



**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, XX de Abril de 2015

# Índice

[Índice 2](#_Toc415578679)

[Introducción 2](#_Toc415578680)

[JavaScript 2](#_Toc415578681)

[Three.js 2](#_Toc415578682)

[Creando la escena 2](#_Toc415578683)

[Escena 2](#_Toc415578684)

[Cámara 2](#_Toc415578685)

[Renderizado 2](#_Toc415578686)

[Función *render* 2](#_Toc415578687)

[Mapbox 2](#_Toc415578688)

[Edición de mapas 2](#_Toc415578689)

[Crear proyecto 2](#_Toc415578690)

[Uso de Mapbox 2](#_Toc415578691)

[Cesium 2](#_Toc415578692)

[Uso de Cesium 2](#_Toc415578693)

[HGT y algoritmo *square-diamond* 2](#_Toc415578694)

[Generador de terreno 2](#_Toc415578695)

[La idea 2](#_Toc415578696)

[Desarrollo 2](#_Toc415578697)

[Otras funciones 2](#_Toc415578698)

[Problemas y soluciones 2](#_Toc415578699)

[Resultado 2](#_Toc415578700)

[Mejoras 2](#_Toc415578701)

[Shapeways 2](#_Toc415578702)

[Conclusiones 2](#_Toc415578703)

[Líneas Futuras 2](#_Toc415578704)

[Bibliografía 2](#_Toc415578705)

# Introducción

Actualmente uno de los temas más recurrentes en las nuevas tecnologías es todo lo relacionado con el diseño en tres dimensiones más conocido como 3D, desde la creación de modelos en el ordenador hasta la creación de impresoras que nos permitan crear esos modelos y que sean reales. La librería *three.js* nos permite crear modelos 3D en el navegador, podemos crear objetos básicos como cubos, planos, textos, cargar objetos prediseñados, crear nuestras propias geometrías y un sinfín de objetos en 3D.

El objeto del proyecto es la creación de una aplicación web para la edición de objetos 3D que vayan a ser impresos en una impresora 3D usando tecnología WebGL con la librería three.js desarrollada en JavaScript.

# JavaScript

JavaScript es un lenguaje de programación interpretado[[1]](#footnote-1) al igual que otros vistos en la diplomatura como php, Lisp , programación en R y Prolog.

Su sintaxis es similar a C, aunque adopta nombres y convenciones del lenguaje de programación Java aunque JavaScript y Java no están relacionados y tienen semánticas y propósitos diferentes.

JavaScript se utiliza principalmente en el lado del cliente añadiendo una mejora al interfaz para el usuario y dando dinamismo a las páginas interactuando con el DOM[[2]](#footnote-2).Con la llegada de Ajax[[3]](#footnote-3) proporcionando llamadas asíncronas al servidor JavaScript se ha convertido en uno de los lenguajes más populares en internet.

Algunos ejemplos del uso de JavaScript son:

* Cargar nuevo contenido para la página o enviar datos al servidor a través de Ajax sin necesidad de recargar la página (por ejemplo, una red social puede permitir al usuario enviar actualizaciones de estado sin salir de la página).
* Animación de los elementos de página, hacerlos desaparecer, cambiar su tamaño, moverlos, etc.
* Contenido interactivo.
* Validación de los valores de entrada de un formulario web para asegurarse de que son aceptables antes de ser enviado al servidor.

La ventaja de usar JavaScript es que al ser ejecutado en el lado del cliente la respuesta es más rápida. Otra ventaja es que JavaScript permite detectar acciones del usuario como pulsaciones de teclas, hacer clic en un botón, pasar por encima de un elemento de la página.

Finalmente otra de las razones por la cual JavaScript es un lenguaje de programación popular es la compatibilidad con la mayoría de los navegadores debido a la estandarización del W3C.[[4]](#footnote-4)

# 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.

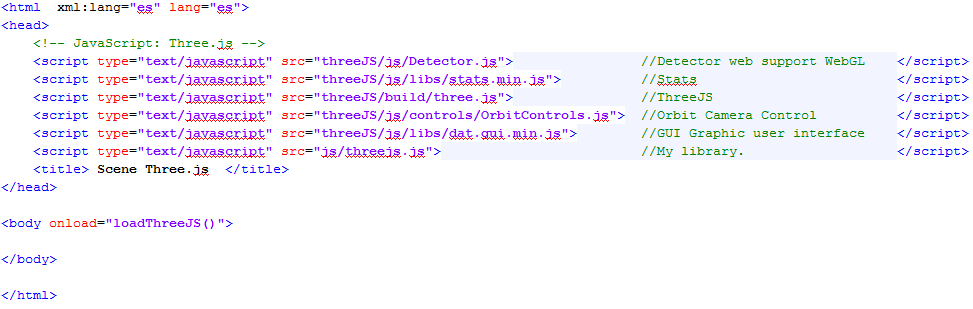
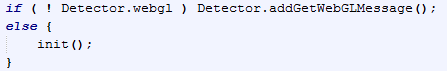


Figura x.x - PFCMyScene.html

Como se puede apreciar la página inicial carece de elementos ya que todos los elementos serán creados mediante JavaScript modificando el DOM, el fichero */js/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.



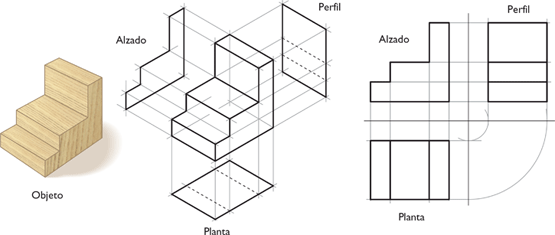
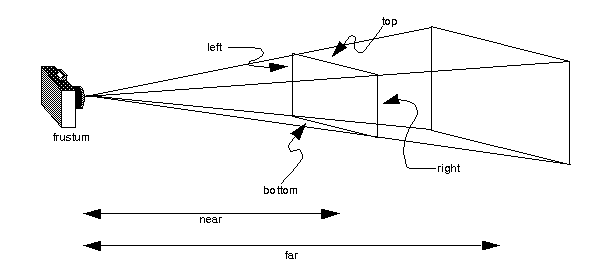
### Escena

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.

