



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: Oscar Ardaiz

Pamplona, 28 de Abril de 2015

Índice

Índice	3
Introducción	5
Antecedentes	5
Objeto.....	5
Tecnologías básicas	6
Qué es una impresión 3D	6
JavaScript.....	8
Three.js.....	9
Mapbox	14
Shapeways.....	19
Desarrollo del trabajo	21
Ficheros HGT	22
Uso Cesium.....	24
Cesium	24
Carga GPX.....	27
Selección Mapa Mapbox.....	29
Mapa y coordenadas.....	30
Información tiles	30
Mapa y recorrido.....	32
Texturas.....	34
Creación de Mesh.....	40
Obtención de ruta	40
Creación de escena y controles gráficos	40
Información tiles	41
Información del terreno	41
Creación del terreno	43
Creación x3d y envío a impresora 3D.....	49
Creación del modelo	49
Subir el modelo	51
Resultados	55
Aplicación Web.....	55
Impresiones 3D	63
Problemas y soluciones.....	66
Conclusiones y líneas futuras	73
Bibliografía	75

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 *threejs* 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.

Antecedentes

Antes de establecer los objetivos de este proyecto podemos encontrar en la web una idea muy interesante en el artículo¹ “3D Print your trek, in color!”, este artículo describe los pasos a seguir para crear un modelo físico tridimensional a partir de una ruta realizada. Sigue los siguientes pasos:

- Descargar las elevaciones de los terrenos².
- Unir varias regiones en un solo mapa con el programa 3DEM³.
- Seleccionar la región a imprimir y guardar el archivo en formato *dem*.
- Convertir el fichero *dem* a *x3d*⁴ con el programa AccuTrans 3D⁵.
- Crear el modelo 3D en Blender⁶ añadiendo efectos con la herramienta.
- Preparar la información del recorrido *gps* usando el programa Qstarz para exportar un fichero en formato *kml*⁷.
- Generar la textura del terreno con *Google Earth*.
- Poner la textura del terreno sobre el modelo 3D con Blender además de editar la textura para que quede perfecta.
- Añadir grosor al modelo con Blender, escalar y exportar el modelo terminado.
- Imprimir el modelo 3D con *Shapeways*.

Objeto

El objetivo de este 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 WebGL⁸ con la librería *threejs* usando como referencia el artículo anteriormente descrito. Usaremos una ruta *gps* y a partir de ella crearemos el recorrido en un objeto tridimensional, de esta manera debo realizar los siguientes pasos:

- Con la ruta *gps* obtener los datos de las elevaciones de los terrenos.
- Textura para el modelado tridimensional con *Mapbox*.
- Creación del objeto tridimensional en el navegador con *ThreeJS*.
- Unir objeto y textura enviando la información a *Shapeways* para su creación física.

¹ <http://www.instructables.com/id/3D-Print-Your-Trek-in-color/>

² <http://earthexplorer.usgs.gov>

³ <http://www.visualizationsoftware.com/3dem.html>

⁴ x3d es una extensión para objetos 3D.

⁵ <http://www.micromouse.ca>

⁶ <http://www.blender.org>

⁷ xml para representar datos geográficos en 3D.

⁸ 3D en el navegador.

Tecnologías básicas

Qué es una impresión 3D

Es la creación de un objeto tridimensional mediante la impresión sucesiva de finas capas de un material, este proceso es denominado fabricación por adición.

Existe un gran número de tecnologías para la impresión 3D, sus principales diferencias se encuentran en la forma en que las diferentes capas son usadas para crear piezas. Cada método tiene sus propias ventajas e inconvenientes, por lo tanto, todo dependerá de las prioridades del cliente. Generalmente las consideraciones principales son:

- Velocidad
- Coste del prototipo impreso
- Coste de la impresora 3D
- Elección y coste de materiales
- Colores

Tipos	Tecnologías	Materiales
Extrusión	Modelado por deposición fundida	Termoplásticos, HDPE, metales eutécticos, materiales comestibles
Hilado	Fabricación por haz de electrones	Casi cualquier aleación
Granulado	Sinterizado directo de metal por láser	Casi cualquier aleación
	Fusión por haz de electrones	Aleaciones de titanio
	Sinterizado selectivo por láser	Polvo termoplástico
	Sinterizado selectivo por calor	Termoplásticos, polvos metálicos, polvos cerámicos
	Proyección aglomerante	Yeso
Laminado	Laminado de capas	Papel, papel de aluminio, capa de plástico
Fotoquímicos	Estereolitografía	Fotopolímero
	Fotopolimerización por luz ultravioleta	Fotopolímero

Tabla comparación tipos de impresiones 3D

Existen un centenar de impresoras 3D la mayoría se diferencian en las tecnologías que usan y en los materiales. Existen además otros factores importantes tales como los ficheros que soportan, el volumen de impresión y uno muy importante como es el precio.



Figura 1 - DIMA LT 200x200x200 stl, obj 1452€

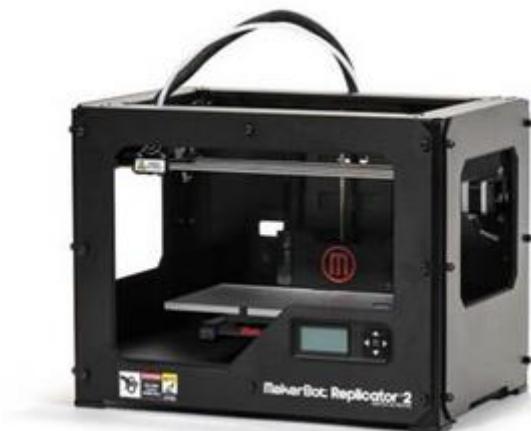


Figura 2 - Makerbot 240x140x150 stl, obj y thing 2399€



Figura 3 - CubeX Single 275x265x240 stl, cubex 2510€



Figura 4 - Witbox 210x297x200 stl 1690€

En las imágenes anteriores hemos podido ver algunos tipos de impresora 3D, la impresora Witbox es desarrollada por la empresa española BQ.

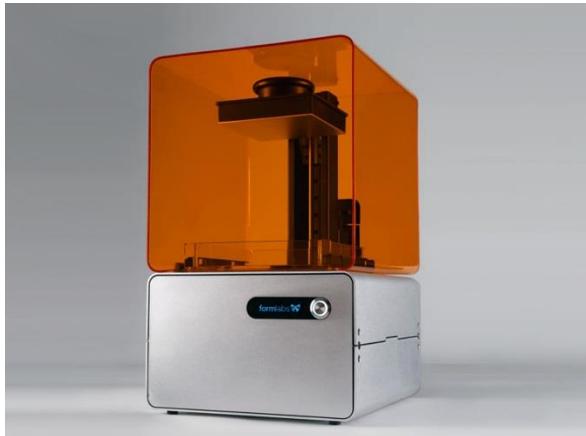


Figura 5 - Form1

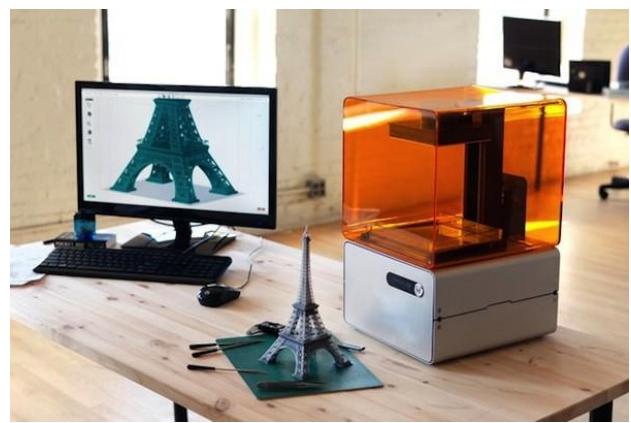


Figura 6 - Objeto 3D por impresora Form1

Impresora Form1 fabricada por FormLabs que imprime objetos usando el método de estereolitografía, su valor aproximado es de unos \$3300, el volumen de impresión es de 125x125x125 y los tipos de ficheros que soporta son .stl y .obj y pesa aproximadamente 8kg.

JavaScript

Es un lenguaje de programación interpretado⁹ 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 DOM¹⁰. Con la llegada de Ajax¹¹ 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.

⁹ Lenguaje interpretado: lenguaje de programación que está diseñado para ser ejecutado por medio de un intérprete.

¹⁰ DOM: Document Object Model.

¹¹ Ajax : Asynchronous JavaScript and XML

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 W3C¹².

Three.js

A grandes rasgos *threejs* es una librería que nos permite crear WebGL¹³, básicamente la diferencia cabe en que si queremos crear un cubo con *threejs* basta con algunas líneas en cambio con JavaScript la cantidad de líneas sería mayor.

Al igual que la mayoría de librerías podemos descargar la librería completa desde la página web oficial o acceder a una versión más ligera a través de su *cdn*¹⁴.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/three.js/r70/three.min.js"> //cdn three.js </script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/three.js/r70/three.js"> //cdn three.js </script>
```

Figura 7 - cdn three.js

Para mostrar gráficos en 3D en el navegador con *threejs* debemos crear una serie de elementos y funciones.

Creando la escena

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.

¹² WWWC World Wide Web Consortium, consorcio internacional que realiza recomendaciones a la red informática mundial conocida como World Wide Web.

¹³ 3D en el navegador.

¹⁴ Content delivery network.

```

<html xml:lang="es" lang="es">
<head>
  <!-- JavaScript: Three.js -->
  <script type="text/javascript" src="threeJS/js/Detector.js"> //Detector web support WebGL </script>
  <script type="text/javascript" src="threeJS/js/libs/stats.min.js"> //Stats </script>
  <script type="text/javascript" src="threeJS/build/three.js"> //ThreeJS </script>
  <script type="text/javascript" src="threeJS/js/controls/OrbitControls.js"> //Orbit Camera Control </script>
  <script type="text/javascript" src="threeJS/js/libs/dat.gui.min.js"> //GUI Graphic user interface </script>
  <script type="text/javascript" src="js/threejs.js"> //My library. </script>
  <title> Scene Three.js </title>
</head>

<body onload="loadThreeJS()">

</body>
</html>

```

Figura 8 - Declaración scripts

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 creará 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.

```

if ( ! Detector.webgl ) Detector.addGetWebGLMessage();
else {
  init();
}

```

Figura 9 - Detector WebGL

Si nuestro navegador permite el uso de WebGL llamamos a la función *init*, 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.

```

var container = document.createElement('div');
document.body.appendChild(container);

```

Figura 10 - Obtener elemento del DOM

Escena

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

```

scene = new THREE.Scene();

```

Figura 11 - Crear escena threejs

Cámara

Threejs 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.

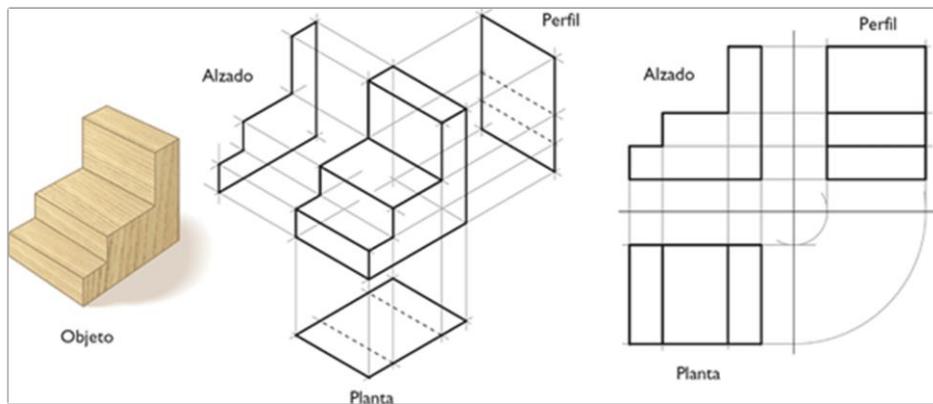


Figura 12 - Proyección Ortogonal

- 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.

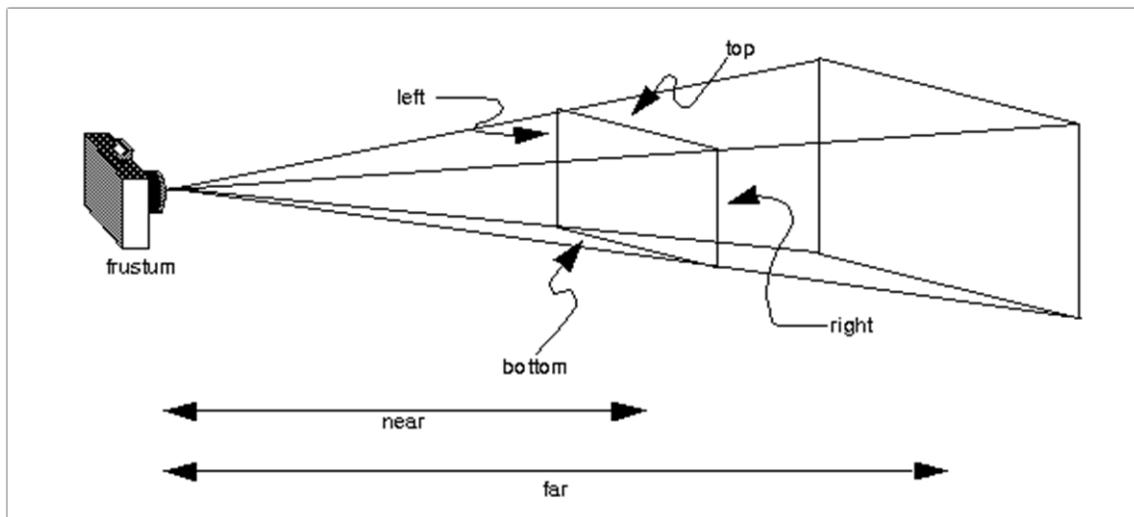


Figura 13 - Proyección Perspectiva

Las cámaras tienen 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, 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.

```
camera = new THREE.PerspectiveCamera(45, window.innerWidth / window.innerHeight, 0.1, 1000);
```

Figura 14 – Creación cámara

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

```
camera.position.z = 100;  
camera.lookAt(scene.position);
```

Figura 15 – Posicionamiento cámara

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.

```
renderer = new THREE.WebGLRenderer();  
renderer.setClearColor(new THREE.Color(0xEEEEEE, 1.0));  
renderer.setSize(window.innerWidth, window.innerHeight);  
renderer.shadowMapEnabled = true;
```

Figura 16 – Creación del Render

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.

```
controlCamera = new THREE.OrbitControls(camera, renderer.domElement);
```

Figura 17 – Creación control de cámara

Una vez creado todos los objetos debemos añadir al DOM el 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.

```
container.appendChild(renderer.domElement);  
render();
```

Figura 18 – Añadir el render al DOM

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.

```
stats = initialStats();

function initialStats() {
    var stat = new Stats();
    stat.setMode(0); // 0: framepersegundo, 1: milisecondo
    // Align top-left
    stat.domElement.style.position = 'absolute';
    stat.domElement.style.left = '0px';
    stat.domElement.style.top = '0px';
    document.body.appendChild(stat.domElement);
    return stat;
}
```

Figura 19 – Creación e inserción de *Stats*

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

```
function render() {
    controlCamera.update();
    stats.update();
    requestAnimationFrame(render);
    renderer.render(scene, camera);
}
```

Figura 20 – Función de renderizado

Cabe mencionar que todas las variables descritas son variables globales de JavaScript, 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.

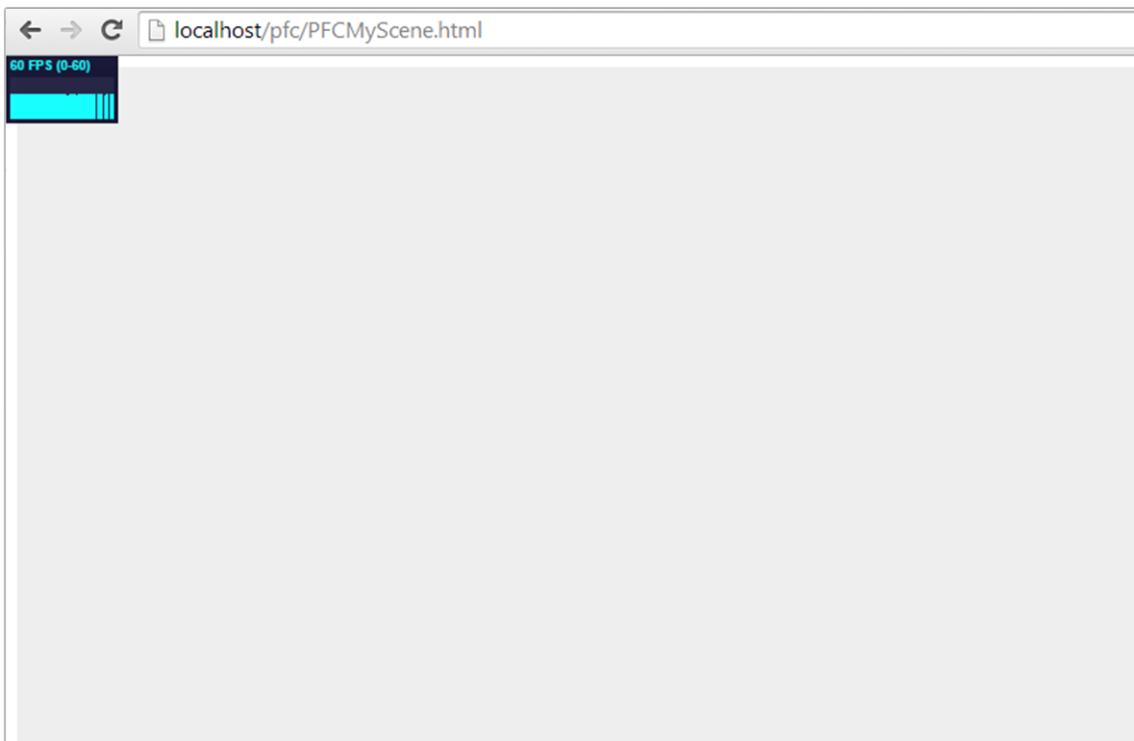


Figura 21 – Escena vacía ThreeJS

Mapbox

Es una librería JavaScript que nos permite crear mapas personalizados, sitios web como Foursquare, Pinteres, Evernote usan esta 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 oficial¹⁵.

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*.

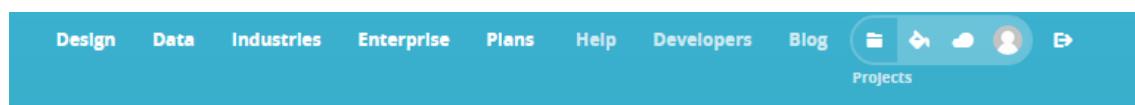


Figura 22 – Panel de control Mapbox

¹⁵ <https://www.mapbox.com/>

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

Projects ?



Figura 23 – Creación proyecto Mapbox

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.

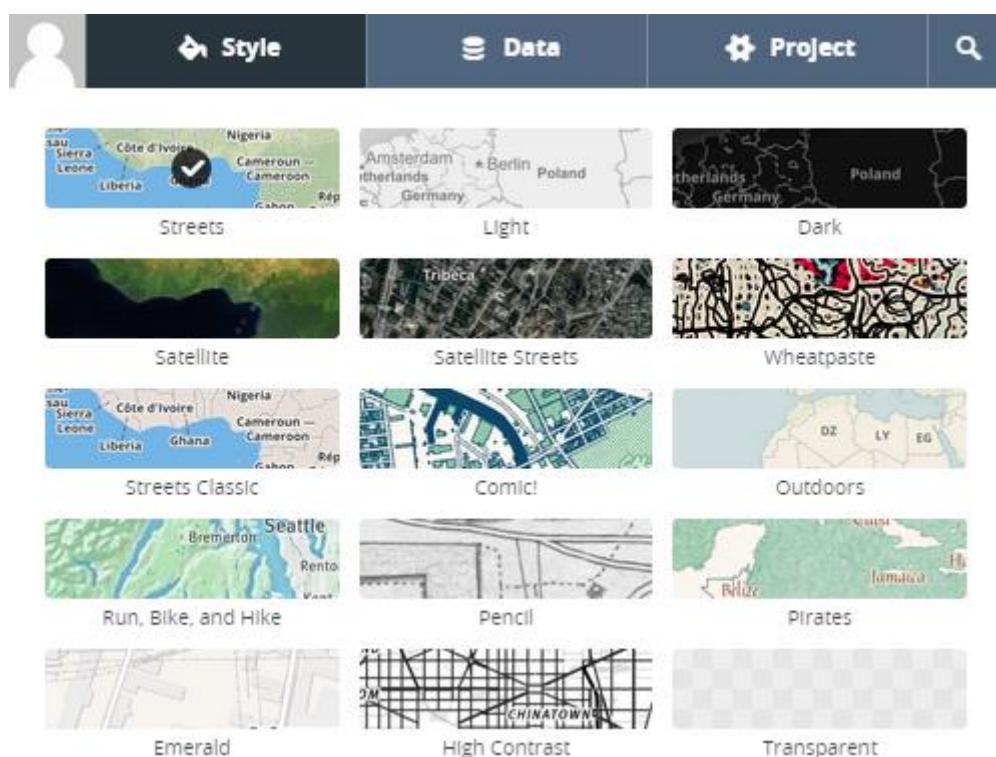


Figura 24 – Selección estilos Mapbox

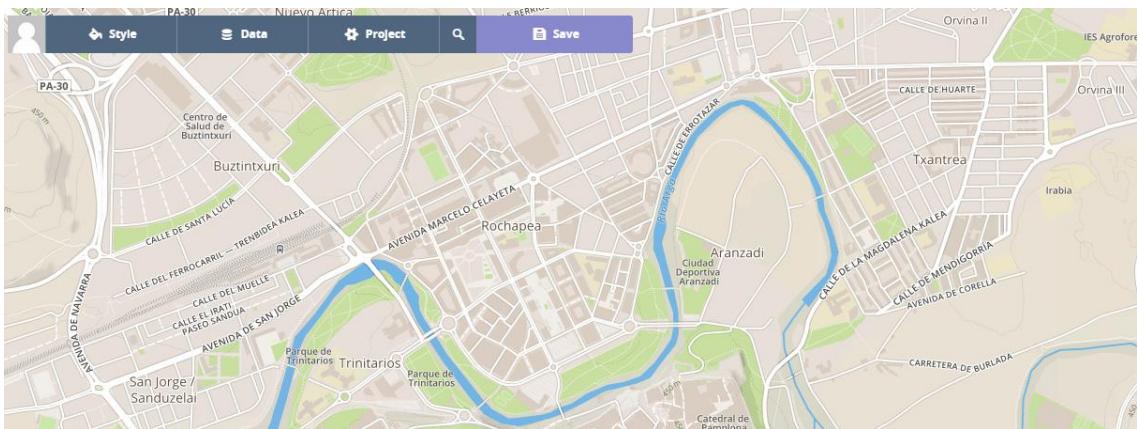


Figura 25 – Mapa personalizado Mapbox

Ahora solo nos queda guardar nuestro proyecto haciendo clic en la opción *Save*.



Figura 26 – Guardar mapa personalizado Mapbox

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.

Projects ?

Filter your projects



Figura 27 – Nombre e id del proyecto

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.

