del cielo y un generador de partículas que nos servirá para representar los fenómenos atmosféricos.

[1] http://middle-earth.thehobbit.com/ (24/01/2016)

[2] http://vr.chromeexperiments.com/ (24/01/2016)

El generador de ruido de Perlin lo usaremos para generar el cielo de forma procedural. Los resultados para este tipo de algoritmos son muy buenos y tienen una eficiencia moderadamente buena, por lo que es ideal para la creación de texturas, ya que esto se puede hacer en la fase de carga de la aplicación. Además de obtener los datos a partir del algoritmo de Perlin, se incluirán varias funciones de renderizado para crear distintos tipos de cielo y su aplicación posterior como textura.

El generador de partículas nos servirá para crear los efectos climatológicos propiamente dichos. La base del generador se hará de forma genérica dependiente de parámetros lo que nos permite crear funciones específicas para cada efecto con mucha facilidad. Esto también es de utilidad a la hora de crear efectos personalizados por parte de los usuarios de la librería. Los efectos incluidos en la librería son: Lluvia, nieve, tormenta de arena, ventisca y viento.

Se presentarán dos aplicaciones de prueba. En la primera se presentan los resultados en 2D de la generación del ruido de Perlin a través del canvas de HTML. En la segunda aplicación, implementaremos, usando three.js, los efectos climatológicos desarrollados.

1.3 Organización de la memoria

Tras esta breve introducción al trabajo, se realizará un breve análisis del estado del arte, véase, de las opciones similares que hay disponibles a día de hoy.

A continuación se detallará la fase de diseño. En este apartado se explicará y entenderá cómo funciona la librería Three.js para determinar qué elementos son necesarios para el desarrollo del proyecto. Además se explicará cómo se han diseñado estos elementos.

En la sección de desarrollo se explicará, de forma concreta, como se ha realizado la implementación de lo diseñado. Veremos los problemas que han surgido así como las soluciones aportadas

Tras conocer como se ha diseñado y desarrollado la librería, se detallaran los entornos de prueba realizados para testear la librería. En el caso del entorno de pruebas 2D se incluirán los resultados de la ejecución y en el entorno de pruebas 3D se detallará el funcionamiento de los controles del mismo.

Para concluir del documento, extraeremos las conclusiones del proyecto, haciendo especial hincapié en las ventajas y las limitaciones de la librería, así como en las futuras mejoras que se le puede aplicar a la misma.

De cara a facilitar la lectura de la memoria, se ha utilizado los siguientes tipos de letra:
Para las palabras en inglés se usará la *cursiva.*
En cierta parte del documento se hace referencia a los ejes x, y, z. En este caso y, para evitar confusiones, cuando nos refiramos a los ejes, se usará la *cursiva.*
En las secciones de código o formulas se usará una fuente `monospace`**.**
Las palabras importantes o encabezados irán en **negrita.**

2 Estado del arte

La facilidad de creación e importación de módulos que tiene javascript ha desembocado en una gran cantidad de librerías. De esta forma no es necesario hacer código que ya está desarrollado y testado. Estos módulos nos permiten, de una manera sencilla, añadir funcionalidades ya desarrolladas por terceros.

Three.js, a pesar de ser una librería, tiene otras librerías que funcionan junto a ella. En el entorno de pruebas 3D, hemos utilizado la librería OrbitControls.js que nos permite manejar la cámara con el ratón. Este es sólo uno de los ejemplos, pero existen una gran cantidad de librerías adicionales, desde una tan sencilla como la que hemos utilizado nosotros, a otras más complejas, como motores de físicas.

A día de hoy no existe ninguna librería centrada en la generación de fenómenos atmosféricos para Three.js, de hecho, no se ha encontrado nada similar para ninguno de los motores 3D más importantes de la actualidad. Lo que si se han visto son ejemplos que se basan en la generación de partículas pero, muy limitados. En algunos casos el acceso al código es difícil y en otros son ejemplos muy básicos que no son de mucha utilidad.

Tras este estudio previo queda claro que una librería de este tipo sería de mucha utilidad para muchos desarrolladores y podría animar a otras personas o grupos de desarrolladores a dar más uso a Three.js o a desarrollar librerías más potentes.

3 Diseño

3.1 Three.js

El generador de fenómenos atmosféricos está diseñado para su uso con Three.js, una librería en javascript desarrollada para renderizar gráficos 3D, a través de WebGL, de una manera sencilla. A la hora de diseñar el generador ha sido necesario conocer cómo funciona three.js en profundidad para adaptarlo a las particularidades de la librería.

3.1.1 WebGL

WebGL viene definido en su propia página web[3] como una API de bajo nivel para gráficos 3D en navegador, basándose en OpenGL ES 2.0 que se apoya en el canvas de HTML5 para renderizar los gráficos.

La librería es mantenida por Khronos Group[4], un consorcio tecnológico sin ánimo de lucro que se encarga de desarrollar y mantener librerías gráficas de código abierto. Cabe destacar que son los responsables de Vulkan[5], la librería gráfica que pretende ser el estándar de código abierto del futuro.

A nivel de utilización de la librería no vamos a entrar, ya que three.js nos permite su uso de forma sencilla. En cualquier caso, se han consultado ejemplos en la red de trabajos hechos con WebGL y sobre todo destacan los *shaders*[6] de gran calidad gráfica como el de la imagen que acompaña este texto.

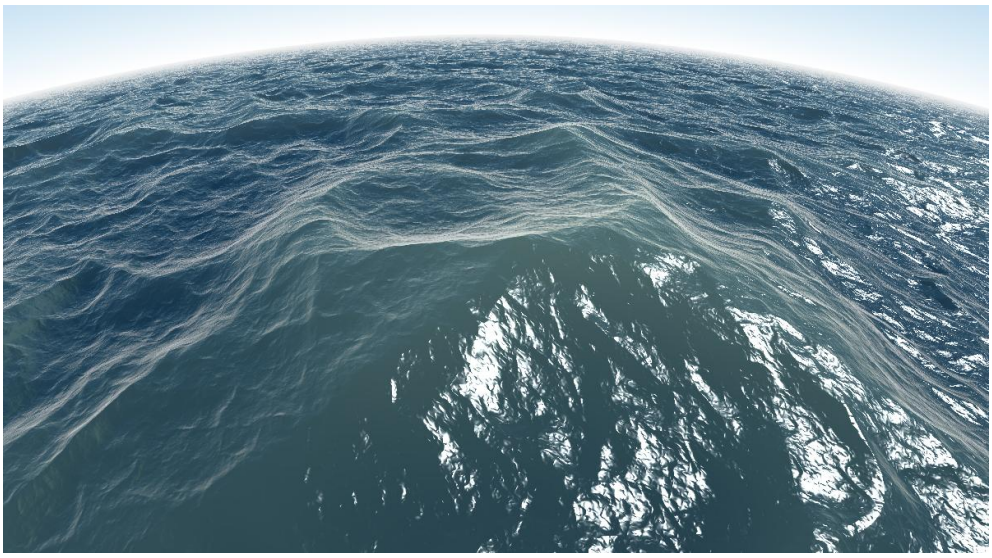

Ilustración 3.1 Shader Webgl

[3] https://www.khronos.org/webgl/ (12/02/2016)

[4] https://www.khronos.org/ (12/02/2016)

[5] https://www.khronos.org/vulkan/ (12/02/2016)

[6] https://www.shadertoy.com/ (08/03/2016)

3.1.2 La librería Three.js

Como comentamos anteriormente, Three.js nos permite abstraernos de la complejidad de WebGL. Está escrita en javascript por lo que es extremadamente sencilla de incorporar a los desarrollos web. La librería es mantenida por Ricardo Cabello a.k.a. mrdoob.