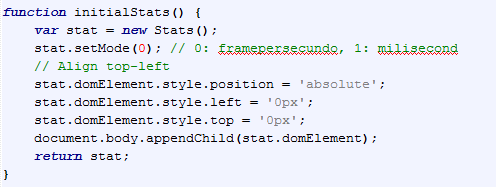


### Función *render*

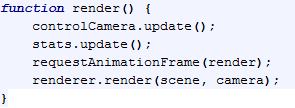
Como objetivo principal esta función asocia al objeto de renderizado la escena y cámara. Si asociamos los elementos descritos anteriormente y añadimos algún objeto como un cubo no veríamos nada ya que además de asociar los objetos debemos actualizar la escena, para ello usaremos la función *requestAnimationFrame* 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.



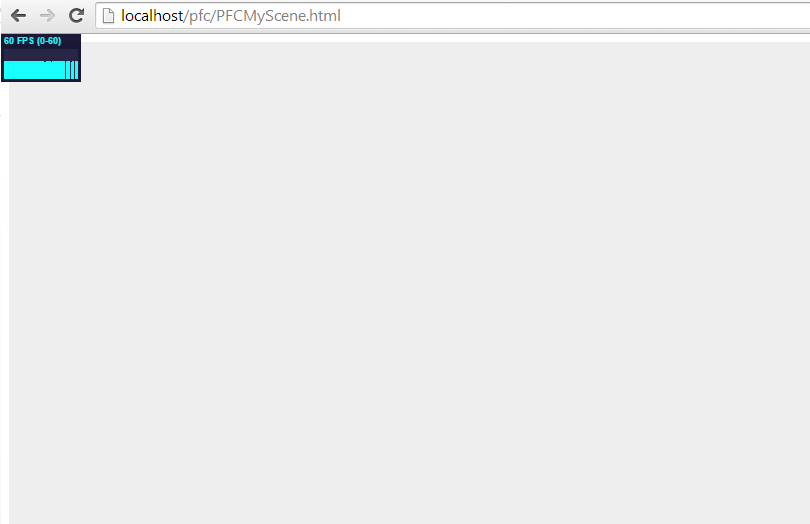


Ya creados los objetos básicos para la escena creamos la función *render*:



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*.

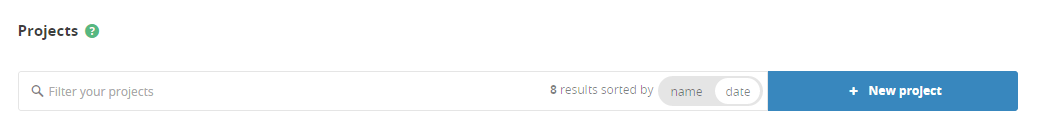
Ya en nuestra sección de *Projects* para crear un proyecto nuevo seleccionamos la opción *+ New Project*.

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.

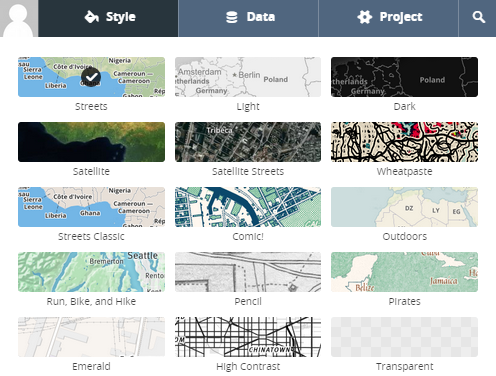


Figura x.x – Mapbox Estilo mapa.

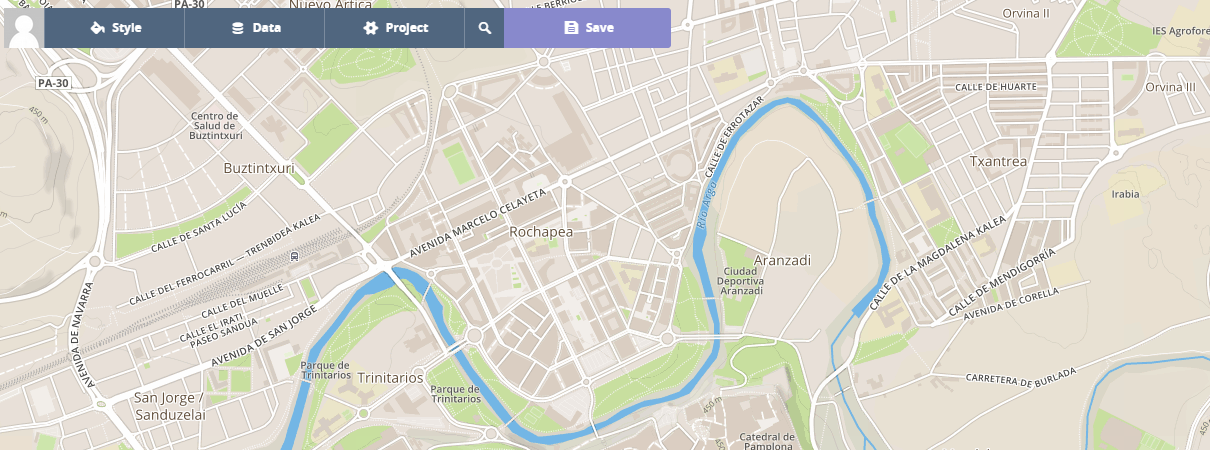


Figura x.x – Mapbox mapa ejemplo.

