



UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Ingeniería Informática

Proyecto Fin de Carrera

SmartPoligraph: Un entorno gráfico interactivo para  
SmartPoliTech



UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Ingeniería Informática

Proyecto Fin de Carrera

SmartPoligraph: Un entorno gráfico interactivo para  
SmartPoliTech

Autor: Jose Luis Martín Ávila

Tutor: Jose Moreno del Pozo

## ÍNDICE GENERAL DE CONTENIDOS

<b>1. Contenido</b>	
2. INTRODUCCIÓN .....	10
a. Descripción del problema .....	10
b. Tecnologías empleadas .....	10
3. PROGRAMACIÓN .....	13
a. Arquitectura de la aplicación .....	13
i. Cliente.....	13
ii. Servidor.....	18
iii. Otros aspectos .....	20
b. Distribución de las habitaciones.....	20
i. Edificio de Servicios Centrales Planta 0 .....	22
ii. Edificio de Servicios Centrales Planta 1 .....	22
iii. Edificio de Servicios Centrales Planta 2.....	23
iv. Edificio de Arquitectura Técnica Planta 0 .....	24
v. Edificio de Arquitectura Técnica Planta 1 .....	25
vi. Edificio de Informática Planta 0 .....	26
vii. Edificio de Informática Planta 1 .....	27
viii. Edificio de Informática Planta Sótano .....	28
ix. Edificio de Obras Públicas Planta 0.....	28
x. Edificio de Obras Públicas Planta 1 .....	29
xi. Edificio de Telecomunicaciones Planta 0 .....	30
xii. Edificio de Telecomunicaciones Planta 1 .....	31
xiii. Edificio de Telecomunicaciones Planta Sótano.....	33
xiv. Edificio de Investigación Comunicaciones Planta 0.....	34
xv. Edificio de Investigación Comunicaciones Planta 1.....	34
xvi. Edificio de Investigación Comunicaciones Planta 2.....	35

xvii.	Edificio de Investigación Comunicaciones Planta Sótano.....	36
xviii.	Anexo Edificio de Investigación Comunicaciones Planta 0 .....	36
xix.	Anexo Edificio de Investigación Comunicaciones Planta 1 .....	37
c.	Algoritmos utilizados .....	37
i.	Servidor .....	37
ii.	Cliente.....	61
iii.	Convertor de ficheros .....	91
4.	FUNCIONAMIENTO. MANUAL DE USUARIO .....	94
a.	Inicio .....	94
b.	Proceso .....	95
i.	Cambiar edificio y planta .....	95
ii.	Mover y girar el modelo .....	95
iii.	Seleccionar una habitación y obtener información de los sensores.....	96
iv.	Mostrar las temperaturas en tiempo real.....	97
v.	Mostrar las humedades en tiempo real .....	98
5.	MANTENIMIENTO .....	99
a.	Puesta en marcha de la aplicación servidor.....	99
b.	Puesta en marcha de la aplicación cliente .....	100
c.	Adición o edición de elementos gráficos .....	100
i.	Cargar el modelo original.....	100
ii.	Editar el modelo.....	101
iii.	Exportar modelos.....	102
iv.	Conversión .....	105
v.	Carga en la aplicación.....	107
d.	Adición de sensores.....	107
e.	Adición de filtros en tiempo real.....	108
f.	Actualizar nombres no nemotécnicos .....	109

6.	CONCLUSIONES Y TRABAJOS FUTUROS .....	111
a.	Conocimientos previos.....	111
b.	Experiencia y conocimientos adquiridos .....	112
c.	Conclusiones .....	113
d.	Posibles ampliaciones futuras .....	113
i.	Mejorar eficiencia.....	113
ii.	Nuevos filtros en tiempo real.....	113
iii.	Kinect.....	113
iv.	Estadísticas.....	113
v.	Panel de administración.....	114
vi.	Ampliar a otros centros de la Universidad de Extremadura .....	115

## **ÍNDICE DE TABLAS**

Tabla 1: Habitaciones de Edificio de Servicios Centrales Planta Baja .....	22
Tabla 2: Habitaciones de Edificio de Servicios Centrales Planta 1 .....	22
Tabla 3: Habitaciones de Edificio de Servicios Centrales Planta 2 .....	23
Tabla 4: Habitaciones de Edificio de Arquitectura Técnica Planta Baja .....	24
Tabla 5: Habitaciones de Edificio de Arquitectura Técnica Planta 1 .....	25
Tabla 6: Habitaciones de Edificio de Informática Planta Baja .....	26
Tabla 7: Habitaciones de Edificio de Informática Planta 1 .....	27
Tabla 8: Habitaciones de Edificio de Informática Planta Sótano.....	28
Tabla 9: Habitaciones de Edificio de Obras Públicas Planta Baja.....	28
Tabla 10: Habitaciones de Edificio de Obras Públicas Planta 1 .....	29
Tabla 11: Habitaciones de Edificio de Telecomunicaciones Planta Baja .....	30
Tabla 12: Habitaciones de Edificio de Telecomunicaciones Planta 1 .....	31
Tabla 13: Habitaciones de Edificio de Telecomunicaciones Planta Sótano .....	33
Tabla 14: Habitaciones de Edificio de Investigación Comunicaciones Planta Baja...	34
Tabla 15: Habitaciones de Edificio de Investigación Comunicaciones Planta 1 .....	35
Tabla 16: Habitaciones de Edificio de Investigación Comunicaciones Planta 2 .....	35
Tabla 17: Habitaciones de Edificio de Investigación Comunicaciones Planta Sótano	36
Tabla 18: Habitaciones de Anexo Edificio de Investigación Comunicaciones Planta Baja .....	36
Tabla 19: Habitaciones de Anexo Edificio de Investigación Comunicaciones Planta 1 .....	37
Tabla 20: Paleta de colores para temperaturas .....	97
Tabla 21: Paleta de colores para los grados de humedad.....	98

## ÍNDICE DE FIGURAS

Figura 1: Carpetas de la aplicación en el lado cliente .....	14
Figura 2: Subcarpetas con modelos de edificios .....	15
Figura 3: Ejemplo de plantas del modelo de Informática .....	16
Figura 4: Ejemplo de Informática Planta 0 .....	16
Figura 5: Ejemplo de selección de habitación. LAB033 Informática Planta 0 .....	17
Figura 6: Ejemplo de modelo sin habitaciones (BASE). Investigación Planta 0 .....	18
Figura 7: Carpetas de la aplicación Servidor .....	18
Figura 8: Plano de Edificio de Servicios Centrales Planta Baja .....	22
Figura 9: Plano de Edificio de Servicios Centrales Planta 1 .....	22
Figura 10: Plano de Edificio de Servicios Centrales Planta 2 .....	23
Figura 11: Plano de Edificio de Arquitectura Técnica Planta Baja .....	24
Figura 12: Plano de Edificio de Arquitectura Técnica Planta 1 .....	25
Figura 13: Plano de Edificio de Informática Planta Baja .....	26
Figura 14: Plano de Edificio de Informática Planta 1 .....	27
Figura 15: Plano de Edificio de Informática Planta Sótano .....	28
Figura 16: Plano de Edificio de Obras Públicas Planta Baja .....	28
Figura 17: Plano de Edificio de Obras Públicas Planta 1 .....	29
Figura 18: Plano de Edificio de Telecomunicaciones Planta Baja .....	30
Figura 19: Plano de Edificio de Telecomunicaciones Planta 1 .....	31
Figura 20: Plano de Edificio de Telecomunicaciones Planta Sótano .....	33
Figura 21: Plano de Edificio de Investigación Comunicaciones Planta Baja .....	34
Figura 22: Plano de Edificio de Investigación Comunicaciones Planta 1 .....	34
Figura 23: Plano de Edificio de Investigación Comunicaciones Planta 2 .....	35
Figura 24: Plano de Edificio de Investigación Comunicaciones Planta Sótano .....	36
Figura 25: Plano de Anexo Edificio de Investigación Comunicaciones Planta Baja .....	36
Figura 26: Plano de Anexo Edificio de Investigación Comunicaciones Planta 1 .....	37
Figura 27: Esquema de funcionamiento de la aplicación servidor .....	39
Figura 28: Ejemplo de formateo activo .....	41
Figura 29: Ejemplo de formateo no activo .....	42
Figura 30: Esquema de funcionamiento de la función getDatos .....	43
Figura 31: Esquema de funcionamiento de la función obtenerTemperaturas .....	47
Figura 32: Ejemplo de fichero nombresHabitacion.js .....	50

Figura 33: Esquema de funcionamiento de la función filtrarTemperaturas.....	57
Figura 34: Ejemplo de llamadas con un array de 10 temperaturas .....	59
Figura 35: Esquema visual de la aplicación cliente .....	61
Figura 36: Algoritmo de funcionamiento de la función Hover.....	68
Figura 37: Algoritmo switch del botón temperaturas .....	87
Figura 38: Visual de la aplicación cliente .....	94
Figura 39: Menú de selección de edificios.....	95
Figura 40: Ejemplo de giro y desplazamiento de modelo 3D .....	96
Figura 41: Ejemplo de lectura de sensores.....	96
Figura 42: Ejemplo de temperaturas en tiempo real .....	97
Figura 43: Ejemplo de humedades en tiempo real .....	98
Figura 44: Puesta en marcha de la aplicación servidor .....	99
Figura 45: Aplicación servidor en ejecución .....	100
Figura 46: Modelo antes de editar.....	101
Figura 47: Modelo después de editar .....	101
Figura 48: Submodelo AUL049.....	102
Figura 49: Opciones de exportación .....	103
Figura 50: Submodelo AUL050.....	103
Figura 51: Submodelos para Informática Planta 0.....	104
Figura 52: Proceso de conversión .....	105
Figura 53: Resultado de la conversión .....	106
Figura 54: Modelo actualizado en la aplicación .....	107
Figura 55: Esquema de funcionamiento swtich de filtros.....	108
Figura 56: Ejemplo de localización de fichero nombresFichero.js.....	110
Figura 57: Ejecución de la función actualizar nombres .....	111
Figura 58: Ejemplo con datos estadísticos .....	114



## **RESUMEN**

El objetivo del proyecto SmartPoliTech es convertir a la Escuela Politécnica de la UEx en un gran ecosistema experimental, un living-lab para el diseño, implantación, integración y validación de sistemas capaces de crear espacios inteligentes, mediante el desarrollo de tecnologías SmartX; un espacio que sea energéticamente eficiente, que facilite la vida social y académica, y que resulte atractivo para el desarrollo individual de sus habitantes y usuarios.

Para conseguir este objetivo a lo largo de los últimos cuatro años se han implantado una serie de sensores que capturan datos del entorno de forma periódica. Posteriormente, estos datos son tratados por multitud de aplicaciones para su interpretación.

En este trabajo, se ha desarrollado un entorno interactivo (SmartPoligraph) para poder consultar en todo momento la información almacenada. Sensores de temperatura, humedad, consumo eléctrico, de agua, CO<sub>2</sub> están disponibles en la aplicación donde se puede acceder a la última lectura de los mismos. Además, la visualización de dicho entorno 3d puede cambiar en función de la información almacenada.

Otra de sus habilidades, se puede visualizar en tiempo real que temperatura posee una localización determinada cambiando el color de las paredes del entorno para poder visualizar mejor la temperatura en todas las zonas delimitadas. Lo mismo se puede aplicar para el grado de humedad y es útil para monitorizar en todo momento que está ocurriendo.