La versión utilizada de la librería en este proyecto ha sido la r77 publicada el 27 de mayo de 2016. En los primeros prototipos se han utilizado distintas versiones de three.js, pero en la implementación final se ha decidido usar la versión r77, de esta forma nos aseguramos la compatibilidad con la última versión estable de la librería.

En la página web[7] de three.js encontramos documentación y ejemplos pero se antojan insuficientes. Al no ser una librería tan popular como puede ser pixi.js[8] (esta para gráficos 2d) los ejemplos que hay a lo largo de ancho de la red son, básicamente, problemas concretos y preguntas acerca de la librería en stackoverflow[9]. Este hecho ha dificultado el desarrollo ya que ha sido necesario hacer muchas pruebas antes de obtener resultados satisfactorios.

Además de la API propiamente dicha, se ofrece un editor gráfico que simplifica aún más el uso de la librería. En nuestro caso se ha realizado todo el trabajo de forma programática, pero esta característica es interesante ya que encaja en la filosofía de facilitar el diseño de aplicaciones 3D para web.

Lo más destacable de la librería es, como hemos comentado anteriormente, la facilidad con la que permite trabajar con gráficos en 3D dando la posibilidad de crear una geometría de las disponibles con una línea de código. Además incluye varios tipos de cámara y luces, que veremos en detalle en el siguiente punto.

[7] http://threejs.org/ (20/01/2016)

[8] http://www.pixijs.com/ (16/04/2016)

[9] http://stackoverflow.com/ (16/04/2016)

3.1.3 Funcionamiento de Three.js

Para desarrollar un proyecto en Three.js necesitamos tres elementos básicos: la escena, la cámara y el motor de renderizado.

La escena es un contenedor en el que se alojan todos los elementos que queremos mostrar en pantalla. Es necesario tras crear un objeto añadirlo de manera explícita a la escena a través el método `add`.

La cámara será la encargada de determinan que objetos se muestran y cuáles no. Existen distintos tipos de cámara. Al no ser algo relevante para nuestro proyecto, se ha usado la *PerspectiveCamera*. Esta cámara utiliza la proyección en perspectiva creando un efecto 3D más marcado.

El motor de renderizado es, como su nombre indica, el motor que se usará para pintar los gráficos en el navegador. Aunque la gracia de three.js es utilizarlo con WebGL, se incluyen otras opciones como el *CanvasRenderer* para evitar incompatibilidades con navegadores antiguos. Si es posible, se recomienda utilizar el *WebGLRenderer* ya que su rendimiento es muy superior.

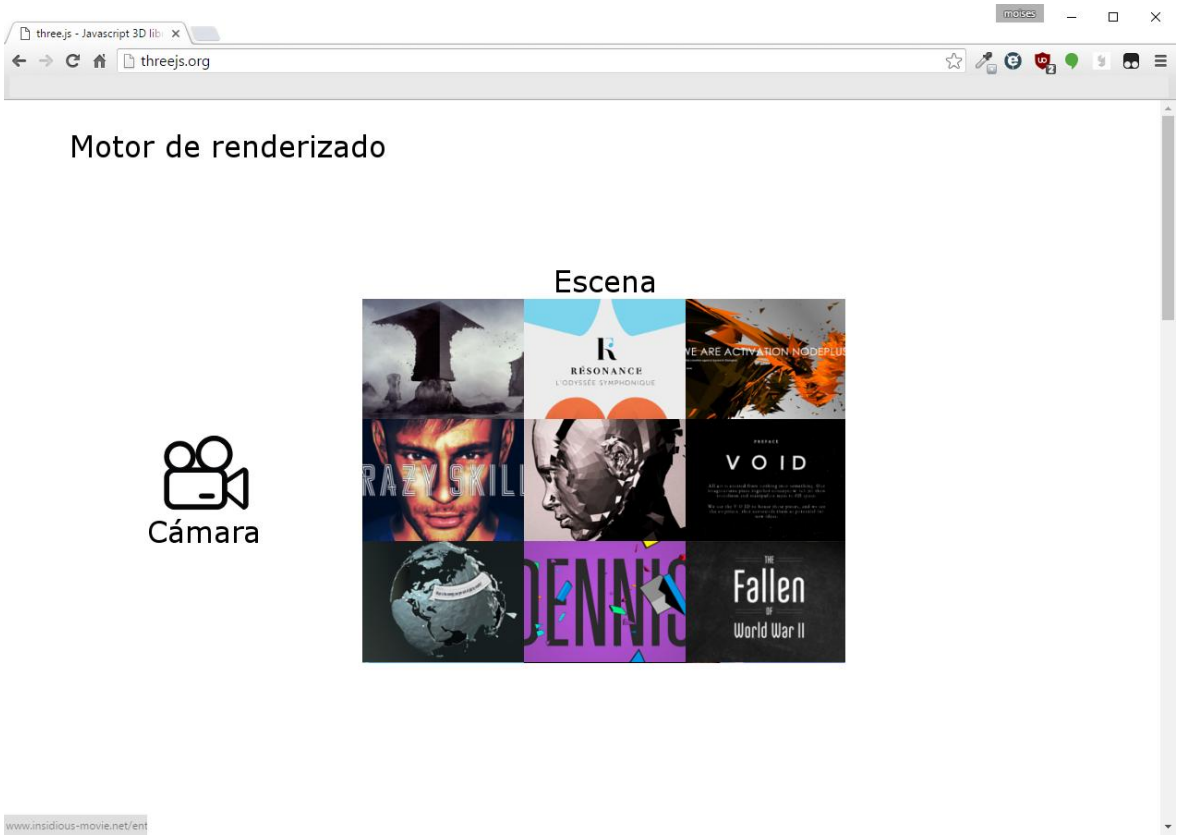

Ilustración 3.2 Esquema básico Three.js

Una vez se configuran estos elementos, lo siguiente es añadir elementos a la escena. Para añadir elementos necesitamos también dominar ciertos conceptos básicos.

Por un lado, tenemos la geometría, que no es otra cosa que la forma que tiene el objeto que estamos definiendo. Three.js dispone por defecto de una gran cantidad de formas geométricas, desde cubos a esferas complejas.

Por otro lado tenemos el material. El material es cómo se verá el objeto. Podemos definirlo de un color liso, con degradados, usando imágenes como texturas, etc.

Una vez definidos estos elementos, tenemos que juntarlos en una *Mesh* (malla) que será lo que añadamos a la escena.

La última pieza del puzle es la iluminación. Three.js dispone de varios tipos de luz. En nuestro caso usaremos una *directional light* (luz direccional). Esta fuente de iluminación actúa como un foco para el que definimos su posición y una serie de parámetros como su intensidad.