```
<script src='https://api.tiles.mapbox.com/mapbox.js/v2.1.5/mapbox.js'></script>
<link href='https://api.tiles.mapbox.com/mapbox.js/v2.1.5/mapbox.css' rel='stylesheet' />
```

Figura 28 –cdn Mapbox

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*¹⁶.



Figura 29 – Clave acceso API Mapbox

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 ubicado en Pamplona, Navarra.

```
<html xml:lang="es" lang="es">
<html>
<head>
  <meta name='viewport' content='initial-scale=1,maximum-scale=1,user-scalable=no' />
  <script src='https://api.tiles.mapbox.com/mapbox.js/v2.1.5/mapbox.js'></script>
  <link href='https://api.tiles.mapbox.com/mapbox.js/v2.1.5/mapbox.css' rel='stylesheet' />
  <style>
    body { margin:0; padding:0; }
    #map { position:absolute; top:0; bottom:0; width:100%; }
  </style>
<title>Mapa Simple</title>
</head>
<body>
  <div id='map'></div>
  <script>
    L.mapbox.accessToken = 'pk.eyJ1IjoiZ9uemFsaXRvIiwiYSI6IlVJTGIweFUifQ.waoF7m8PZbBM6u8Tg_rR7A';
    var map = L.mapbox.map('map', 'gonzalito.lg2cg21').setView([42.8, -1.634], 17);
  </script>
</body>
</html>
```

Figura 30 – Ejemplo mapa Mapbox

¹⁶ API Access token.

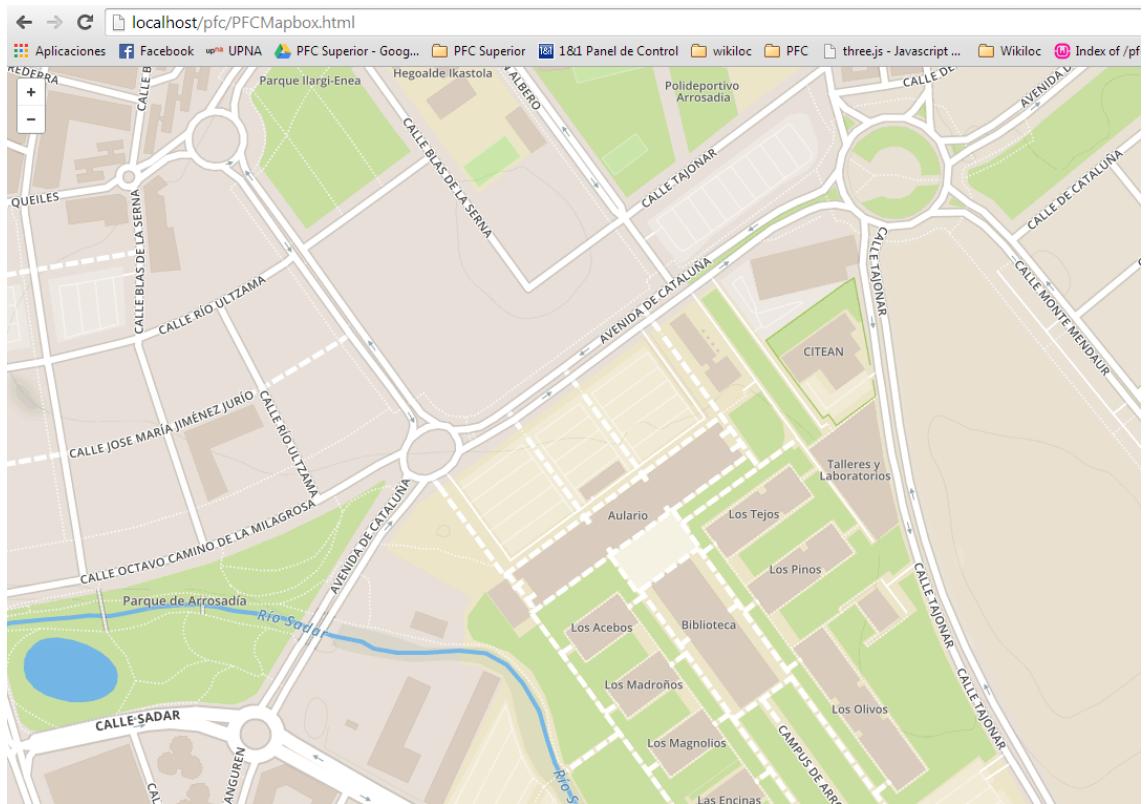
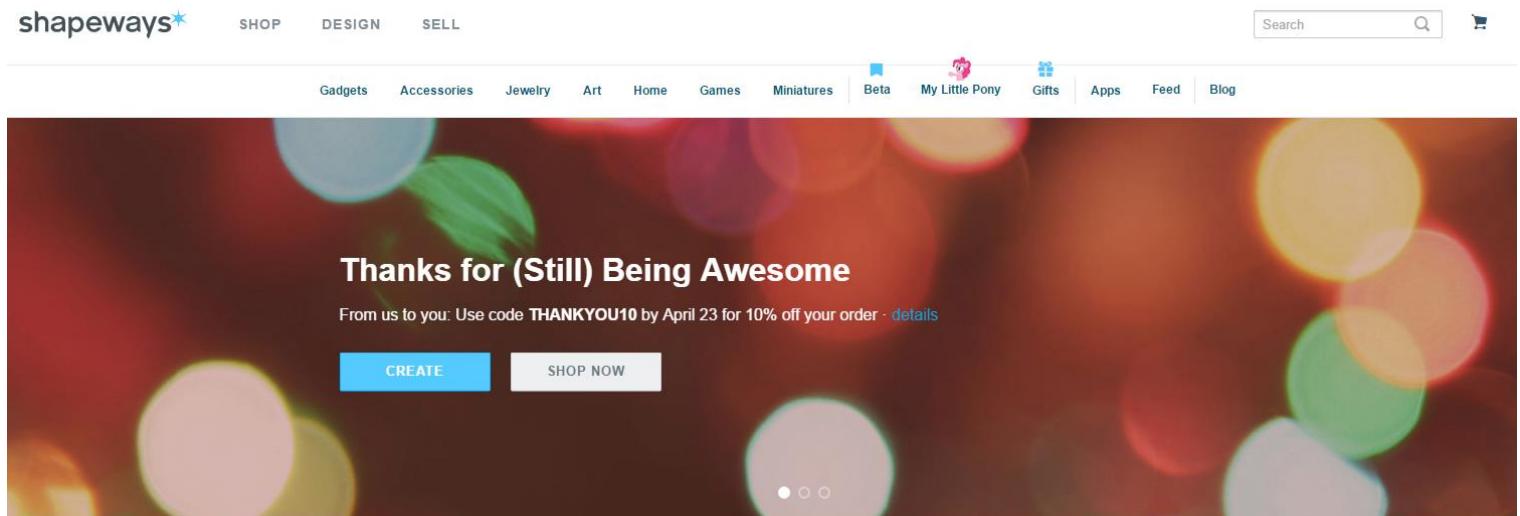


Figura 31 – Ejemplo web Mapbox

Shapeways

Es una compañía cuya función principal es la impresión en 3D, nos podemos crear una cuenta usando un perfil de *Facebook*. Al disponer de una cuenta podemos subir nuestros modelos 3D creados con *threejs* y verificar si están bien construidos y si cumplen sus estándares para ser impresos. Se pueden elegir un sinfín de materiales lo cual implicaría un aumento o disminución de su precio, además de otros factores que alteran el precio como el tamaño del modelo.



The World's Leading 3D Printing Service & Marketplace

Shapeways Enables Everyone to Bring Their Ideas to Life

Figura 32 – Portada Shapeways

Shapeways dispone de distintos tipos de impresoras 3D a continuación una serie de imágenes de impresoras que existen en su fábrica.



Figura 33 - Impresora 3D Stratasys FDM 400mc

El modelo Stratasys es uno de los más grandes y rápidos a la hora de crear prototipos.



Figura 34 - Impresora 3D Eden 500

El modelo Eden producido en Israel es para uso doméstico o de oficina y se caracteriza por su gran nivel de detalle, en comparación con la anterior es mucho más pequeño.



Figura 35 - Impresora 3D Dimension sst768

Dimension sst 768 es creada por la compañía Stratasys, tiene un aspecto de una máquina de café y es para imprimir modelos pequeños.

Desarrollo del trabajo

En este apartado se explicará el desarrollo y funcionamiento de la aplicación web. Se explicará cómo se han obtenido los datos de las elevaciones de los terrenos, la textura a usar y la creación e importación del modelado tridimensional.

La arquitectura de la aplicación web será la siguiente

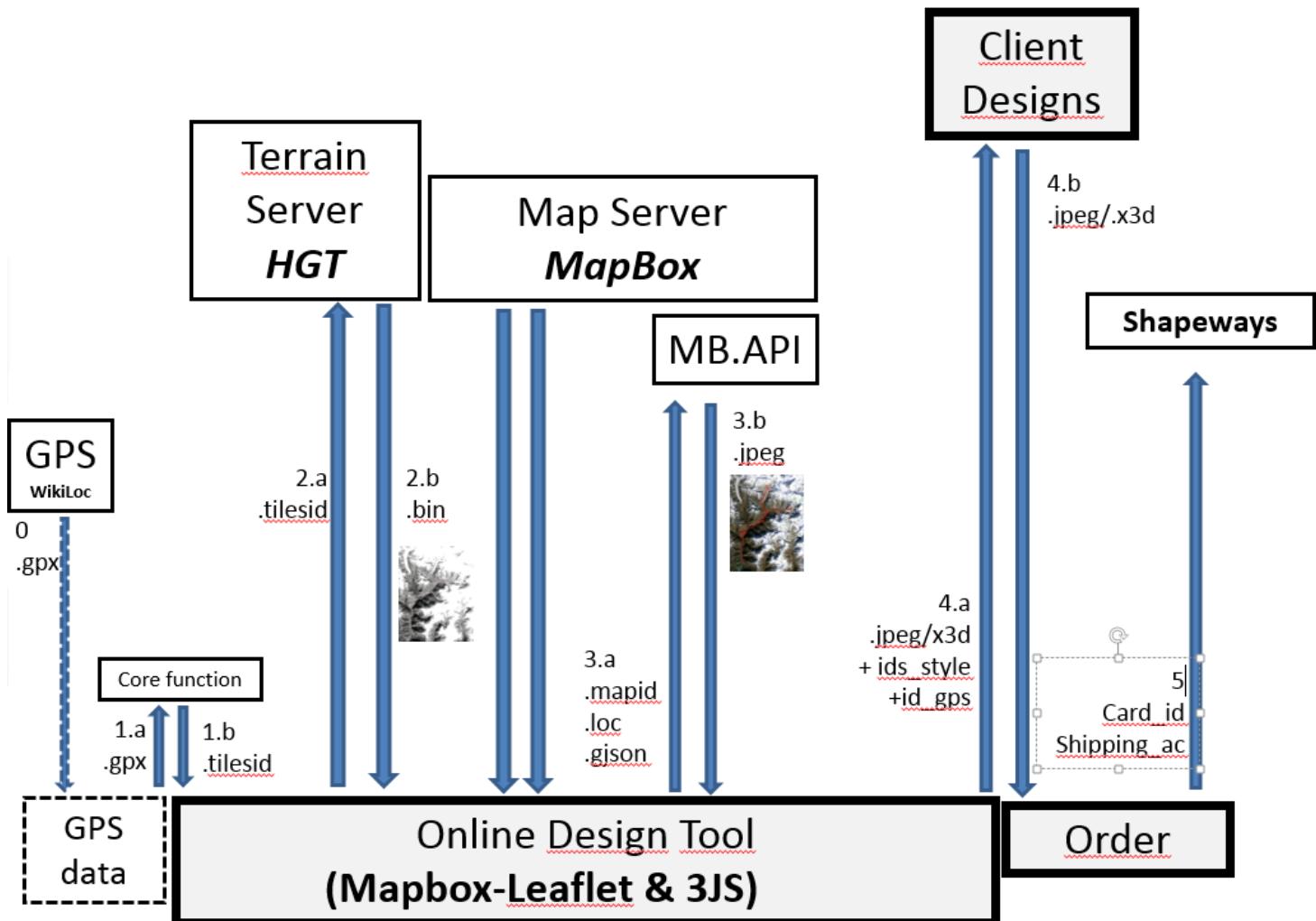


Figura 36 - Arquitectura Aplicación Web Inicial

Ficheros HGT

Estos ficheros contienen las elevaciones de la superficie de la tierra por regiones. Cada fichero describe un tile con la altura en grados de latitud y el ancho en grados de longitud, el nombre de cada fichero hace referencia a la latitud y longitud en la esquina inferior izquierda del tile, un ejemplo el fichero *N43W002.hgt*.

Realice un estudio de cómo interpretar la información de estos ficheros obteniendo los siguientes resultados basándome en el algoritmo *square-diamond* para la creación de terrenos aleatorios, este tipo de algoritmo se usa para la creación de terrenos en los videojuegos.

Junto a la memoria se adjunta los ficheros JavaScript para la creación de terrenos aleatorios sin base, con lo visto en el creador de terreno con *Cesium* se podría añadir fácilmente los laterales y el fondo creando una geometría completa. Imágenes de terrenos creados aleatoriamente:

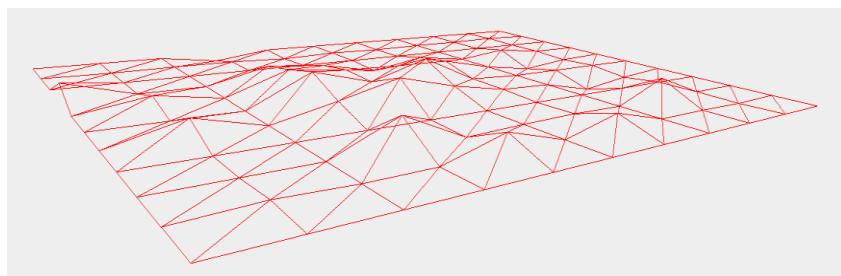


Figura 37 - Terreno Aleatorio 1

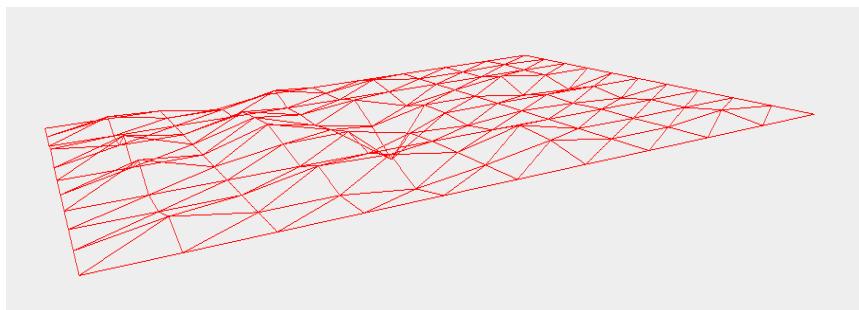


Figura 38 - Terreno Aleatorio 2

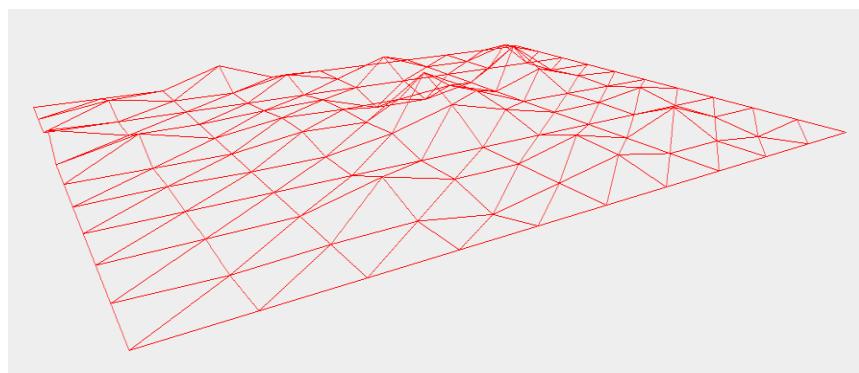


Figura 39 - Terreno Aleatorio 3

Imagen correspondiente a un fichero *hgt* y representación gráfica con *threejs*

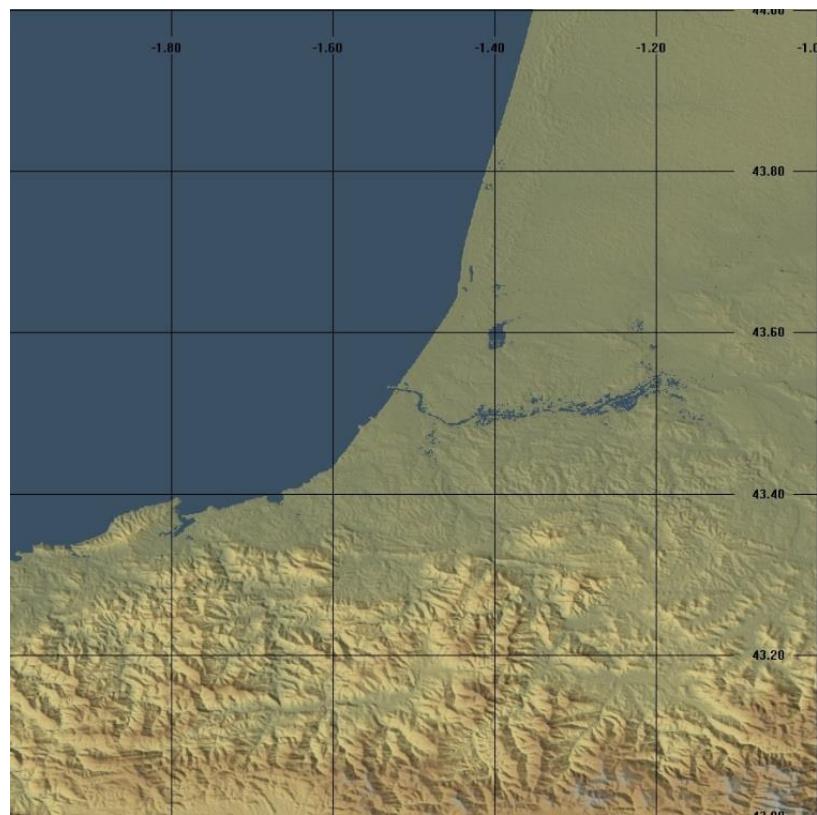


Figura 40 - Fichero *hgt* visualizado en el programa 3DEM

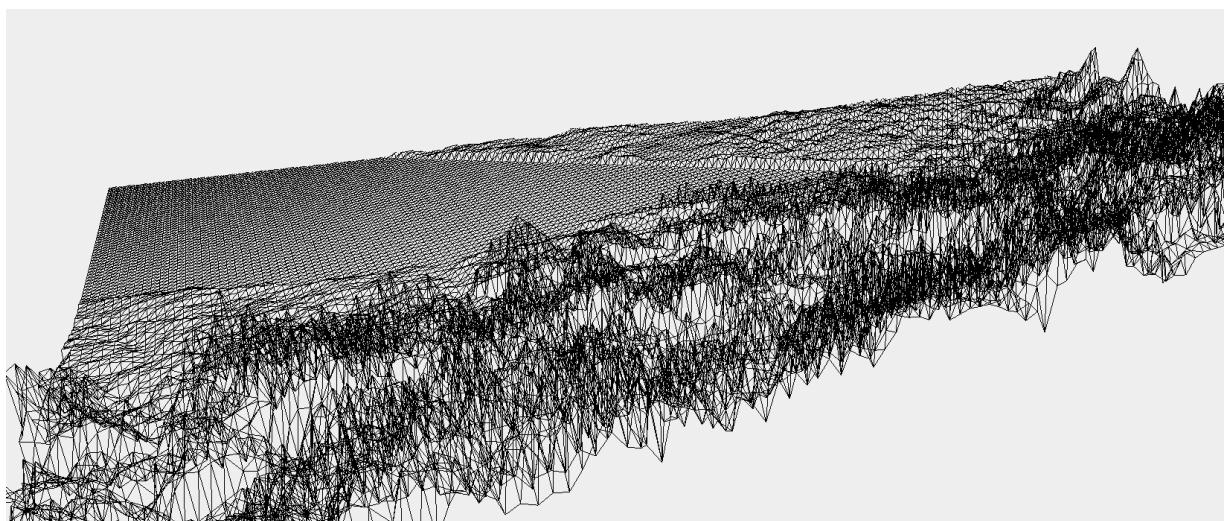


Figura 41 – Fichero *hgt* modelado con Threejs

Uso Cesium

A la hora de usar los ficheros *hgt* me encontré con los siguientes problemas:

- Habría que descargar todos los ficheros correspondientes al plantea completo, gigabytes de información.
- Dificultad a la hora de trabajar con los datos debido a la gran cantidad de datos que son contenidos en ellos.
- Ubicación de coordenadas dentro del fichero.
- Gran cantidad de datos que ralentizaba la visualización del modelo en el navegador, muchos datos para el motor gráfico del navegador.

Debido a los problemas mencionados anteriormente en los ficheros *hgt* se optó por utilizar la librería *Cesium JS*.

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 acceso¹⁷ 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 oficial¹⁸.

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 los tiles la graficaremos usando *threejs*.

Un ejemplo

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.

¹⁷ Open Source

¹⁸ <http://cesiumjs.org/downloads.html>