Adicionalmente, contiene todos los modelos en 3d de todos los edificios de la Escuela Politécnica, así como su división en función de un plano que se adjunta en esta misma documentación, por si fuera necesario añadir o sustituir sensores que ayuden al control de SmartPoliTech.

En esta memoria se explicará qué tecnologías se han utilizado y el motivo de las mismas. También cómo funciona la aplicación y como se pueden modificar los aspectos más importantes (nuevos sensores, modificación de colores o modelos).

## **2. INTRODUCCIÓN**

### **a. Descripción del problema**

Los principales objetivos de SmartPoliTech son la eficiencia energética para facilitar la vida social y académica y el desarrollo de aplicaciones que la mejoren. Los datos recopilados por SmartPoliTech quedan almacenados en una base de datos que no es pública y que, además, solo muestra datos recogidos sin nada que permita interpretarlos fácilmente.

Tampoco existe alguna forma pública de explorar en 3 dimensiones los entornos observados por SmartPoliTech

Este proyecto utiliza los datos recogidos y los combina con un entorno 3D que permite a cualquier usuario ver la información recopilada en la zona que le interese. De esta forma también se puede facilitar el acceso a los datos y una mejor interpretación de los mismos en todo momento ya que estos aparecen de una forma visualmente atractiva y clasificados según su función.

### **b. Tecnologías empleadas**

Se han utilizado diversas tecnologías según el problema que se requería resolver:

**3D:** Existen multitud de herramientas para modelar en 3 dimensiones. Algunas incluso más orientadas al modelado de edificios. Para ello se contemplaron las siguientes opciones:

- **Sketchup:** es un programa de diseño gráfico y modelado en tres dimensiones (3D) basado en caras. Para entornos de arquitectura, ingeniería civil, diseño industrial, diseño escénico, GIS, videojuegos o películas. Es un programa desarrollado por @Last Software, empresa adquirida por Google en 2006 y finalmente vendida a Trimble en 2012. La principal característica es poder realizar diseños en 3D de forma sencilla e intuitiva. Permite conceptualizar y modelar imágenes en 3D de edificios y cualquier objeto o artículo que imagine el diseñador o dibujante, además de que el programa incluye una galería de

objetos, texturas e imágenes listas para descargar y utilizar en el modelado de la figura.

- **Blender:** Blender es un programa informático multi plataforma, dedicado especialmente al modelado, iluminación, renderizado, animación y creación de gráficos tridimensionales. También de composición digital utilizando la técnica procesal de nodos, edición de vídeo, escultura (incluye topología dinámica) y pintura digital.
- **Revit:** Es un software de Modelado de información de construcción (BIM, Building Information Modeling), para Microsoft Windows, desarrollado actualmente por Autodesk. Permite al usuario diseñar con elementos de modelación y dibujo paramétrico. BIM es un paradigma del dibujo asistido por computador que permite un diseño basado en objetos inteligentes y en tercera dimensión. De este modo, Revit provee una asociatividad completa de orden bi-direccional. Un cambio en algún lugar significa un cambio en todos los lugares, instantáneamente, sin la intervención del usuario para cambiar manualmente todas las vistas. Un modelo BIM debe contener el ciclo de vida completo de la construcción, desde el concepto hasta la edificación.

De estas tres tecnologías para el diseño de los modelos 3D utilizados por la aplicación se ha decidido optar por Sketchup debido a las siguientes razones:

- Existe un modelo ya realizado de la escuela politécnica realizado por un alumno como Proyecto Fin de Carrera que podía reutilizarse para el desarrollo de esta aplicación. Solo requería adaptar el modelo antiguo al estado actual de los edificios, dividir correctamente las zonas según las habitaciones y modelar otros edificios que no fueron realizados anteriormente como es el Pabellón de Telecomunicaciones y Edificio de Investigación con su correspondiente Anexo. Utilizar otras aplicaciones significaría diseñar de 0 los edificios y de Revit solo disponía del pabellón de informática. Además, el motor gráfico (del que se hablará más adelante) no es compatible aún con BIM por lo que habría que seguir trabajando con modelos en OBJ.
- Facilidad de diseño. De los programas mencionados anteriormente, Sketchup es el que permite diseñar edificios de una forma más sencilla dibujando los

planos y utilizando la herramienta de extrusión para crear paredes, puertas y ventanas. El diseño en Revit es bastante más complejo y además generaría mucha información que no es necesaria para la aplicación a desarrollar. Blender está más enfocado al diseño de animaciones que de edificios por lo cual fue descartado.

**Aplicación web:** La aplicación consta de un HTML etiquetado correctamente y un motor en JavaScript que trabaja con la librería gráfica y realiza todos los cambios a nivel de usuario, como es el funcionamiento de los botones y la muestra de datos en las tablas. Se utiliza AJAX para recuperar los datos de la base de datos y obtener los modelos gráficos de la aplicación.

**Servidor web:** Se ha creado una API REST en node.js que hace de intermediario entre la aplicación y la base de datos. De esta forma, las credenciales de conexión de la base de datos son ocultas al usuario mejorando la seguridad y cada una de las funciones añadidas devuelve un json con los datos que pida la aplicación web cliente. Se ha preparado para que la aplicación consuma los menos recursos disponibles relegando todo lo posible al servidor, para aligerar la aplicación del cliente que ya es bastante pesada trabajando con el entorno 3D. Las funciones del servidor son devolver los datos de los sensores de una zona en concreto, obtener todas las temperaturas y grados de humedad de un piso completo y actualizar el nombre de las habitaciones (el cual se explicará más adelante).

**Gráficos 3D:** Se ha utilizado para ello la librería gráfica Three.js. Three.js es una biblioteca liviana escrita en JavaScript para crear y mostrar gráficos animados por ordenador en 3D en un navegador Web y puede ser utilizada en conjunción con el elemento canvas de HTML5, SVG ó WebGL. El código fuente está alojado en un repositorio en GitHub y se ha popularizado como una de las más importantes para la creación de las animaciones en WebGL. Three.js está disponible bajo la licencia MIT. Se ha contemplado el uso de otras librerías gráficas como Babylon, pero al final se opta por Three.js ya que es la que mejor documentada está, más opciones tiene y más

fácil es buscar información y ejemplos de funcionamiento. Además, es bastante eficiente y actualmente va por su versión 87.

**Conversor de modelos:** Los modelos en 3D exportados de Sketchup están en el estándar OBJ, el cual la aplicación puede funcionar con ellos. Sin embargo, si en lugar de estar en formato OBJ están en formato json, su carga es más rápida por lo cual se ha desarrollado una aplicación que convierte automáticamente los OBJ en ficheros json optimizados.

**Base de datos:** Se utiliza una base de datos InfluxDB donde están almacenados los datos recopilados por los sensores. La base de datos ya existe por lo que no es creada por este proyecto y solo hace uso de ella.

**Frameworks y otras tecnologías empleadas:** Se ha utilizado JQuery para facilitar la implementación de las funciones JavaScript que manejan la interfaz, así como las peticiones AJAX. También se ha utilizado C++ para desarrollar una aplicación que ejecuta el conversor de modelos por cada fichero OBJ que encuentre. El conversor de modelos está proporcionado por Three.js y es un script en Python, por lo que requiere su instalación.

### **3. PROGRAMACIÓN**

#### **a. Arquitectura de la aplicación**

La aplicación consta de dos partes, un cliente basado en HTML y JavaScript y un servidor implementado en Node.js.

##### **i. Cliente**

Dentro del cliente tenemos la web HTML, maquetada con Bootstrap y donde cargamos todos los scripts JavaScript necesarios para el funcionamiento. Además de JQuery y Bootstrap, se detallan el resto de librerías utilizadas:

- *three.keyboardstate.js*: Para los controles de teclado de la aplicación

- *three.min.js*: El más importante. Contiene toda la API gráfica y sus funciones más importantes.
- *detector.js*: Librería que nos permite saber que objeto dentro del entorno gráfico se ha seleccionado con el ratón.
- *Projector.js*: Requerida por el detector para saber dónde se ha hecho click dentro del entorno gráfico
- *OrbitControls.js*: Librería para controlar la cámara del entorno gráfico con el ratón.

Después, cargamos el motor creado en este proyecto, *render.js* que contiene todas las funciones necesarias para la carga de los modelos al entorno gráfico, así como su tratamiento y cambios del mismo, todas las funciones necesarias para la lectura y carga de datos, y las funciones para cambiar la interfaz que ve el usuario en el HTML.

Además del fichero index.html, hay 4 carpetas:

Nombre	Fecha de modifica...	Tipo	Tamaño
css	15/08/2017 5:23	Carpeta de archivos	
img	31/08/2017 23:58	Carpeta de archivos	
js	05/09/2017 23:50	Carpeta de archivos	
modelos	05/09/2017 2:49	Carpeta de archivos	
index.html	06/09/2017 4:50	Archivo HTML	4 KB

Figura 1: Carpetas de la aplicación en el lado cliente

- *css*: Carpeta donde se almacenan los ficheros de estilo css de la aplicación
- *img*: Carpeta donde se almacenan los ficheros de imágenes e iconos de la aplicación
- *js*: Carpeta donde se almacenan todos los scripts necesarios para el funcionamiento de la aplicación, ya mencionados anteriormente.
- *Modelos*: Carpeta donde se almacenan los modelos ya convertidos a .js desde .OBJ. Dentro de la misma tendremos varias subcarpetas cada una

correspondiente al nombre nemotécnico formado por 3 siglas de cada edificio de la Escuela Politécnica








Nombre	Fecha de modifica...	Tipo
 AIN	05/09/2017 2:51	Carpeta de archivos
 ATE	23/08/2017 0:19	Carpeta de archivos
 INF	03/09/2017 5:37	Carpeta de archivos
 INV	03/09/2017 4:34	Carpeta de archivos
 OPU	24/08/2017 3:48	Carpeta de archivos
 SCO	28/08/2017 18:42	Carpeta de archivos
 TEL	03/09/2017 16:06	Carpeta de archivos

Figura 2: Subcarpetas con modelos de edificios

Los nombres nemotécnicos corresponden a los siguientes edificios:

- AIN: Edificio Anexo de Investigación Comunicaciones
- ATE: Pabellón de Arquitectura Técnica y Edificación
- INF: Pabellón de Informática
- INV: Edificio de Investigación
- OPU: Pabellón de Obras Públicas
- SCO: Pabellón de Servicios Comunes
- TEL: Pabellón de Telecomunicaciones

Se han utilizado estos nombres nemotécnicos para que coincidan con el nombre asignado a los sensores, los cuales se explicarán más adelante.

Por cada carpeta de modelos, tendremos varias subcarpetas correspondientes a cada planta, donde P00 equivale a planta baja, P01 a primera planta, P02 a segunda planta y PS1 a primera planta de sótano. Esta nemotécnica se utiliza al igual que la anterior, para que coincida con la utilizada por los sensores para facilitar la implementación y mantenimiento de la aplicación.