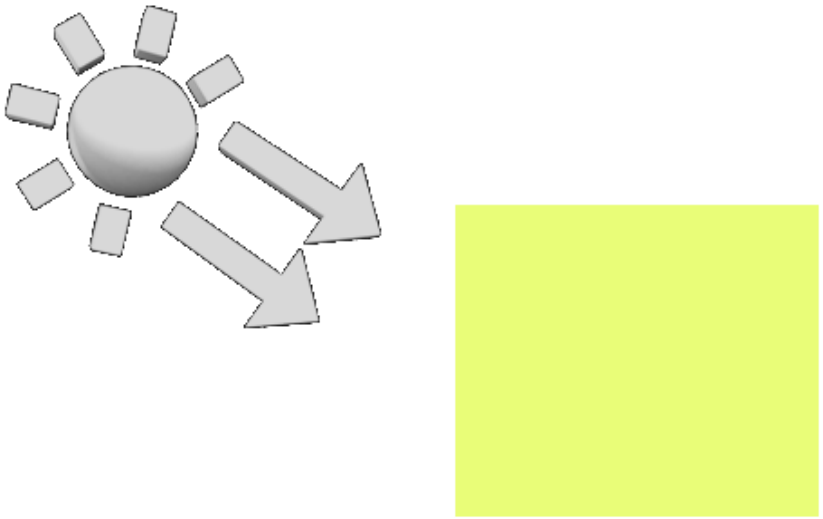

Ilustración 3.3 Luz direccional

Con estos conceptos básicos nos hacemos a la idea de cómo funciona la librería y de los elementos que son necesarios para nuestro proyecto.

También nos gustaría destacar un punto negativo que, en cierto modo, han afectado al proyecto.

Three.js es una librería, no es un framework propiamente dicho. De hecho, existen frameworks para three.js, como por ejemplo Whitestorm.js[10]. Esto nos limita, ya que no disponemos de un motor de físicas asociado a la librería, es algo que depende de la implementación concreta en cada caso. Se han visto proyectos con múltiples motores de físicas, como por ejemplo cannon.js[11] o physi.js[12], por lo tanto el diseño se tiene que adaptar a esta circunstancia.

3.1.4 Elementos necesarios

Una vez conocida la forma en la que trabaja Three.js, podemos plantear, de forma general los elementos que necesitamos.

La primera parte del desarrollo consistirá en crear una textura que haga de cielo. En las primeras versiones se utilizaba una imagen, pero el resultado no era muy satisfactorio, por lo que ha optado por generar la textura del cielo de forma procedural. Para crear la textura hemos utilizado el algoritmo de ruido de Perlin.

El otro subsistema se dedicará a los fenómenos en sí mismos, y debido a la problemática del motor de físicas, consistirá en un motor de partículas que, aparte de generar las partículas, las aporta movimiento, dando sensación de que tienen física.

Tanto el algoritmo de generación de ruido de Perlin como el motor de partículas son detallados en los puntos siguientes.

[10] http://whitestormjs.xyz/ (3/05/2016)

[11] http://www.cannonjs.org/ (25/02/2016)

[12] http://chandlerprall.github.io/Physijs/ (25/02/2016)

3.2 Fenómenos atmosféricos

Una vez decidido como se iba a desarrollar y debido a la naturaleza del proyecto, ha sido muy sencillo separar las distintas partes que lo componen. De esta forma, se permite al futuro usuario de la librería mayor libertad a la hora de utilizarla.
Las partes que componen la librería son las siguientes:

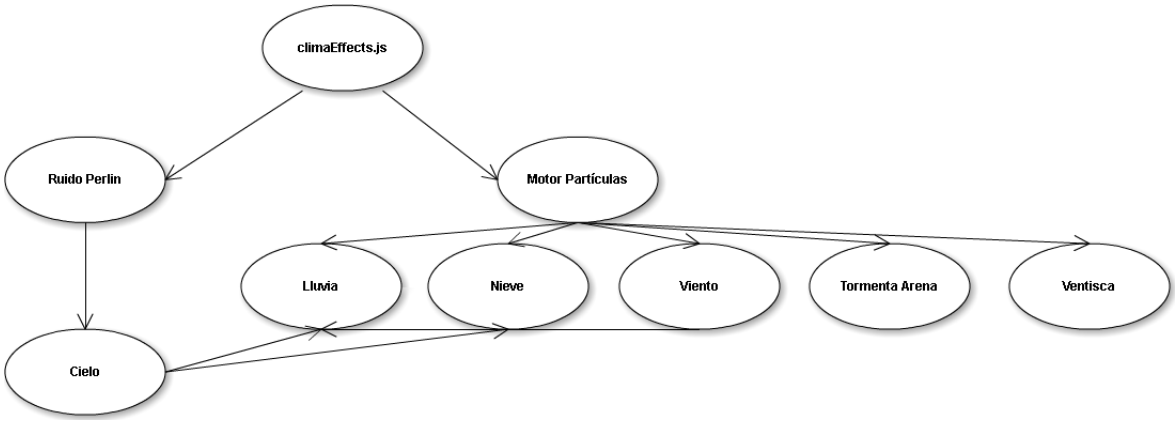

Ilustración 3.4 Esquema librería

En el diagrama podemos ver que los fenómenos son independientes, pero la lluvia y la nieve pueden depender del cielo y la lluvia se puede ver afectada por el viento.

3.2.1 Ruido de Perlin

El algoritmo de ruido de Perlin permite generar series de números aleatorios con cierta continuidad, lo que es muy útil a la hora de crear efectos como las nubes, ya que gracias a él se obtienen unos resultados muy naturales.

En 1982, Ken Perlin[13] diseñó este algoritmo para utilizarlo en la generación de texturas para la película Tron[14]. Este hecho no le sirvió para llevarme una estatuilla de la industria del cine americana, pero su algoritmo ha sido extensamente utilizado y versionado desde entonces. En el año 1997 la academia del cine americano le concedió un premio honorífico por este motivo.

El algoritmo de ruido de Perlin tiene un objetivo concreto. Obtener series numéricas que, aun siendo aleatorias, tienen continuidad.
El funcionamiento de este algoritmo es sencillo. Básicamente se trata de crear una serie de números aleatorios con una determinada frecuencia. A estos puntos se les aplica una función de interpolación para rellenar todo el dominio en el que se define la función.
A continuación se realiza la misma operación aumentando la frecuencia y reduciendo la amplitud de forma progresiva, tantas veces como consideremos.
Finalmente se suman todas las series de números obtenidas para generar una onda.

[13] https://mrl.nyu.edu/~perlin/ (12/06/2016)

[14] http://www.filmaffinity.com/es/film488334.html (12/06/2016)

Para generar la textura del cielo, implementaremos el algoritmo de ruido de Perlin para dos dimensiones, de esta forma nos será muy sencillo aplicarlo como textura posteriormente.

La amplitud inicial de la función será de 128. De esta forma, será muy sencillo normalizar los valores finales a la escala RGB(0...255).

Se van a desarrollar varias funciones de generación de combinación RGB en función de los datos obtenidos. De esta forma se pueden pintar cielos muy diferentes entre sí variando un parámetro.

3.2.2 Efectos de partículas

Como comentábamos anteriormente, la variedad de motores de físicas que se usan con three.js, nos suponía un problema, ya que los fenómenos atmosféricos se ven afectados por la física. Debido a esto, se ha creado un motor de partículas que no depende de ningún motor de físicas.

Este motor se ha realizado de forma genérica, para después con implementaciones y valores concretos, desarrollar las funciones para cada efecto.

Para crear el motor nos apoyaremos en el objeto `Points` de three.js. Este objeto está pensado para este uso en concreto y se apoya en el tipo `GL_POINTS` de WebGL.