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.

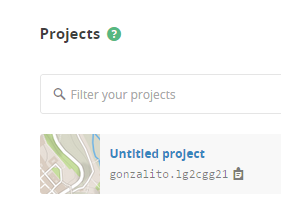


Figura x.x – Mapbox *identificador* mapa*.*

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.



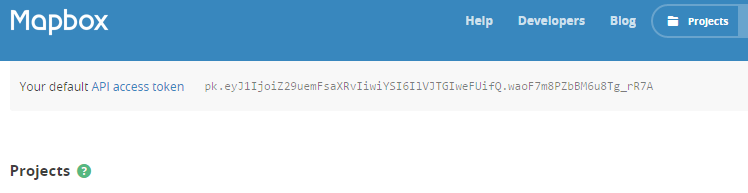
 Con nuestra cuenta creada anteriormente tan solo nos basta indicar nuestra *clave de acceso* y el *identificador* del mapa para poder visualizar nuestro mapa. Para saber nuestro *identificador* de usuario debemos ir a la sección de *Projects* y fijarnos en la parte de arriba donde aparecerá nuestro *clave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

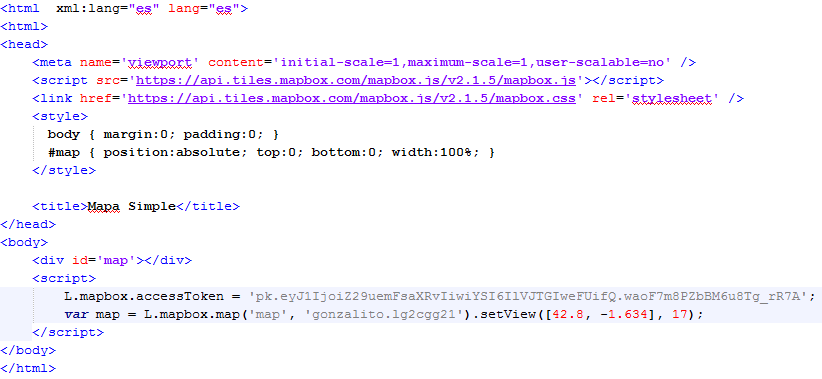


Figura x.x – Mapbox PFCMapbox.html código*.*

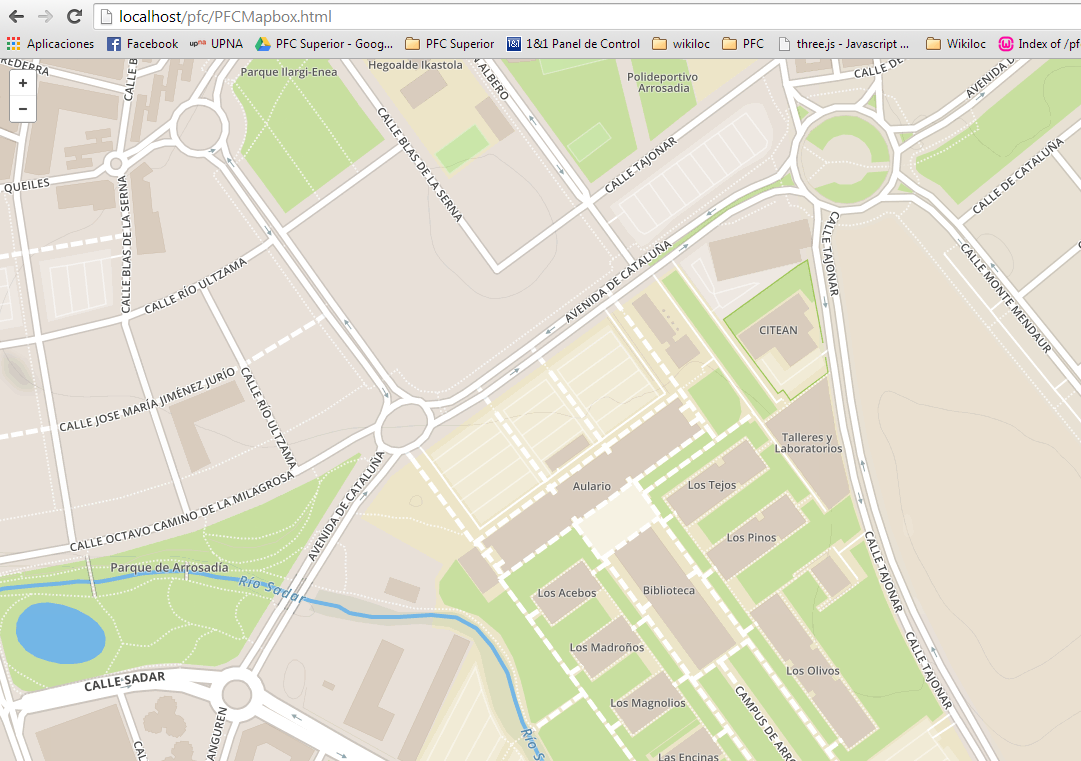


Figura x.x – Mapbox PFCMapbox.html navagador.

<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.

