



**PROYECTO FIN DE CARRERA**

“EDICIÓN E IMPRESIÓN DE MODELOS 3D”

Titulación: Ingeniería Informática (2º ciclo)

Departamento de Ingeniería Matemática e Informática

Alumno: Gonzalo Andrade Benavente

Teléfono: 677562988

Tutor: Óscar Ardaiz

Pamplona, 10 de septiembre de 2014

# Índice

[Índice 3](#_Toc414896825)

[Resumen 4](#_Toc414896826)

[Introducción 5](#_Toc414896827)

[JavaScript 6](#_Toc414896828)

[Three.js 7](#_Toc414896829)

[Creando la escena 7](#_Toc414896830)

[Escena 8](#_Toc414896831)

[Cámara 8](#_Toc414896832)

[Renderizado 9](#_Toc414896833)

[Función *render* 10](#_Toc414896834)

[Mapbox 13](#_Toc414896835)

[Edición de mapas 13](#_Toc414896836)

[Crear proyecto 13](#_Toc414896837)

[Uso de Mapbox 15](#_Toc414896838)

[Cesium 18](#_Toc414896839)

[Uso de Cesium 18](#_Toc414896840)

[Shapeways 21](#_Toc414896841)

[Bibliografía 22](#_Toc414896842)

# Resumen

Hoy en día un tema recurrente es el 3D, desde la creación de modelos en el ordenador hasta la creación de una impresora que nos permita crear esos modelos y que sean reales.

# Introducción

# JavaScript

# Three.js

A grandes rasgos three.js es una librería que nos permite crear WebGL (3D en el navegador), básicamente la diferencia cabe en que si queremos crear un cubo con three.js basta con algunas líneas en cambio con JavaScript la cantidad de líneas sería mayor.

Al igual que la mayoría de librerías (frameworks) podemos descargar la librería completa desde la página web oficial o acceder a una versión más ligera a través de su cdn (content delivery network).



Figura x.x - cdn three.js

Para mostrar gráficos en 3D en el navegador (WebGL) con three.js debemos crear una serie de elementos y funciones.

## Creando la escena

Para lograr un código más legible y manejable he realizado durante todo el proyecto una programación modular, así cada módulo resuelve un problema particular.

Para visualizar gráficos en 3D con three.js como elementos básicos necesitamos una escena, una cámara y un renderizado de la escena con la cámara. La página *html* contiene las etiquetas básicas de cualquier página, si nos hemos descargado la librería debemos indicar cada fichero javaScript que usemos durante la programación.

Para un código más ordenado indicaremos las librerías javaScript entre la etiqueta *<head>* y en la etiqueta *<body>* llamaremos al método *onload* indicando la función con la cual cargaremos todos los elementos de la página.

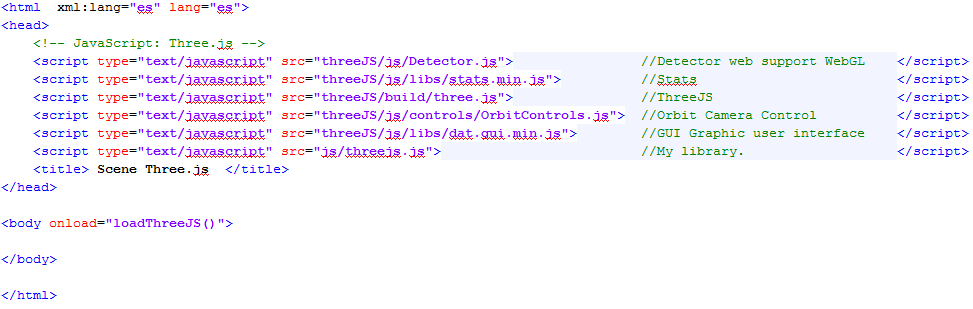
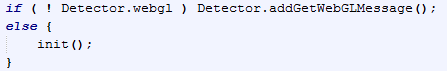


Figura x.x - PFCMyScene.html

Como se puede apreciar la página inicial carece de elementos ya que todos los elementos serán creados mediante javaScript modificando el DOM (Document object model), el fichero *threejs.js* es aquel que creara la escena.