```

<html xml:lang="es" lang="es">
<head>
    <script type="text/javascript" src="js/Cesium.js"> //Cesium </script>
    <script type="text/javascript" src="js/my_cesium.js"> //Cesium </script>
    <title> PFC Cesium </title>
</head>

<body onload="requestTilesWhenReady()">

</body>

</html>

```

Figura 42 – Declaración Script y función inicial

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*.

```

var aCesiumTerrainProvider = new Cesium.CesiumTerrainProvider({
    url : '//cesiumjs.org/stk-terrain/tilesets/world/tiles'
});

function requestTilesWhenReady() {
    var lat = 42.8, lon = -1.634, level = 14;
    if (aCesiumTerrainProvider.ready) {
        console.log("[PFC my_cesium_mesh.js]: Cesium Server Terrain Provider ready");
        showData(lat, lon, level);
    } else {
        setTimeout(requestTilesWhenReady, 10);
    }
}

```

Figura 43 – Acceso a Cesium

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.

```

function showData(lat, lon, level) {
    document.write("[PFC my_cesium.js]: Show data");
    var positionLonLat, positionTileXY, requestPromise, tilePromise, promise = [], cesium_data;
    positionLonLat = Cesium.Cartographic.fromDegrees(lon, lat);
    positionTileXY = aCesiumTerrainProvider.tilingScheme.positionToTileXY(positionLonLat, level);
    requestPromise = aCesiumTerrainProvider.requestTileGeometry(positionTileXY.x, positionTileXY.y, level, false).then(function(data){
        cesium_data = data;
    });
    tilePromise = Cesium.when(requestPromise);
    promise.push(tilePromise);

    Cesium.when.all(promise, function() {
        console.log("[PFC my_cesium.js]: All request asynchronous complete.");
        console.log(cesium_data);
    });
}

```

Figura 44 – Función showData

En la función `showData` realizamos lo siguiente:

- `Cesium.Cartographic.fromDegrees(longitude, latitude, height optional, result 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.

```
▶ _boundingSphere: h
  _childTileMask: 0
  _createdByUpsampling: false
▶ _eastIndices: Uint16Array[15]
  _eastSkirtHeight: 23.519085626208074
  _encodedNormals: undefined
▶ _heightValues: Uint16Array[177]
▶ _horizonOcclusionPoint: o
▶ _indices: Uint16Array[909]
  _maximumHeight: 502.2276306152344
  _minimumHeight: 474.3674621582031
▶ _northIndices: Uint16Array[11]
  _northSkirtHeight: 23.519085626208074
▶ _quantizedVertices: Uint16Array[531]
▶ _southIndices: Uint16Array[12]
  _southSkirtHeight: 23.519085626208074
▶ _uValues: Uint16Array[177]
▶ _vValues: Uint16Array[177]
  _waterMask: undefined
▶ _westIndices: Uint16Array[15]
  _westSkirtHeight: 23.519085626208074
```

Figura 45 – Objeto respuesta 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 se guarda la longitud, latitud y altura.
- `indices` que nos indica cómo se unen los vértices para formar el terreno.
- Con estos datos crearemos las geometrías de los terrenos, esto se explicará más adelante en el desarrollo del proyecto.

Carga GPX

A partir de este apartado se explicará el funcionamiento de los distintos interfaces de la aplicación Web.

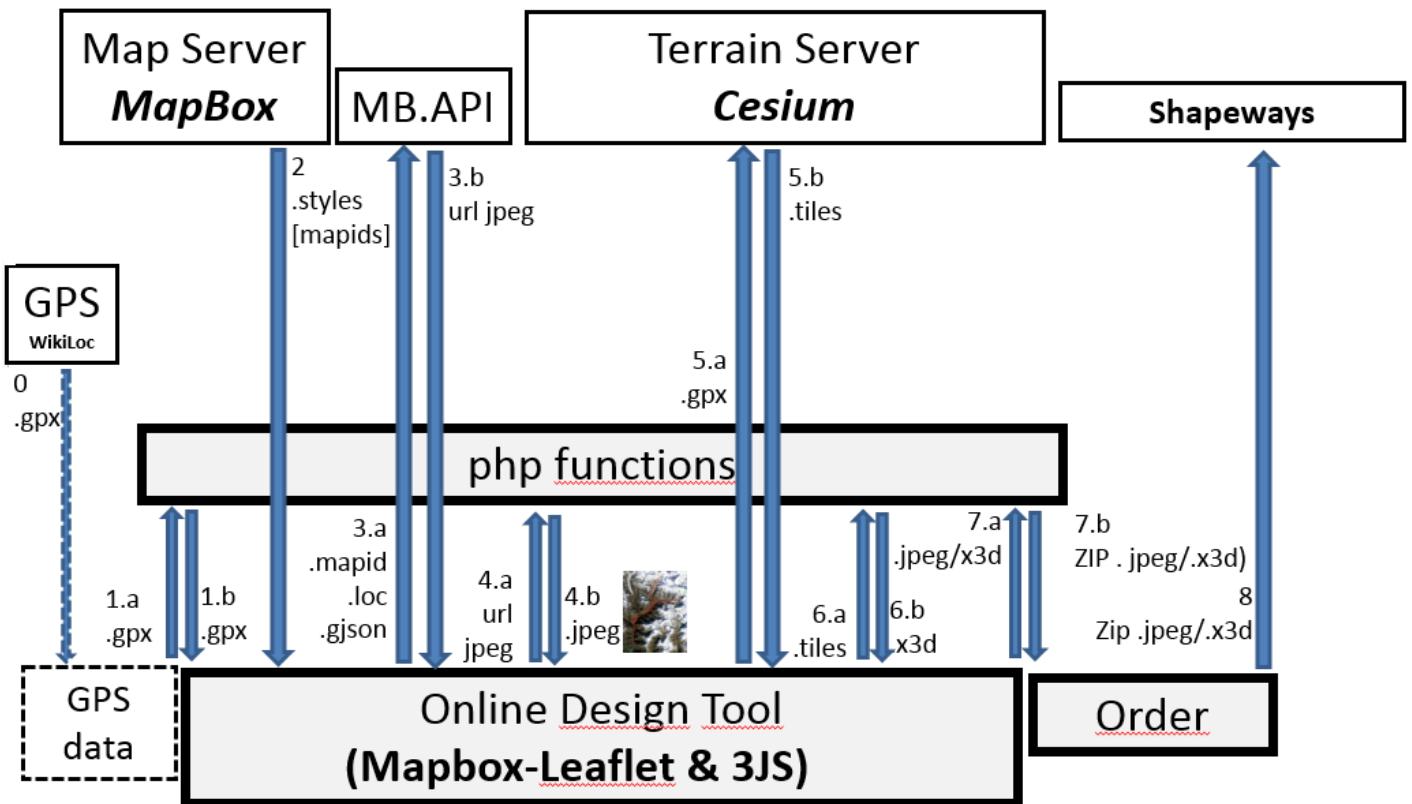


Figura 46 – Arquitectura Aplicación Web Final

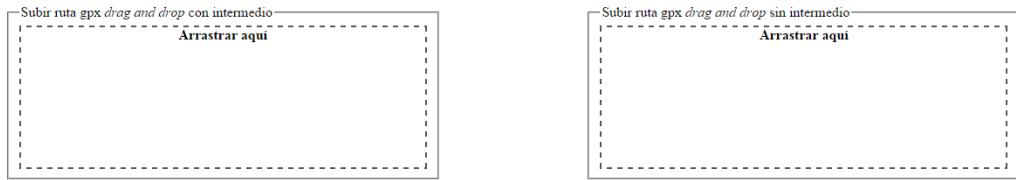
Se carga la página *PFCIndex.html* donde se subirá el fichero *gpx* al servidor mediante *ajax* y *php*, se ha controlado que solo se puedan subir ficheros *gpx* y cada vez que se cargue la página inicial se borrarán todos los datos en el servidor.

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

```
sessionStorage.setItem("rute", response);
```

Para acceder a la variable almacenada en sesión solo debemos indicar el nombre de la variable.

```
var name = sessionStorage.rute.substring(sessionStorage.rute.indexOf("/") + 1, sessionStorage.rute.length);
```



Mensaje de estado: Navegador soporta File-FileList-FileReader

Figura 47 – PFCIndex.html

Podremos elegir dos formas de subir el fichero, la primera nos permite elegir el tipo de mapa y visualizar la ruta antes de obtener el modelado 3D y la segunda cargaría el modelado 3D por defecto.

Selección Mapa Mapbox

Si hemos elegido la primera opción que nos permite elegir el tipo de mapa y visualizar la ruta antes de crear el modelado 3D podremos interactuar en la página *PFCMyRute.html*, en caso contrario se cargará la página y se bloqueará hasta obtener todas las imágenes correspondiente a la ruta y se abrirá la página *PFCMyMesh.html* donde se creará el modelado 3D.

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



Figura 48 – PFCMyRute.html estilo Satelite

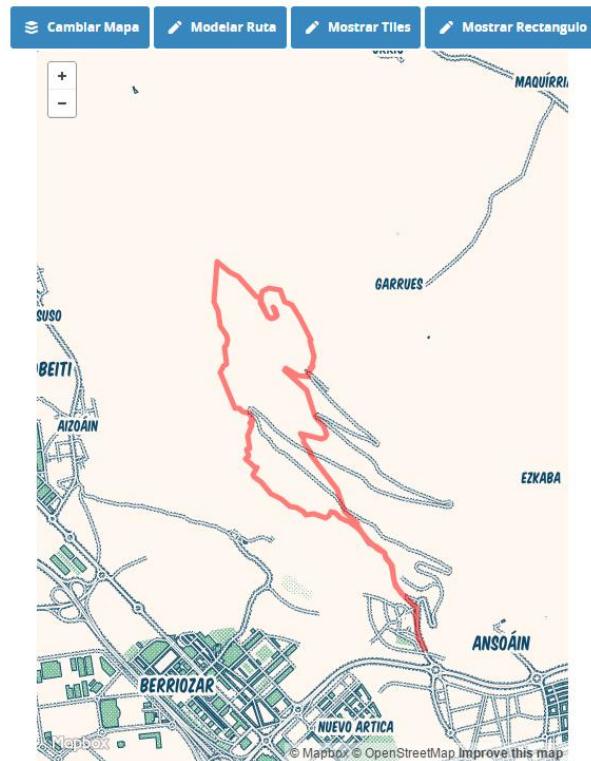


Figura 49 – PFCMyRute.html estilo personalizado

En este apartado se explicará cómo se obtienen todos datos para la creación del terreno e imágenes del mismo.

Mapa y coordenadas

Cuando la conexión a Cesium esté lista llamamos a la función `getRoute()` con la cual obtendremos las coordenadas de la ruta 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

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 un área rectangular del mapa. 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.

```

function InfoTile(x, y, cardinality, northwest_latitude, northwest_longitude, southeast_latitude, southeast_longitude, longitude, latitude, index) {
    this.x = x;
    this.y = y;
    this.bounds = [[northwest_latitude, northwest_longitude], [southeast_latitude, southeast_longitude]];
    this.cardinality = cardinality;
    /*
     * Reverse, to create correctly GeoJson.
     * Normal is [latitude, longitude];
    */
    this.coordinate = [longitude, latitude];
    this.index = index;
    this.geometry;
}

```

Figura 50 – Código función InfoTile

```

function checkTile(coord, level) {
    var i, j, positionLonLat, positionTileXY, tile;
    positionLonLat = Cesium.Cartographic.fromDegrees(coord[0][1], coord[0][0]);
    positionTileXY = aCesiumTerrainProvider.tilingScheme.positionToTileXY(positionLonLat, level);
    tile = getTile(positionTileXY.x, positionTileXY.y, level);
    info_tiles.push(new InfoTile(positionTileXY.x, positionTileXY.y, "c", tile.northwest.latitude, tile.northwest.longitude,
        tile.southeast.latitude, tile.southeast.longitude, coord[0][1], coord[0][0], 0));
    for (i = 1; i < coord.length; i++) {
        var positionLonLat_actual, positionTileXY_actual;
        positionLonLat_actual = Cesium.Cartographic.fromDegrees(coord[i][1], coord[i][0]);
        positionTileXY_actual = aCesiumTerrainProvider.tilingScheme.positionToTileXY(positionLonLat_actual, level);
        if ((positionTileXY.x != positionTileXY_actual.x) || (positionTileXY.y != positionTileXY_actual.y)) {
            tile = getTile(positionTileXY_actual.x, positionTileXY_actual.y, level);
            if (positionTileXY.x < positionTileXY_actual.x)
                info_tiles.push(new InfoTile(positionTileXY_actual.x, positionTileXY_actual.y, "checkTile", tile.northwest.latitude, tile.northwest.longitude,
                    tile.southeast.latitude, tile.southeast.longitude, coord[i][1], coord[i][0], i));
            else if (positionTileXY.x > positionTileXY_actual.x)
                info_tiles.push(new InfoTile(positionTileXY_actual.x, positionTileXY_actual.y, "checkTile", tile.northwest.latitude, tile.northwest.longitude,
                    tile.southeast.latitude, tile.southeast.longitude, coord[i][1], coord[i][0], i));
            else if (positionTileXY.y < positionTileXY_actual.y)
                info_tiles.push(new InfoTile(positionTileXY_actual.x, positionTileXY_actual.y, "checkTile", tile.northwest.latitude, tile.northwest.longitude,
                    tile.southeast.latitude, tile.southeast.longitude, coord[i][1], coord[i][0], i));
            else if (positionTileXY.y > positionTileXY_actual.y)
                info_tiles.push(new InfoTile(positionTileXY_actual.x, positionTileXY_actual.y, "checkTile", tile.northwest.latitude, tile.northwest.longitude,
                    tile.southeast.latitude, tile.southeast.longitude, coord[i][1], coord[i][0], i));
            positionTileXY = positionTileXY_actual;
        }
    }
    /*
     * Remove repeat elements.
    */
    var info_tiles_clone = info_tiles;
    for(i = 0; i < info_tiles_clone.length; i++) {
        var actual = info_tiles_clone[i];
        var pos = new Array();
        for(j = 0; j < info_tiles.length; j++) {
            if ((actual.x == info_tiles[j].x) && (actual.y == info_tiles[j].y))
                pos.push(j);
        }
        /*
         * is j=1 because the first never remove.
        */
        for(j = 1; j < pos.length; j++)
            info_tiles.splice(pos[j], 1);
    }
    /*
     * Compare the two last.
    */
    if (info_tiles.length > 1)
        if ((info_tiles[info_tiles.length-1].x == info_tiles[info_tiles.length-2].x) && (info_tiles[info_tiles.length-1].y == info_tiles[info_tiles.length-2].y))
            info_tiles.splice(info_tiles.length-1, 1);
}

```

Figura 51 – Código función checkTile

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)`. Esta 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 arreglo `rectangle_tiles`.