Nombre	Fecha de modifica...	Tipo	Tamaño
 P00	05/09/2017 5:33	Carpeta de archivos	
 P01	05/09/2017 5:59	Carpeta de archivos	
 PS1	03/09/2017 5:36	Carpeta de archivos	

Figura 3: Ejemplo de plantas del modelo de Informática

Y en cada planta tenemos los ficheros OBJ ya convertidos a JS, así como las texturas utilizadas, el fichero de metadatos y el fichero de nombres de habitación.































































 _7.jpg	 COM014.js	 datos.js	 LAB012.js
 _Metal_Aluminum_Anodized_5.jpg	 COM017.js	 DES005.js	 LAB016.js
 _Metal_Rough_8.jpg	 COM026.js	 DES010.js	 LAB018.js
 _Stone_Granite_Midnite_2.jpg	 COM029.js	 DES011.js	 LAB021.js
 _Tile_Ceramic_Natural_1.jpg	 COM031.js	 DES015.js	 LAB032.js
 _Tile_Ceramic_Natural_3.jpg	 COM034.js	 DES022.js	 LAB033.js
 _Tile_Ceramic_Natural_5.jpg	 COM038.js	 DES023.js	 LAB036.js
 _Tile_Ceramic_Natural_14.jpg	 COM044.js	 DES024.js	 LAB037.js
 AUL028.js	 COM046.js	 DES025.js	 LAB040.js
 AUL030.js	 COM048.js	 DES027.js	 LAB041.js
 BASE.js	 CUA002.js	 DES035.js	 nombresHabitacion.js
 COM001.js	 CUA004.js	 DES039.js	 Stone_Brushed_Khaki.jpg
 COM003.js	 CUA019.js	 DES042.js	 Suede_Green.jpg
 COM006.js	 CUA020.js	 DES043.js	 Tile_Ceramic_Earthy.jpg
 COM009.js	 CUA045.js	 LAB007.js	
 COM013.js	 CUA047.js	 LAB008.js	

Figura 4: Ejemplo de Informática Planta 0

Para simplificar, a partir de ahora llamamos *habitación* a cualquier zona delimitada según los planos de la localización de los sensores. Dicha habitación puede ser un aula, un despacho, un pasillo, una zona común, etc., cerrada y donde se han instalado o se instalarán sensores para monitorizar la actividad y parámetros meteorológicos. La distribución está detallada en el apartado *b*, *distribución de las habitaciones* y que sigue la nemotécnica de los sensores

Todos los modelos están subdivididos en las habitaciones correspondientes, mientras que la zona del modelo no seleccionable y que no posee datos recibe el nombre de BASE. El resto, recibe nombres referentes a la habitación correspondiente. Hay que destacar dos ficheros más que son el fichero *datos.js* y *nombreshabitacion.js*. El primero contiene un array de datos en formato json donde cada elemento tiene dos



parámetros y un tercero opcional: *nombreArchivo*, *habitación* y el opcional nombre. *nombreArchivo* hace referencia al fichero del modelo donde están los datos de la habitación y *habitación* es el nombre nemotécnico que tendrá la misma una vez que se ha cargado. Estos 2 parámetros son generados automáticamente utilizando el conversor de modelos al convertir OBJ en JS. Por último, el parámetro nombre sirve para el apartado visual de la aplicación y hace mostrar un nombre no nemotécnico en caso de que existiera. Por ejemplo, mostraría en la aplicación *RoboLab* en lugar de *LAB033*.

El conversor de modelos no genera los nombres automáticamente ya que este solo se limita a convertirlos y a crear el fichero de metadatos necesario para su uso. Para añadir los nombres se puede hacer de forma manual o utilizando la herramienta incluida en la API REST que se explicará en el apartado de mantenimiento.

Por último, el fichero *nombresHabitacion.js* contiene un array de nombres y la habitación a la que hacen referencia, para ser mostrado en la parte visual de la aplicación.

Para crear los modelos se ha utilizado Sketchup 2016 y para dividirlos en las habitaciones correspondientes, se han seleccionado y agrupado las partes internas de la misma. Luego se exporta a OBJ cada grupo por separado y cuando se han exportado todos los grupos de cada habitación, se hace una última exportación sin los grupos de habitación que será el llamado BASE, carente de información.

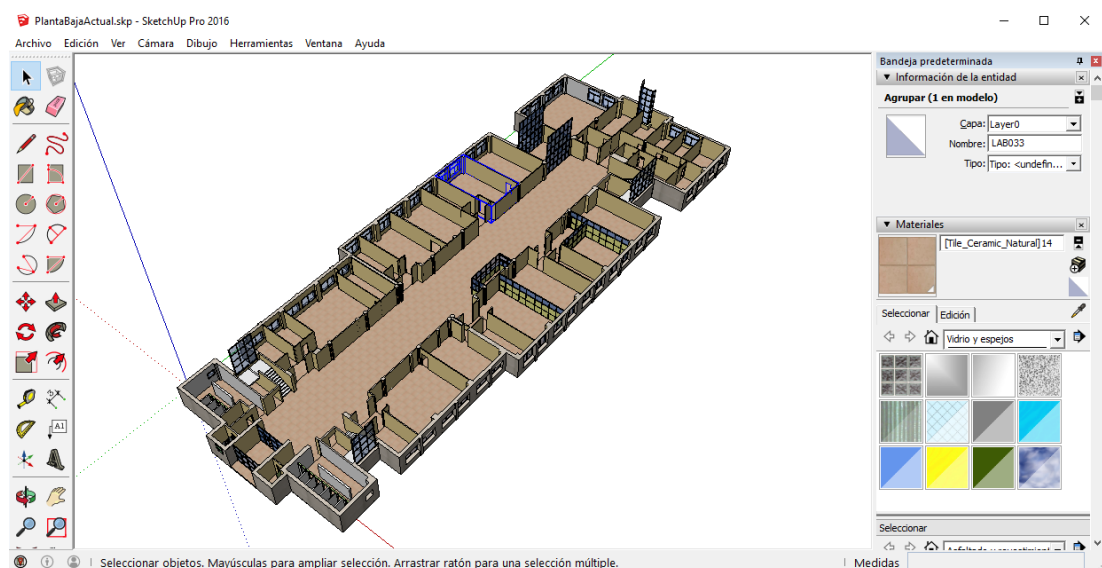


Figura 5: Ejemplo de selección de habitación. LAB033 Informática Planta 0

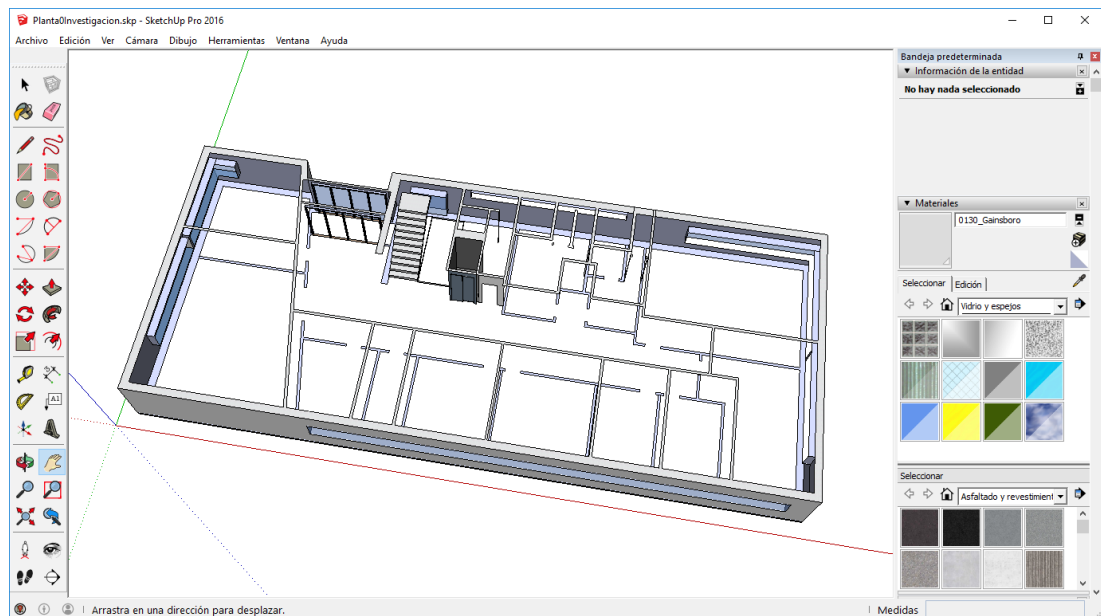


Figura 6: Ejemplo de modelo sin habitaciones (BASE). Investigación Planta 0

El motor de la aplicación cliente está definido en el fichero render.js. Realiza todas las acciones para manejar el modelo 3d, petición de datos, control de eventos y manejo de la interfaz de usuario. Lo primero que encontramos es toda la declaración de variables globales que se van a utilizar, seguida de los listeners para los eventos de JavaScript, declaración de funciones necesarias y por último la ejecución de las mismas para dar funcionamiento al motor.

## ii. Servidor

En el lado del servidor tenemos una API REST desarrollada en Node.js que actúa de intermediario entre la aplicación y la base de datos para responder a todas las peticiones de la aplicación. La división de carpetas y ficheros es la siguiente:

Nombre	Fecha de modifica...	Tipo	Tamaño
controladores	15/08/2017 5:24	Carpeta de archivos	
node_modules	05/09/2017 3:17	Carpeta de archivos	
public	01/09/2017 0:24	Carpeta de archivos	
rutras	15/08/2017 5:24	Carpeta de archivos	
app.js	15/08/2017 5:29	Archivo JS	2 KB
index.js	15/08/2017 5:24	Archivo JS	1 KB
package.json	05/09/2017 3:17	Archivo JSON	1 KB
package-lock.json	05/09/2017 3:17	Archivo JSON	62 KB

Figura 7: Carpetas de la aplicación Servidor

- Controladores: Carpeta donde se almacenan los controladores. En este caso solo existe dentro un fichero (lectura.js) que se trata del controlador lectura, que recibe las peticiones de la aplicación, pide los datos necesarios a la base de datos, los procesa y los devuelve a la aplicación para que esta haga uso de los mismos.
  - Node\_modules: Carpeta donde se almacenan todos los paquetes de node.js necesarios para que el servidor funcione:
    - *body-parser*: Para usos futuros por si fuera necesario mejorar la aplicación servidor.
    - *express*: Para crear la aplicación servidor.
    - *file-system*: Para trabajar con el sistema de archivos. Utilizado para juntar los ficheros de datos.js y nombresHabitacion.js.
    - *influx*: Componente para la conexión con la base de datos y realizar consultas.
    - *request*: Para la lectura de las peticiones
- Además, se incorpora el fichero package.json para volver a instalar los paquetes y la versión utilizada en caso de que estos se borren.
- Public: Contiene los datos de la aplicación cliente, ya explicados anteriormente
  - Rutas: Contiene las URLs creadas de la API REST, así como la función del controlador que se ejecuta al hacer uso de esas rutas.
  - app.js: Contiene los datos necesarios para la creación de la aplicación servidor
  - index.js: Establece el puerto de escucha de la aplicación servidor y la ejecuta.
  - package.json: Metadatos de la aplicación servidor.