Antes de crear la escena es muy útil llamar a la librería *Detector* ubicada en /*js/Detector.js* para indicarnos si nuestro navegador soporta WebGL, si todo va bien llamaremos a la función *init* que se encargará de cargar todos los elementos antes descritos más otros para lograr una escena más completa.



Si nuestro navegador permite el uso de WebGL llamamos a la función *init* que como su nombre lo indica se encarga de inicializar todos los elementos necesarios para crear una escena. Crearemos un objeto contenedor donde vamos a *renderizar* la escena, este será una etiqueta *div* (bloque de contenido o sección de página) y la añadimos a nuestra página.



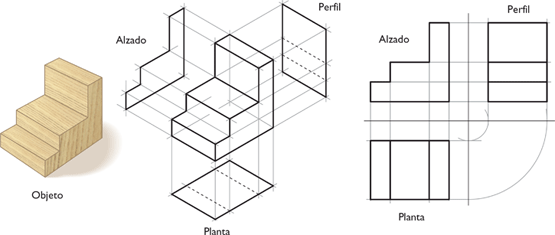
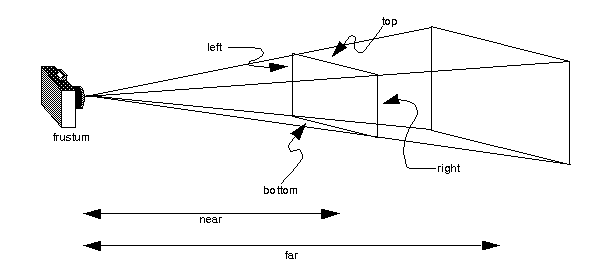
### Escena

La creación de la escena es bastante simple, solamente debemos crear el objeto y para añadir objetos usaremos la función *add*.



### Cámara

Three.js nos ofrece cuatro tipos de cámaras:

* Camera: esta es una clase abstracta de la cual heredan las otras.
* CubeCamera: Crea 6 cámaras para renderizar un tipos especiales de gráficos 3D, es una cámara que no use durante mi proyecto.
* OrthographicCamera: Cámara con una proyección ortográfica. Una proyección ortográfica es un sistema de representación gráfica que consiste en representar elementos geométricos o volúmenes en un plano mediante proyección ortogonal.
* PerspectiveCamera: Cámara con una proyección perspectiva. Este método consiste en proyectar puntos hacia el plano de visión, con este tipo de proyección parece todo más real ya que es la manera de formar las imágenes en el lente de las cámaras, como lo ve el ojo humano.  
  

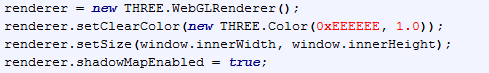
 Las cámaras tienes atributos similares como *near* y *far*,tal como lo indica su traducción es la posición de la cámara hacia el plano cercano y lejano respectivamente y nos indica que en ese intervalo serán mostrados los objetos en la escena. Para la cámara en perspectiva se necesitan dos valores más para los atributos *fov* y *aspect*, el primero es el campo de visión desde el fondo hasta el inicio en grados y el segundo es el radio de aspecto que siempre tendrá el valor del ancho de la ventana entre la altura de la misma.

Con la cámara creada debemos establecer su posición y hacia donde apunta, en este caso apuntará hacia la escena.



### Renderizado

En este punto ya hemos creado la cámara y la escena, ahora debemos crear el objeto de renderizado y añadirlo al DOM, es decir, añadirlo a nuestra página.



Como se puede ver en el fragmento de código solo hemos creado el objeto de renderizado añadiendo algunos valores a los atributos color de fondo, tamaño y muestra de sombra habilitada.

Para que nuestra escena sea dinámica he añadido un control orbital para la cámara, la idea es que podamos manejar a nuestro antojo la cámara por la escena. Necesitaremos añadir la librería ubicada en /*js/controls/OrbitControls.js* y crear el objeto *OrbitControls* asociando la cámara y el renderizado.



Una vez creado todos los objetos debemos añadir al DOM él renderizado y luego asociar la escena y la cámara, para esta acción crearemos una función llamada *render* la cual además de asociar el renderizado con la escena y la cámara nos proporcionará una escena más dinámica.