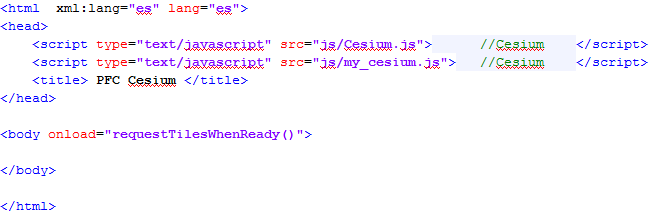


Figura x.x – PFCCesium.html

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*

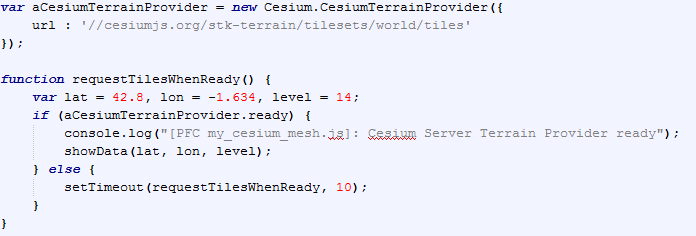


Figura x.x – Cesium función recursiva.

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.

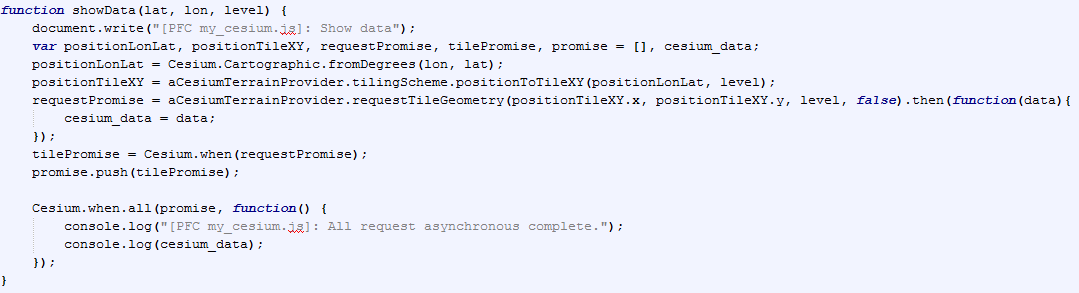


Figura x.x – Cesium función obtención de datos.

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.

# HGT y algoritmo *square-diamond*

# Generador de terreno

## La idea

A partir de una ruta *gpx* modelar el recorrido en 3D con las elevaciones del terreno. Se podrá elegir uno de los mapas creados en Mapbox que será la textura de nuestro modelado.

## Desarrollo

#### PFCIndex.html

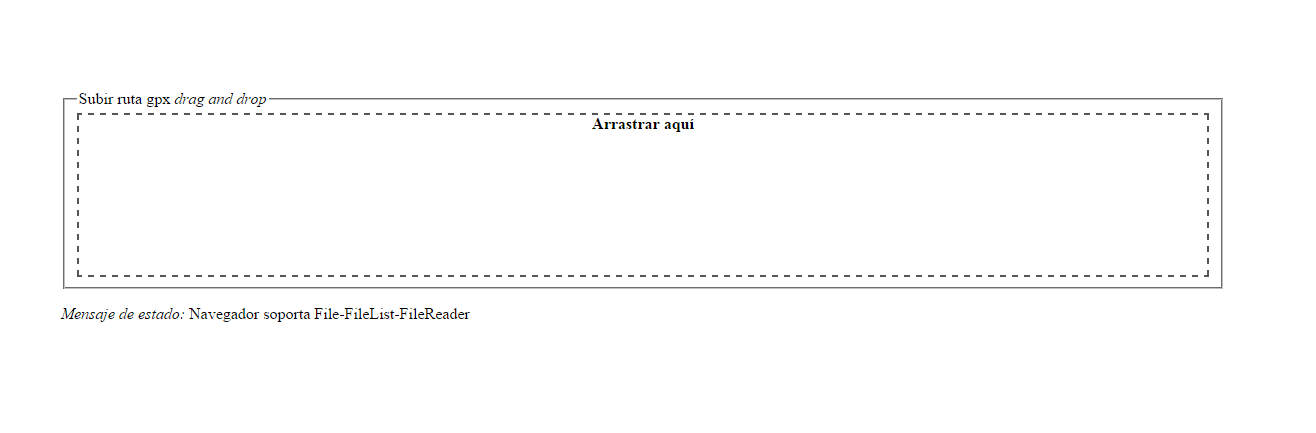
 Se carga la página *PFCIndex.html* donde se subirá el fichero *gpx* al servidor mediante ajax y php, se ha controlado que solo se puedan subir ficheros *gpx* y cada vez que se cargue la página inicial se borraran todos los datos en el servidor.

Figura x.x – Drag and drop *gps*.

#### PFCMyRute.html

Cuando el fichero es cargado exitosamente en el servidor cargaremos la página *PFCMyRute.html* que nos mostrará la ruta *gpx* cargada en el mapa de Mapbox.



Figura x.x – Mapbox ruta gps.

El nombre del fichero subido al servidor quedará almacenado en una variable de sesión de JavaScript.

Cesium nos ofrece los datos mediante tiles, estos contiene una gran cantidad de información, entre esos datos tenemos las elevaciones del terreno. Un tile corresponde a una área rectangular del mapa, por cada coordenada verifico al tile que pertenece obteniendo de esta manera todos los tiles correspondientes a la ruta.

Por cada tile almaceno sus límites Sur-Oeste, Sur-Este, Nor-Oeste, Nor-Este. Con estos datos en Mapbox puedo dibujar un rectángulo para identificar cada tile de la ruta.



Figura x.x – Mapbox tile ruta.

La idea es obtener un modelado rectangular del recorrido, por lo tanto, a partir de la ruta he añadido más tiles identificando los máximos y mínimos laterales de la ruta.

Para obtener la información de cada tile debemos indicar sus coordenadas y nivel. En este proyecto se usó el nivel 14 ya que es el mayor nivel soportado para la zona Europea, en zonas de Estados Unidos soporta un nivel de 15, mientras mayor es el nivel más detallado son los datos obtenidos.