```
function createRectangle(info, level) {
    info_tiles_rectangle = new Array();
    var min_x = info[0].x, max_x = info[0].x, min_y = info[0].y, max_y = info[0].y;
    for(i = 1; i < info.length; i++) {
        if (info[i].x < min_x)
            min_x = info[i].x;
        if (info[i].x > max_x)
            max_x = info[i].x;
        if (info[i].y < min_y)
            min_y = info[i].y;
        if (info[i].y > max_y)
            max_y = info[i].y;
    }
    /*
     * Create a rectangle.
    */
    for(i = min_x; i <= max_x ; i++) {
        for (j = max_y; j >= min_y ; j--) {
            tile = getTile(i, j, level);
            info_tiles_rectangle.push(new InfoTile(i, j, "createRectangle",tile.northwest.latitude, tile.northwest.longitude, tile.southeast.latitude, tile.southeast.longitude, 0, 0, 0));
        }
    }

    for(i = 0; i < info.length; i++){
        for(j = 0; j < info_tiles_rectangle.length; j++) {
            if((info[i].x == info_tiles_rectangle[j].x) && (info[i].y == info_tiles_rectangle[j].y)) {
                info_tiles_rectangle[j] = info[i];
            }
        }
    }
}

console.log("[PFC functions.js]: Rectangle mesh ready with " + info_tiles_rectangle.length + " tiles.");
return info_tiles_rectangle;
}
```

Figura 52 – Código función `createRectangle`

Mapa y recorrido

Con el mapa creado y con toda la información necesaria para la creación de nuestro modelado en el arreglo `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 se dibujaba bien la ruta.



Figura 53 – Ruta gpx

Por cada coordenada verifico al tile que pertenece obteniendo de esta manera todos los tiles correspondientes a la ruta. Además almaceno los límites Sur-Oeste, Sur-Este, Nor-Oeste, Nor-Este de cada tile, así en *Mapbox* puedo dibujar un rectángulo para identificar cada tile de la ruta.



Figura 54 – Tiles 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 55 – Tiles nivel 14



Figura 56 – Tiles nivel 15

Texturas

En este apartado se explicará cómo se crea la imagen de nuestro recorrido.

Para obtener 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.

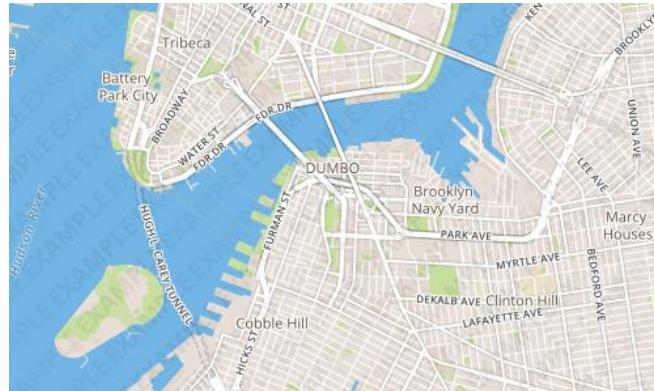


Figura 57 – Textura simple

- Imágenes de mapas con marcadores.

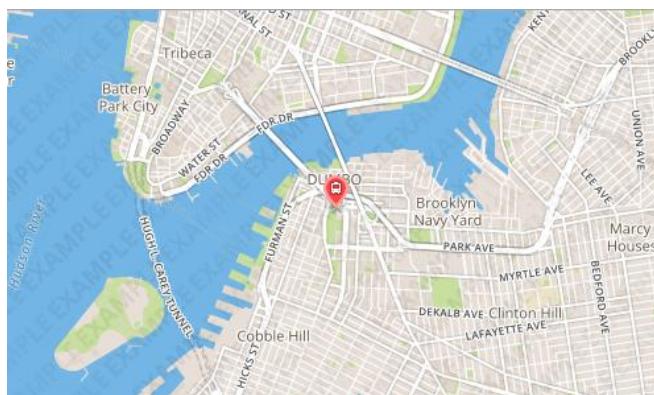


Figura 58 – Textura con marcador

- Imágenes de mapas con rutas mediante GeoJSON.

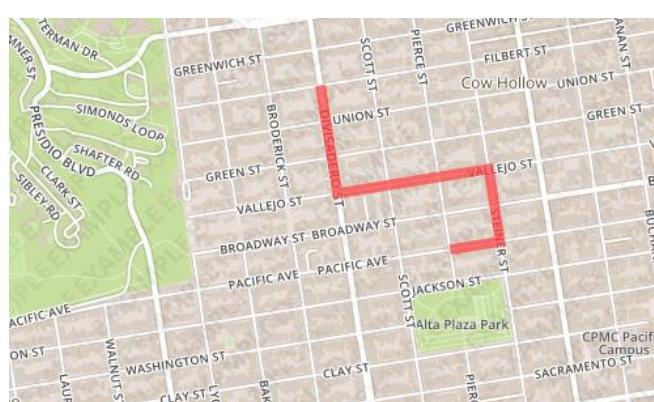


Figura 59 – Textura con GeoJSON

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

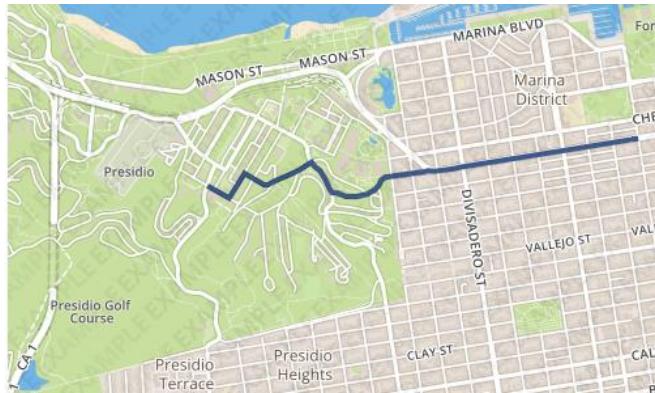


Figura 60 – Textura con Encode poluline

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 aplico la función *fixCoordinates()* que reduce la cantidad de coordenadas sin perder el sentido de la ruta.

Para obtener las texturas de cada tile he creado la función *drawRoute()*, esta función recorre todos los tiles almacenados en el arreglo *rectangle_tiles* y por cada tile se realiza lo siguiente:

```

function drawRoute(){
    console.clear();
    var i, j, bounds, out_bounds = 0, static_image_json;
    var coordinate_out, polyline;
    var json_coordinates = new Array(), reverse_line_points = new Array();
    var url = new Array();
    var polyline;

    var rest = (180/Math.PI *Math.log(Math.tan(Math.PI/4 + (rectangle_tiles[0].bounds[0][0]) *(Math.PI/180)/2))) -
        (180/Math.PI *Math.log(Math.tan(Math.PI/4 + (rectangle_tiles[0].bounds[1][0]) *(Math.PI/180)/2)));
    var width = 510, height = Math.round(width * rest * 100 - 75);
    var zoom_map = 16;
    var name = sessionStorage.rute.substring(sessionStorage.rute.indexOf("/") + 1, sessionStorage.rute.length);
    var file_name = name.substring(0, name.indexOf("."));
    var coordinate_after, coordinate_before;
    var fails, fails_total;

    fails_total = coordinates.length;

    var pos_out, cambio;
    var short_coordinates;
    for (i = 0; i < rectangle_tiles.length; i++) {
        json_coordinates = new Array();
        reverse_line_points = new Array();
        short_coordinates = new Array();
        out_bounds = 1;
        fails = 0;
        pos_out = 1;
        cambio = 0;
        //Mesh with coordinates.

```

Figura 61 – Código función drawRoute 1

1. Compruebo que el tile corresponde a la ruta y no a los tiles que cree como relleno, por cada uno de ellos:
 - 1.1. 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, de este modo no perdemos el trazado original.
 - 1.2. 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. Compruebo que las coordenadas estén dentro de los límites de cada tile. En el estudio para obtener la ruta pensando en todas las rutas posibles que puede realizar un usuario me encontré con los siguientes problemas:
 - 1.2.1. 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.
 - 1.2.2. Limitación a la hora de crear texturas con Mapbox por cualquiera de los métodos descritos tenemos un límite de coordenadas.

1.2.3. Almacenar más coordenadas de las debidas. Cuento la cantidad de posiciones que se ha salido, de este modo luego puedo saber la cantidad de coordenadas que pertenecen a la ruta, esto lo hago ya que al principio almacenaba todas las coordenadas lo cual al final provocaba fallos.

```

if ( (rectangle_tiles[i].coordinate[0] != 0) && (rectangle_tiles[i].coordinate[1] != 0) ) {
    //Add coordinate of previous tile.
    if (rectangle_tiles[i].index != 0) {
        coordinate_after = coordinates[rectangle_tiles[i].index-1];
        json_coordinates.push(coordinate_after);
    }
    for(j = rectangle_tiles[i].index; j < coordinates.length; j++) {
        if ((coordinates[j][0] < rectangle_tiles[i].bounds[0][0]) && (coordinates[j][0] > rectangle_tiles[i].bounds[1][0]) &&
            (coordinates[j][1] > rectangle_tiles[i].bounds[0][1]) && (coordinates[j][1] < rectangle_tiles[i].bounds[1][1])) {
            out_bounds = 0;
            json_coordinates.push(coordinates[j]);
            pos_out++;
        } else {
            //in - out
            if (out_bounds == 0)
                cambio++;
            out_bounds = 1;
            if (fails < fails_total) {
                json_coordinates.push(coordinates[j]);
                fails++;
            }
        }
    }
}

```

Figura 62 – Código función drawRoute 2

- 1.3. Si no han habido cambios significa que al ruta se ha mantenido dentro del tile, si el valor de la variable cambio es 1 significa que ha salido una sola vez, por lo tanto, acortamos las coordenadas hasta la última posición antes de salir.
- 1.4. Invierto las coordenadas para que sean reconocidas por *Mapbox* para crear la ruta.
- 1.5. 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 para poder ser pasado por *url* con la función *encodeURIComponent*.
- 1.6. Creo el enlace con el cual luego descargaremos la imagen correspondiente para cada tile.
- 1.7. 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*, al realizar cada llamada algunas texturas no se creaban y a la hora de unir todas las texturas quedaban espacios.

```

if (json_coordinates.length > 0) {
    bounds = [[rectangle_tiles[i].bounds[0][0], rectangle_tiles[i].bounds[0][1]], [rectangle_tiles[i].bounds[1][0], rectangle_tiles[i].bounds[1][1]]];
    L.rectangle(bounds, {color: "#0C14F7", weight: 2, fillOpacity: 0}).addTo(map);
    map.setMaxBounds(bounds);
    for(j = 0; j < pos_out; j++)
        short_coordinates.push(json_coordinates[j]);
    if (cambio == 1)
        json_coordinates = short_coordinates;
    for(j=0; j < json_coordinates.length; j++)
        reverse_line_points.push([json_coordinates[j][1], json_coordinates[j][0]]);
    while (reverse_line_points.length > 120)
        reverse_line_points = fixCoordinates(reverse_line_points);
    coordinate_after = 0;
    coordinate_before = 0;
    var geo_json = { "type": "Feature", "properties": { "stroke": "#ff0000", "stroke-width": 20 },
                    "geometry": { "type": "LineString", "coordinates": reverse_line_points} };
    encode_json = JSON.stringify(geo_json);
    encode_json_uri = encodeURIComponent(encode_json);
    if (id_map == -1)
        static_image_json = 'https://api.tiles.mapbox.com/v4/' + id_maps[id_maps.length-1] + '/geojson(' + encode_json_uri + ')' + map.getCenter().lng + ',' + map.getCenter().lat + ',' +
                           + zoom_map + '/' + width + 'x' + height + '.png?access_token=' + L.mapbox.accessToken;
    else
        static_image_json = 'https://api.tiles.mapbox.com/v4/' + id_maps[id_map] + '/geojson(' + encode_json_uri + ')' + map.getCenter().lng + ',' + map.getCenter().lat + ',' +
                           + zoom_map + '/' + width + 'x' + height + '.png?access_token=' + L.mapbox.accessToken;
    console.log(i + " - " + static_image_json);
    url.push(new Request(static_image_json, file_name+i));
}
} else {

```

Figura 63 – Código función drawRoute 3

2. Compruebo que el tile que no corresponde a la ruta tenga o no coordenadas. Durante las pruebas encontré tiles de relleno que contienen coordenadas de la ruta.

 - 2.1. Realizo los mismos pasos mencionados anteriormente en el punto 1 por si el tile tiene coordenadas.

3. Creo la imagen del tile sin coordenadas y la almaceno.

```

//Without coordinates
bounds = [[rectangle_tiles[i].bounds[0][0], rectangle_tiles[i].bounds[0][1]], [rectangle_tiles[i].bounds[1][0], rectangle_tiles[i].bounds[1][1]]];
L.rectangle(bounds, {color: "#0C14F7", weight: 2, fillOpacity: 0}).addTo(map);
map.setMaxBounds(bounds);
if (id_map == -1)
    static_image_json = 'https://api.tiles.mapbox.com/v4/' + id_maps[id_maps.length-1] + '/'+ map.getCenter().lng + ',' + map.getCenter().lat + ',' +
                       + zoom_map + '/' + width + 'x' + height + '.png?access_token=' + L.mapbox.accessToken;
else
    static_image_json = 'https://api.tiles.mapbox.com/v4/' + id_maps[id_map] + '/'+ map.getCenter().lng + ',' + map.getCenter().lat + ',' +
                       + zoom_map + '/' + width + 'x' + height + '.png?access_token=' + L.mapbox.accessToken;
url.push(new Request(static_image_json, file_name+i));

```

Figura 64 – Código función drawRoute 4

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 definido para todos los lugares por igual dando errores dependiendo de la zona, para que no suceda eso se usó el algoritmo de *Proyección de Mercator* que hace una relación entre el ancho de la imagen junto con la diferencia de latitud del tile. Cuando el servidor termine de crear las texturas se cargará la página *PFCMyMesh.html* donde aparecerá el modelado en 3D con *threejs*.

Asumiendo que la tierra tiene forma esférica (elipsoide) se busca transformar del sistema longitud-latitud al sistema cartesiano, ese es el principio de la Proyección de Mercator.

```

var rest = (180/Math.PI *Math.log(Math.tan(Math.PI/4 + (rectangle_tiles[0].bounds[0][0]) *(Math.PI/180)/2)))
- (180/Math.PI *Math.log(Math.tan(Math.PI/4 + (rectangle_tiles[0].bounds[1][0]) *(Math.PI/180)/2)));
var width = 510, height = Math.round(width * rest * 100 - 75);

```

Figura 65 – Transformación Proyección Mercator

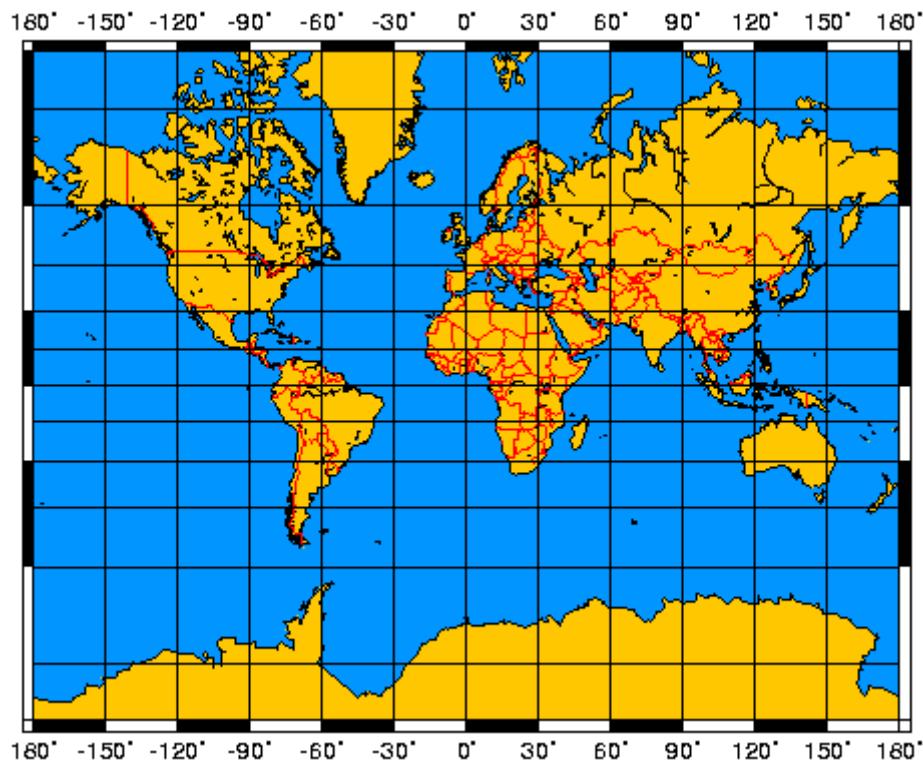


Figura 66 – Mapa de la tierra Proyección Mercator

Creación de Mesh

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*, */js/my_cesium_mesh.js*, */js/clases.js*, */js/functions.js* y */js/Cesium.js* este último es el fichero ofrecido por Cesium.

Creamos las variables locales que usaremos y esperamos hasta lograr la conexión a *Cesium*.

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*.

```
function init() {  
  
    var container = document.createElement('div');  
    document.body.appendChild(container);  
    stats = initialStats();  
    //----- Creación de la escena que contendrá los objetos, cámaras, luces.  
    scene = new THREE.Scene();  
    //----- Creamos la cámara  
    camera = new THREE.PerspectiveCamera(45, window.innerWidth / window.innerHeight, 0.1, 1000);  
    //----- Creamos el render y definimos el tamaño.  
    renderer = new THREE.WebGLRenderer();  
    renderer.setClearColor(new THREE.Color(0xEEEEEE, 1.0));  
    renderer.setSize(window.innerWidth, window.innerHeight);  
    renderer.shadowMapEnabled = true; //Sombra en el renderizado  
    //----- posición y punto de la cámara para centrar la escena  
    camera.position.set(0,100,200);  
    //camera.lookAt(scene.position);  
    controlCamera = new THREE.OrbitControls(camera);  
    //controlCamera.noPan = true;  
    container.appendChild(renderer.domElement);  
}  
  
function render() {  
    controlCamera.update();  
    stats.update();  
    requestAnimationFrame(render);  
    renderer.render(scene, camera);  
}
```

Figura 67 – Código crear escena Threejs

Información tiles

Realizaremos los mismos pasos del apartado 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)`. Esta 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 de la función `requestTileGoemetry()`

```
▶ _boundingSphere: h
▶ _childTileMask: 0
▶ _createdByUpsampling: false
▶ _eastIndices: Uint16Array[15]
▶ _eastSkirtHeight: 23.519085626208074
▶ _encodedNormals: undefined
▶ _heightValues: Uint16Array[177]
▶ _horizonOcclusionPoint: o
▶ _indices: Uint16Array[909]
▶ _maximumHeight: 502.2276306152344
▶ _minimumHeight: 474.3674621582031
▶ _northIndices: Uint16Array[11]
▶ _northSkirtHeight: 23.519085626208074
▶ _quantizedVertices: Uint16Array[531]
▶ _southIndices: Uint16Array[12]
▶ _southSkirtHeight: 23.519085626208074
▶ _uValues: Uint16Array[177]
▶ _vValues: Uint16Array[177]
▶ _waterMask: undefined
▶ _westIndices: Uint16Array[15]
▶ _westSkirtHeight: 23.519085626208074
```

Figura 68 – Retorno de valores 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`: especifica 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.

Los tres primeros arreglos son almacenados conjuntamente en el arreglo `_quantizedVertices` agrupados en vector de 3 elementos. 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:

1. Creamos una geometría con `threejs`.
2. 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.
3. Creamos las caras de la geometría usando la variable `_indices` obteniendo el terreno.
4. Almacenaremos la geometría creada en el atributo `geometry` del arreglo `rectangle_tiles`.

Por cada iteración del bucle llamaré a la función `associateGeometry(data, 1000)`, la cual recibe como parámetros de entrada toda la información de *Cesium* y un valor para escalar los datos. Tendremos una variable global `index_tile` la cual hace referencia a que tile estamos tratando en la función `associateGeometry()`, se hace todo esto ya que el resultado de la función `requestTileGeometry()` es asíncrono.

```
function associateGeometry(data, scale) {
  var mesh, facesQuantized, geometry;
  var xx = data._uValues,
    yy = data._vValues,
    heights = data._heightValues;
  facesQuantized = data._indices;
  var geometry = new THREE.Geometry();
  var new_height;
  for(var i=0; i < heights.length; i++) {
    new_height = (((data._maximumHeight - data._minimumHeight) * heights[i]) / 32767) + data._minimumHeight;
    geometry.vertices.push( new THREE.Vector3(Math.round(xx[i]/scale),Math.round(yy[i]/scale), new_height/30));
  }
  for(var i=0; i < facesQuantized.length; i=i+3)
    geometry.faces.push(new THREE.Face3(facesQuantized[i], facesQuantized[i+1], facesQuantized[i+2]));
  rectangle_tiles[index_tile].geometry = geometry;
}
```

Figura 69 – Código función `associateGeometry`

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.