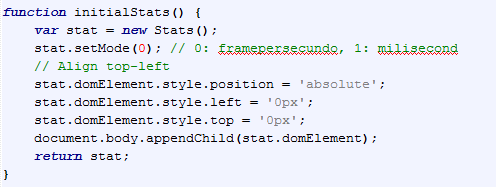


### Función *render*

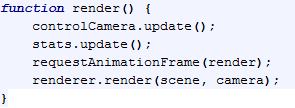
Como objetivo principal esta función asocia al objeto de renderizado la escena y cámara. Si asociamos los elementos descritos anteriormente y añadimos algún objeto como un cubo no veríamos nada ya que además de asociar los objetos debemos actualizar la escena, para ello usaremos la función *requestAnimationFrame* pasando como argumento la función *render*. La función requestAnimationFrame dibujara lo renderizado en la escena 60 veces por segundo (60 fotogramas por segundo), como se actualizará el renderizado también debemos actualizar el valor del objeto *OrbitControls* para lograr el efecto de control de la cámara.

Para ver la cantidad de fotogramas que nuestro navegador trabaja vamos a añadir un objeto *Stats* cuya librería está ubicada en */js/libs/stats.min.js*, este también debe ser actualizado en la función *render*. Crearemos un objeto y lo asociaremos a una etiqueta *div* creada dinámicamente como se hizo anteriormente, para ello usaremos la función *initialStats* que nos creará el bloque en el DOM y nos devolverá el objeto con las estadísticas.



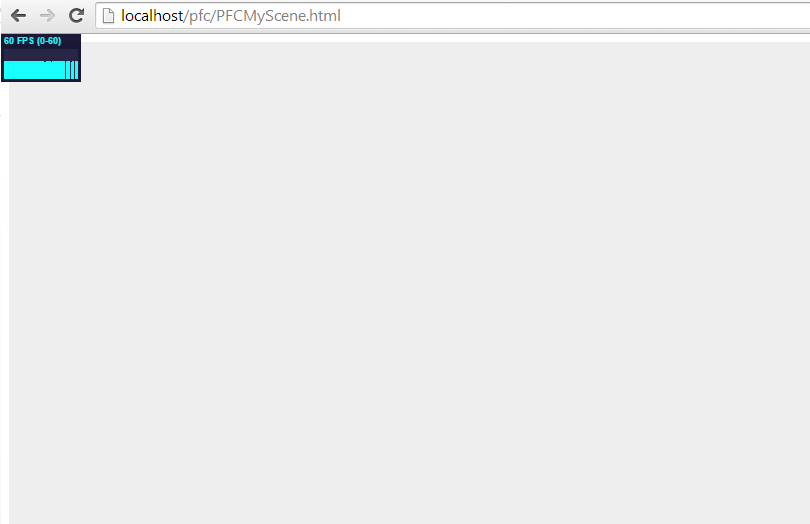


Ya creados los objetos básicos para la escena creamos la función *render*:



Cabe mencionar que todas las variables descritas son variables globales de javasScript, gracias a ello podemos modificar sus valores fácilmente en las funciones.

Con todo lo descrito anteriormente deberíamos crear una escena sin ningún elemento teniendo un aspecto que se puede apreciar en la siguiente imagen.



Crear unas escena: <http://threejs.org/docs/index.html#Manual/Introduction/Creating_a_scene>

WebBL: <http://es.wikipedia.org/wiki/WebGL>

Cdn: <http://es.wikipedia.org/wiki/Red_de_entrega_de_contenidos>

Página web: <http://threejs.org/>

<http://es.wikipedia.org/wiki/Proyecci%C3%B3n_ortogr%C3%A1fica>

<http://es.wikipedia.org/wiki/Proyecci%C3%B3n_ortogonal>

<http://sabia.tic.udc.es/gc/teoria/Proyecciones/Punto5.htm>

Ejemplos

<http://www.smartjava.org/content/all-109-examples-my-book-threejs-threejs-version-r63>

<http://stemkoski.github.io/Three.js/>

<http://learningthreejs.com/blog/2011/08/14/dat-gui-simple-ui-for-demos/>

<http://mrdoob.github.io/three.js/examples/canvas_interactive_cubes.html>

<http://srchea.com/apps/terrain-generation-diamond-square-threejs-webgl/>

Idea principal

<http://www.instructables.com/id/3D-Print-Your-Trek-in-color/?ALLSTEPS>

# Mapbox

Es una librería javaScript que nos permite crear mapas personalizados, sitios web como Foursquare, Pinteres, Evernote usan está librería. La información que nos ofrece Mapbox proviene de fuentes de información de libre acceso (open data) tales como OpenStreetMap y Nasa. Su tecnología está basada en Node.js, CouchDB, Mapnik, Gdal y Leafletjs.