De esta forma crearemos una serie de puntos. A partir de dichos puntos se asignarán distintas posiciones y propiedades en función de los parámetros que se le pasan al motor.

Adicional a la creación de las partículas, se ha añadido una función de animación de las mismas para suplir el problema de la ausencia de motor de físicas. Esta animación se basará en las propiedades de los puntos creados.

Se recrearán los siguientes fenómenos atmosféricos a partir del motor base aplicando distintos parámetros: Lluvia, nieve, tormenta de arena, ventisca y viento. Además, se podrá modificar el comportamiento de los mismos a través de parámetros, permitiendo usar texturas o colores definidos para las partículas y elegir el número de partículas en función de la calidad deseada.

El viento se ha implementado sólo para que funcione con la lluvia. Su implementación se ha realizado de una forma muy básica y sirve, sobre todo, para demostrar la sencillez a la hora de modificar el comportamiento de las partículas.

El último punto a destacar es que se permitirá la opción de que las partículas de lluvia o nieve caigan de las nubes en función de la textura generada. Esto hace que tenga un comportamiento mucho más realista y se haga un mayor aprovechamiento de la textura generada.

De esta forma construimos un motor de partículas versátil, ligero e independiente de un motor de físicas.

3.3 Entornos de pruebas

De cara a mostrar el uso de la librería, se van a crear dos entornos de pruebas, uno en 2D y el otro en 3D.

3.3.1 Entorno de pruebas 2D

En este entorno de pruebas, visualizaremos la textura generada para el cielo en 2 dimensiones. Se permitirá cambiar el tamaño de la textura así como la función de renderizado a utilizar.
Este entorno de pruebas también nos servirá para recoger los datos de tiempos de generación del cielo incluidos en el apartado 5 de este documento.

3.3.2 Entorno de pruebas 3D

En el entorno de pruebas 3D podremos ver en funcionamiento los fenómenos atmosféricos en una aplicación realizada con Three.js
En esta aplicación, un plano hará las veces de cielo y se le aplicará la textura generada. A partir de ahí, se podrán elegir aplicar los distintos efectos climatológicos, así como las variantes de los mismos.

4 Desarrollo

4.1 Generación de ruido de Perlin

En esta sección detallaremos la implementación del ruido de Perlin, así como las distintas funciones de normalizado y renderizado desarrolladas.

4.1.1 Implementación del ruido de Perlin

La implementación del algoritmo de ruido de Perlin ha consistido en dos grandes pasos, la creación de las ondas y la interpolación. El algoritmo consiste en la creación de una serie de valores a partir de una función dada e interpolarlos para rellenar todo el dominio de la función. A partir de ahí se generan ondas sucesivas reduciendo la amplitud y se suman a la inicial. En este apartado hablaremos de la generación de ondas, ya que la interpolación viene detallada en el anexo A.

En la parte de diseño hablamos de utilizar una amplitud máxima en la función de 128 para facilitar la normalización en RGB. Los resultados obtenidos con esta amplitud no han sido los esperados. La amplitud se reducía mucho en las últimas iteraciones del bucle dando lugar a valores muy pequeños. Reduciendo el número de iteraciones, se perdía realismo en el resultado por lo que se modificó este comportamiento para mantener 8 iteraciones que, tras las pruebas, otorga una buena relación tiempo de proceso/calidad.

La primera iteración sigue teniendo una amplitud de 128 pero, las siguientes, en vez de reducirse a la mitad de forma recurrente a partir de 128, se reducen desde el tamaño de la textura. De esta forma, obtenemos valores fuera del intervalo [-128,128] que nos servirán para que sean más evidentes a la hora de renderizar las secciones de nubes y de cielo.

Cabe destacar que la función elegida para generar los datos es un número aleatorio en el intervalo determinado por la amplitud.

4.1.2 Normalización y función de renderizado

A pesar de la modificación de la amplitud explicada en el apartado anterior, las funciones de normalizado y de renderizado no ha sido necesario modificarlas.

La función de normalizado recorre los datos generados anteriormente y los normaliza usando la siguiente formula

```
newValue = floor(128 + oldValue);
```

No ha sido necesario modificarla ya que, al crear el color RGB si le pasamos un valor por encima de 255, automáticamente lo interpreta como 255 y si le pasamos un valor por debajo de 0, automáticamente lo interpretará como 0. Esto, aparte de facilitarnos el trabajo, nos permite enfatizar más las zonas tanto de nubes, como de cielo despejado.

Se han desarrollado tres funciones de renderizado que generan cielos diferentes a partir de los mismos datos. A la hora de generar el cielo se le pasa a la función una constante como parámetro que representa el tipo de cielo que se quiere pintar.

Cielo Azul

Para pintar un cielo azul, llamaremos a la función con la constante BLUE como argumento.

La manera en la que se construye el color RGB es la siguiente:

```
RGB(min(value,230),min(value,230),255)
```

Las componentes *Red* y *Green* se limitan a 230 como máximo para evitar la saturación del color blanco, de esta forma se obtienen resultado más naturales. La componente *Blue* se mantiene fija en 255 de esta forma, cuando las otras componentes se alejan de 230, se generan siempre colores con tendencia azul. En la imagen que acompaña al texto se puede observar como al fijar la componente *Blue* en 255, los blancos que se generan tienen cierto tono azulado, por lo que se consigue un efecto más realista al dar la sensación de transparencia.

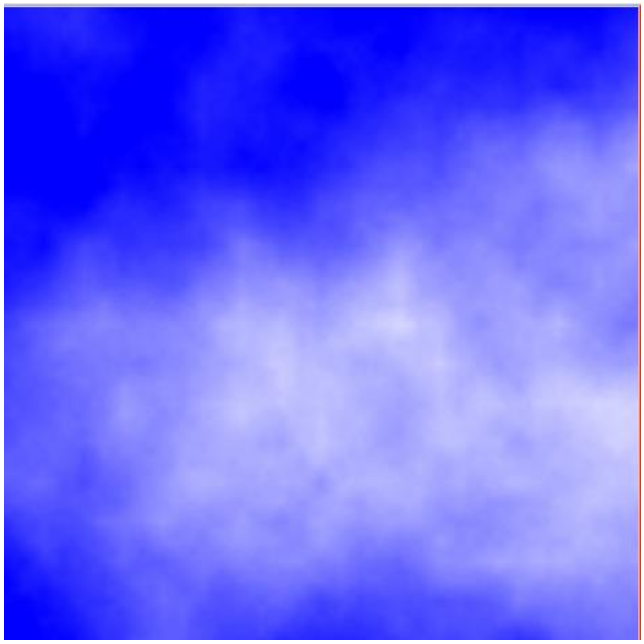

Ilustración 4.1 Cielo azul

Cielo Oscuro
Para pintar un cielo oscuro, llamaremos a la función con la constante DARK como argumento.
La manera en la que se construye el color RGB es la siguiente:

```
RGB(min(max(25,value),230),min(max(25,value),230),min(max(25,value),230))
```

Las componentes tres componentes se limitan entre 25 y 230 para evitar la saturación del color blanco y del negro, de esta forma obtenemos resultados más naturales. Al ser el mismo valor para las tres componentes se consiguen colores en la escala de grises, dejando el cielo de color negro y las nubes blancas.