### **iii. Otros aspectos**

Además de las aplicaciones cliente y servidor, se ha desarrollado una aplicación Conversor de Modelos que se puede encontrar dentro de la carpeta “Conversor de Modelos” en el CD entregado que convierte los modelos de objeto OBJ a JS, optimizando la carga de la aplicación. El conversor crea una carpeta Convertido y busca todos los ficheros OBJ y ejecuta un script desarrollado en Python por los creadores de la librería Three.js que convierte los modelos OBJ en JS almacenando en esa carpeta el fichero de salida. Además, guarda todas las texturas utilizadas por los modelos dentro de esa misma carpeta para no repetir el mismo fichero por cada modelo diferente.

### **b. Distribución de las habitaciones**

La distribución de las habitaciones está reglada con respecto a unos planos que se entregan en este mismo CD y para indicar si es un laboratorio, aula, despacho, etc., se sigue esta regla nemotécnica formada por 3 letras y tres números.

- AULXXX: Aula
- COMXXX: Zona común, el cual puede ser un cuarto de baño, un pasillo u otras zonas
- CUAXXX: Cuartos privados como pueden ser almacenes, cuartos de contadores, servidores, limpieza etc.
- DESXXX: Despacho
- SEMXXX: Seminario
- CAFXXX: Cafetería

Donde XXX es el número de habitación indicado en el plano. Algunas distribuciones no coinciden en plano a como es actualmente, así que se ha priorizado la actualidad frente al plano.

Se sigue esta regla nemotécnica para que coincida con la utilizada por los sensores ya existentes. Cada sensor está formado por las siguientes reglas:

UEXCC\_EDI\_PLA\_HABITA\_NUMERO\_TIP

Donde:

- EDI: Edificio en donde se encuentra, el cual es el nombre nemotécnico ya descrito anteriormente. Es decir, ATE para Arquitectura Técnica, INF para Informática, SCO para Edificio de Usos Comunes, OPU para Obras Públicas, TEL para Telecomunicaciones, INV para Edificio de Investigación y AIN para su ANEXO.
- PLA: Planta donde se encuentra el sensor. A saber:
  - P00: Planta Baja
  - P01: Planta 1
  - P02: Planta 2
  - PS1: Planta Sótano
- HABITA: La habitación donde se encuentra y las cuales se indicarán todas en el apartado siguiente
- NOMBRE: Número del sensor, los cuales serán SEN001, SEN002, etc.
- TIP: Tipo de sensor, los cuales son los siguientes:
  - THV: Temperatura, humedad y ventana
  - THR: Temperatura, humedad y CO2
  - THC: Temperatura y humedad
  - STV
  - AGU: Agua
  - ELE: Electricidad
  - CGT

Por ejemplo, un sensor llamado UEX\_INF\_P00\_LAB033\_SEN001\_THV indica un sensor en el edificio de Informática, Planta 0, habitación LAB033 que realiza una lectura de temperatura, humedad y estado de la ventana (abierta o cerrada). A continuación, vamos a indicar los nombres otorgados a cada edificio y planta:

## i. Edificio de Servicios Centrales Planta 0

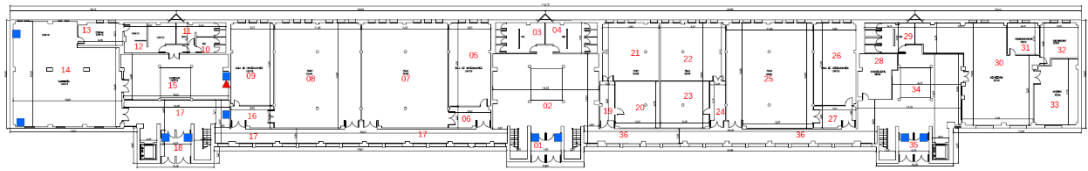


Figura 8: Plano de Edificio de Servicios Centrales Planta Baja

Tabla 1: Habitaciones de Edificio de Servicios Centrales Planta Baja

COM001	CUA013	AUL025
COM002	CAF014	AUL026
COM003	COM015	COM027
COM004	LAB016	COM028
AUL005	COM017	COM029
COM006	COM018	COM030
AUL007	COM019	CUA031
AUL008	AUL020	CUA032
AUL009	AUL021	CUA033
COM010	LAB022	COM034
COM011	AUL023	COM035
CUA012	COM024	COM036

## ii. Edificio de Servicios Centrales Planta 1

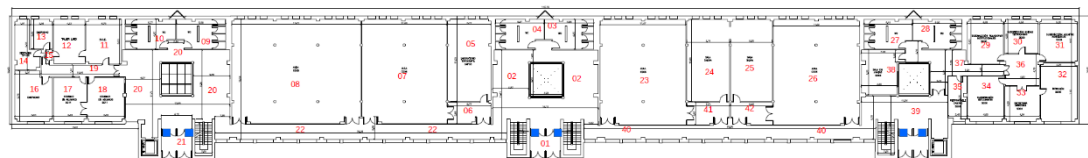


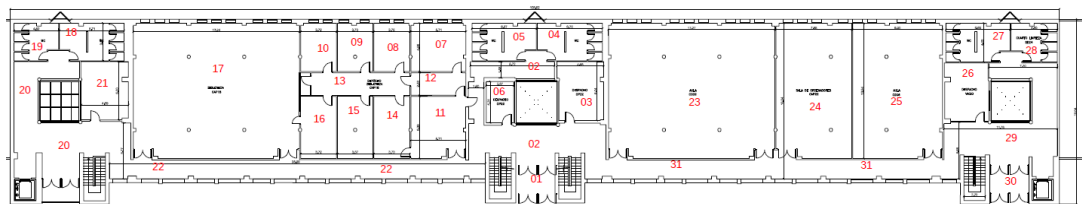
Figura 9: Plano de Edificio de Servicios Centrales Planta 1

Tabla 2: Habitaciones de Edificio de Servicios Centrales Planta 1

COM001	COM002	COM003
--------	--------	--------

COM004	DES017	DES031
LAB005	DES018	DES032
COM006	COM019	DES033
AUL007	COM020	DES034
BASE	COM021	DES035
AUL008	COM022	COM036
COM009	AUL023	COM037
COM010	AUL024	DES038
DES011	AUL025	COM039
AUL012	AUL026	COM040
DES013	COM027	COM041
DES014	COM028	COM042
COM015	DES029	
DES016	DES030	

### iii. Edificio de Servicios Centrales Planta 2



*Figura 10: Plano de Edificio de Servicios Centrales Planta 2*

*Tabla 3: Habitaciones de Edificio de Servicios Centrales Planta 2*

COM001	DES006	COM011
COM002	COM007	COM012
DES003	COM008	COM013
COM004	COM009	COM014
COM005	COM010	COM015

COM016	COM022	COM027
AUL017	AUL023	COM028
COM018	BASE	COM029
COM019	AUL024	COM030
COM020	AUL025	COM031
COM021	DES026	

#### iv. Edificio de Arquitectura Técnica Planta 0



Figura 11: Plano de Edificio de Arquitectura Técnica Planta Baja

Tabla 4: Habitaciones de Edificio de Arquitectura Técnica Planta Baja

COM001	DES011	DES020
COM002	COM012	DES021
CUA003	LAB013	DES022
COM004	DES014	LAB023
CUA005	DES015	COM024
CUA006	LAB016	COM025
COM007	CUA017	COM026
DES008	COM017	DES027
DES009	LAB018	DES028
DES010	DES019	LAB029



DES030	COM034	CUA038
COM031	COM035	COM039
LAB032	DES036	CUA040
DES033	DES037	

**v. Edificio de Arquitectura Técnica Planta 1**

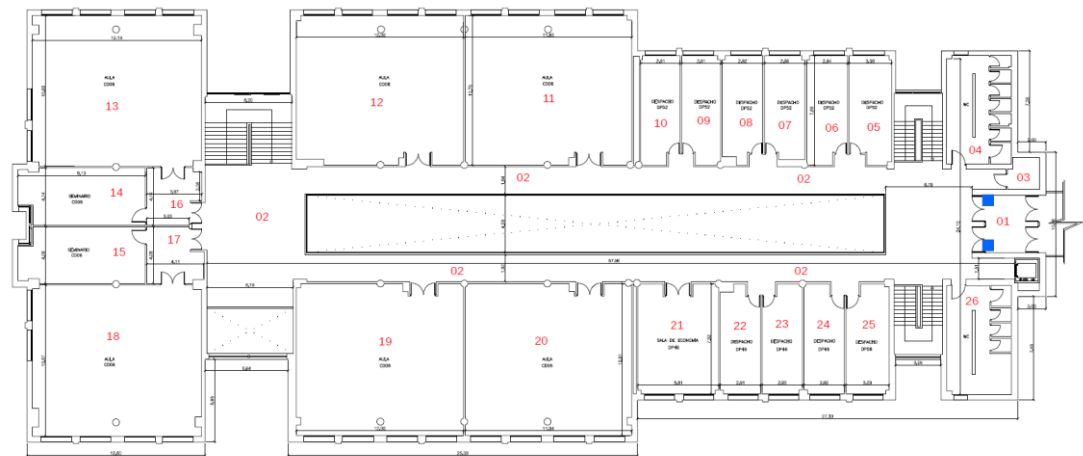


Figura 12: Plano de Edificio de Arquitectura Técnica Planta 1

Tabla 5: Habitaciones de Edificio de Arquitectura Técnica Planta 1

COM001	DES010	AUL019
COM002	AUL011	AUL020
CUA003	AUL012	DES021
COM004	AUL013	DES022
DES005	SEM014	DES023
DES006	SEM015	DES024
DES007	COM016	DES025
DES008	COM017	COM026
DES009	AUL018	

## vi. Edificio de Informática Planta 0



*Figura 13: Plano de Edificio de Informática Planta Baja*

*Tabla 6: Habitaciones de Edificio de Informática Planta Baja*

COM001	COM017	LAB033
CUA002	LAB018	COM034
COM003	CUA019	DES035
CUA004	CUA020	LAB036
DES005	LAB021	LAB037
COM006	DES022	COM038
LAB007	DES023	DES039
LAB008	DES024	LAB040
COM009	DES025	LAB041
DES010	COM026	DES042
DES011	DES027	DES043
LAB012	AUL028	COM044
COM013	COM029	CUA045
COM014	AUL030	COM046
DES015	COM031	CUA047
LAB016	LAB032	COM048

## vii. Edificio de Informática Planta 1



Figura 14: Plano de Edificio de Informática Planta 1

Tabla 7: Habitaciones de Edificio de Informática Planta 1

COM001	DES17	DES033
CUA002	DES018	DES034
COM003	DES019	DES035
DES004	DES020	DES036
DES005	COM021	DES037
DES006	LAB022	DES038
DES007	AUL023	DES039
DES008	AUL024	COM040
DES009	COM025	AUL041
COM010	DES026	DES042
COM011	COM27	DES043
DES012	LAB028	DES044
DES013	AUL029	DES045
DES014	AUL030	DES046
DES015	AUL031	COM047
DES016	CUA032	

## viii. Edificio de Informática Planta Sótano



Figura 15: Plano de Edificio de Informática Planta Sótano

Tabla 8: Habitaciones de Edificio de Informática Planta Sótano

CUA001	CUA003
CUA002	CUA004

## ix. Edificio de Obras Públicas Planta 0



Figura 16: Plano de Edificio de Obras Públicas Planta Baja

Tabla 9: Habitaciones de Edificio de Obras Públicas Planta Baja

COM001	DES005	DES009
CUA002	COM006	LAB010
COM003	LAB007	COM011
CUA004	COM008	DES012

DES013	COM023	DES033
LAB014	LAB024	COM034
CUA015	DES025	COM035
LAB016	DES026	COM036
DES017	COM027	DES037
LAB018	COM028	DES038
DES019	LAB029	CUA039
COM020	COM030	COM040
DES021	COM031	CUA041
LAB022	DES032	COM042