Podemos usar algunos mapas que nos ofrece Mapbox por defecto sin la necesidad de crear una cuenta, para crear mapas y cambiar su diseño necesitamos crear una cuenta en la página oficial (<https://www.mapbox.com/>).

## Edición de mapas

Crearemos una cuenta gratuita y editaremos algunos mapas, para ello debemos rellenar el típico formulario y tras la confirmación de la cuenta vía email tendremos acceso a nuestra cuenta Mapbox.

## Crear proyecto

Nuestros diseños serán proyectos, para crear un proyecto debemos ir a nuestra cuenta y seleccionar la opción de *projects*.



Figura x.x – Mapbox *Projects*.

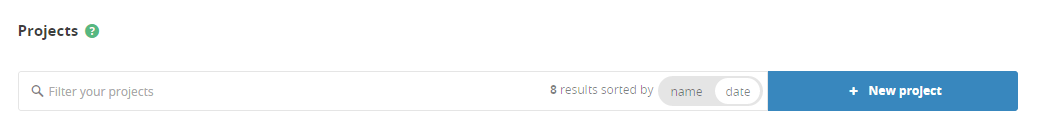
Ya en nuestra sección de *Projects* para crear un proyecto nuevo seleccionamos la opción *+ New Project*.

Figura x.x – Mapbox *New project*.

Al seleccionar la opción *New project* nos aparecerá un mapamundi junto con los diseños que nos ofrece Mapbox. Por defecto esta seleccionada la opción *Street* la cual utiliza colores clásicos que hacen referencia a los elementos del mapa, por ejemplo los ríos están con un color azul claro.

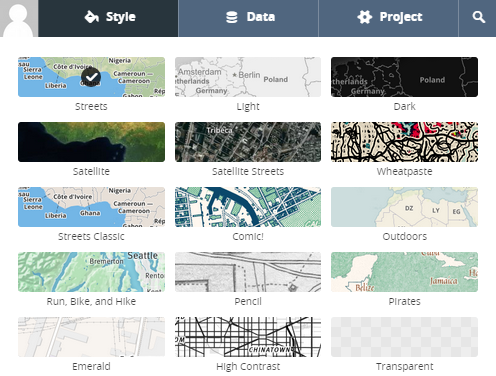


Figura x.x – Mapbox Estilo mapa.

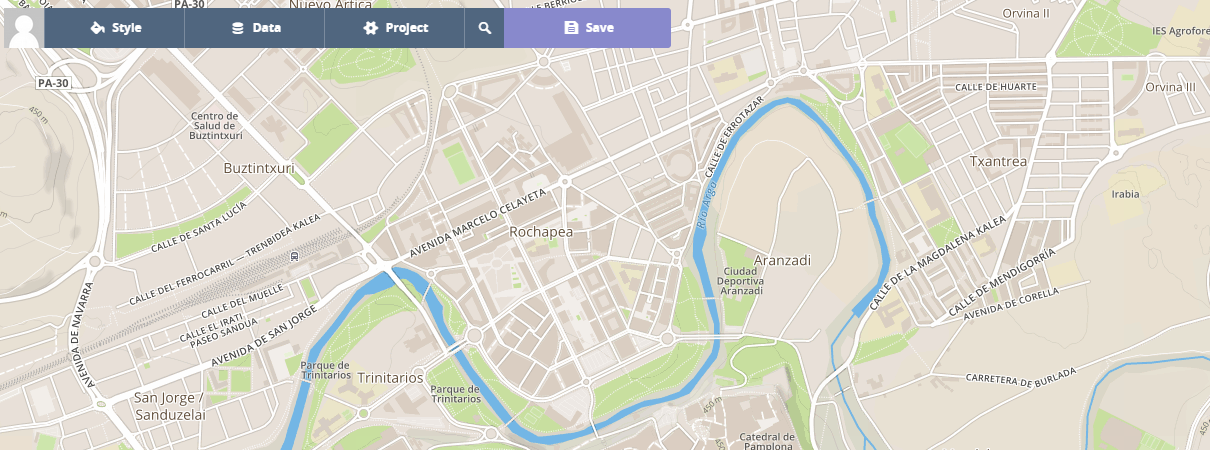


Figura x.x – Mapbox mapa ejemplo.