```

function load(coord) {
    /*
        info_tiles contains all information about tiles of the route gpx.
        Each element of the array is a InfoTile element.
    */
    //Create scene ThreeJs with threejs.js
    loadThreeJS();
    //Create Graphic User Interface with gui.js
    createGUI();
    var level = 14;
    info_tiles = new Array();
    checkTile(coord, level);
    rectangle_tiles = createRectangle(info_tiles, level);
    var promise = [];
    for(i = 0; i < rectangle_tiles.length; i++) {
        requestPromise = aCesiumTerrainProvider.requestTileGeometry(rectangle_tiles[i].x, rectangle_tiles[i].y, level, false).then(function(data) {
            associateGeometry(data, 1000);
            index_tile++;
        });
        var tilePromise = Cesium.when(requestPromise);
        promise.push(tilePromise);
    }
    Cesium.when.all(promise, function() {
        createTerrain();
    });
}

}

```

Figura 70 – Código función load

Creación del terreno

Ahora debemos unir todas las geometrías para obtener una única geometría que será nuestra ruta recorrida. La función que se encargará de crear el terreno es *createTerrain()*.

```

function createTerrain() {
    var i, j;
    console.log('[PFC my_ceosium_mesh.js]: Geometries created in rectangle_tiles');
    var rectangle = maxMinTileXY();
    var x = 0, y = 0, z = 0;
    var geometry, material;
    for(i = rectangle[0][0]; i <= rectangle[1][0]; i++) {
        for(j = rectangle[1][1]; j >= rectangle[0][1]; j--) {
            material = new THREE.MeshBasicMaterial( { color: "rgb(255,0,0)", wireframe: true ,side:THREE.DoubleSide} );
            geometry = getGeometry(i,j);
            addGeometryScene(geometry, x, y, z, material);
            y = y + 33;
        }
        y = 0;
        x = x + 33;
    }
    showCombinedGeometry();
}

```

Figura 71 – Código función createTerrain

createTerrain() creará el terreno realizando las siguientes acciones:

1. Obtener el máximo y menor tile con la función *maxMinTileXY()* para poder unir cada geometría de un tile con su vecina y recorrer todos los tiles de forma ordenada.

```
function maxMinTileXY() {
    var min_x = rectangle_tiles[0].x, max_x = rectangle_tiles[0].x, min_y = rectangle_tiles[0].y, max_y = rectangle_tiles[0].y;
    for(i = 1; i < rectangle_tiles.length; i++) {
        if (rectangle_tiles[i].x < min_x)
            min_x = rectangle_tiles[i].x;
        if (rectangle_tiles[i].x > max_x)
            max_x = rectangle_tiles[i].x;
        if (rectangle_tiles[i].y < min_y)
            min_y = rectangle_tiles[i].y;
        if (rectangle_tiles[i].y > max_y)
            max_y = rectangle_tiles[i].y;
    }
    return [[min_x, min_y], [max_x, max_y]];
}
```

Figura 72 – Código función manMinTileXY

2. Con los valores obtenidos de la función *maxMinTileXY()* crearemos un bucle que recorra todos los tiles que pertenecen a la ruta.
3. 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. Para ello usaremos la función *merge()* de *threejs* que une mallas, una malla es la unión de una geometría con una textura que en este caso será una textura básica ya que al final añadiremos la textura de la ruta completa.

```
function getGeometry(x, y) {
    var geometry;
    for (i = 0; i < rectangle_tiles.length; i++) {
        if ((rectangle_tiles[i].x == x) && (rectangle_tiles[i].y == y)) {
            geometry = rectangle_tiles[i].geometry;
            break;
        }
    }
    return geometry;
}
```

Figura 73 – Código función getGeometry

```
function addGeometryScene(geometry, x, y, z, material) {
    mesh = new THREE.Mesh( geometry, material );
    mesh.position.set(x, y, z);
    mesh.updateMatrix();
    combined_geometry.merge(mesh.geometry, mesh.matrix);
}
```

Figura 74 – Código función addGeometryScene

4. Actualizaremos los valores de las variables por cada iteración del bucle.
5. 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()* :

1. 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.
2. 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 de *Proyección de Mercator*.
3. Cuando la imagen de toda la ruta ha sido creada el servidor nos devuelve su ubicación en el servidor.
4. Creamos la textura para nuestra geometría final.
5. Como es una geometría creada desde cero debemos indicar como la textura debe implantarse en cada cara, para ello usamos la función *addFaceVertexUvs()*.

```
function addFaceVertexUvs(geometry) {  
    geometry.computeBoundingBox();  
    var max = geometry.boundingBox.max,  
        min = geometry.boundingBox.min;  
    var offset = new THREE.Vector2(0 - min.x, 0 - min.y);  
    var range = new THREE.Vector2(max.x - min.x, max.y - min.y);  
    geometry.faceVertexUvs[0] = [];  
    for (i = 0; i < geometry.faces.length ; i++) {  
        var v1 = geometry.vertices[geometry.faces[i].a], v2 = geometry.vertices[geometry.faces[i].b], v3 = geometry.vertices[geometry.faces[i].c];  
        geometry.faceVertexUvs[0].push(  
            [  
                new THREE.Vector2((v1.x + offset.x)/range.x ,(v1.y + offset.y)/range.y),  
                new THREE.Vector2((v2.x + offset.x)/range.x ,(v2.y + offset.y)/range.y),  
                new THREE.Vector2((v3.x + offset.x)/range.x ,(v3.y + offset.y)/range.y)  
            ]);  
    }  
    geometry.uvsNeedUpdate = true;  
    return geometry;  
}
```

Figura 75 – Código función addFaceVertexUvs

6. Para crear el volumen del terreno debemos añadir una base, para ello usamos la función *addBase()*.

```
function addBase(geometry){
    //In geometry2 have the old data of geometry.
    var geometry2 = geometry.clone();
    geometry2.computeBoundingBox();
    for(var i = 0; i < geometry2.vertices.length; i++) {
        geometry.vertices.push(new THREE.Vector3(geometry2.vertices[i].x, geometry2.vertices[i].y, geometry2.boundingBox.min.z - 10));
    }
    var southVertices = new Array(), northVertices = new Array(), eastVertices = new Array(), westVertices = new Array(), i;
    //Save the index of the limits(west, east, north, south).
    geometry2.computeBoundingBox();
    // Vertices of mesh.
    for(i = 0; i < geometry2.vertices.length; i++) {
        if (geometry.vertices[i].y === geometry2.boundingBox.min.y)
            southVertices.push(i);
        else if (geometry.vertices[i].x === geometry2.boundingBox.min.x)
            westVertices.push(i);
        else if (geometry.vertices[i].x === geometry2.boundingBox.max.x)
            eastVertices.push(i);
        else if (geometry.vertices[i].y === geometry2.boundingBox.max.y)
            northVertices.push(i);
    }
    //displace, value to fin her vector.-slue in the base.
    var displace = geometry2.vertices.length;
    //Sort vertices to draw faces correctly.
    southVertices = sortVector(geometry2, southVertices, "x");
    westVertices = sortVector(geometry2, westVertices, "y");
    eastVertices = sortVector(geometry2, eastVertices, "y");
    northVertices = sortVector(geometry2, northVertices, "x");
    //Add the faces.
    addRelieve(geometry, southVertices, displace);
    addRelieve(geometry, westVertices, displace);
    addRelieve(geometry, eastVertices, displace);
    addRelieve(geometry, northVertices, displace);
    /*
     * West and east miss a point and north two point.
    */
}
```

Figura 76 – Código función addBase 1

```
if (!isNaN(southVertices[0]+displace)){
    //West-south
    geometry.faces.push(new THREE.Face3(westVertices[0], westVertices[0]+displace, southVertices[0]+displace));
    geometry.faces.push(new THREE.Face3(southVertices[0], westVertices[0], southVertices[0]+displace));
    //East-south
    geometry.faces.push(new THREE.Face3(eastVertices[0], eastVertices[0]+displace, southVertices[southVertices.length-1]+displace));
    geometry.faces.push(new THREE.Face3(southVertices[southVertices.length-1], eastVertices[0], southVertices[southVertices.length-1]+displace));
}
if (!isNaN(northVertices[0]+displace)){
    //North-West
    geometry.faces.push(new THREE.Face3(westVertices[westVertices.length-1], westVertices[westVertices.length-1]+displace, northVertices[0]+displace));
    geometry.faces.push(new THREE.Face3(northVertices[0], westVertices[westVertices.length-1], northVertices[0]+displace));
    //North-East
    geometry.faces.push(new THREE.Face3(eastVertices[eastVertices.length-1], eastVertices[eastVertices.length-1]+displace, northVertices[northVertices.length-1]+displace));
    geometry.faces.push(new THREE.Face3(northVertices[northVertices.length-1], eastVertices[eastVertices.length-1], northVertices[northVertices.length-1]+displace));
}
//Add Base geometry
geometry.faces.push(new THREE.Face3(southVertices[0]+displace, eastVertices[eastVertices.length-1]+displace, southVertices[southVertices.length-1]+displace));
geometry.faces.push(new THREE.Face3(southVertices[0]+displace, westVertices[westVertices.length-1]+displace, eastVertices[eastVertices.length-1]+displace));

return geometry;
}
```

Figura 77 – Código función addBase 2

7. Creamos el material indicando la textura que usará y que la textura aparezca en ambos lados de la geometría.
8. Creamos la malla que es la unión de la geometría y el material.
9. Especificamos su ubicación en la escena. Esta ubicación debe ser dinámica siempre para que el usuario pueda interaccionar correctamente con el terreno.
10. Rotamos la malla, ya que su creación ha sido modificando todos los vértices.
11. Añadimos la malla a la escena.

Código completo de la función *ShowCombinedGeometry*.

```

function showCombinedGeometry() {
    $("#dialog-geometry").dialog( "open" );
    var rectangle = maxMinTileXY();
    var columns = 0, rows = 0;
    for(var i = rectangle[0][0]; i <= rectangle[1][0]; i++) {
        columns++;
    }
    for(var j = rectangle[1][1]; j >= rectangle[0][1]; j--) {
        rows++;
    }
    var name = sessionStorage.rute.substring(sessionStorage.rute.indexOf("/") + 1, sessionStorage.rute.length);
    var file_name = name.substring(0, name.indexOf("."));
    var xhr = new XMLHttpRequest();
    var url = "createImage.php";
    var rest = (180/Math.PI *Math.log(Math.tan(Math.PI/4 + (rectangle_tiles[0].bounds[0][0]) *(Math.PI/180)/2))) -
    (180/Math.PI *Math.log(Math.tan(Math.PI/4 + (rectangle_tiles[0].bounds[1][0]) *(Math.PI/180)/2)));
    rest = Math.round(rest * 100 * 510);
    var contenido = "rows="+rows+"&columns="+columns+"&name="+file_name+"&rest="+rest;
    xhr.open("GET", url+"?"+contenido, true);
    xhr.send();
    xhr.onreadystatechange = function () {
        if (xhr.readyState == 4 && xhr.status == 200) {
            console.log("[PFC my_ceesium_mesh.js] Texture create in " + xhr.responseText);
            var texture, material;
            texture = THREE.ImageUtils.loadTexture(xhr.responseText);
            combined_geometry = addFaceVertexUvs(combined_geometry);
            combined_geometry = addBase(combined_geometry);
            material = new THREE.MeshBasicMaterial( { map: texture, wireframe: true, side:THREE.DoubleSide} );
            mesh = new THREE.Mesh(combined_geometry, material);
            mesh.position.set(-combined_geometry.boundingBox.max.x + (combined_geometry.boundingBox.max.x/2), 0, 50);
            mesh.rotation.x = Math.PI / 180 * (-90);
            scene.add(mesh);
            $("#dialog-geometry").dialog( "close" );
        }
    }
}

```

Figura 78 – Código función *showCombinedGeometry*

Contenido del fichero *createImage.php*

```
<?php
    ini_set('memory_limit', '128M');
    $type;
    $finfo = finfo_open(FILEINFO_MIME_TYPE); // return mime type ala mimetype extension
    foreach ( glob('textures/*') as $filename) {
        $type = finfo_file($finfo, $filename);
    }
    finfo_close($finfo);

    $posslash = strrpos($type, '/');
    $type = substr($type, $posslash+1);

    $rows = (int) $_GET['rows'];
    $columns = (int) $_GET['columns'];
    $file = $_GET['name'];
    $width = (510 * 0.5) - 1;
    $h = (int) $_GET['rest'];
    $heighth = ( ($h - 75) * 0.5 ) - 1;

    $background_image = imagecreatetruecolor( $width * $columns, $heighth * $rows);
    $times = 0;
    $x = 0;
    $y = $heighth * ($rows - 1);
    for($i = 0; $i < $columns; $i++) {
        for($j = 0; $j < $rows; $j++) {
            $source_image = 'textures/'.$file.$times.'.jpeg';
            $image = imagecreatefromjpeg($source_image);
            $size = getimagesize($source_image);
            imagecopy($background_image, $image, $x, $y, 0, 0, $size[0], $size[1]);
            imagedestroy($image);
            $y = $y - $heighth;
            $times++;
        }
        $x = $x + $width;
        $y = $heighth * ($rows - 1);
    }
    imagejpeg($background_image,'export/'.$file.'.jpeg');
    imagedestroy($background_image);
    echo 'export/'.$file.'.jpeg';

?>
```

Figura 79 – Código *createImage.php*

Creación x3d y envío a impresora 3D

En este apartado se explicará cómo se crea el objeto y como debe ser enviado a *Shapeways*.

Creación del modelo

Para subir un objeto debemos crear un *zip* que incluya el objeto en formato *x3d* y su material en algún formato de imagen, para ello usaremos la librería *x3dExporter.js* con la cual transformamos nuestra malla y la añadimos a nuestro *x3d* de la siguiente manera:

- Debemos crear el objeto exporter.
- Aplicar la función *parse* pasando como argumento la malla que une nuestra geometría y material de la ruta.
- Ahora enviamos la información mediante *ajax* a la página *createX3D.php* que se encarga de crear el objeto *x3d*. La imagen ya la tenemos en el servidor ya que la usamos para crear el terreno en el navegador.
- El servidor envía la respuesta del servidor con el cual cargamos la página *downloadShapeways.php* que se encarga de generar el *zip* uniendo el *x3d* y la imagen el cual es descargado en nuestro ordenador.

```
this.export = function () {
    console.clear();
    var a, exporter;
    var mesh, material;
    console.log("[PFC Gui.js]: Export with THREE.X3dExporter.");
    exporter = new THREE.X3dExporter();
    if (combined_geometry) {
        console.log(combined_geometry);
        material= new THREE.MeshBasicMaterial( { side:THREE.DoubleSide} );
        mesh = new THREE.Mesh(combined_geometry, material);
        a = exporter.parse(mesh);
        var formData = new FormData();
        var name = sessionStorage.rute.substring(sessionStorage.rute.indexOf("/") + 1, sessionStorage.rute.length);
        var file_name = name.substring(0, name.indexOf("."));
        formData.append('file_name', file_name);
        formData.append('indexed_face_set', a);
        var xhr2 = new XMLHttpRequest();
        xhr2.open("POST", 'createX3D.php', true);
        xhr2.send(formData);
        xhr2.onreadystatechange = function () {
            if (xhr2.readyState == 4 && xhr2.status == 200) {
                console.log(xhr2.responseText);
                window.open('./downloadShapeways.php?file_name='+file_name, "_self");
            }
        }
    }
}
```

Figura 80 – Código función export

Fichero *createX3D.php*

```
<?php
$xml = fopen("export\\\".$_POST['file_name'].".x3d", "w") or die("Unable to open file!");
fwrite($xml, "<?xml version='1.0' encoding='UTF-8'?>");
//fwrite($xml, "<!DOCTYPE X3D PUBLIC 'ISO//Web3D//DTD X3D 3.0//EN' 'http://www.web3d.org/specifications/x3d-3.0.dtd'>");
fwrite($xml, "<X3D xmlns:xsd='http://www.w3.org/2001/XMLSchema-instance' version='3.0' profile='Immersive' xsd:noNamespaceSchemaLocation='http://www.w3.org/2001/XMLSchema-instance'>");
fwrite($xml, "<head>");
    fwrite($xml, "<meta name='filename' content='".$_POST['file_name']."' />");
    fwrite($xml, "<meta name='generator' content='3xdExport.js' />");
fwrite($xml, "</head>");
fwrite($xml, "<Scene>");
    fwrite($xml, "<Transform scale='0.4 0.4 0.4'>");
        fwrite($xml, "<Shape>");
            fwrite($xml, "<Appearance>");
                fwrite($xml, "<ImageTexture url='".$_POST['file_name'].".jpeg' />");
                fwrite($xml, "<Material diffuseColor='0 0 1' ambientIntensity='1.0' />");
            fwrite($xml, "</Appearance>");
            fwrite($xml, $_POST['indexed_face_set']);
        fwrite($xml, "</Shape>");
    fwrite($xml, "</Transform>");
fwrite($xml, "</Scene>");
fwrite($xml, "</X3D>");
fclose($xml);

$zip = new ZipArchive();
$filename = "export/".$_POST['file_name'].".zip";

if ($zip->open($filename, ZipArchive::CREATE)!==TRUE) {
    echo("cannot open <$filename>\n");
} else {
    echo "[PFC createX3D.php]: Create X3D complete.";
}
$zip->addFile("export/".$_POST['file_name'].".jpeg", $_POST['file_name'].".jpeg");
$zip->addFile("export/".$_POST['file_name'].".x3d", $_POST['file_name'].".x3d");
$zip->close();
?>
```

Figura 81 – Código *createX3D.php*

Fichero *downloadShapeways.php*