#### x. Edificio de Obras Públicas Planta 1

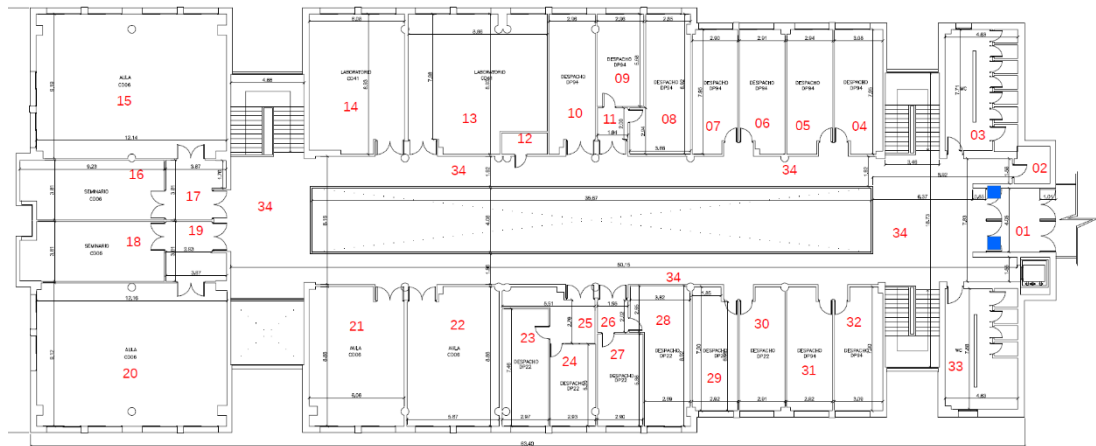


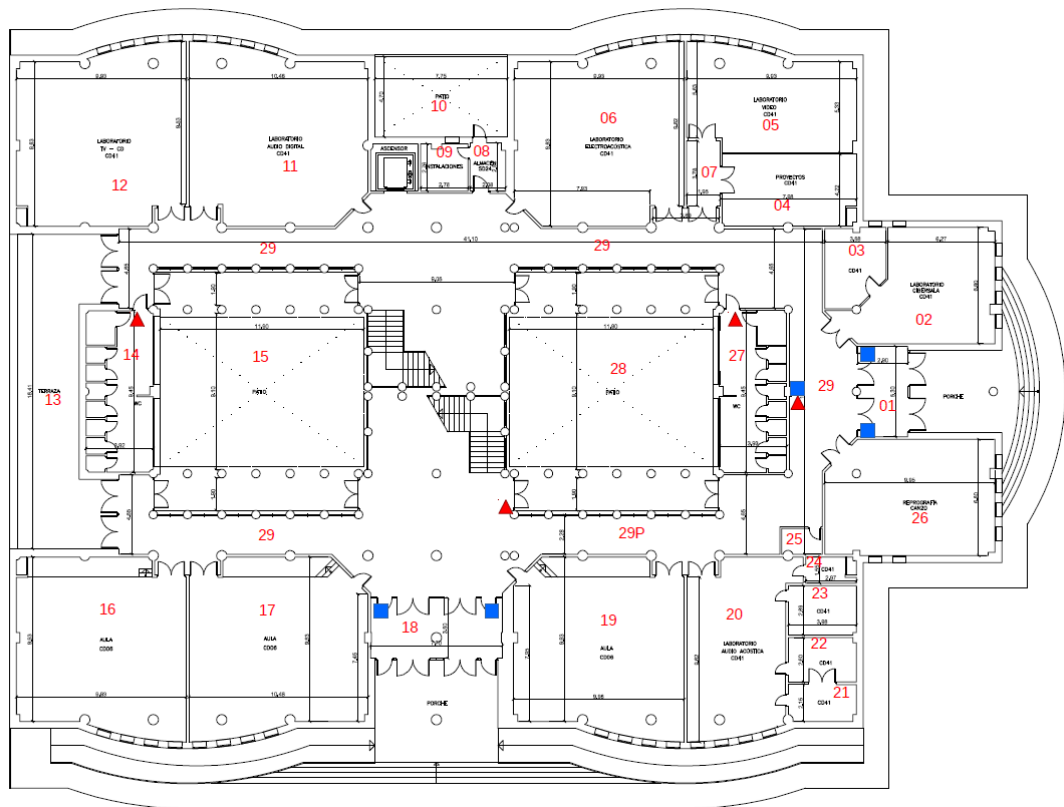
Figura 17: Plano de Edificio de Obras Públicas Planta 1

Tabla 10: Habitaciones de Edificio de Obras Públicas Planta 1

COM001	DES007	AUL013
CUA002	DES008	LAB014
COM003	DES009	AUL015
DES004	DES010	SEM016
DES005	COM011	COM017
DES006	COM012	SEM018

COM019	COM025	DES031
AUL020	COM026	DES032
AUL021	DES027	COM033
AUL022	DES028	COM034
DES023	DES029	
DES024	DES030	

### **xi. Edificio de Telecomunicaciones Planta 0**



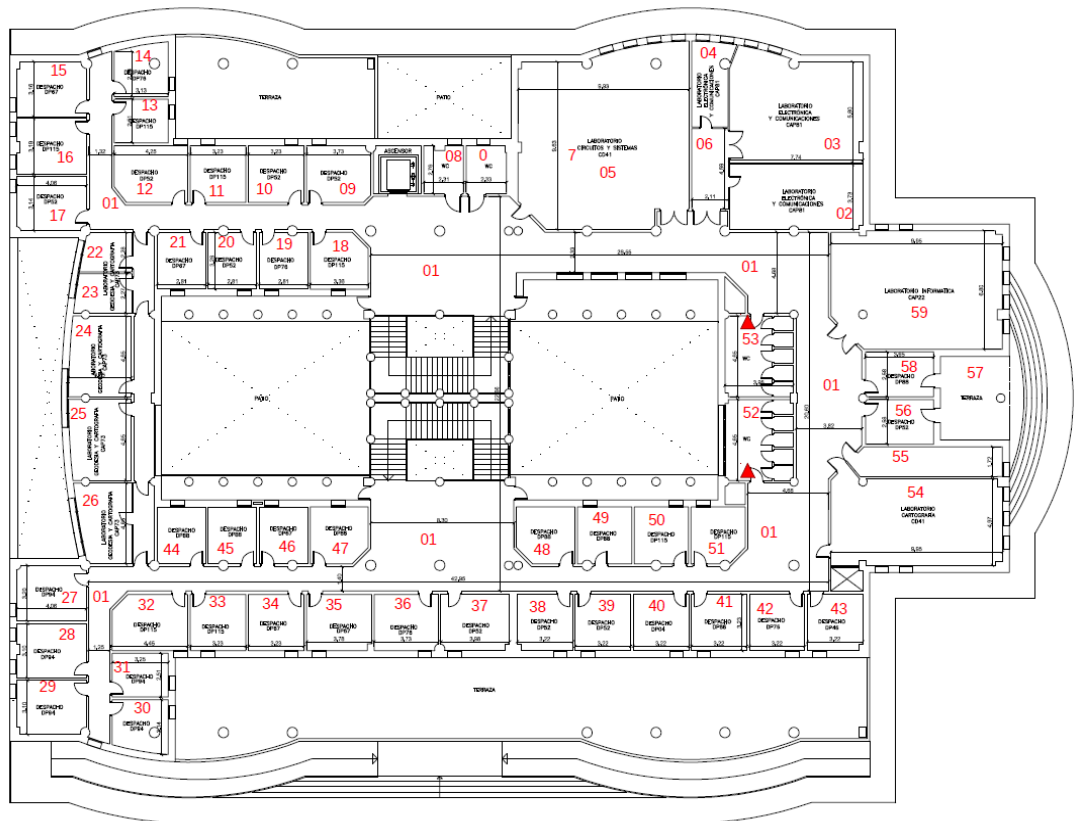
*Figura 18: Plano de Edificio de Telecomunicaciones Planta Baja*

*Tabla 11: Habitaciones de Edificio de Telecomunicaciones Planta Baja*

COM001	LAB006	LAB011
LAB002	COM007	LAB012
LAB003	CUA008	COM013
COM004	CUA009	COM014
LAB005	COM010	COM015

AUL016	CUA021	COM026
AUL017	CUA022	CUA027
COM018	CUA023	COM028
AUL019	CUA024	COM029
LAB020	CUA025	

## xii. Edificio de Telecomunicaciones Planta 1



*Figura 19: Plano de Edificio de Telecomunicaciones Planta 1*

*Tabla 12: Habitaciones de Edificio de Telecomunicaciones Planta 1*

COM001	COM006	DES011
LAB002	COM007	DES012
LAB003	COM008	DES013
LAB004	DES009	DES014
LAB005	DES010	DES015

DES016	DES031	DES046
DES017	DES032	DES047
DES018	DES033	DES048
DES019	DES034	DES049
DES020	DES035	DES050
DES021	DES036	DES051
LAB022	DES037	COM052
LAB023	DES038	COM053
LAB024	DES039	LAB054
LAB025	DES040	COM055
LAB026	DES041	DES056
DES027	DES042	COM057
DES028	DES043	DES058
DES029	DES044	LAB059
DES030	DES045	



### xiii. Edificio de Telecomunicaciones Planta Sótano

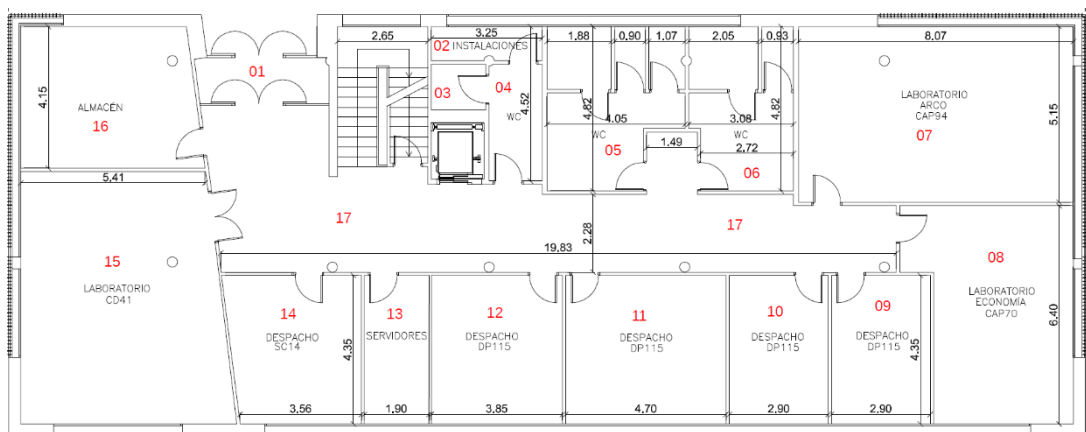


Figura 20: Plano de Edificio de Telecomunicaciones Planta Sótano

Tabla 13: Habitaciones de Edificio de Telecomunicaciones Planta Sótano

CUA001	CUA004	CUA007
CUA002	CUA005	CUA008
CUA003	CUA006	

#### xiv. Edificio de Investigación Comunicaciones Planta 0



*Figura 21: Plano de Edificio de Investigación Comunicaciones Planta Baja*

*Tabla 14: Habitaciones de Edificio de Investigación Comunicaciones Planta Baja*

BASE	COM006	DES012
COM001	LAB007	CUA013
CUA002	LAB008	DES014
CUA003	DES009	LAB015
CUA004	DES010	CUA016
COM005	DES011	COM017