Ahora solos nos queda guardar nuestro proyecto hacienda clic en la opción *Save.*





Figura x.x – Mapbox guardar mapa*.*

Ya tenemos nuestro diseño de mapa creado, en la sección de proyectos nos aparecerá y si nos fijamos bien como título tiene *Untitled Project* ya que no le hemos puesto ningún nombre, en este caso no es relevante ya que para referirnos a ese mapa usaremos su identificador que aparece debajo de cada proyecto.

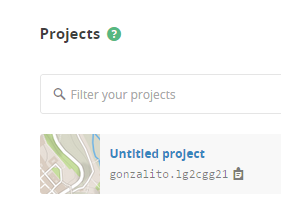


Figura x.x – Mapbox *identificador* mapa*.*

Si hemos seguido los pasos anteriormente descritos ya tenemos nuestro primer diseño de ejemplo, en el siguiente apartado se explica el uso de Mapbox en el navegador con el cual visualizaremos el mapa creado.

## Uso de Mapbox

Al ser un framework ligero podemos disfrutar de todas sus opciones indicando su cdn sin la necesidad de descargar ningún fichero, cargaremos el cdn de la página de estilos de Mapbox y haremos referencia a su librería javaScript.



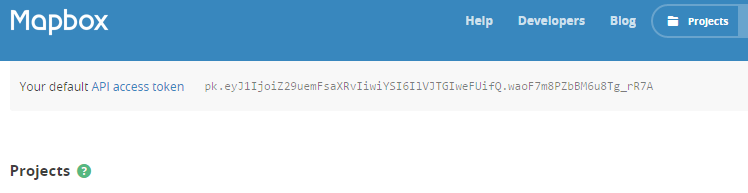
 Con nuestra cuenta creada anteriormente tan solo nos basta indicar nuestra *clave de acceso* y el *identificador* del mapa para poder visualizar nuestro mapa. Para saber nuestro *identificador* de usuario debemos ir a la sección de *Projects* y fijarnos en la parte de arriba donde aparecerá nuestro *clave de acceso* (API Access token).

Figura x.x – Mapbox *clave acceso.*

Para ver el mapa debemos crear una simple página web con un elemento donde será cargado el mapa, en este caso usaremos un elemento del tipo *div* como bloque contenedor y cuando carguemos el mapa indicaré que sus coordenadas sean el Campus de Arrosadía.