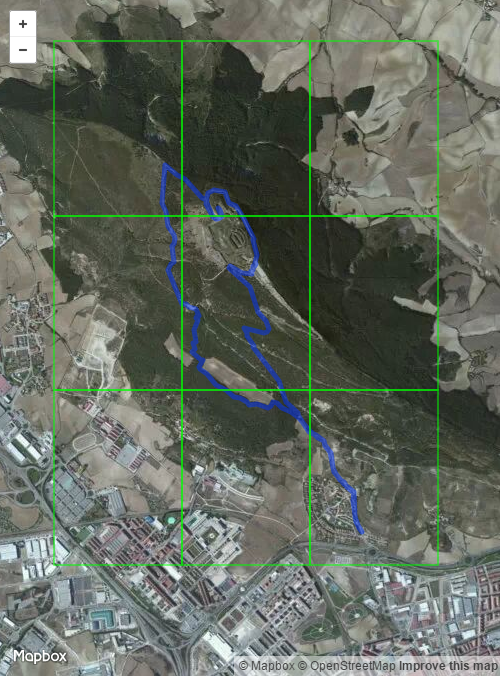


Figura x.x – Mapbox nivel 14. Figura x.x – Mapbox nivel 13.

La página *PFCMyRute.html* a su vez trabaja con varios ficheros externos en JavaScript ubicados en la carpeta *js*. El fichero */js/my\_cesium\_rute.js* junto a los ficheros */js/clases.js, /js/functions.js* y */js/Cesium.js* este último es el fichero ofrecido por Cesium.

#### my\_cesium\_rute.js

#### Mapa y coordenadas

Si el fichero subido corresponde al tipo *gpx* se cargarán las coordenadas a través de *ajax* en el arreglo *coordinates* y se llamará a la función *createMap*() que creará el mapa y asignará los valores de las variables globales que usaremos en la aplicación.

##### Información tiles

Usando la función *checkTile(coordinates, level)* creamos el arreglo *info\_tiles*,la función lleva como parámetros de entrada las coordenadas y el nivel de Mapbox. El arreglo guarda objetos *InfoTile* donde cada uno de ellos contiene la siguiente información:

* *bounds:* Límites del tile.
* *cardinality:* En un principio se usó para la creación del terreno, luego por modificaciones se ha usado para guardar la referencia de donde fue creado la variable, en que función del código.
* *coordinate:* coordenadas iniciales de la ruta.
* *Index:* posición inicial de la ruta correspondiente al arreglo de coordenadas *coordinates.*
* *x:* coordenada x de Cesium para solicitar la información de un tile.
* *y:* coordenada Y de Cesium para solicitar la información de un tile.

En este punto tenemos los tiles que corresponden a la ruta y como se mencionó antes la idea es crear un terreno rectangular, por lo tanto, llamamos a la función *createRectangle(info\_tiles, level)*. Está función nos devuelve un arreglo que contiene objetos *InfoTile* correspondientes a la ruta y añade nuevos valores completando un rectángulo, como parámetros de entrada la función recibe los tiles del recorrido y el nivel de Mapbox. El resultado de la función lo guardaremos en la variable *rectangle\_tiles*.

##### Mapa y recorrido

Con el mapa ya creado y con toda la información necesaria para la creación de nuestro modelado en la variable *rectangle\_tiles* llamaremos a la función *loadGpx()* que se encargará de centrar el mapa con el primer y último tile del arreglo *rectangle\_tiles*, es posible que no se visualice toda la ruta en el mapa, para ello se puede acercar o alejar la vista del mapa usando el ratón. Además de centrar el mapa la función *loadGPX()* dibujará la ruta sobre el mapa para verificar que los datos cargados sean correctos, en principio se usó el método *omnivore* para cargar rutas *gpx* en los mapas pero en algunos casos no me dibujaba bien la ruta así que usando las coordenadas dibujo una línea que corresponde a la ruta.

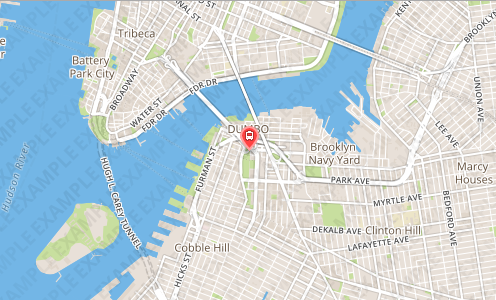
##### Texturas

Para crear la textura final debemos crear la textura por cada tile y luego unir todas las texturas creadas. Para extraer las imágenes usamos el API *Static maps* de Mapbox, nos permite obtener distintos tipos de imágenes:

* Imágenes de mapas simples.



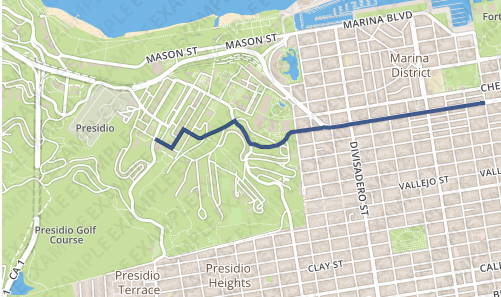
* Imágenes de mapas con marcadores.



* Imágenes de mapas con rutas mediantes GeoJSON.



* Imágenes de mapas con ruta mediante codificación de líneas *Encoded polylines*.



Para obtener la textura de cada tile con su ruta podemos usar uno de los dos últimos métodos. En un principio utilice el método de codificación de líneas pero surgían muchos problemas por este método ya que había que usar una librería de *google* y a veces no se dibujaba la ruta correctamente, al final decidí usar el método pasando la ruta en formato GeoJSON.