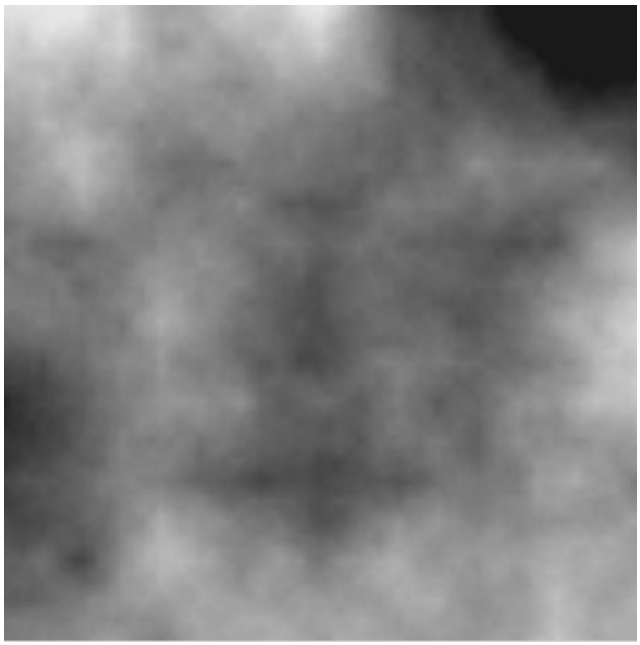

Ilustración 4.2 Cielo oscuro

Cielo Extraño
Para pintar un cielo extraño, llamaremos a la función con la constante STRANGE como argumento.
La manera en la que se construye el color RGB es la siguiente:

```
RGB(max(100,value),value,max(100,value))
```

Las componentes *Red* y *Blue* se limitan a 100 como máximo para que el color morado sea el predominante. La componente *Green* simplemente usa el valor, sin limitaciones, ya que no tenemos el problema de la saturación debido a la limitación de las otras componentes.

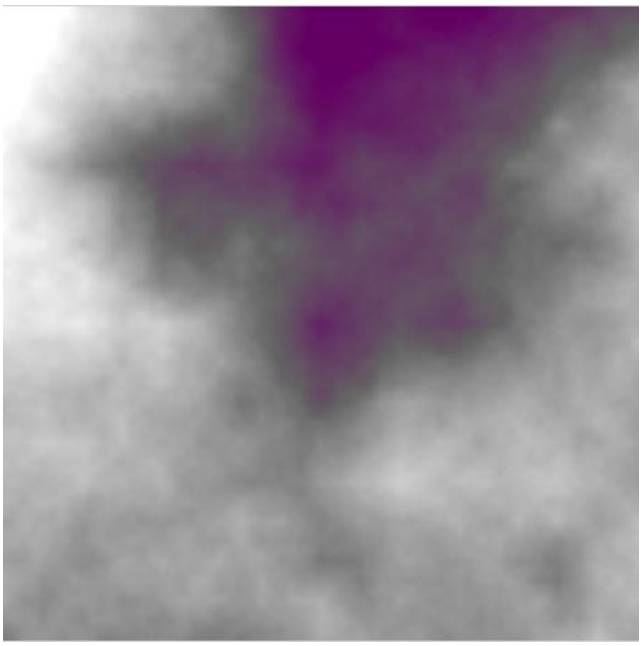

Ilustración 4.3 Cielo extraño

En cada punto se pinta el valor obtenido en un canvas que hemos creado previamente. Una vez terminado el proceso, la función devuelve el canvas ya pintado para su posterior uso.

A la hora de aplicar la textura en Three.js se hicieron pruebas de pintado de los puntos directamente sobre el material pero, los resultados eran muy inferiores a los obtenidos al pintar en el canvas. Three.js permite utilizar un canvas como textura, por lo tanto se decidió utilizar esa técnica al obtenerse mucho mejores resultados.

4.2 Generador de partículas

El otro punto clave del proyecto es el generador de partículas. En esta sección veremos en detalle como se ha llevado a cabo la implementación, así como los problemas que han ido surgiendo y las soluciones alcanzadas.

4.2.1 Motor de partículas

El motor es el núcleo del generador y es el encargado de la creación de las partículas. Más adelante hablaremos de las implementaciones concretas de cada efecto, pero a nivel general, el motor crear una serie de partículas en posiciones aleatorias y les añade una gravedad aleatoria. Además se incluyen varias propiedades al sistema de partículas que serán de utilidad de cara a las animaciones.

Para crear las partículas, utilizaremos una geometría de Three.js a la que iremos añadiendo puntos definidos en los ejes *x*, *y*, *z*.
Para determinar las posiciones iniciales se han determinado dos grupos con sus particularidades.
Por un lado tenemos los efectos que caen del cielo como la lluvia o la nieve. Estos efectos tienen dos formas de generar su posición inicial.

La forma más básica consiste en, determinando unas coordenadas *x*, *y*, *z*, un tamaño y el valor de *y* al que se alcanza el suelo, se generaran los puntos en los ejes *x*,*z* calculando un valor aleatorio en el intervalo [-tamaño/2 + (k), tamaño/2 + (k)] siendo k el valor inicial de *x* ó *z* en cada caso. Para el eje *y*, se generar las partículas desde la *y* inicial hasta una altura máxima de *y* + el valor de y al alcanzar el suelo. De esta forma se evita la perdida de continuidad en la primera oleada de partículas.

La forma avanzada de crear las partículas permite su creación en función de los datos del cielo. Para poder decidir las posiciones en las que se pueden crear las partículas, esta implementación exige un paso previo.
Este paso consiste en normalizar las posiciones del array original de datos al tamaño del elemento en el que se aplica, es decir, aunque la textura generada sea de 1024x1024, esta se puede aplicar a una geometría de, por ejemplo, 2048x2048 por lo que es necesaria esta normalización. Para llevarla a cabo, se ha definido un objeto que contiene 2 arrays, uno para las posiciones de *x* y otro para las de *z*. Se recorre la serie de datos original buscando valores mayores de 0. Este valor se ha elegido al ser un punto intermedio, pero se podría modificar para conseguir otros efectos. Una vez se encuentra un punto que cumple la condición se normaliza multiplicando utilizado la siguiente fórmula:

```
newValue = index * (size/dataLength) - (size/2);
```

Simplemente lo que hacemos es multiplicar la posición en la que se encuentra el valor inicial por la relación entre el tamaño de la geometría y el tamaño de la textura original. A continuación se le resta la mitad del tamaño de la geometría. Esto es necesario ya que, para three.js, las coordenadas de la geometría representan el centro de la misma, por lo que tenemos que expandirnos la mitad del tamaño de la geometría en las 4 direcciones.
Una vez terminado este proceso, a la hora de elegir los valores *x,z* iniciales del punto, se coge un valor al azar entre 0 y la longitud de los nuevos arrays y se aplica el valor que contiene el array en ese punto. Para la posición inicial en *y* se utiliza el mismo método que en la forma básica.

El otro grupo de efectos son aquellos que vienen desde el horizonte, es decir, las tormentas de nieve y de arena.
Para calcular la posición inicial en los ejes *x,z* de estas partículas se utiliza el mismo método que en el modo básico de las que caen del cielo, ya que al situarse por todo el área definida, no se produce perdida de continuidad en la animación. Para calcular la posición inicial en *y*, se genera un valor aleatorio entre la *y* de origen y el suelo. De esta forma se cubre de partículas toda la superficie definida en los tres ejes.