#### xv. Edificio de Investigación Comunicaciones Planta 1

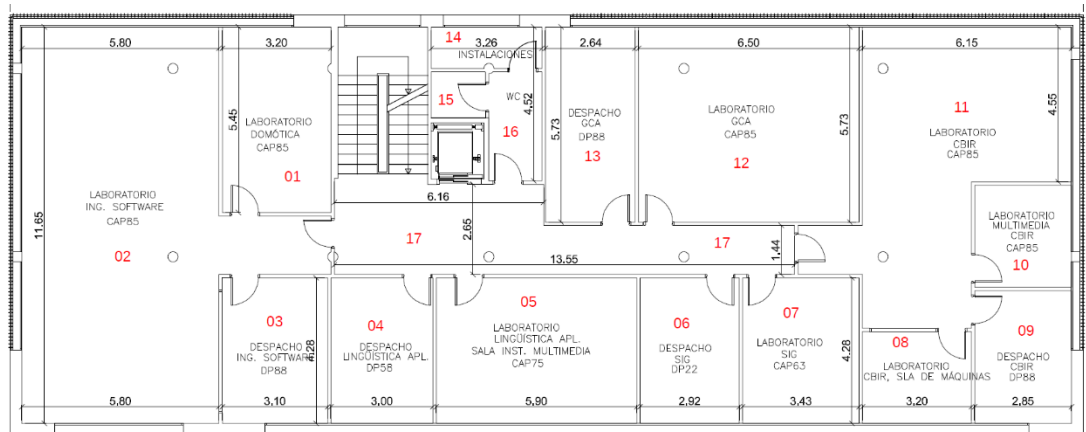


Figura 22: Plano de Edificio de Investigación Comunicaciones Planta 1

Tabla 15: Habitaciones de Edificio de Investigación Comunicaciones Planta 1

LAB001	LAB007	DES013
LAB002	LAB008	CUA014
DES003	DES009	CUA015
DES004	LAB010	CUA016
LAB005	LAB011	COM017
DES006	LAB012	

## xvi. Edificio de Investigación Comunicaciones Planta 2

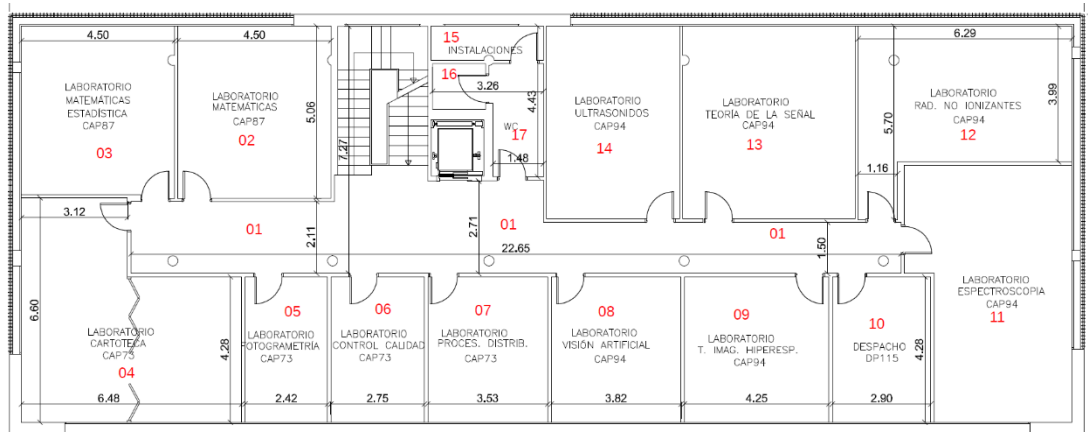


Figura 23: Plano de Edificio de Investigación Comunicaciones Planta 2

Tabla 16: Habitaciones de Edificio de Investigación Comunicaciones Planta 2

COM001	LAB007	LAB013
LAB002	LAB008	LAB014
LAB003	LAB009	CUA015
LAB004	DES010	CUA016
LAB005	LAB011	CUA017
LAB006	LAB012	

*Tabla 17: Habitaciones de Edificio de Investigación Comunicaciones Planta Sótano*

CUA003

Detailed architectural floor plan of the first floor of the 'Edificio de la Facultad de Ingeniería'. The plan shows five rooms, each with its function and dimensions:

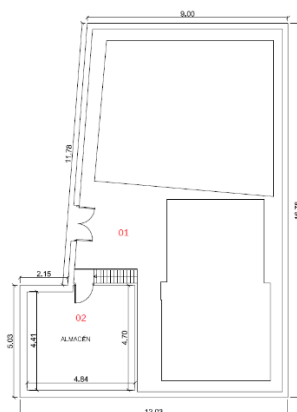
- Room 01:** LABORATORIO ABS. ACÚSTICA CAM. REVERBERANTE. Dimensions: 6.91 (width), 6.39 (depth), 1.76 (entrance width).
- Room 02:** DESPACHO ABS. ACÚSTICA. Dimensions: 4.41 (width), 4.00 (depth), 4.84 (entrance width), 5.00 (entrance depth).
- Room 03:** LABORATORIO ABS. ACÚSTICA CAM. RECEPCIÓN. Dimensions: 5.54 (width), 4.00 (depth), 1.10 (entrance width).
- Room 04:** LABORATORIO ABS. ACÚSTICA CAM. EMISIÓN. Dimensions: 5.00 (width), 6.00 (depth).
- Room 05:** LABORATORIO ABS. ACÚSTICA CAM. REVERBERANTE. Dimensions: 6.91 (width), 6.39 (depth).

Overall building dimensions: 9.00 (total width), 12.03 (total depth), 5.00 (entrance depth), 2.15 (entrance width).

*Tabla 18: Habitaciones de Anexo Edificio de Investigación Comunicaciones Planta Baja*

LAB004

## **xix. Anexo Edificio de Investigación Comunicaciones Planta 1**



*Figura 26: Plano de Anexo Edificio de Investigación Comunicaciones Planta 1*

*Tabla 19: Habitaciones de Anexo Edificio de Investigación Comunicaciones Planta 1*

COM001

CUA002

### **c. Algoritmos utilizados**

En esta parte se detallan los algoritmos utilizados para el motor del cliente y del servidor, así como fragmentos de código y su explicación a la hora de implementarlos. Empezamos por la parte del servidor, la cual es más sencilla

#### **i. Servidor**

El servidor es una aplicación API REST en node.js para procesar las peticiones de la aplicación cliente. Su implementación está distribuida en los ficheros estándar de rutas, controladores, app, index y package.json.

En el fichero *app.js* se inicializan los controladores express, bodyparser app para crear la aplicación que hará de servidor, así como los parámetros correspondientes para procesar las peticiones http. Indicamos, además, que todas las peticiones que se hagan a la API deben ir precedidas de /api/ para organizar mejor.

En el fichero de rutas, ubicado en rutas/lectura.js se inicializan los parámetros de la aplicación y el gestor de rutas, así como las 4 rutas POST de la API que se van a utilizar para procesar las peticiones. A saber:

- `/api/obtenerDatos`: Recibe por parámetros POST un edificio, planta y habitación determinada utilizando la nemotécnica descrita en el apartado anterior y devuelve un json procesado con la petición.
- `/api/obtenerTemperaturas`: Recibe por parámetros POST un edificio y una planta, y devuelve un json con dos parámetros. Dichos parámetros son el nombre de la habitación de la cual existen datos de temperatura y el color correspondiente a la susodicha. De esta forma evitamos a la aplicación cliente calcular el color necesario, haciéndola más ligera al usuario.
- `/api/obtenerHumedades`: Igual que el apartado anterior, pero con los grados de humedad en lugar de la temperatura
- `/api/actualizarNombresHabitación`: Función que recibe tres parámetros POST. A saber, edificio, planta y una clave. Cuando la función se ejecuta, actualiza el fichero de datos *datos.js* de un modelo determinado y le añade el parámetro nombre para cambiar el aspecto visual de la aplicación. Por ejemplo, que en lugar de LAB033 aparezca *Robolab* o que en lugar de LAB041 aparezca *Laboratorio de Física*. Los nombres estarán guardados en el fichero *nombresHabitacion.js*.