Con ambos métodos cargo una nueva página pasando en el *url* toda la información y en ambos si se supera una cantidad aproximada de 120 coordenadas no se devuelve correctamente la imagen del mapa, de este modo cuando un tile tiene un número muy grande de coordenadas aplique una función de reducción que intenta no perder la ruta original.

Para obtener las texturas de cada tile he creado la función *drawRute()*, esta función recorre todos los tiles almacenados en la variable *rectangle\_tiles* y por cada tile se realiza lo siguiente:

* Compruebo que el tile corresponde a la ruta y no a los tiles que cree como relleno para obtener una forma rectangular, por cada uno de ellos:
  + Si no es el primer tile donde se inicia la ruta añado la coordenada anterior para que la línea de la ruta no tenga cortes entre tiles.
  + En *rectangle\_tiles* tengo el índice donde se inicia la ruta en el tile con respecto al arreglo de coordenadas, a partir de esa posición compruebo las coordenadas que pertenecen a ese tile y las almaceno. Las coordenadas las compruebo con los límites de cada tile almacenado también en *rectangle\_tiles*. Durante el desarrollo del proyecto me encontré con los siguientes problemas a la hora de crear las rutas en los tiles:
    - Si la ruta en el tile no tiene cortes el algoritmo inicial funciona sin problemas. Cuando hablo de cortes me refiero a que la ruta sale del tile y vuelve a entrar, para ello establecí una tasa de fallo que sigue almacenando la ruta hasta que lleguemos a un número elevado de fallos. Este paso es necesario porque a la hora de dibujar la ruta en Mapbox no puede tener cortes ya que no dibujaría la ruta tal cual es.
    - Otro problema o limitación es que a la hora de crear texturas con Mapbox por cualquiera de los métodos descritos tenemos un límite de coordenadas así que aplique una función para reducir las mismas sin perder el sentido de la ruta original.
  + Añado al final si no es el último tile la primera coordenada del siguiente tile para que la línea de la ruta no tenga cortes entre tiles.
  + Invierto las coordenadas para que sean reconocidas por Mapbox para crear la ruta.
  + Creo el objeto GeoJson para enviar por *url* para la creación de la textura del mapa con la ruta. Codifico el objeto como JSON con la función *JSON.stringify()* y codifico nuevamente el objeto para poder ser pasado por *url* con la función *encodeURIComponent*.
  + Creo el enlace con el cual luego descargaremos la imagen correspondiente para cada tile.
  + Para lograr un mejor orden en el código almaceno el enlace y su índice en un arreglo para luego crear las texturas con una sola llamada *ajax*
* Compruebo que el tile que no corresponde a la ruta tenga o no coordenadas en él ya que en el desarrollo del proyecto hay tiles de relleno que contienen coordenadas de la ruta.
  + Realizo los mismos pasos mencionados anteriormente.
* Creo la imagen del tile sin coordenadas y la almaceno.

Con el arreglo envío todas las texturas mediante *ajax* y se crean las texturas en el servidor mediante *php*~~. Al principio las imágenes las cree con un tamaño de 509 de ancho por 702 de alto pero en los casos de rutas muy largas el servidor se queda sin memoria, por la tanto, en el mismo servidor modifico el tamaño de cada imagen sin que afecte en la visualización posteriormente.~~ Cuando el servidor termine de crear las texturas se cargará la página *PFCMyMesh.html* donde aparecerá el modelado en 3D con *threejs*.

#### PFCMyMesh.html

Con las texturas en el servidor nos queda crear la geometría y añadir la ruta recorrida. En esta etapa del proyecto es donde mezclaremos el uso de la librería *Cesium* con los datos de las elevaciones y *threejs* para crear el terreno de la ruta recorrida.

La página *PFCMyMesh.html* a su vez trabaja con varios ficheros externos en JavaScript ubicados en la carpeta *js*. El fichero */js/my\_cesium\_mesh.js* junto a los ficheros */js/clases.js, /js/functions.js* y */js/Cesium.js* este último es el fichero ofrecido por Cesium.

#### my\_cesium\_mesh.js

Creamos las variables locales que usaremos y esperamos hasta lograr la conexión a *Cesium* (milésimas de segundos).

##### Obtención de ruta

Cuando la conexión a Cesium esté lista llamamos a la función *getRute()* con la cual obtendremos las coordenadas de la ruta y llamaremos a la función *load()* que se encargará de crear la escena en 3D, crear algunos controles gráficos y crear el terreno de la ruta.

##### Creación de escena y controles gráficos

Llamaremos a la función *loadThreeJS()* y a la función *createGUI()*. La función loadThreeJS creará la escena, posicionará la cámara y aplicará el renderizado al añadir el terreno. La función *createGUI()* se encargará de añadir los botones para volver al inicio, visualizar el mapa y exportar el modelo para ser impreso en *Shapeways* (imprimir objetos en 3D).

##### Información tiles

Realizaremos los mismos pasos que en la página anterior para obtener los datos, usando la función *checkTile(coordinates, level)* creamos el arreglo *info\_tiles*,la función lleva como parámetros de entrada las coordenadas y el nivel de Mapbox. El arreglo guarda objetos *InfoTile* explicados anteriormente.

En este punto tenemos los tiles que corresponden a la ruta y como se mencionó antes la idea es crear un terreno rectangular, por lo tanto, llamamos a la función *createRectangle(info\_tiles, level)*. Está función nos devuelve un arreglo que contiene objetos *InfoTile* correspondientes a la ruta y añade nuevos valores completando un rectángulo, como parámetros de entrada la función recibe los tiles del recorrido y el nivel de Mapbox. El resultado de la función lo guardaremos en la variable *rectangle\_tiles*.