A la hora de asignar la gravedad a las partículas, nos encontramos de nuevo con los mismos grupos. En ambos casos, es necesario definir las gravedades base en los tres ejes a la hora de llamar a la función del motor de partículas.
Para los efectos que caen del cielo, se les asigna una aceleración en el intervalo [-k,k] siendo k el valor del parámetro de la gravedad base, ya sea en *x* o en *z*. De este modo se consigue un efecto de caída más natural que si solo se desplazara en el eje *y*. Respecto a la aceleración en el eje y, se situará en el intervalo [k,2k] siendo k la aceleración base. De esta forma nos aseguramos que siempre tenga la misma dirección, pero distintas aceleraciones.
En el caso de los efectos que vienen desde el horizonte, el proceso es similar pero invirtiendo el comportamiento de los ejes *z*, *y*.

Por último el motor guarda varias propiedades heredadas de los parámetros recibidos de cara a usarlos en la función de animación de una manera más sencilla. Las propiedades son las siguientes:

`size`: Tamaño de la superficie del motor de partículas
`rootX`: Punto origen en *X*
`rootY`: Punto origen en *Y*
`rootZ`: Punto origen en *Z*
`floor`: Distancia desde la *Y* origen hasta el suelo
`Xgravity`: Gravedad base en *X*
`Ygravity`: Gravedad base en *Y*
`Zgravity`: Gravedad base en *Z*
`wind`: Si hay viento activo
`windConfig`: Objeto vacio para su uso posterior en caso de necesitar viento
`fromData`: Objeto que contiene los arrays de posiciones validas en función de los datos

4.2.2 Animación de partículas

La animación de partículas consiste, básicamente, en una función que recorre los puntos del sistema de partículas y les asigna una nueva posición en función de la aceleración.

Antes de asignar la nueva posición, se comprueba si la partícula ha superado el límite marcado como suelo. En el caso de los efectos que caen del cielo, el límite se controla en el eje *y*, en el caso de los fenómenos que vienen desde el horizonte, se controla en el eje *z*.
Si la partícula ha alcanzado el suelo, en vez de destruirla y crear una nueva, lo que hacemos es desplazarla hasta una posición inicial aleatoria de la misma forma que se creaban al principio. La única particularidad respecto a la forma de designar la posición inicial en este caso es la manera en la que se calcula la y inicial. En el caso de la creación de las partículas, la y variaba desde la posición base en *y* hasta el la distancia al cielo (en este caso tomada por encima para evitar discontinuidades). En este caso, la *y* varía desde la *y* base hasta 1/5 de la distancia al suelo. Esta altura es suficiente para evitar que se pierda la continuidad.
Además de la posición, también se reinicia la aceleración. En este caso se repite completamente el proceso utilizado en la creación de las partículas. Con este cambio de gravedad, acentuamos aún más la sensación de aleatoriedad típico de este tipo de fenómenos, consiguiendo un resultado más natural.

A la hora de aplicar la gravedad a las partículas, se ha añadido un sistema de viento. Este viento sólo afecta a la lluvia ya que se ha creado a efectos demostrativos y tiene que ser creado explícitamente como se explica en el anexo B.
Se han creado tres tipos de viento por defecto: *Fitfully* (racheado), *Uniform* (uniforme) y *Variable* (variable).
El viento uniforme aplica una aceleración constante en los ejes *x* y *z*.
El viento racheado aplica una aceleración variable en los ejes *x* y *z*. Esta aceleración varía creciendo y decreciendo pero siempre en la misma dirección.
El viento variable aplica una aceleración variable en los ejes *x* y *z*. Esta aceleración se mantiene constante en valor pero varía la dirección en la que se aplica.

En el momento en el que se aplica la gravedad a la partícula se aplica la aceleración base de la misma y, si está activado, la aceleración provocada por el viento. Aplicar la aceleración consiste en incrementar la posición de la partícula en cada uno de los tres ejes con los valores anteriormente indicados.

4.2.3 Funcionalidades adicionales

Existen dos parametrizaciones adicionales a las comentadas anteriormente.
La primera de ellas es la calidad a la que queremos renderizar. Este parámetro se ha añadido para que se pueda utilizar de forma sencilla en equipos modestos sin afectar al rendimiento. En función de la calidad se crea un número de partículas determinado. Se han definido constantes que determinan la cantidad de partículas tal y como se indica en la siguiente tabla:

Constante	# de partículas
LOW	300
MEDIUM	3000
HIGH	30000
ULTRA	300000

Tabla 4-1 Cantidad de partículas en función de la calidad

La segunda parametrización sirve para pintar las partículas a partir de colores o de texturas. Se han definido las siguientes constantes por defecto pero pueden ser modificadas en caso de ser necesario.

Colores

Constante	Valor
RAINCOLOR	0x0000ff
SNOWCOLOR	0xffffff
SANDCOLOR	0x96612D

Tabla 4-2 Constantes de color definidas

Texturas

Constante	Valor
RAINTEXTURE	./images/rain.png
SNOWTEXTURE	./images/snow.png
SANDTEXTURE	./images/sand.png

Tabla 4-3 Ruta de texturas por defecto

Ilustración 4.4 Detalle de texturas utilizadas

De cara a facilitar el trabajo con la librería, se han creado una serie de funciones con valores por defecto para aplicarlas sin necesidad de conocer cómo funciona por debajo. En el anexo B, se explicarán en detalle estas funciones, así como un uso más avanzado de la librería.

5 Integración, pruebas y resultados

En esta sección detallaremos como se han desarrollado los entornos de pruebas y los resultados obtenidos. Estos entornos de pruebas forman parte de la demostración que se hará en la defensa del proyecto.

5.1 Entorno de pruebas 2D

Para la creación del entorno de pruebas en 2D se ha creado un archivo HTML llamado index2d.html.
Este entorno consiste en un canvas que nos servirá para pintar el cielo generado y unos controles que nos permitirán cambiar el tamaño y la función de renderizado a usar.
Para crear los controles se ha utilizado la librería dat.gui.js que facilita mucho esta labor. Esta librería es la más comúnmente usada en los desarrollos de este tipo.

Además de servirnos para visualizar los resultados del cielo, este entorno de pruebas se ha utilizado para calcular los tiempos de ejecución del algoritmo para distintos tamaños. En el tiempo de ejecución se ha incluido la renderización ya que a la hora de aplicarlo como textura en el entorno 3D, se ha de renderizar previamente, por lo tanto es necesario tener este tiempo en cuenta. En la siguiente tabla se muestran los resultados para los distintos tamaños.