### Controlador de lectura

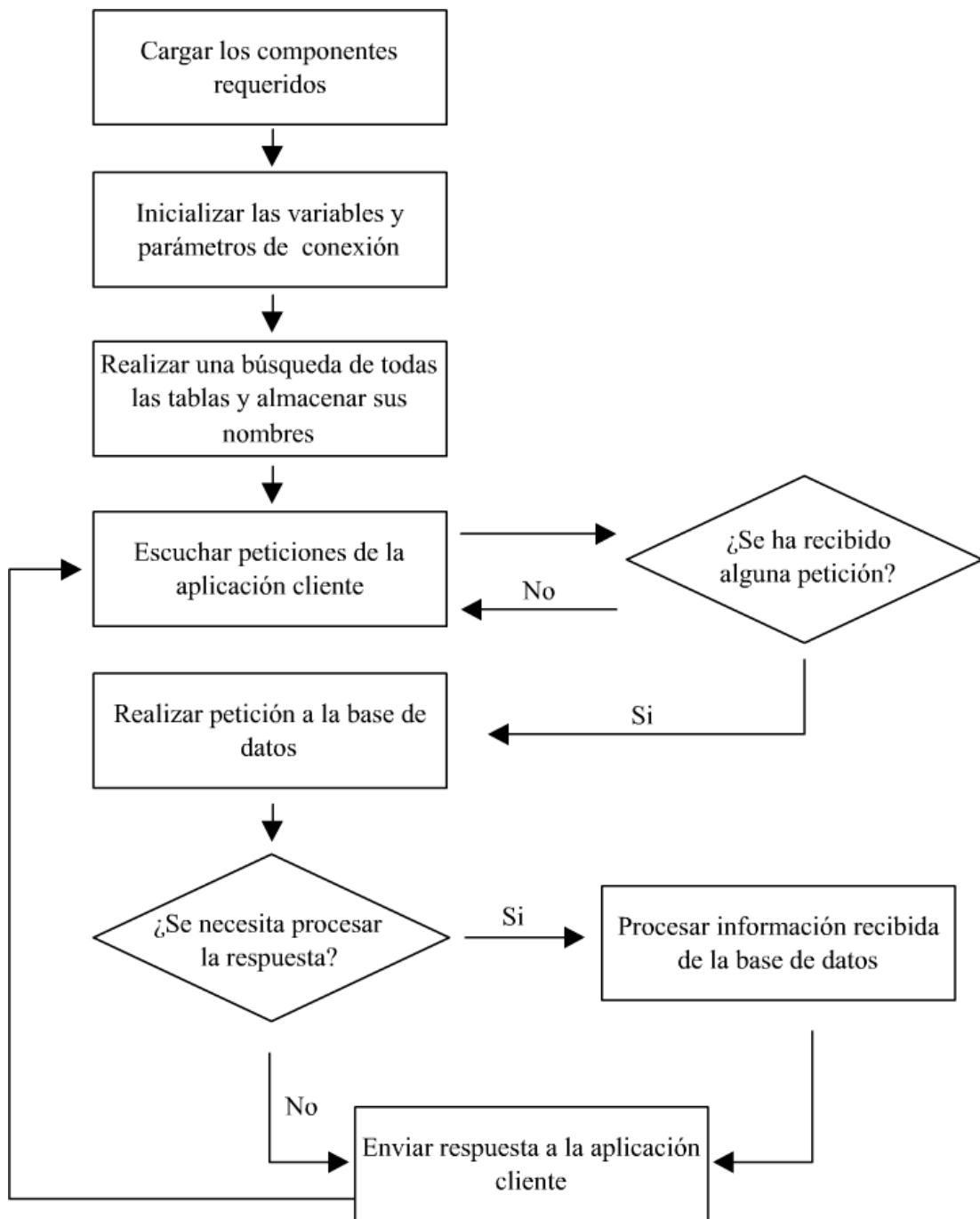


Figura 27: Esquema de funcionamiento de la aplicación servidor

El controlador de lectura implementa todas las funciones descritas anteriormente. Veamos primero las variables globales definidas:

```
var request = require('request');  
var fs = require('fs');  
const Influx = require('influx');
```

```
var path = require("path");

var tablas = [];
var FILTRO_TABLAS = "UEXCC";
var FORMATEO_DATOS = true;
var CLAVE = "GzsRjy2uYrpyJ-oqp38K";

/* Array con los colores para mostrar en tiempo real las
temperaturas o la humedad */
var COLORES = ["#d8d8ff", "#9393ff", "#5a5aff", "#1c1cff",
"#3dff3d", "#01c101", "#ffaa00", "#ff3f00", "#ff0000",
"#840101"];

/* Inicializar los parámetros de conexión */
const influx = new Influx.InfluxDB(
{
    database: 'sensors',
    host: '158.49.112.125/query?pretty=true',
    port: 80,
    username: 'guest',
    password: 'smartpolitech'
})

/* Cargamos todas las tablas disponibles que posean datos de
sensores, para buscar posteriormente las indicadas por
parámetros en la URL */
influx.query('SHOW MEASUREMENTS').then(results =>
{
    let maximo = results.length;

    for (var i = 0; i < maximo; i++)
    {
        if (results[i].name.indexOf(FILTRO_TABLAS) !== -1)
        {
            tablas.push(results[i].name);
        }
    }

    /* Ahora en tablas tenemos una lista con todas las tablas
en las que se puede buscar los sensores de una habitación
escogida*/
}
```



```
console.log("LISTADO DE TABLAS:");  
console.log(tablas);  
console.log("Total de tablas: " + tablas.length);  
});
```

Lo primero es inicializar los componentes necesarios, como son request, fs, influx y path. A continuación, declaramos un array vacío *tablas*, el cual almacenará el nombre de todas las tablas existentes en la base de datos. De esta forma, cuando queramos obtener todos los datos referentes a alguna habitación, busquemos previamente en ese array las tablas correspondientes y de esta manera ahorrarnos siempre un acceso extra a la base de datos para preguntar cuántas tablas hay referentes a esa habitación.

También declaramos un filtro, que en este caso tendrá el valor UEXCC para que a la hora de guardar los nombres de las tablas evitemos almacenar nombres como prueba que no aportan valor alguno y ahorramos tiempo de búsqueda. Activamos también la variable FORMATEO\_DATOS a verdadero para que los datos de algunos sensores aparezcan visualmente más atractivos:

### Laboratorio de Física (LAB041)

Sensor	SEN001_THR
Thu Sep 07 2017 01:02:07 GMT+0200 (Hora de verano romance)	
Temperatura	28°C
Humedad	36.1%
Bateria	2.68V

Figura 28: Ejemplo de formateo activo

### Laboratorio de Física (LAB041)

Sensor	SEN001_THR
Thu Sep 07 2017 01:02:07 GMT+0200 (Hora de verano romance)	
hum	36.1
temp	28
vbat	2.68

Figura 29: Ejemplo de formateo no activo

Y por último inicializamos la variable CLAVE que tendrá la variable con la cual se comparará la clave en la agregación de nombres de habitación. Una vez inicializadas las variables, se procede a la inicialización de los parámetros de conexión a la base de datos influxDB en la cual están almacenados los datos correspondientes a los sensores. Por último, realizamos una consulta para averiguar cuantas tablas disponibles hay y almacenamos en el array tablas, filtrando las que no comienzan por UEXCC. Con esto, ya tenemos el servidor inicializado y listo para escuchar las peticiones de la aplicación cliente.

### Función `getDatos`

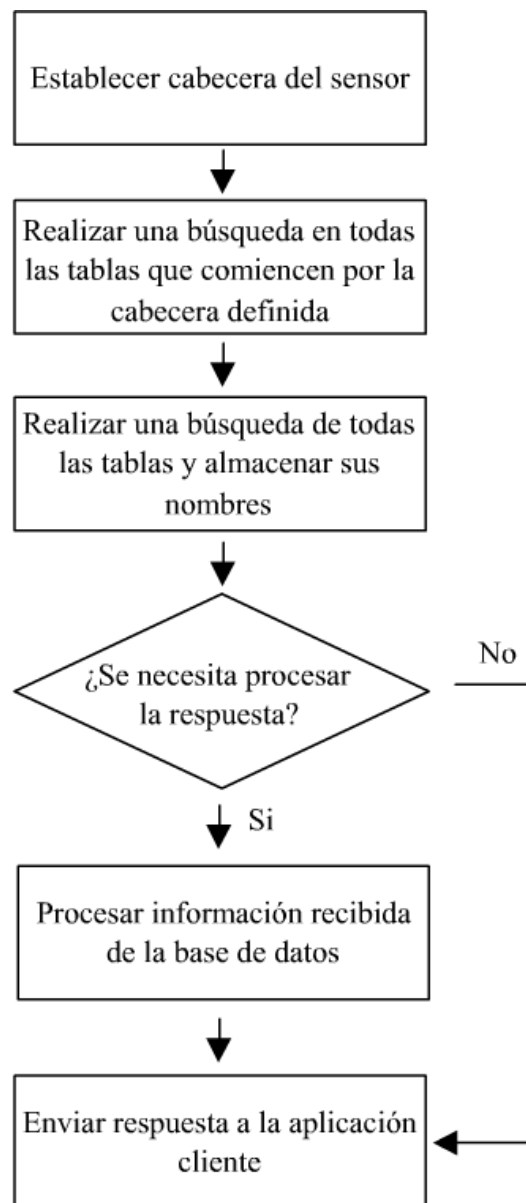


Figura 30: Esquema de funcionamiento de la función `getDatos`

```
function getDatos(req, res)
{
    let sensor = FILTRO_TABLAS + "_" + req.body.edificio +
    "_" + req.body.planta + "_" + req.body.habitacion;
    let tablasABuscar = [];
    let tablasRealizadas = 1;
    var resultados = [];

    for (var i = 0; i < tablas.length; i++)
```

```
{
    /* En caso de que encuentre alguna tabla que
    contenga datos de la habitacion la añadimos */
    if (tablas[i].indexOf(sensor) !== -1)
    {
        tablasABuscar.push(tablas[i]);
    }
}

if (tablasABuscar.length == 0)
{
    res.status(200).send(JSON.stringify({error: "Lo
    sentimos, no existen sensores para la habitción " +
    req.body.habitacion}));
}

/* Con este truco escogemos el último valor capturado
por el sensor (ultima fila de la base de datos) */
for (var i = 0; i < tablasABuscar.length; i++)
{
    let nombreSensor = tablasABuscar[i].substr(21, 20);
    influx.query('SELECT * FROM ' + tablasABuscar[i] +
    ' GROUP BY * ORDER BY DESC LIMIT 1').then(results =>
    {
        let datoSensor = results[0];
        datoSensor["sensor"] = nombreSensor;
        datoSensor["time"] =
        datoSensor["time"].toString();
        resultados.push(datoSensor);

        if (tablasRealizadas < tablasABuscar.length)
        {
            tablasRealizadas++;
        }
        else
        {
            if (FORMATEO_DATOS)
                formatearDatos(resultados, res);
            else
```

```
        res.status(200).send(JSON.stringify(resultados));
    }
    });
}
}
```

La función recibe por parámetros una *request* y una *response*. La primera tarea consiste en crear un filtro para buscar todas las tablas que lo cumplan ya que contendrá la información que se va a mostrar en la parte del cliente. Este filtro consiste en juntar los tres parámetros enviados por la aplicación cliente (pabellón, planta y habitación) junto con el filtro inicial (UEXCC en este caso) para crear el comienzo de las tablas que contengan los datos de esa habitación.

Por ejemplo, si queremos buscar todos los sensores que hay para la habitación LAB033 de la planta 0 del pabellón de informática, el filtro será UEXCC\_INF\_P00\_LAB033. Una vez establecido hacemos una búsqueda por el array de tablas para ver cuales coinciden con ese filtro y las guardamos en otro array, llamado *tablasABuscar*. En caso de no encontrar ninguna enviamos un mensaje de error diciendo que no se han encontrado datos para esa habitación.

Una vez que hemos terminado la búsqueda, realizamos una consulta por cada tabla que cumpla el filtro con la sentencia

```
SELECT * FROM tabla GROUP BY * ORDER BY DESC LIMIT 1
```

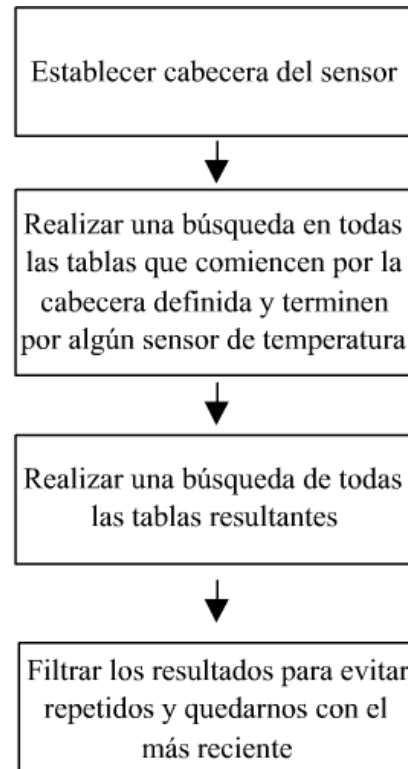
La cual nos envía siempre el último valor añadido en dicha tabla, ya que no todos almacenan datos con la misma periodicidad. También sería posible indicar que nos enviara los valores durante un lapso de tiempo, pero en este caso nos interesa obtener el último almacenado, ya que es el más reciente. Una vez que hemos obtenido el resultado de la consulta, almacenamos en ella también el nombre del sensor utilizado y convertimos el parámetro *time* en un formato más parecido al lenguaje humano.

Para seleccionar el nombre del sensor, utilizaremos el nombre de la tabla restando el inicial filtro. Por ejemplo, si se tratara del sensor UEXCC\_INF\_P01\_LAB024\_SEN001\_THC nos quedaríamos con todos los caracteres a partir del 21 (todos los sensores tienen el mismo número de caracteres, por lo cual es fácil), que en este caso sería SEN001\_THC.