```
<?php
//Download php
$archivo = $_GET['file_name'].".zip";
$path = 'export/';
$ruta = $path.$archivo;
if (is_file($ruta)) {
    header('Content-Type: application/force-download');
    header('Content-Disposition: attachment; filename='.$archivo);
    header('Content-Transfer-Encoding: binary');
    header('Content-Length: '.filesize($ruta));
    readfile($ruta);
} else
    exit();

unlink($path.$_GET['file_name'].".zip");
unlink($path.$_GET['file_name'].".x3d");
unlink($path.$_GET['file_name'].".jpeg");
?>
```

Figura 82 – Código *dowloadShapeways.php*

Subir el modelo

Ya creado el objeto solo nos queda subir el *zip* a *Shapeways*, logeados como usuarios nos dirigimos a nuestro perfil y seleccionamos la opción *Models*.

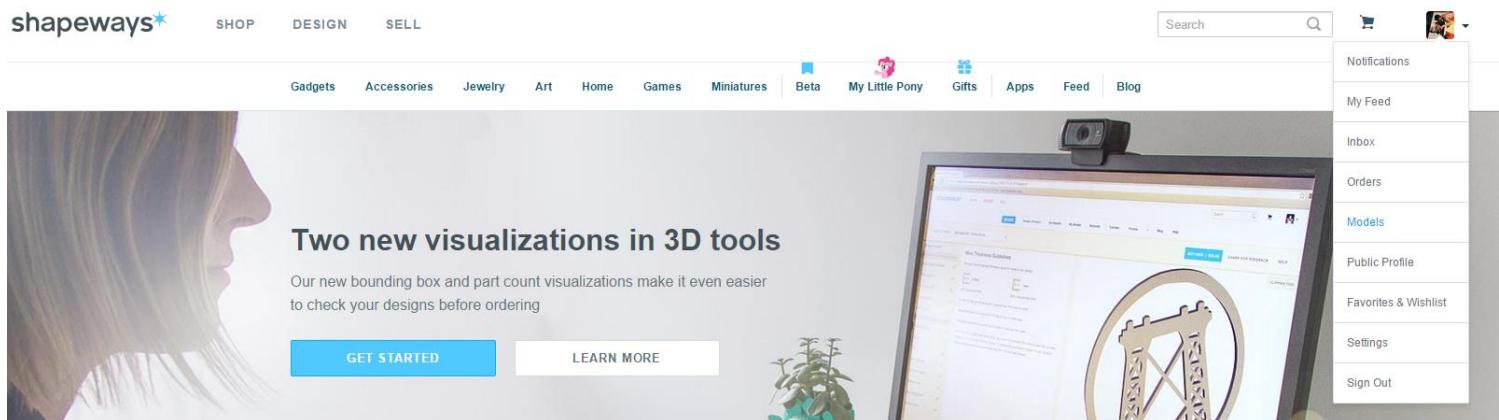


Figura 83 – Portada Shapeways

Ya en nuestros modelos en la esquina superior derecha nos encontramos con el botón *Upload* el cual nos permitirá subir nuestro modelo.

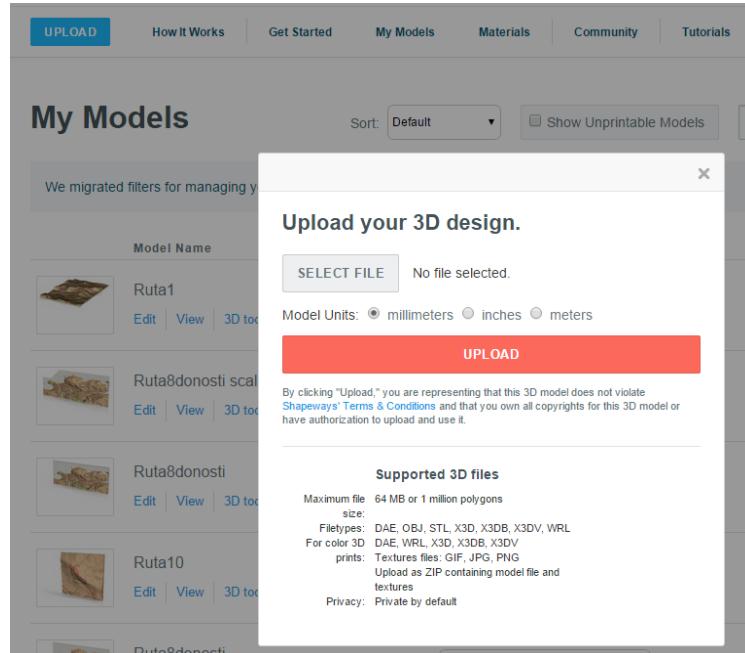


Figura 84 – Upload 3D Shapeways

Mientras el fichero es procesado *Shapeways* nos indica sus características como el nombre del fichero, el tamaño, su volumen entre otras características. Cuando el objeto ha sido cargado por completo a la derecha de la descripción aparecerá una imagen con el modelo cargado y hacia abajo todas las opciones de material, referencia a si el objeto cumple los estándares de *Shapeways*.

The screenshot shows the Shapeways website interface. At the top, there is a navigation bar with links: UPLOAD, How It Works, Get Started, My Models, Materials, Community, Tutorials, Blog, and Help. The UPLOAD button is highlighted in blue. Below the navigation bar, the page title is "Ruta10" with a "VIEW PRODUCT" button to its right. There are three tabs: Model (which is selected), Details, and Selling. Under the Model tab, there is a preview image of the model labeled "Ruta10". To the left of the image, there are technical specifications: File: ruta10.x3d, Size: Cm: 3.96 w x 3.96 d x 1.004 h, In: 1.559 w x 1.559 d x 0.395 h, Part Count: 1, Material Volume: 9.6341cm³, and Machine Space: 13.8348cm³. Below the preview image, there are three more tabs: Materials, Details, and History. The Materials tab is selected, showing a table for "Strong & Flexible Plastic". The table has columns for Material Finish (White), Auto Checks (Loading), Manual Checks (Pass), Success Rate (Pass), Price (Pass), and Qty (Pass). An "ADD TO CART" button is located at the bottom right of the table.

Figura 85 – Modelo subido

Shapeways nos ofrece una herramienta para comprobar si los modelos cumplen los requisitos mínimos para ser impresos, esta herramienta es *View 3D tolos*.



Figura 86 – Modelo verificado

Haciendo clic en el enlace cargaremos la página que nos indicará si el modelo cumple los requisitos y si no los cumple nos indicará que debemos arreglar. En la imagen anterior podemos ver que para el material *White* nuestro modelo ha cumplido todos los requisitos.

Welcome to 3D tools

Discover how your 3D model is automatically and manually checked at Shapeways. Open checks and learn potential problems that may affect the 3D printing of your product.

When you buy your product, all checks are subject to review by a Shapeways 3D Print Engineer.

Strong & Flexible Overview

See full material information for Strong & Flexible Plastic [here](#).

How it's 3D printed

To print in this material, we start with a bed of nylon powder and sinter the powder with a laser layer by layer, solidifying the powder as we go. Because of this layer by layer process, some products may see a staircase effect. How much you see this effect depends on how your model is oriented in the print tray. Our production planners are working hard to orient models optimally to ensure efficient and good lookin' trays.

Look and Feel

This material is incredibly versatile, and can be used for a wide variety of applications, from iPhone cases to jewelry, remote controlled quadcopters to wearable bikinis. When thin, it's flexible enough for hinges and springs. When thick, it's strong enough for structural components.

Handling and Care

CLIPPING TOOL

Figura 87 – Comprobación modelo

Seleccionando la opción *Clipping Tool* podemos recortar nuestro modelo y ver posibles fallos.

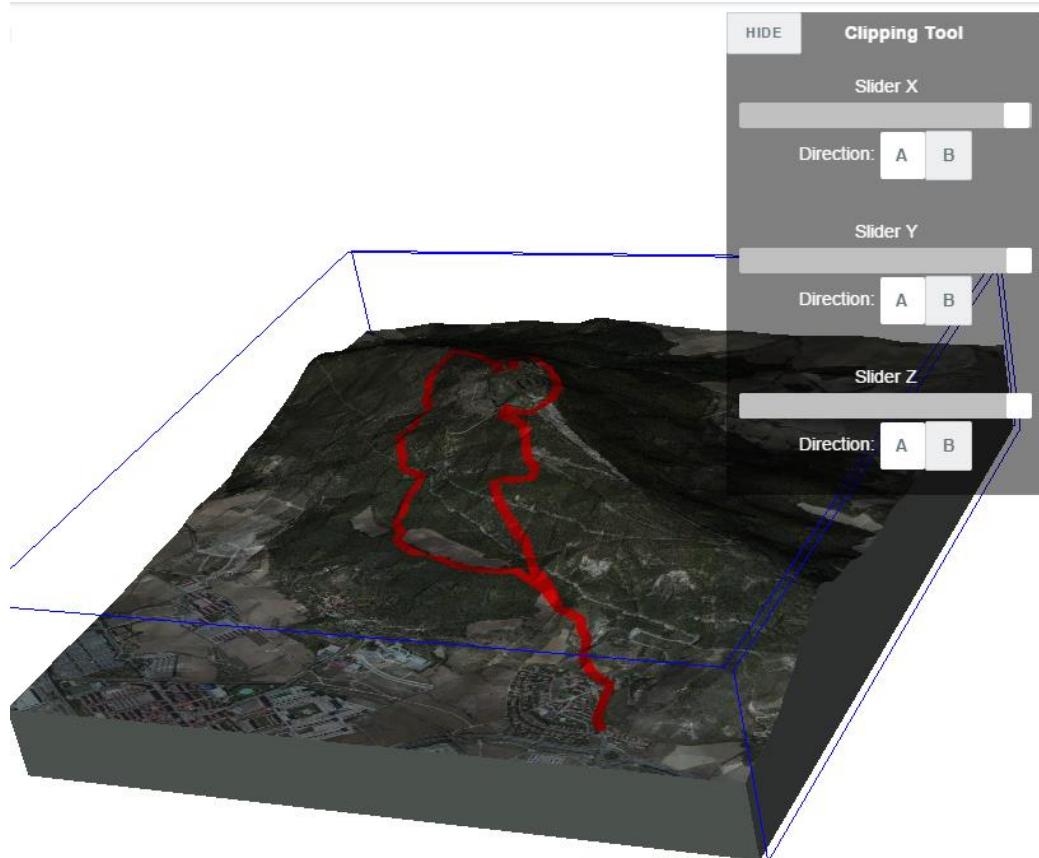


Figura 88 – Edición modelo

Resultados

En este apartado se mostrarán los resultados gráficos en el navegador, resultados físicos tras su impresión en *Shapeways* y los problemas junto con las soluciones que tuve durante el desarrollo del proyecto.



Figura 89 – Ruta

Aplicación Web

A partir de la ruta seleccionada se van creando las geometrías correspondientes de cada tile hasta completar todo el recorrido.

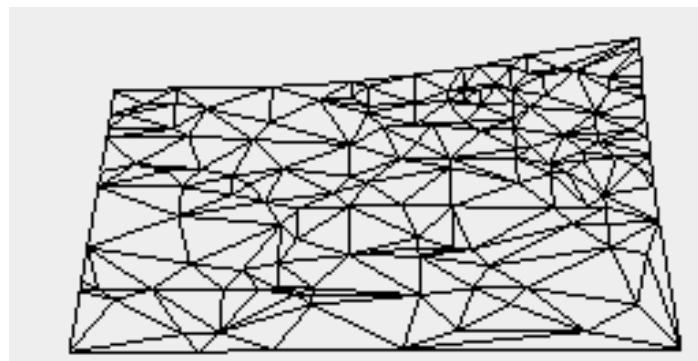


Figura 90 – Geometría 1 tile

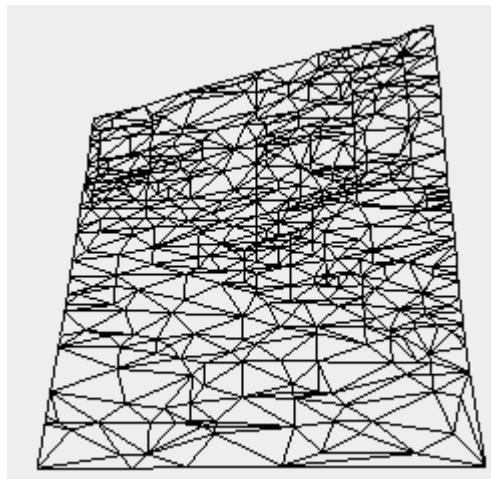


Figura 91 – Geometría 2 tiles

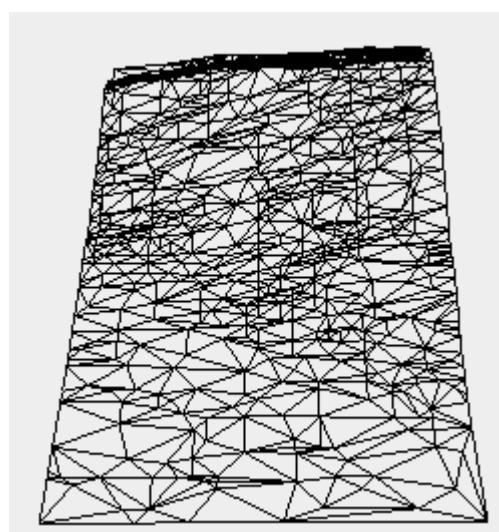


Figura 92 – Geometría 3 tiles

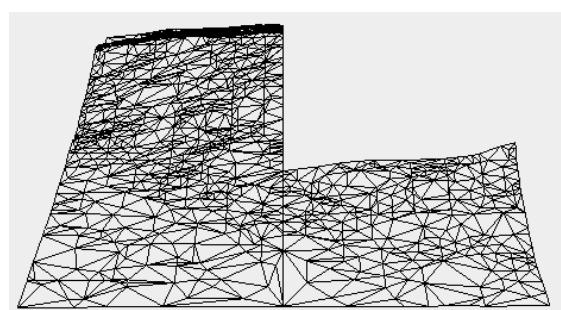


Figura 93 – Geometría 4 tiles

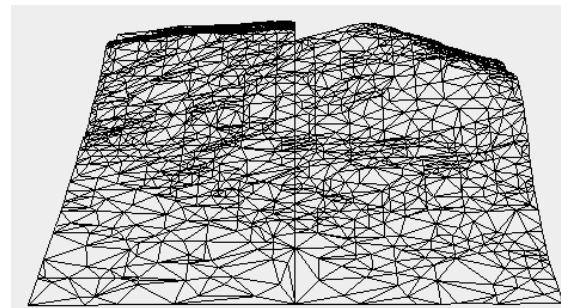


Figura 94 – Geometría 5 tiles

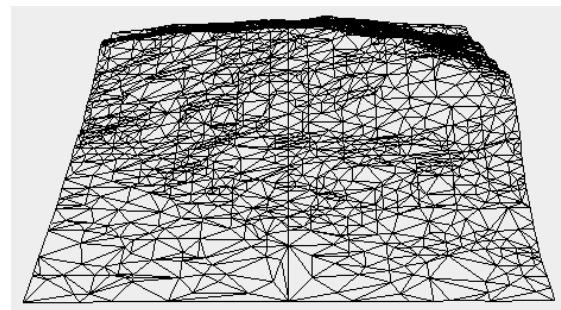


Figura 95 – Geometría 6 tiles

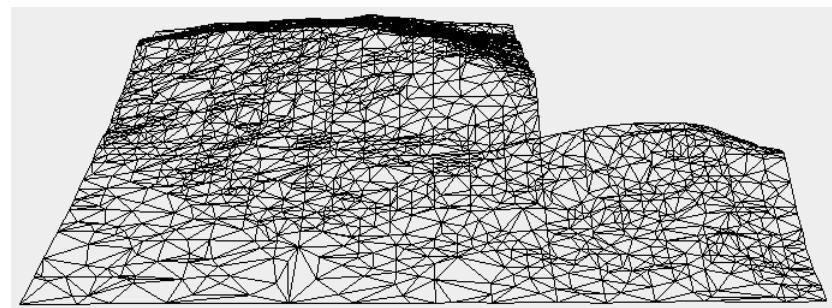


Figura 96 – Geometría 7 tiles

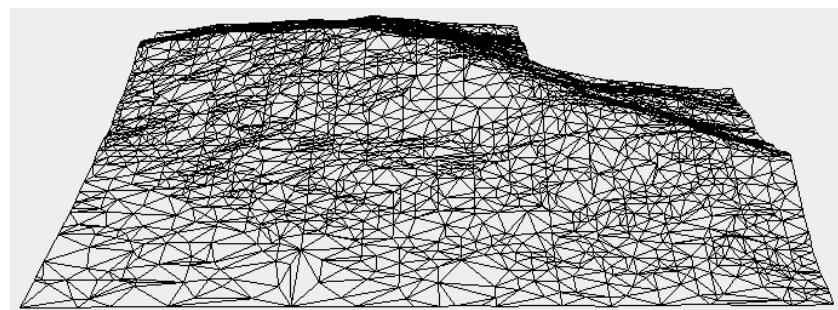


Figura 97 – Geometría 8 tiles

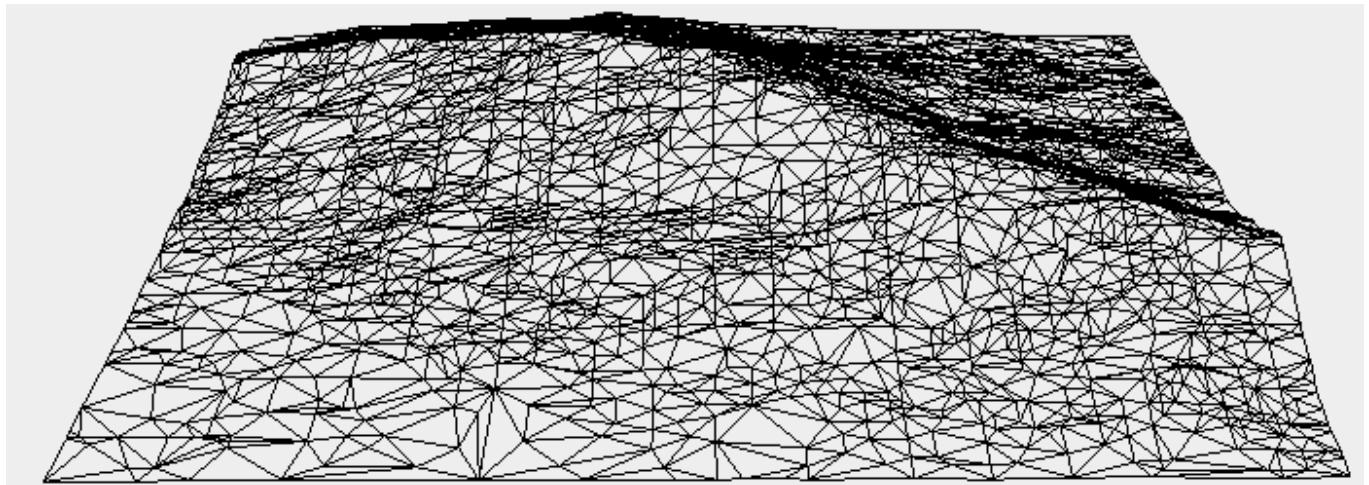


Figura 98 – Geometría 9 tiles

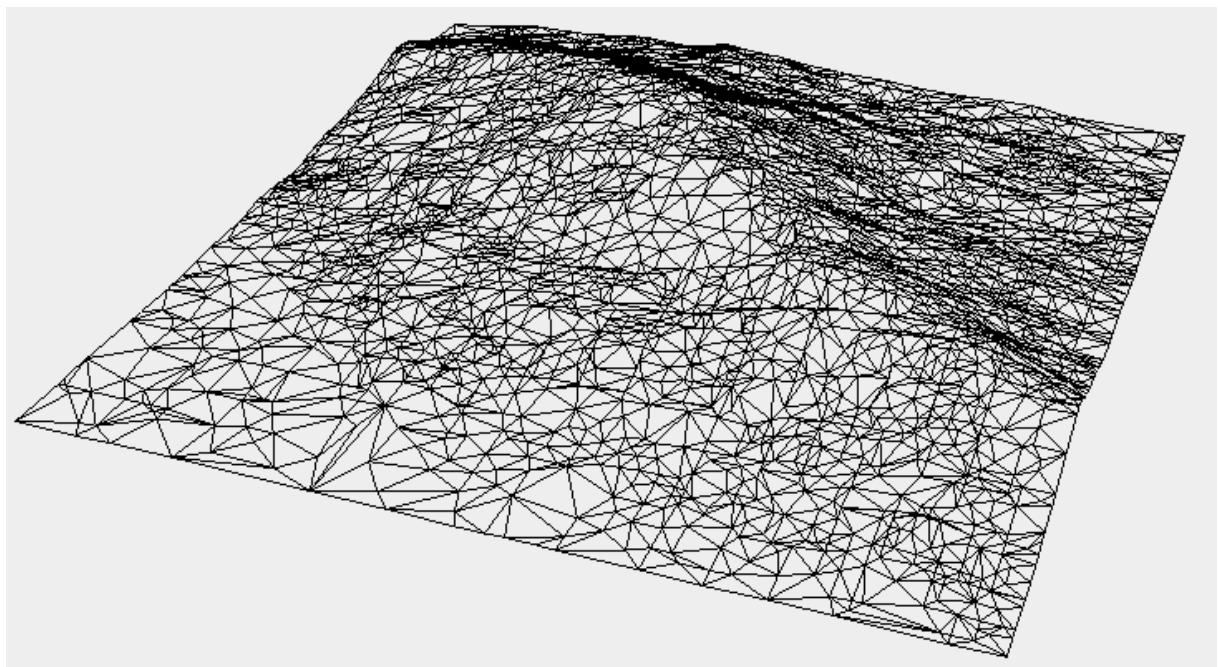


Figura 99 – Geometría completa

Una vez creada la geometría añadimos los laterales y la base para obtener una geometría completa.

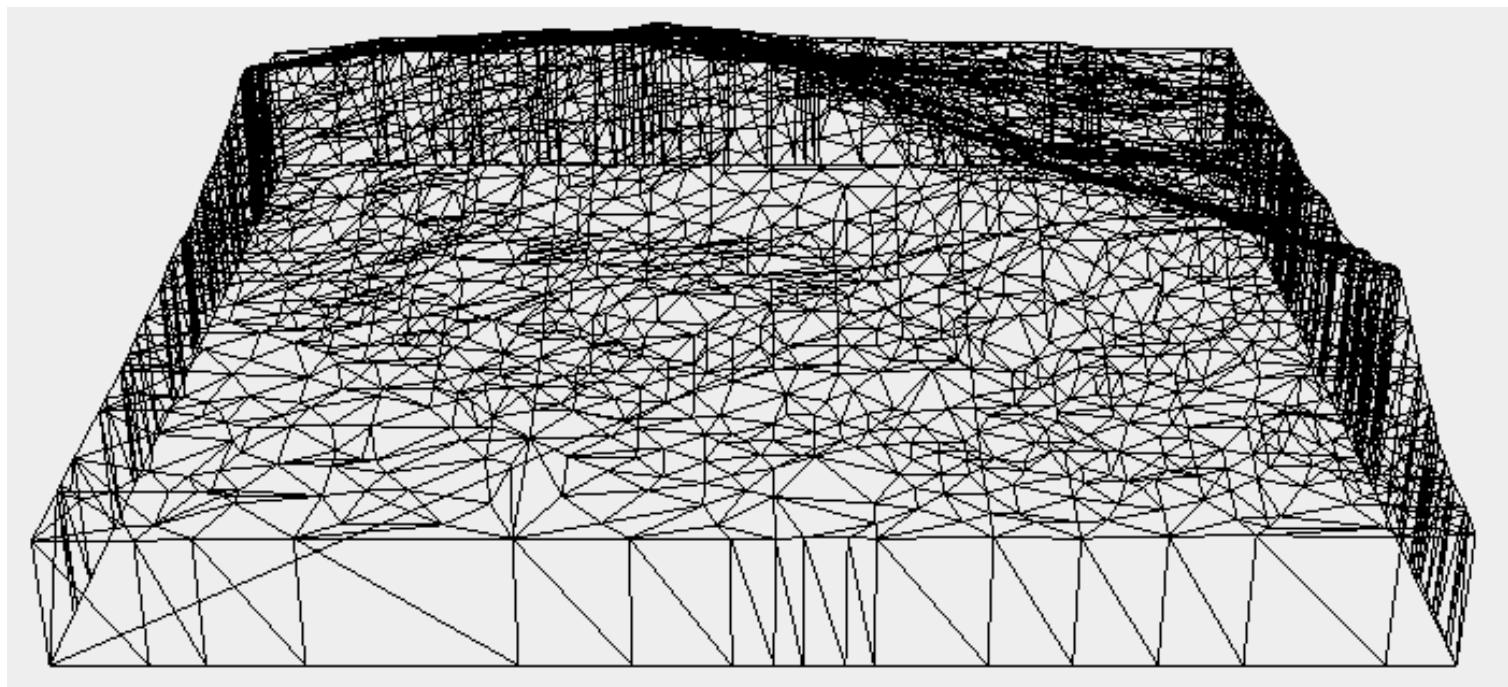


Figura 100 – Geometría completa con base y laterales

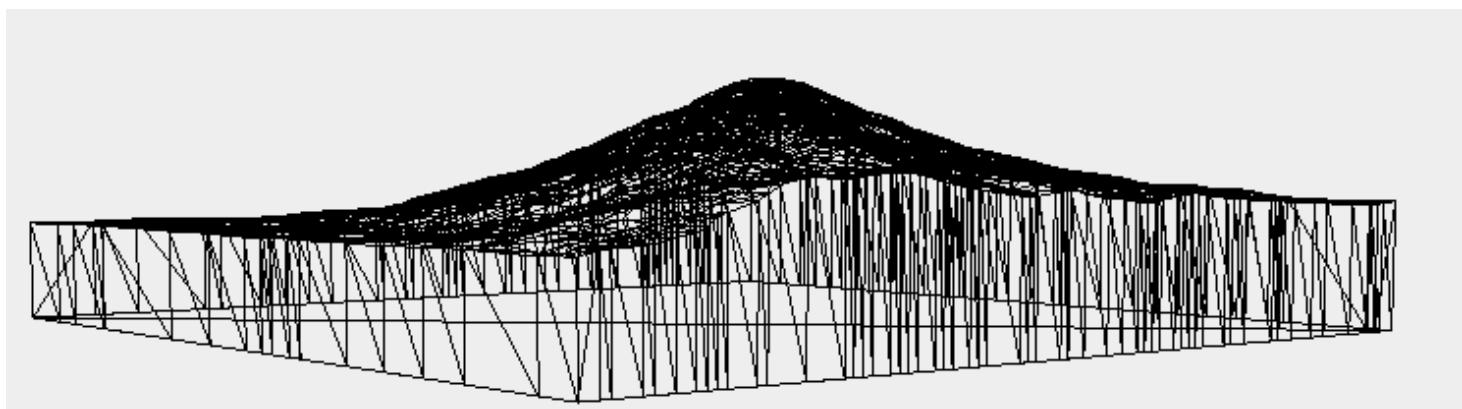


Figura 101 – Geometría completa con base y laterales

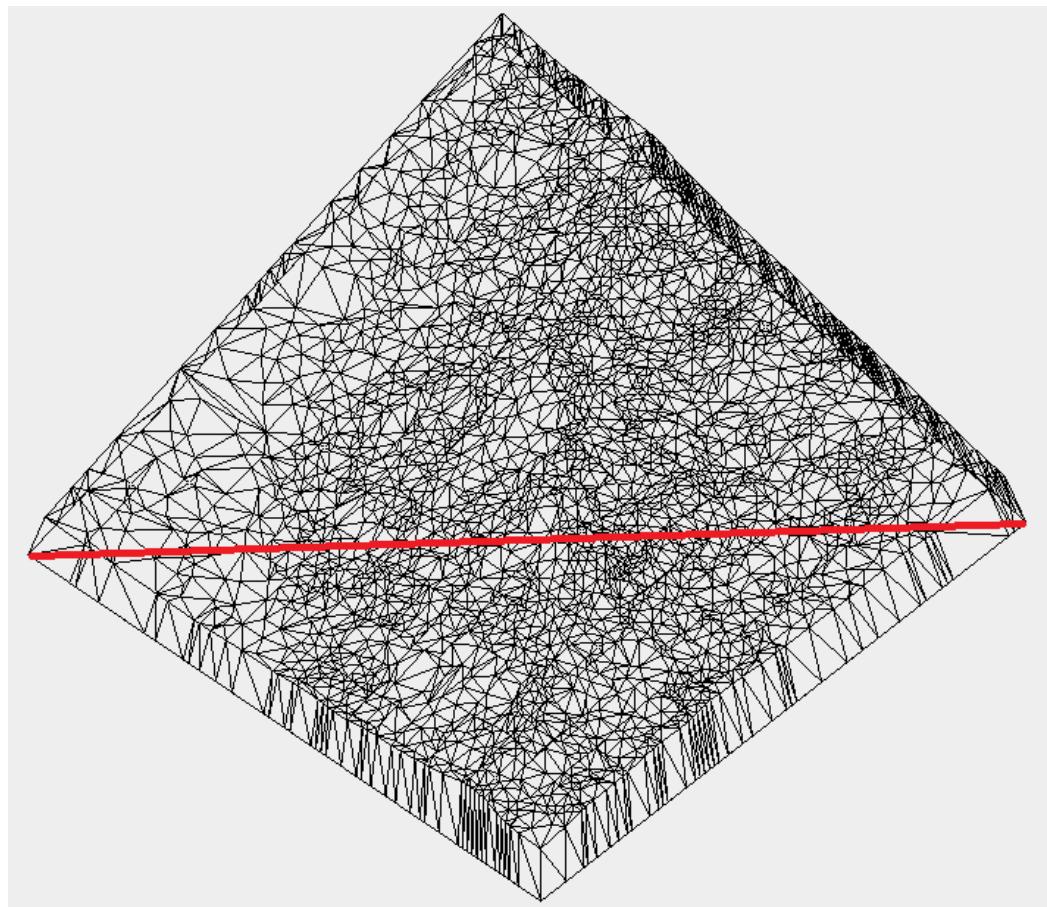


Figura 102 – Geometría completa con base y laterales

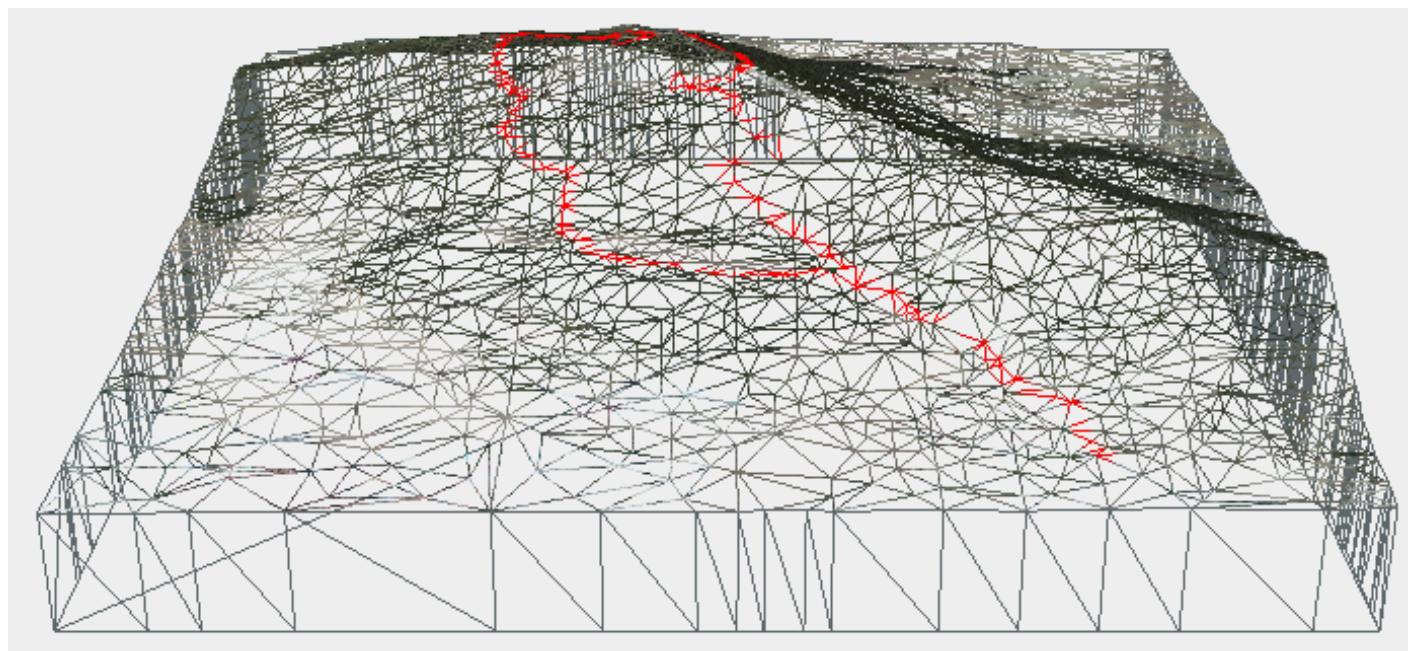


Figura 103 – Geometría completa con base y laterales.

Añadimos la textura a la geometría obteniendo el modelado final.



Figura 104 – Geometría completa con textura



Figura 105 – Geometría completa con textura

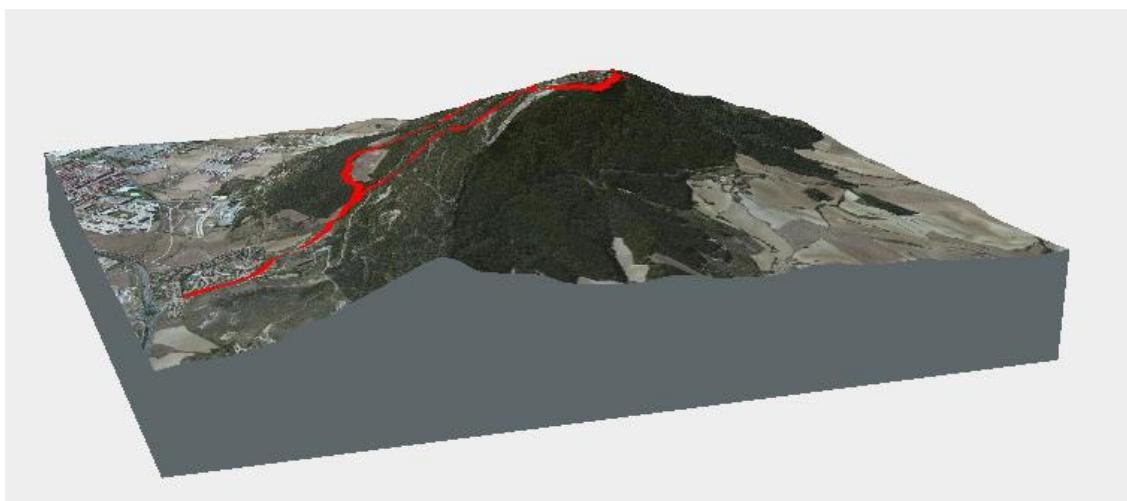


Figura 106 – Geometría completa con textura

Impresiones 3D

En las siguientes imágenes podemos ver los resultados de los objetos enviados a *Shapeways*.



Figura 107 – Resultado en shapeways

USA 3D

File	otros1.x3d
Size	Cm: 5.94 w x 4.95 d x 1.21 h In: 2.339 w x 1.949 d x 0.476 h
Part Count	1
Material Volume	17.4867cm ³
Machine Space	25.4044cm ³

[UPDATE FILE](#) By uploading a new version of this product, your success rate will be reset. [Why?](#)

Materials Details History

Strong & Flexible Plastic

Material Finish	Auto Checks	Manual Checks	Success Rate	Price	Qty.
White View 3D tools	Passed	—	—	€13.19	<input type="button" value="1"/> ADD TO CART
Black View 3D tools	Passed	—	—	€14.31	<input type="button" value="1"/> ADD TO CART

Figura 108 – Resultado en shapeways



Figura 109 – Resultado en shapeways

Donosti 3D

File ruta8donosti.x3d

Size Cm: 8.884 w x 3.808 d x 0.712 h
In: 3.498 w x 1.499 d x 0.28 h

Part Count 1

Material Volume 16.0916cm³

Machine Space 24.8067cm³

UPDATE FILE By uploading a new version of this product, your success rate will be reset. [Why?](#)

Materials Details History

Strong & Flexible Plastic

Material Finish	Auto Checks	Manual Checks	Success Rate	Price	Qty.	Add To Cart
White View 3D tools	Passed	—	—	€12.62	1	ADD TO CART
Black View 3D tools	Passed	—	—	€13.75	1	ADD TO CART

Figura 110 – Resultado en shapeways

Modelos físicos impreso por *Shapeways*, la primera imagen pertenece a una ruta en los Estados Unidos y la segunda a una ruta realizada en San Sebastián.

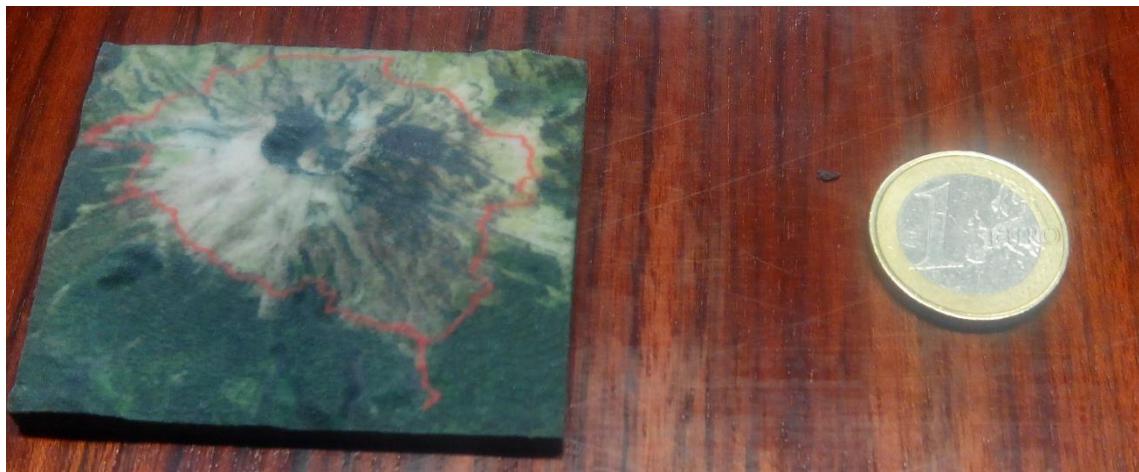


Figura 111 - Resultado en shapeways



Figura 112 – Resultado en shapeways

Problemas y soluciones

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

- *threejs*: Las imágenes que serán usadas como texturas en materiales por obligación deben estar alojadas en el servidor, si no son alojadas *threejs* no reconoce la textura.
- *threejs*: Cuando se crea una geometría se deben indicar los vértices y como están conectados a través de caras, sino no se visualizará la geometría.
- *threejs*: Cuando se utiliza la función *merge()* de una geometría como parámetros debemos usar la geometría y matriz de una malla, antes debemos actualizar la matriz de esa malla con la función *updateMatrix()*.
- *threejs*: 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:

```
256 [.WebGLRenderingContext]GL ERROR :GL_INVALID_OPERATION : glDrawElements: attempt to access out of range vertices in attribute 1
⚠ WebGL: too many errors, no more errors will be reported to the console for this context.
```

Figura 113 – Error texturas

Revisando otras geometría encontré que el arreglo *faceVertexUvs* debía tener la misma cantidad de elementos que el atributo *faces*, para solucionar esto se aplicó sobre la geometría la función *addFaceVertexUvs()* que rellena manualmente los valores del arreglo *faceVertexUvs*.

```
▼ faceVertexUvs: Array[1]
  ► 0: Array[0]
    length: 1
    ► proto : Array[0]
▼ faces: Array[1247]
  ► [0 ... 99]
  ► [100 ... 199]
  ► [200 ... 299]
  ► [300 ... 399]
  ► [400 ... 499]
  ► [500 ... 599]
  ► [600 ... 699]
  ► [700 ... 799]
  ► [800 ... 899]
  ► [900 ... 999]
  ► [1000 ... 1099]
```

Figura 114 – faces de la geometría

- *mapbox*: Al principio la ruta y tiles que conforman la ruta no se identificaban bien, esto se debía a que *Mapbox* se debe indicar como primer parámetro la longitud y luego la latitud. Para resolver el problema use la clase *Rectangle* de la librería y dibujaba los rectángulos correspondientes por cada tile asegurándome que la elección de los tiles pertenece a la ruta.

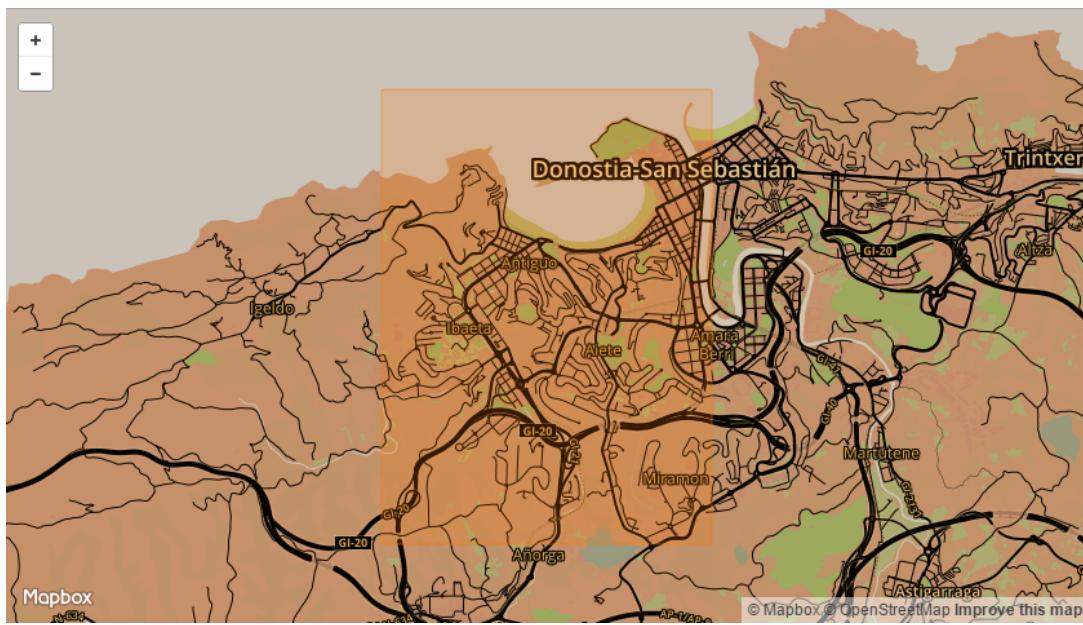


Figura 115 – Tile nivel 13

- *mapbox*: Inclusión obligatoria del mapa en el navegador para cargar la ruta y obtener las imágenes. Además se especifica el tamaño del mapa para poder obtener las coordenadas centrales, estas coordenadas centrales nos ayudarán a obtener las imágenes estáticas.
- *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, de este modo si unimos varios tiles estos coincidirán perfectamente y se podrá visualizar el terreno sin cortes.

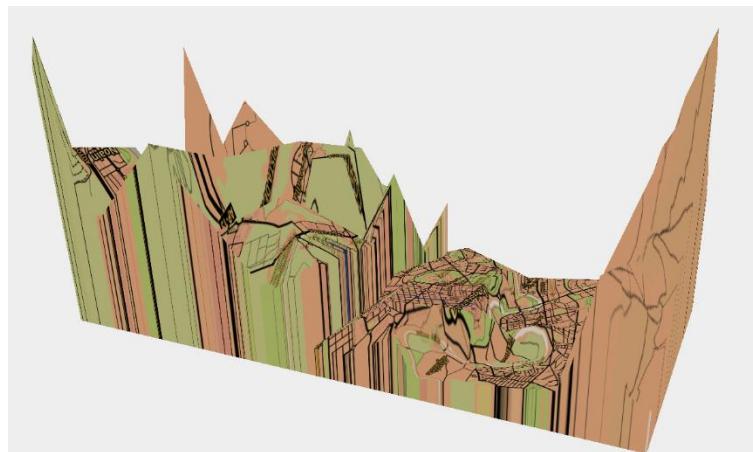


Figura 116 – Tiles no coincidentes

- *cesium*: Las funciones ofrecidas por esta librería son asíncronas, 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.