Tamaño (px)	Tiempo (ms)
256x256	173
512x512	1014
1024x1024	3363
2048x2048	15588

Tabla 5-1 Tiempos de ejecución

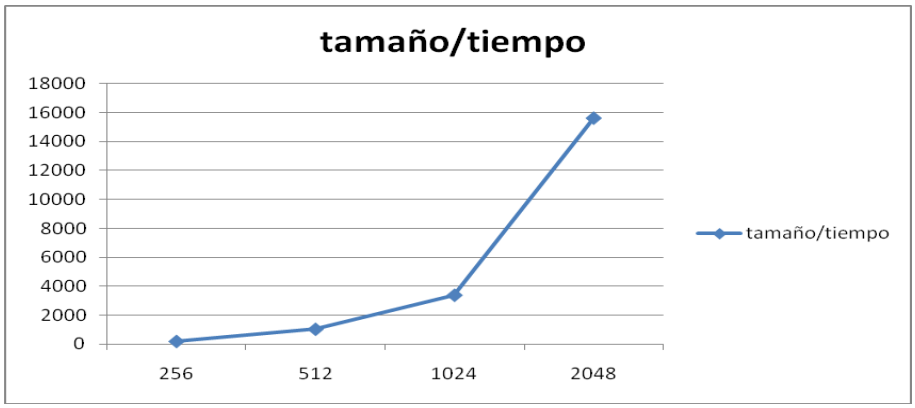

Ilustración 5.1 Gráfica de tiempos de ejecución

Como vemos en la gráfica, el crecimiento se intuye exponencial como era lo esperado. El tiempo de creación de una textura de 1024x1024 que es la que usamos en el tester3D ronda los 3 segundos lo que supone una buena relación calidad/tiempo.

5.2 Entorno de pruebas 3D

El principal uso de este entorno de pruebas ha sido el de ajustar el comportamiento de los efectos para hacerlos más naturales y comprobar que funcionan correctamente.

El entorno de pruebas 3D cuenta con un plano en el que se pintará la textura y que servirá como referencia para los fenómenos que se muestren en pantalla.
Para realizar los controles del entorno nos hemos apoyado, al igual que en el entorno de pruebas 2D, en la librería dat.gui.js. Además, para controlar la cámara se ha recurrido a la librería OrbitControls,js, de amplio uso en aplicaciones realizadas con Three.js.

Los controles disponibles son los siguientes:
Quality: Permite seleccionar las calidades disponibles que, a su vez, varían el número de partículas simultaneas en pantalla. Las calidades disponibles son *Low*, *Medium*, *High* y *Ultra* y responden a los valores indicados en la tabla 4-1
Type: Esta opción nos permite cambiar la función de renderizado que se aplica al cielo. Están disponibles las tres opciones comentadas anteriormente, es decir, *Blue*, *Dark* y *Strange.*
Effect: Con esta opción elegimos el fenómeno atmosférico que queremos visualizar. Nos permite elegir entre *Rain* (lluvia), *Snow* (nieve), *SandStorm* (Tormenta de arena) y *Blizzard* (Ventisca).
Texture: Opción booleana que controla si se aplica textura a las partículas o no.
Wind: Activa o desactiva el viento.
FromData: Genera los efectos a partir de los datos o de forma genérica. De esta forma se pueden visualizar las diferencias entre un método y otro.

Ilustración 5.2 Tester 3D

6 Conclusiones y trabajo futuro

6.1 Conclusiones

De cara a analizara las conclusiones, nos hemos decidido centrar, por un lado, en las ventajas y limitaciones del trabajo realizado y, por otro en el posible trabajo futuro ya que este trabajo no es más que una pequeña parte de lo que podría ser una librería completa.

6.1.1 Ventajas

La facilidad de uso es la ventaja principal de esta librería ya que no se necesita conocer su funcionamiento interno para usarla. También en este sentido, si se quiere un uso más avanzado, también se da la posibilidad, lo que la convierte en una herramienta muy versátil.
Al no depender de un motor de físicas, se ha conseguido una librería más eficiente y, al contar con varias calidades definidas por defecto, permite que se adapte y consiga buen rendimiento en equipos de todas las gamas.

6.1.2 Limitaciones

Por un lado nos hemos encontrado con limitaciones gráficas. De cara a crear fenómenos complejos como por ejemplo, un huracán, necesitaríamos trabajar directamente a bajo nivel con WebGL ya que las herramientas que nos proporciona Three.js son limitadas.
Por otra parte, al no depender de un motor de físicas, lo que es una ventaja se convierte también en un inconveniente, ya que las partículas no tienen interacción con el resto del entorno. Quedaría en manos del programador que implemente la librería integrar estas partículas con el motor de físicas que esté utilizando.

6.1.3 Trabajo futuro

Como comentábamos al principio de esta sección, para tener una librería completa hay mucho posible trabajo por hacer. A continuación se detallan los puntos más importantes que han quedado fuera de la librería final.

Motor de físicas
A pesar de que lo ideal es que se mantenga independiente de un motor de físicas, de cara a crear efectos complejos, nos sería de gran ayuda. Por lo tanto podría realizarse un estudio acerca de los motores más utilizados y aquellos que van integrados en los frameworks basados en Three.js y seleccionar el más utilizado. Otra solución más compleja sería hacer varias implementaciones para varios motores de físicas, aunque no parece una opción muy recomendable.

Niebla
Three.js dispone de una niebla por defecto pero más que una niebla como fenómeno atmosférico, es una niebla que permite ahorrar recursos a la hora de renderizar. Un posible desarrollo sería crear una niebla de mejor calidad en términos visuales que la que ofrece Three.js

Tormentas

Otra de las posibles mejoras, sería crear efectos que aparte de visuales, sean sonoros. Por ejemplo, podrían crearse tormentas, que al crearse un rayo aparte de verse, tenga un sonido de trueno acompañándolo. Three.js incluye un gestor de audio muy básico por lo que lo ideal sería utilizar alguno más avanzado. Además habría que controlar la distancia del rayo respecto a la cámara para calcular el tiempo que tarda en reproducirse el audio. Es un desarrollo de bastante complejidad pero que podría aportar muy buenos resultados.

Referencias

Libros

1.- Dirksen J. (2015) *Learning Three.js The JavaScript 3D Library for WebGL* Second Edition

2.- Sukin, Isaac (2013) *Game Development with Three.js*

Recursos web

3.- Three.js - Javascript 3D Library. http://threejs.org/

4.-Wikipedia la enciclopedia libre. https://es.wikipedia.org

5.- Ruido de Perlin Lambda lab.85 http://www.lanshor.com/ruido-perlin/ (22/01/2016)

6.- Stemkoski Three.js examples https://stemkoski.github.io/Three.js/ (13/02/2016)

Glosario

- **API** Application Programming Interface. Conjunto de funciones que ofrece una librería para su uso en programas externos.

- **Librería** Conjunto de funciones.

- **Shader** Porción de código utilizada para aplicar transformaciones o efectos en una superficie.

- **Proyección en perspectiva** Tipo de proyección que consiste en dirigir los objetos de la escena hacia un punto de fuga. De esta forma los objetos lejanos se verán más pequeños y los cercanos más grandes

- **Framework** Entorno de trabajo. En ocasiones estos frameworks se basan en lenguajes concretos, en otras ocasiones giran en torno a una temática (videojuegos, por ejemplo) o incluso se desarrollan frameworks específicos para la integración de varias librerías.

- **Procedural** Generado a partir de un procedimiento. En nuestro caso concreto, generamos la textura de forma procedural.

- **RGB** Red/Green/Blue. Modelo de colores basado en la suma de sus tres componentes, que representan los colores primarios (Rojo, Verde y Azul)

- **Canvas** Elemento HTML usado para renderizar. La traducción literal es lienzo.

Anexos

A. Interpolación bilineal

La interpolación, como concepto general, nos permite aproximar el valor de una función en un punto a partir de los valores que lo rodean. De esta forma, podemos aproximar los valores que desconocemos a lo largo del dominio de una función.