##### Información del terreno

En el arreglo *rectangle\_tiles* tenemos toda la información necesaria para crear el terreno, con un simple bucle solicitaremos a *Cesium* los datos de cada tile con la función *requestTileGeometry()* mencionada anteriormente, en este punto explicaremos los datos que hemos utilizado de *Cesium* cuando realizamos la petición por cada tile. Además iremos almacenando cada petición en un arreglo para controlar el programa debido a que *requestTileGeometry()* es una función asíncrona, por lo tanto, solo se creará el terreno cuando hayamos procesado todos los tiles.

Por cada llamada a la función *requestTileGeometry() Cesium* devuelve el siguiente objeto:



Figura x.x – Petición *Ceisum* .

Para la creación del terreno usaremos los arreglos:

* *\_uValues*: contiene el valor de las coordenadas horizontales en el tile, si es 0 indica que es el borde Oeste del tile. Cuando el valor es 32767 nos indica el borde Este del tile. Para el resto de los valores este vector es una interpolación lineal entre las longitudes de los bordes del este y oeste del tile.
* *\_vValues*: contiene el valor de las coordenadas verticales en el tile, si es 0 indica que es el borde Sur del tile. Cuando el valor es 32767 nos indica el borde Norte del tile. Para el resto de los valores este vector es una interpolación lineal entre las longitudes de los bordes del este y oeste del tile.
* *\_heightValues*: contiene las alturas del tile, si es 0 indica que es la menor altura del tile. Cuando el valor es 32767 nos indica la mayor altura del tile. Para el resto de los valores este vector es una interpolación lineal entre la mínima y máxima altura. Para obtener las alturas debemos tener en cuenta los valores *\_minimumHeight* y *\_maximumHeight* y establecer una regla de tres para obtener las alturas reales.
* *\_indices:* específica como los vértices van unidos entre sí en triángulos, cada tres índices especifican un triángulo, este valor es el más importante a la hora de crear los terrenos.

Con los datos mencionados anteriormente ya tenemos todo lo que necesitamos para crear cada tile que formará el terreno. La creación de cada tile es de la siguiente manera:

* Creamos una geometría con *threejs*.
* Añadimos los vértices a la geometría. Cada vértice será el valor de la coordenada horizontal *\_uValues*, coordenada vertical *\_vValues* y altura *\_heightValues* escalada usando los valores de altura máxima y mínima de cada tile.
* Creamos las caras de la geometría usando la variable *\_indices* obteniendo el terreno.
* Almacenaremos la geometría creada en el atributo *geometry* del arreglo *rectangle\_tiles*.

Al terminar el bucle en el arreglo *rectangle\_tiles* tendremos por cada tile su geometría, el bucle terminará cuando todas las peticiones hayan devuelto la información solicitada ya que un principio no todas las peticiones se completaban y algunas geometrías no se creaban.

##### Creación del terreno

Ahora debemos unir todas las geometrías para obtener una única geometría que será nuestra ruta recorrida. ~~Para ello usaremos la función~~ *~~merge()~~* ~~de~~ *~~treejs~~* ~~que une mallas, una malla es la unión de una geometría con una textura~~. La función que se encargará de crear el terreno es *createTerrain()*.

*createTerrain()* creará el terreno realizando las siguientes acciones:

* Obtener el máximo y menor tile con la función *maxMinTileXY()* para poder unir cada geometría de un tile con su vecina.
* Con los valores obtenidos de la función *maxMinTileXY()* crearemos un bucle que recorra todos los tiles que pertenecen a la ruta.
* Utilizare una serie de variables para la localización de cada geometría en la escena, para obtener la geometría usaremos la función *getGeometry()* que nos devuelve la geometría del tile que indiquemos. Usaremos la función *addGeometryScene()* para crear una geometría que ira uniendo cada geometría de cada tile hasta obtener una geometría final que corresponderá a la ruta.
* Actualizaremos los valores de las variables por cada iteración del bucle.
* Finalmente llamaremos a la función *showCombinedGeometry()* la cual creará la textura para la geometría y la malla que será visualizada en la escena.

A continuación explicaré la función *showCombinedGeometry()* :

* Usamos la función maxMinTileXY() anteriormente mencionada, al tener los máximos y mínimos de la ruta recorremos todos los tiles para obtener cuantas filas y columnas tendrá nuestra ruta completa. Se hace este proceso para unir todas las imágenes de los tiles creando una sola imagen que usaremos como textura a nuestra geometría final.
* Realizamos una petición asíncrona al servidor enviando como datos la cantidad de columnas, cantidad de filas, nombre del fichero en uso y la medida de alto que debería tener cada imagen (algoritmo latitud).
* Cuando la imagen de toda la ruta ha sido creada el servidor nos devuelve su ubicación en el servidor.
* Creamos la textura para nuestra geometría final.
* Como es una geometría creada desde cero debemos indicar como la textura debe implantarse en cada cara, para ello usamos la función *addFaceVertexUvs()*.
* Para crear el volumen del terreno debemos añadir una base, para ello usamos la función *addBase().*
* Creamos el material indicando la textura que usará y que la textura aparezca en ambos lados de la geometría.
* Creamos la malla que es la unión de la geometría y el material.
* Especificamos su ubicación en la escena. Esta ubicación debe ser dinámica siempre para que el usuario pueda interaccionar correctamente con el terreno.
* Rotamos la malla, ya que su creación ha sido modificando todos los vértices.
* Añadimos la malla a la escena.