```

var promise = [];
for(i = 0; i < rectangle_tiles.length; i++) {
    requestPromise = aCesiumTerrainProvider.requestTileGeometry(rectangle_tiles[i].x, rectangle_tiles[i].y, level, false).then(function(data) {
        associateGeometry(data, 1000);
        index_tile++;
    });
    var tilePromise = Cesium.when(requestPromise);
    promise.push(tilePromise);
}

Cesium.when.all(promise, function() {
    createTerrain();
});

```

Figura 117 – Llamadas asíncronas

- *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. En las zonas que no existe el nivel 15 nos devuelve una imagen en negro.



Figura 118 – Mapbox nivel 14

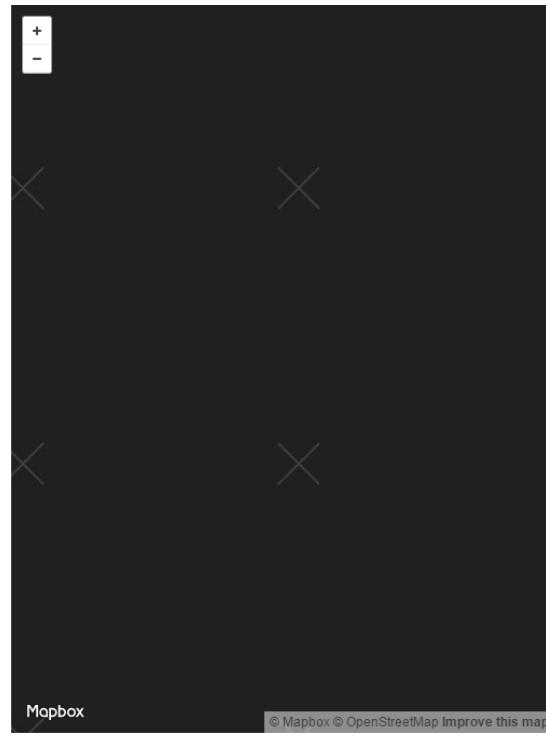


Figura 119 – Mapbox nivel 15

- *mapbox + three*: Se definió una medida estándar para las imágenes, luego al unir las imágenes para obtener la textura final la conexión entre ellas no era precisa y había cortes. Como solución para obtener medidas dinámicas según la ruta se usó la “Proyección de

Mercator" el cuál aproxima el tamaño de las imágenes en función de la latitud.

- *mapbox*: Para la creación de imágenes se optó por la creación de imágenes estáticas con GeoJSON debido a que por el método polyline estaba más limitado y a veces daba errores.
- *mapbox*: En algunos tiles existe un gran número de coordenadas que luego a la hora de crear la imagen estática *Mapbox* nos indica que la petición es demasiado grande, se creó una función que reduce esa cantidad de coordenadas sin perder el sentido de la ruta.



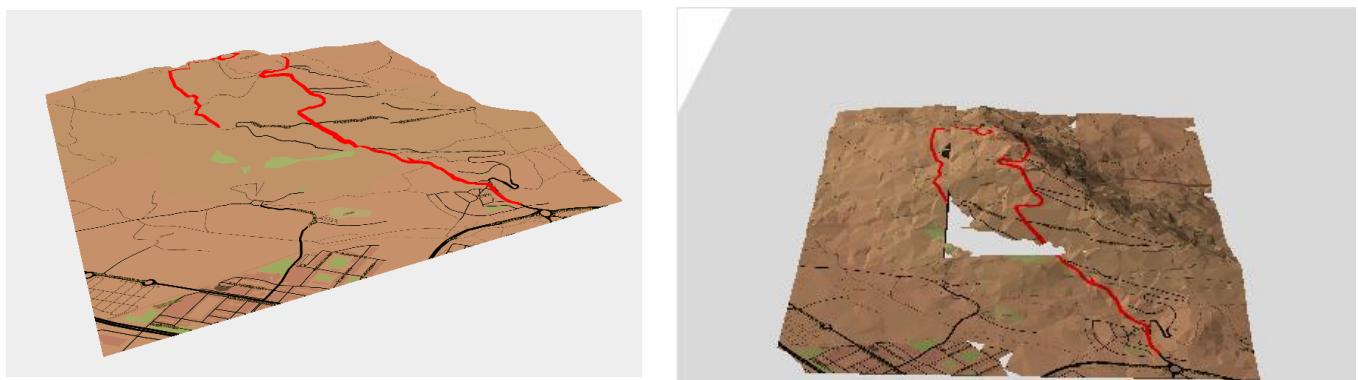
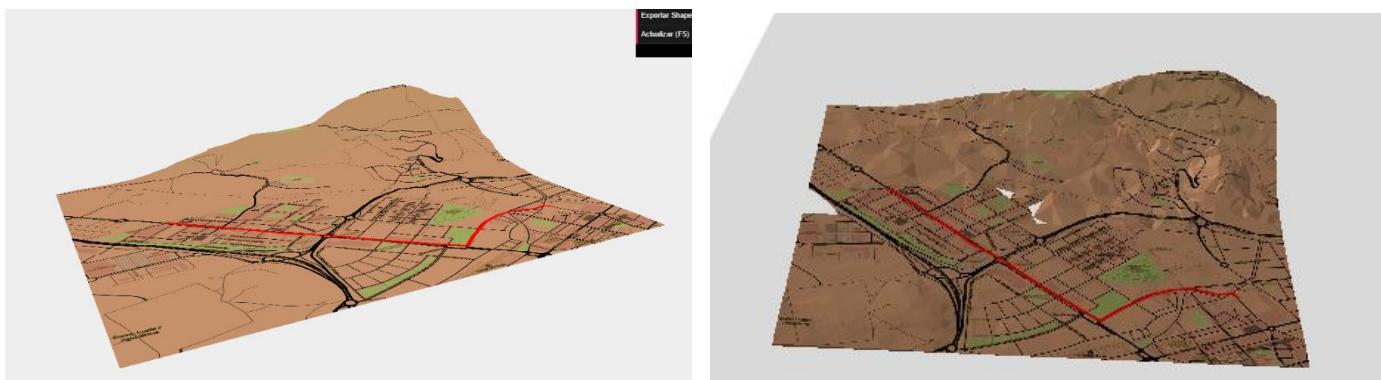
A screenshot of a web browser window. The address bar shows the URL [https://api.tiles.mapbox.com/v4/gonzalito.k53kcf3d/geojson\(%7B%22type%22%3A%22Feature%22,%22geometry%22%3A%22LineString%22,%22properties%22%3A%22%7B%22](https://api.tiles.mapbox.com/v4/gonzalito.k53kcf3d/geojson(%7B%22type%22%3A%22Feature%22,%22geometry%22%3A%22LineString%22,%22properties%22%3A%22%7B%22). Below the address bar, a message box displays the JSON object:

```
{"message": "Request path must be <= 4096 characters."}
```

Figura 120 – Error GeoJSON

- *three*: Durante el desarrollo del proyecto use distintas formas de creación de terrenos:
 - Crear la geometría de cada tile y añadir sus laterales y fondo, cada vez que se creaba una geometría se unía con la siguiente.
 - Especificaba distintos tamaños de grosor de laterales.

Los métodos mencionados anteriormente a nivel gráfico mostraban geometrías completas pero luego al enviar los objetos a *Shapeways* aparecían huecos.



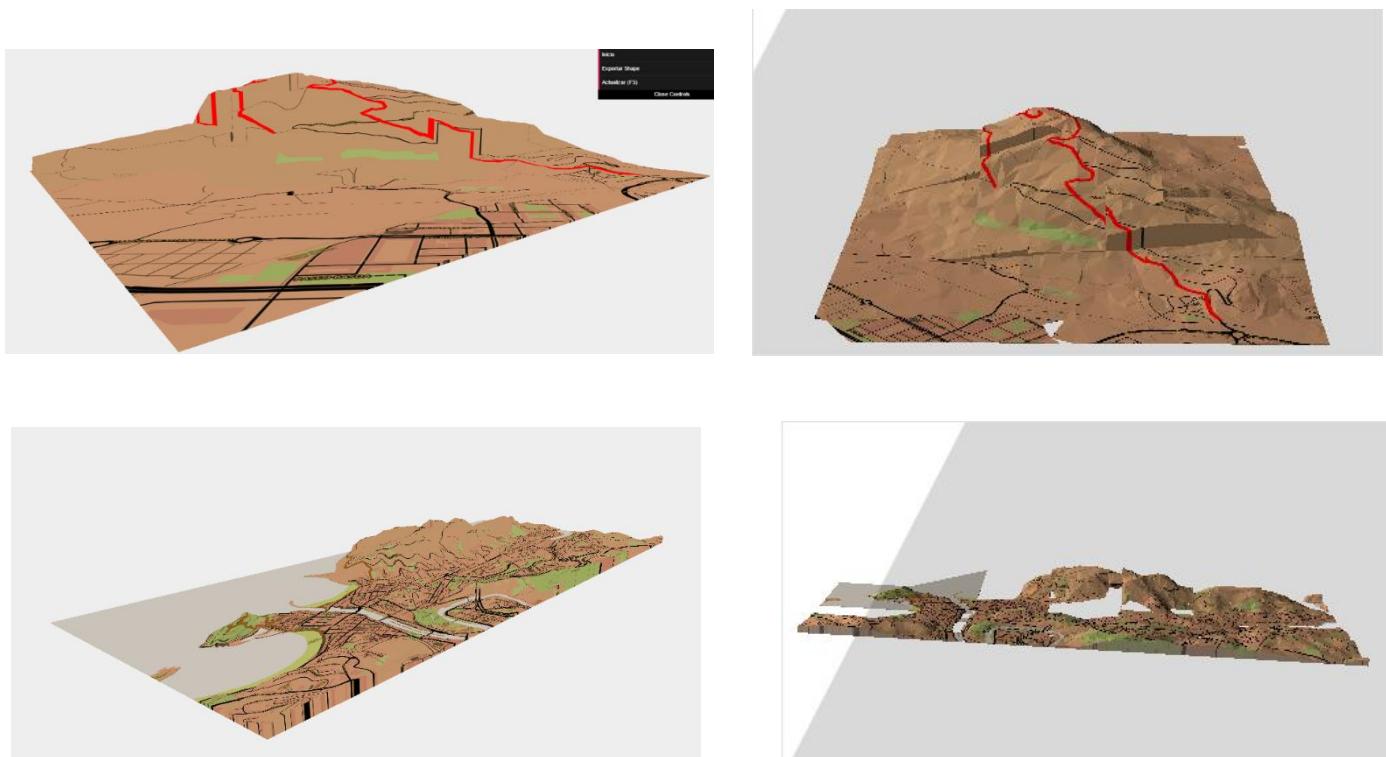


Figura 121 – Imágenes de error de modelado

Como solución primero se crean las geometrías de cada tile y se van uniendo, al finalizar la unión se añade los laterales y fondo obteniendo como resultado en *Shapeways* una geometría completa y sin huecos.

- *php*: Problemas con la memoria a la hora de crear imágenes, como solución tuve que disminuir el tamaño de las mismas. También había problemas de memoria a la hora de hacer peticiones *post* al servidor, para ello se modificó el fichero *php.ini* dando más capacidad al enviar a través del método *post* cambiando el valor de la variable *post_max_size*.
- *cesium*: Por razones desconocidas cuando se crea el terreno final con las texturas en ocasiones las geometrías no coinciden, la única solución es actualizar la página de este modo se soluciona el problema.

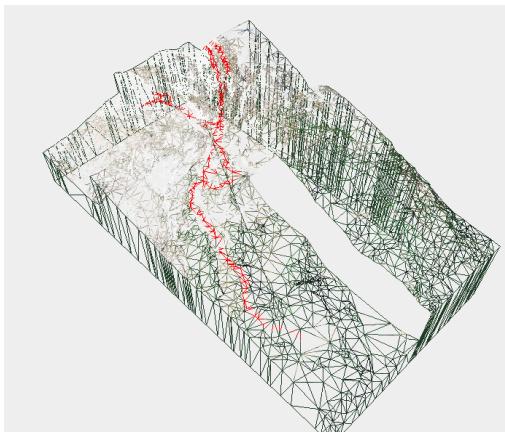


Figura 122 – Error datos cesium

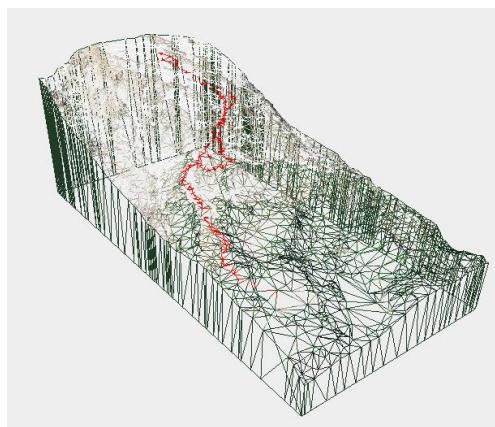


Figura 123 – Datos correctos cesium

Conclusiones y líneas futuras

Conclusiones

Se ha cumplido el objetivo creando una aplicación web que crea un objeto tridimensional con las elevaciones del terreno a partir de una ruta realizada.

He logrado mezclar distintas tecnologías para obtener el objeto tridimensional, *Mapbox* me ha aportado la textura del terreno, *Cesium* las elevaciones del terreno y *threejs* la posibilidad de crear el objeto tridimensional desde cero.

El objeto creado es enviado a *Shapeways* donde es validado e impreso obteniendo el modelo real.

Líneas futuras

Posibilidad de crear rutas personalizadas como lo hace *wikiloc*.

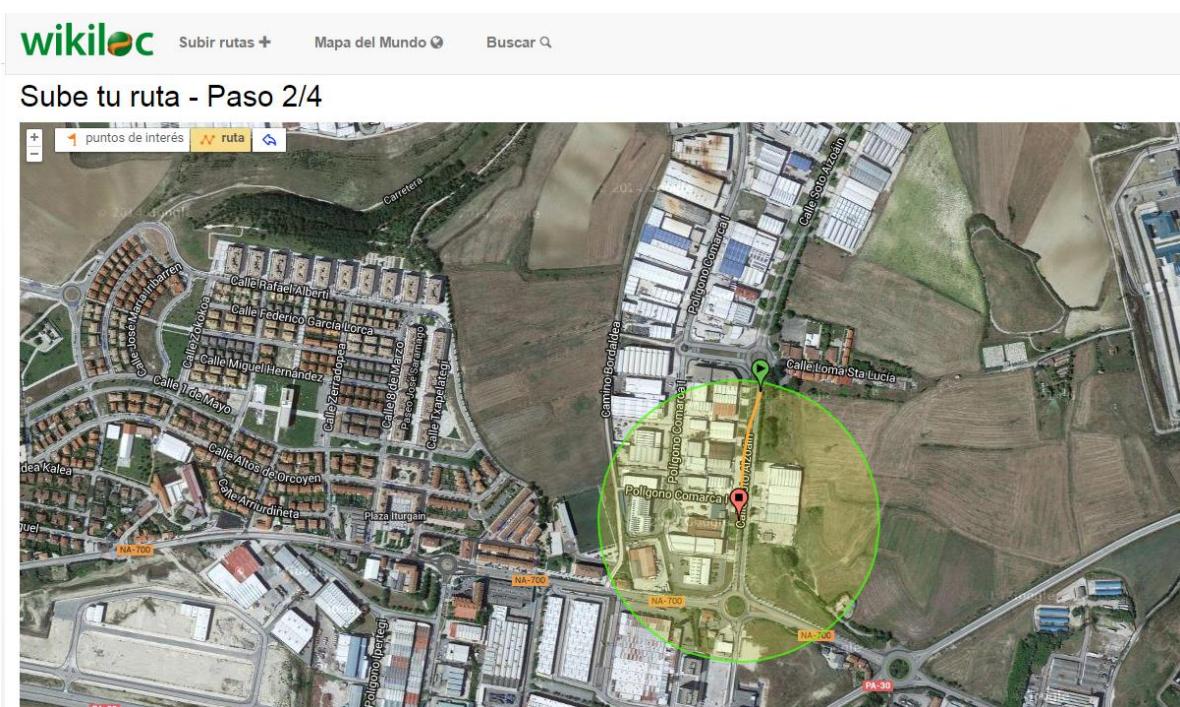


Figura 124 – Dibujar ruta wikilocs

Implementar otras fuentes de datos más potentes como google maps.

Cargar la ruta recorrida en otros objetos 3D, como por ejemplo una taza o modelar en una forma esférica

Implementar conexión directa de la aplicación web con *Shapeways*.

Buscar datos de profundidades marítimas e implementar.

Bibliografía

Ricardo Cabello. ThreeJS <<http://threejs.org/>>

Mapbox. <<https://www.mapbox.com/>>

Cesium. <<http://cesiumjs.org/>>

Drhatch. 3D Print Your Trek, in color!.

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

Sann-Remy Chea. Terrain generation using diamond-square algorithm, Three.js
<<http://srchea.com/apps/terrain-generation-diamond-square-threejs-webgl/>>

Stemkoski. Three.js – examples <<http://stemkoski.github.io/Three.js/>>

Jos Dirksen. All 109 Examples from my book on Three.js for Three.js version r63
<<http://www.smartjava.org/content/all-109-examples-my-book-threejs-threejs-version-r63>>

Eugenio Rodríguez. Fabricación por adición, el camino hacia la próxima revolución de la
tecnología aeroespacial.
<<http://www.fierasdelaingenieria.com/fabricacion-por-adicion-el-camino-hacia-la-proxima-revolucion-de-la-tecnologia-aeroespacial/>>

HGT <<http://fileformats.archiveteam.org/wiki/HGT>>

Coverage map viewfinderpanoramas.org
<http://www.viewfinderpanoramas.org/Coverage%20map%20viewfinderpanoramas_org3.htm>

Paul Groves. Texture coordinates. <<http://paulyg.f2s.com/uv.htm>>

Wikipedia. Proyección de mercator.
<http://es.wikipedia.org/wiki/Proyecci%C3%B3n_de_Mercator>

Comparativa 3D.
<http://www.dima3d.com/wp-content/uploads/2015/02/comparativa_DIMA_LT_p1.jpg>

Shapeways blog. Our Machines
<<https://www.shapeways.com/blog/archives/41-our-machines-part-i-stratasys-fdm-400mc.html>>