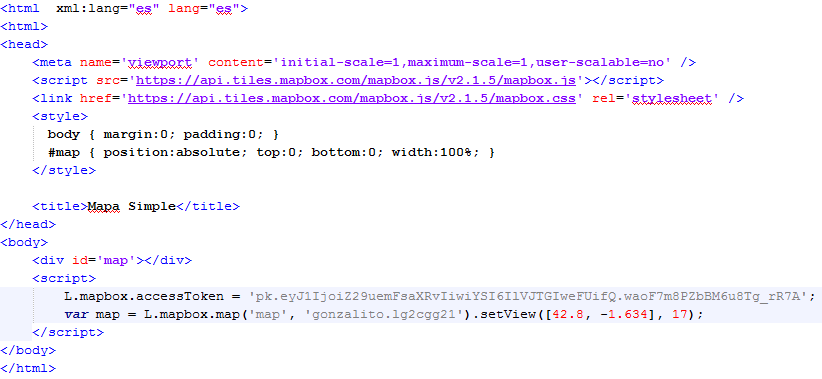


Figura x.x – Mapbox PFCMapbox.html código*.*

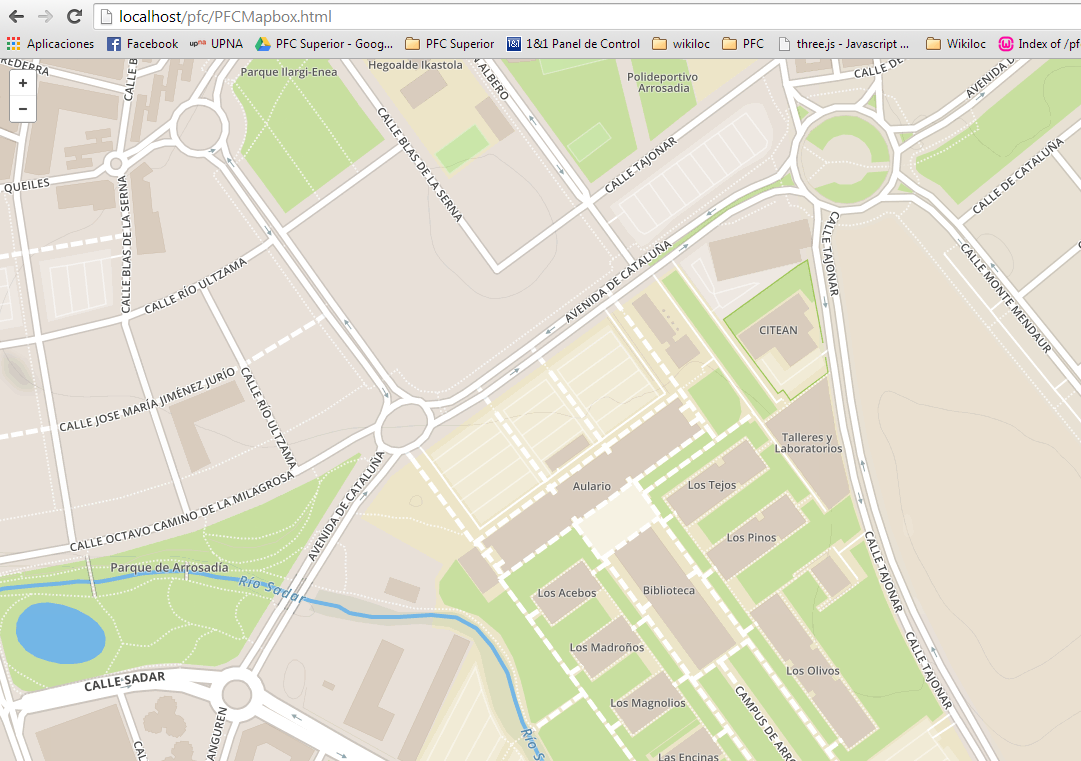


Figura x.x – Mapbox PFCMapbox.html navagador.

<http://en.wikipedia.org/wiki/Mapbox>

<http://wiki.openstreetmap.org/wiki/Mapbox>

<https://www.mapbox.com/mapbox.js/api/v2.1.5/>

# Cesium

Librería basada en javaScript para la creación de globos terrestres en 3D y mapas en 2D en el navegador sin la necesidad de algún plugin, es de libre acceso (open source) bajo licencia Apache 2.0.

A diferencia de los otros frameworks basados en javaScript este no posee *cdn*, por lo tanto, para su uso debemos descargarnos el framework desde la página oficial ( <http://cesiumjs.org/downloads.html> ).

Esencialmente utilizaremos la librería para acceder a los datos de las elevaciones del terreno. Resumiendo el uso de cesium:

* Con las coordenadas de la ruta accedemos a los tiles que nos ofrece cesium. Los tiles son rectángulos que guardan información acerca de los datos del terreno.
* La información obtenida en lo tiles la graficaremos usando *threeJS*.

## Uso de Cesium

En este ejemplo de Cesium usaremos las coordenadas de una zona de Pamplona, en específico latitud *42.8* y longitud *-1.634*.

Para obtener los datos de cesium debemos seguir los siguientes pasos:

* Declarar un proveedor de datos de Cesium, haremos un bucle hasta que el proveedor esté listo.
* Con las coordenadas obtenemos la posición del tile y de este obtenemos la información. Para obtener la información del tile usaremos una llamada asíncrona que nos proporciona Cesium, para mayor comodidad almacenaré cada petición en un arreglo y usare una función que ofrece Cesium que se ejecutará cuando las peticiones hayan acabado.