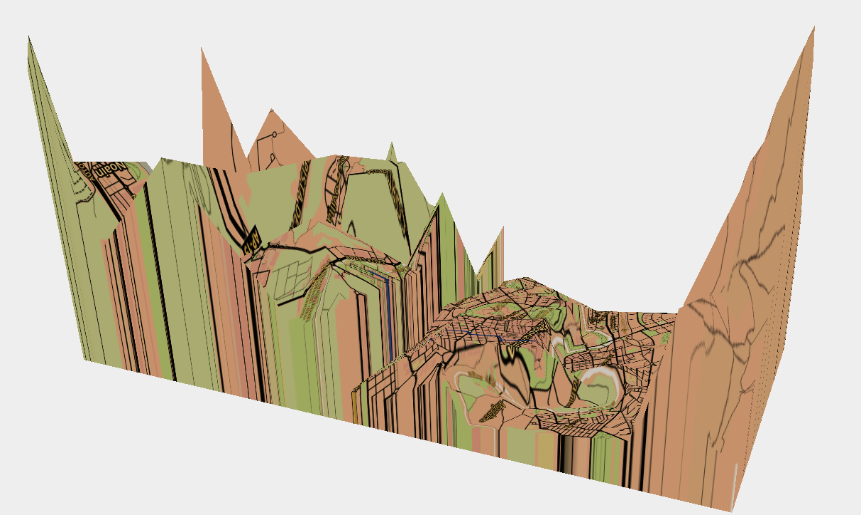
## Otras funciones

En este proyecto también se han usado los siguientes ficheros JavaScript y php:

* */js/clases.js*
* */js/dnd.js*
* */js/gui.js*
* */js/my\_jquery*
* */js/x3dExporter.js*
* *createImage.php*
* *createX3D.php*
* *deleteFiles.php*
* *downloadShapeways.php*
* *getRute.php*
* *getTexture.php*
* *saveFile.php*
* *upload.php*

## Problemas y soluciones

En este apartado explicaré los problemas y soluciones que aparecieron durante el proyecto.

* *three*: Las imágenes que serán usadas como texturas en materiales por obligación deben estar alojadas en el servidor.
* *three*: Cuando se crea una geometría se deben indicar los vértices y a partir de ellos como están conectados a través de caras.
* *three*: Cuando se utiliza la función *merge()* de una geometría como parámetros debemos usar una malla pero antes debemos actualizar la matriz de esa malla con la función *updateMatrix().*
* *three*: Como se mencionó antes cuando creamos una geometría desde cero debemos indicar como se implantará la textura en nuestro objeto, al principio apareció este error:  
    
  Revisando la geometría encontré que el atributo *faceVertexUvs* debía tener la misma cantidad de elementos que el atributo *faces*, para solucionar esto se aplicó sobre la geometría la función *addFaceVertexUvs()* encontrada en internet y que soluciona el problema. Esta función fue extraída de la librería *threejs* por un usuario. 
* *mapbox*: para identificar si estaba eligiendo bien los tiles de los cuales obtengo la información use la clase *Rectangle* de la librería y dibujaba los rectángulos correspondientes por cada tile.
* *mapbox:* Inclusión obligatoria del mapa en el navegador para cargar la ruta e imágenes. Además se especifica el tamaño del mapa para poder obtener las coordenadas centrales para obtener las imágenes por tiles, este tamaño no puede ser cambiado.
* *cesium:* Escalar los datos de las alturas ya que son valores muy altos y ajustar esos valores con los máximos y mínimos valores de las alturas de cada tile obtenido de esta librería.   
    
  
* *cesium*: Las llamadas a funciones son asíncronas así que se deben manejar cuidadosamente los datos para que no haya errores a la hora de crear las geometrías. Se implementó una función que espera que todas las peticiones se hayan completado.
* *three*: Se añade a cada geometría una base lateral para obtener un efecto de fondo y luego se cierra la geometría para que pueda ser reconocida como una geometría completa sin huecos.
* *cesium*: Se definió el nivel 14 para la obtención de datos por ser el más cercano a lo real, se intentó usar el nivel 15 pero no está implementado en todo el mundo y en lugares de Estados Unidos no produce error pero en otros lugares como San Sebastián produce error por no tener datos.
* *three*: Para aplicar completar la geometría final debemos usar la función *computeBoundingBox()* que añade los máximos y mínimos valores de la misma, de este modo podemos crear los laterales y el fondo.
* *mapbox + three:* Problemas a la hora de obtención y unión de imágenes ya que no hay una medida estándar, estás se ven afectadas por la latitud. Para ello se utilizo este algoritmo de *Proyección de Mercator*.

 <http://es.wikipedia.org/wiki/Proyecci%C3%B3n_de_Mercator>

* *mapbox*: Para mostrar las coordenadas en los mapas, se deben invertir el orden e indicar primero la longitud y luego la latitud.
* *mapbox*: Para la creación de imágenes se optó por la creación de imágenes estáticas con GeoJSON ya que por método de polyline estaba más limitado y a veces daba errores. Limitado ya que al codificar las coordenadas se crean muchos caracteres que no son reconocidos por el navegador a la hora de crear la imagen.
* *mapbox:* en algunos tiles la cantidad de coordenadas sobrepasa las 120 permitidas aproximadamente a la hora de crear una imagen estática, se creó una función que reduce esa cantidad de coordenadas sin perder el sentido de la ruta.

**

## Resultado

## Mejoras

# Shapeways

# Conclusiones

# Líneas Futuras

Mejorar algoritmo de ruta para que no de fallos.

# Bibliografía

<http://threejs.org/>

1. Lenguaje interpretado: lenguaje de programación que esta diseñado para ser ejecutado por medio de un intérprete. [↑](#footnote-ref-1)
2. DOM: Document Object Model. [↑](#footnote-ref-2)
3. Ajax : Asynchronous JavaScript and Xml [↑](#footnote-ref-3)
4. WWWC World Wide Web Consortium, consorcio internacional que realiza recomendaciones a la red informática mundial conocida como World Wide Web. [↑](#footnote-ref-4)