Existen varias funciones de interpolación pero, en nuestro caso, hemos optado por la interpolación bilineal al ser la más sencilla. Los resultados obtenidos son suficientemente buenos como para no complicar el proceso y para cumplir el compromiso entre el tiempo de proceso y calidad del resultado.

La interpolación bilineal se aplica en las dimensiones *x* e *y* siendo una extensión de la interpolación lineal. Primero se aplica en el eje *y* y a continuación en el eje *x* dando lugar a una función cuadrática.

La interpolación lineal es un método muy sencillo que consiste en trazar una recta que une dos puntos dados y calcular los puntos intermedios a partir de esta recta obtenida.

A continuación se detalla la función de interpolación. Los datos y la imagen que acompaña al texto han sido extraídos de la wikipedia.

De esta forma, si buscamos el valor del P = (x, y) y conocemos el valor de f en los cuatro puntos Q11 = (x1, y1), Q12 = (x1, y2), Q21 = (x2, y1) y Q22 = (x2, y2) obtenemos la siguiente formula.

$$
\begin{aligned}
f(x,y) \approx & \frac{f(Q_{11})}{(x_2-x_1)(y_2-y_1)}(x_2-x)(y_2-y) + \\
& \frac{f(Q_{21})}{(x_2-x_1)(y_2-y_1)}(x-x_1)(y_2-y) + \\
& \frac{f(Q_{12})}{(x_2-x_1)(y_2-y_1)}(x_2-x)(y-y_1) + \\
& \frac{f(Q_{22})}{(x_2-x_1)(y_2-y_1)}(x-x_1)(y-y_1) \\
= & \frac{1}{(x_2-x_1)(y_2-y_1)}\Big(f(Q_{11})(x_2-x)(y_2-y) + \\
& \qquad f(Q_{21})(x-x_1)(y_2-y) + \\
& \qquad f(Q_{12})(x_2-x)(y-y_1) + \\
& \qquad f(Q_{22})(x-x_1)(y-y_1) \quad \Big).
\end{aligned}
$$

Ilustración A.1 Formula interpolación bilineal

Podemos observar como en este caso, en vez de obtener el punto a través de los valores de una recta, lo hacemos a través de áreas. En la imagen siguiente se explica de forma gráfica.

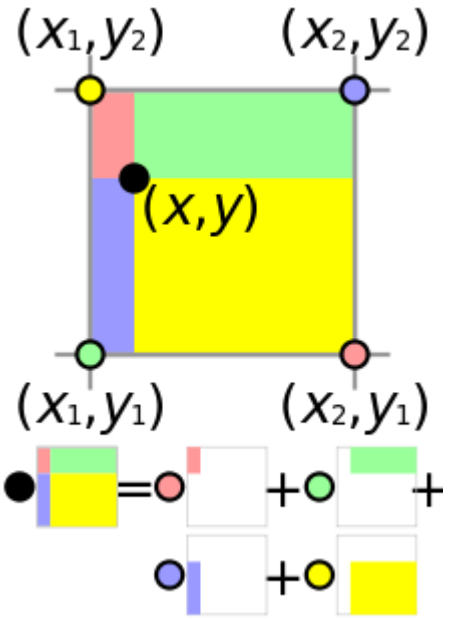

Ilustración A.2 Interpolación bilineal gráfica

Aplicando esta técnica en nuestro proyecto conseguimos que, a partir de los valores iniciales que se calculan de forma aleatoria, los valores adyacentes tengan continuidad.

B. Manual del programador

Manual Básico

Para poder utilizar la librería se debe importar al archivo HTML el fichero climaeffects.js

```
<script src="climaeffects.js"></script>
```

Para crear la textura del cielo hay que llamar a la función *makeCloudCanvas()* queda a elección del programador la parte de la aplicación en la que llamar a esta función. Teniendo en cuenta los tiempos de ejecución sería recomendable llamarla en la fase de carga de la aplicación.

```
//Creamos la textura usando el metodo makeCloudCanvas()
var dt = new THREE.Texture(makeCloudCanvas());

//Le decimos que es necesario actualizar para mostrar la
//textura generada
dt.needsUpdate = true;
```

Para crear los efectos meteorológicos se debe llamar a la siguiente función:

```
createEffect(quality, effect, texture);
```

El parámetro `quality` es de tipo *string*. Los valores permitidos son los de las constantes `LOW, MEDIUM, HIGH, ULTRA.`

El parámetro `effect` es de tipo *string*. Los valores permitidos son los de las constantes `RAIN, SNOW, SANDSTORM, BLIZZARD.`

Manual avanzado

Para poder utilizar los datos de la textura del cielo a lo largo del desarrollo, se da la posibilidad de generar primero los datos y luego la textura. Para ello utilizaremos la siguiente función.

```
var textureData = getCloudData();
```

Esta función nos devuelve un array bidimensional de 1024x1024. Para renderizarlo, llamaremos a la función:

```
textureData.toThree(renderType)
```

El parámetro `renderType` es de tipo *string*. Los valores permitidos son los de las constantes `BLUE, DARK, STRANGE.`

Además de para elegir el tipo de renderizado, esto nos servirá para crear los efectos en función de los datos de la textura a través de la siguiente función.

```
createEffectFromData(quality,effect,texture,textureData)
```

La diferencia entre esta función y `createEffect()` es el último parámetro que corresponde con el array de datos generado previamente.

Para crear viento primero hay que crear el efecto *RAIN* y a continuación hay que llamar a la función:

```
setupWind(type);
```

El parámetro `type` es de tipo *string*. Los valores permitidos son los de las constantes `FITFULLY, UNIFORM, VARIABLE.`

Adicional a esto, se da la opción de llamar a la función que crea los efectos directamente para que los programadores más avanzados puedan personalizar sus efectos. El prototipo de la función es el siguiente:

```
function createParticleSystem(quality,size,floorLimit,
rootX,rootY,rootZ,XbaseGravity,YbaseGravity,
ZbaseGravity,particleColor,particleSize,withTexture,sand
,fromData,rawData)
```

El parámetro `quality` es de tipo *string*. Los valores permitidos son los de las constantes `LOW, MEDIUM, HIGH, ULTRA.`

El parámetro `size` es de tipo *integer*. Define la superficie que cubre tanto en *x* como en *z* el efecto.

El parámetro `floorLimit` es de tipo *integer*. Define la distancia entre el punto que se generan los efectos en *y* y el suelo.

Los parámetros `rootX,rootY,rootZ` son de tipo *integer*. Definen las coordenadas iniciales para el efecto.

Los parámetros `XbaseGravity,YbaseGravity,ZbaseGravity` son de tipo *integer*. Definen la gravedad base en los tres ejes.

El parámetro `particleColor` es de tipo *string*. Define el color que tendrá la partícula, también soporta rutas para definir texturas.

El parámetro `particleSize` es de tipo *integer*. Define el tamaño de las partículas.

El parámetro `withTexture` es de tipo *boolean*. Si se pasa el valor *false,* el valor de `particleColor` tiene que ser un color, si no, tiene que ser la ruta a una textura.

El parámetro `sand` es de tipo *boolean*. Define si las partículas caen desde el cielo (*false)* o vienen desde el horizonte (*true*).

El parámetro `fromData`es de tipo *boolean*. Si se van a usar los datos para generar el efecto, debe valer *true*.

El parámetro `rawData` es de tipo *integer[][]*. Array que contiene los datos de la textura del cielo.