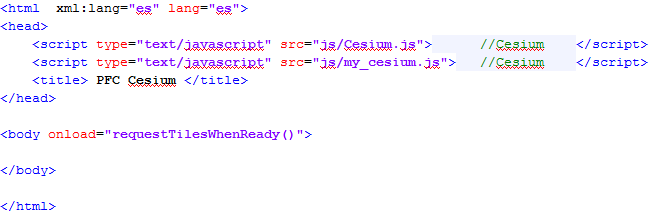


Figura x.x – PFCCesium.html

Se llama a la función *requestTileWhenReady* que se encargará de establecer la conexión con Cesium usando un objeto *CesiumTerrainProvider*. Se creará una función recursiva que llamará tantas veces sea necesario a la función hasta obtener la conexión, todas las funciones y código javaScript estará en el fichero */js/my\_cesium.js*

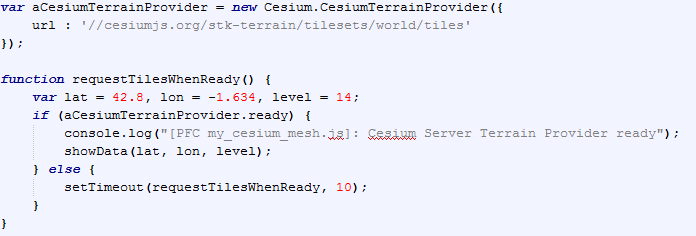


Figura x.x – Cesium función recursiva.

Cuando se haya establecida la conexión llamaremos a la función *showData* pasando como argumentos la latitud, longitud y el nivel. El nivel es la aproximación real de los datos, es decir, mientras mayor sea el nivel más específico son los datos y dependiendo de la ubicación geográfica hay más o menos niveles.

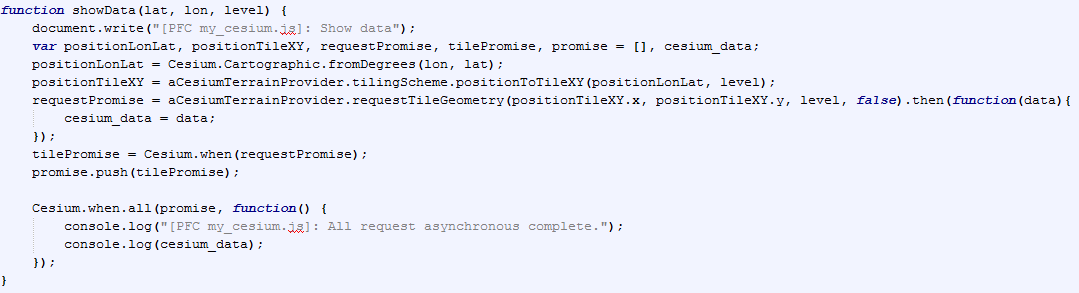


Figura x.x – Cesium función obtención de datos.

En la función *showData* realizamos lo siguiente:

* *Cesium.Cartographic.fromDegrees(longitude, latitude, height optional, reult optional)*: Creamos una instancia del tipo *Cartographic* (objeto Cesium) a partir de una longitud y latitud en grados, el objeto resultante está en radianes.
* *Cesium.CesiumTerrainProvider.tilingScheme.positionToTileXY(position, level, result optional)*: Calculamos las coordenadas x e y del tile según la posición de las coordenadas.
* *Cesium.CesiumTerrainProvider.requestTileGeometry(x, y, leve, throttleRequests optional)*: Obtenemos la geometría a partir del tile indicado.
* Almacenaremos la información en la variable *cesium\_data*.
* Crearemos por cada petición una variable usando la función *Cesium.when(petición)* y la almacenaremos en el arreglo *promise*. Implementamos el método *Cesium.when.all()* que nos indicará que las peticiones se han realizado y mostraremos la información por la consola del navegador.



Figura x.x – Datos Cesium.

De los datos ofrecidos por cesium usaremos los siguientes:

* *maximumHeight* y *minimumHeight* como los valores mínimos y máximos de alturas del terreno.
* *uValues*, *vValues* y *heightValues* almacenados en el objeto *quantizedVertices* que contiene tripletes donde guardamos la longitud, latitud y altura.
* Con estos datos crearemos las geometrías de los terrenos, esto se explicará más adelante en el desarrollo del proyecto.

# Shapeways

# Bibliografía

<http://threejs.org/>