Al tratarse de consultas asíncronas, es necesario crear un mecanismo de control para ir tratándolas según lleguen sin terminar la ejecución de la función. Esto se soluciona estableciendo un bucle con tantas iteraciones como consultas necesite hacer almacenando en un array cara resultado (junto con el añadido del nombre del sensor y el cambio de fecha) y contando cuantas iteraciones se han completado. Una vez llegado al máximo se devuelve el valor de ese array (*resultados*) convertido en JSON al realizar todas las iteraciones. Dicho array puede ser devuelto en bruto o formateado a un texto más amigable al usuario.

### **Función obtenerTemperaturas**

Esta función recibe una *request* y una *response*. Dentro de la *request* tenemos los dos parámetros POST enviados por la aplicación cliente, que son un *edificio* y una *planta*. La función retornará un array en formato JSON con cada una de las habitaciones que tengan sensores de temperatura y cuyos elementos del array están compuestos por el código de habitación y el color que tomará la habitación. Dicho color variará según la temperatura de la habitación, la cual siempre utilizamos la lectura más reciente en caso de haber más de un sensor por habitación. De esta forma, retornando el color y no la temperatura, evitamos que sea la aplicación cliente quien tenga que seleccionarla liberándola de esta tarea, que se ejecuta en el servidor.



*Figura 31: Esquema de funcionamiento de la función obtenerTemperaturas*

```
function obtenerTemperaturas(req, res)
{
    let sensor = FILTRO_TABLAS + "_" + req.body.edificio +
    "_" + req.body.planta;
    let tablasABuscar = [];
    let tablasRealizadas = 1;
    var resultados = [];

    for (var i = 0; i < tablas.length; i++)
    {
        /* En caso de que encuentre alguna tabla que
        pertenezca al edificio y planta seleccionados, comprobamos si
        posee sensor de temperatura */
        if (tablas[i].indexOf(sensor) !== -1)
        {
            /* Escojemos los últimos 3 caracteres de la
            tabla para ver si corresponde a un sensor de temperatura */

            let tipoSensor = tablas[i].substr(-3, 3);
```

```
/* Nos quedamos únicamente con los sensores
que tengan valores de temperatura */
    if (tipoSensor == "THV" || tipoSensor == "TEH"
|| tipoSensor == "THC" || tipoSensor == "THR" || tipoSensor ==
"HV2")
    {
        tablasABuscar.push(tablas[i]);
    }
}

if (tablasABuscar.length == 0)
{
    let datoTemperatura = {};
    datoTemperatura["error"] = "No hay sensores de
temperatura para " + req.body.edificio + "_" +
req.body.planta;

    res.status(200).send(JSON.stringify(datoTemperatura));
}

/* Ahora realizamos la búsqueda en las tablas
seleccionadas */
for (var i = 0; i < tablasABuscar.length; i++)
{
    let nombre = tablasABuscar[i].substr(14, 6);
    influx.query('SELECT temp FROM ' + tablasABuscar[i]
+ ' GROUP BY * ORDER BY DESC LIMIT 1').then(results =>
    {
        let datoTemperatura = {};
        datoTemperatura["habitacion"] = nombre;
        datoTemperatura["temperatura"] =
results[0].temp;
        datoTemperatura["fecha"] = results[0].time;
        resultados.push(datoTemperatura);

        if (tablasRealizadas == tablasABuscar.length)
        {
```



```
        res.status(200).send(JSON.stringify(filtrarTemperaturas(
resultados)));
    }
    else
    {
        tablasRealizadas++;
    }
    });
}
}
```

Lo primero que hacemos es crearnos un filtro que contenga el edificio y la planta, para así seleccionar todas las tablas que pertenezcan a las habitaciones de la misma. Una vez que hemos encontrado alguna habitación correspondiente, activamos un segundo filtro para saber si el sensor que hemos encontrado es de temperatura (THV, TEH, THC, THR o THV2), de esta forma descartamos sensores, por ejemplo, de agua o electricidad que no nos son útiles. Si la tabla encontrada cumple esos dos requisitos, la añadimos a la lista de tablas de la cual vamos a necesitar la información. En caso de no encontrar ninguna, devolvemos un mensaje de error para informar al usuario.

Una vez que tenemos todas las tablas necesarias, realizamos todas las consultas de búsqueda correspondiente y al igual que la función anterior, es necesario llevar un control de la asincronía. Una vez que tenemos la respuesta a una consulta procesamos esa información para crear una estructura formada por 3 elementos:

- Nombre nemotécnico de la habitación: habitación
- Temperatura de la habitación: temperatura
- Fecha de lectura de la temperatura: fecha

Luego, añadimos esa estructura a un nuevo array llamado resultados. Una vez completada la búsqueda devolvemos dicho array en formato JSON, no sin antes ser filtrado para evitar la duplicidad de los valores. Esto ocurre porque algunas habitaciones pueden tener más de un sensor de temperatura, por lo cual nos quedamos siempre con la lectura más reciente. Además, al filtrar el resultado, en lugar de devolver el valor numérico de la temperatura, lo que se devuelve es el color en formato

hexadecimal correspondiente a la temperatura, que será el que tome la habitación en la aplicación cliente. De la función de filtrado, se hablará más adelante.

### **Función obtenerHumedades**

Esta función es exactamente igual a la anterior, pero utilizando como discriminador el grado de humedad en lugar de la temperatura.

### **Función actualizarNombresHabitacion**

Esta función añade los nombres no nemotécnicos de habitaciones al fichero de datos *datos.js* de cada modelo de edificio y planta, para que de esta manera se cargue y se pueda visualizar en la aplicación cliente en lugar del nombre nemotécnico, no conocido por los usuarios. Por ejemplo, que aparezca Robolab en lugar de LAB033 o Laboratorio de Física en lugar de LAB041. Un ejemplo de ese fichero encontrado en la planta 0 del edificio de Informática es el siguiente:

```
1 {  
2   "nombres": [  
3     {"habitacion": "LAB033", "nombre": "RoboLab"},  
4     {"habitacion": "COM003", "nombre": "Cuarto de baño mujeres"},  
5     {"habitacion": "LAB041", "nombre": "Laboratorio de Física"}  
6   ]  
7 }
```

*Figura 32: Ejemplo de fichero nombresHabitacion.js*

Como se puede ver, consta de un array en formato JSON con los elementos habitación y nombre no nemotécnico.

La función recibe por parámetros una request y una response, donde dicha request está formada por 3 parámetros en formato POST que deben ser enviados a la url

SERVIDOR/api/actualizarNombresHabitación

Los tres parámetros que debe recibir son los siguientes:

- Edificio: Nombre del edificio en su formato nemotécnico
- Planta: Planta del edificio en su formato nemotécnico

- Clave: Una clave de seguridad que deben conocer los administradores y así evitar que cualquier usuario no autorizado utilice esta función. En este caso, la clave establecida (totalmente aleatoria) es *GzsRjy2uYrpyJ-oqp38K*

Para enviar estos tres parámetros pueden utilizarse aplicaciones como *Postman* o cualquier otra que permita el envío de parámetros a una URL. Una vez comprobado que todos los parámetros son correctos, se cargan en memoria los dos ficheros (*datos.js* y *nombresHabitacion.js*) y se mezclan, volcando en el fichero *datos.js* la unión de los dos conjuntos de datos.

Se contemplaron más posibilidades para realizar esta función. Una de ellas es que sea el propio conversor de ficheros el que añada los nombres, sin embargo, es posible que estos no existan o no quieran ser mostrados por lo cual creo conveniente que el conversor se limite únicamente a su función de convertir modelos. Otra opción es que los ficheros *datos.js* y *nombresHabitacion.js* se lean por separado al cargar el modelo en la aplicación cliente, pero esto provocaría siempre el doble de lecturas, peticiones y accesos, por lo que no sería una solución eficiente, siendo mejor realizar una única lectura que contenga todos los datos necesarios. La opción finalmente tomada fue la de aprovechar una API REST ya existente para realizar la unión de los dos ficheros, y que además se pueden tener backups del fichero de nombres por si fuera necesario, de forma independiente a los modelos. Veamos el código y como trabaja la función:

```
function actualizarNombresHabitacion(req, res)
{
    var edificio = req.body.edificio;
    var planta = req.body.planta;

    if (req.body.clave != CLAVE)
    {
        res.status(200).send({error: "Error. La clave de
activación no es válida"}});
    }
    else
    {
        var rutaFicheroDatos = "./public/modelos/" + edificio +
"/" + planta + "/datos.js"
        var habitaciones;
        fs.readFile(rutaFicheroDatos, 'utf8', function (err,
data)
        {
```

```
        if (err)
        {
            res.status(200).send({error: "Error. Los
parámetros no son correctos"});
        }
        else
        {
            habitaciones = JSON.parse(data);
            /* Estando aquí ya hemos cargado en
habitaciones todos los metadatos necesarios. Ahora cargamos el
fichero que contiene los nombres, para mezclarlos */

            var rutaFicheroNombres = "./public/modelos/"
+ edificio + "/" + planta + "/nombresHabitacion.js"
            var nombresHabitacion;

            fs.readFile(rutaFicheroNombres, 'utf8',
function (error, nombres)
            {
                if (error)
                {
                    res.status(200).send({error:
"Error. No existe fichero de nombres"});
                }
                else
                {
                    nombresHabitacion =
JSON.parse(nombres).nombres;
                    /* Si estamos aquí, es que todo
funciona correctamente. Ahora, actualizamos el fichero de datos con
los nombres */

                    for (var i = 0; i <
nombresHabitacion.length; i++)
                    {
                        var encontrado = false;
                        for (var j = 0; j <
habitaciones.Edificio.length && encontrado == false; j++)
                        {
                            if
(habitaciones.Edificio[j].habitacion ==
nombresHabitacion[i].habitacion)
                            {
```

```

                                                                    encontrado =
true;

    habitaciones.Edificio[j].nombre = nombresHabitacion[i].nombre;
                                }
                                }
                                }

/* Ahora convertimso ese array en
JSON listo para ser grabado en un fichero y además, lo mostramos por
pantalla */

    habitaciones =
JSON.stringify(habitaciones, null, 2);

    fs.writeFile(rutaFicheroDatos,
habitaciones, 'utf8', function (err)
    {
        if (err)
        {

            res.status(200).send({error: "Error. No se ha podido guardar
el fichero"});

        }
        else
        {

            res.status(200).send(habitaciones);

        }

    });

}

});

}

}
}
```

Lo primero que hace es almacenar en variables edificio y planta los valores enviados en la request para así trabajar con ellos más fácil posteriormente. A continuación, comprueba que la clave enviada coincide que la establecida en la aplicación servidor al inicio de la misma y si no es correcta, informar al usuario. En caso de clave correcta, continuamos con la ejecución.

Después cargamos el fichero *datos.js* del edificio y planta recibidos por parámetro, informando de error si este no se encontrara y cargando en memoria el

fichero en caso correcto. A continuación, realizamos otra lectura del fichero *nombresHabitacion.js* para cargar en memoria su contenido si existe o informando de error en caso de no existir. Una vez los dos cargados, procedemos a mezclarlos, realizando una búsqueda en bucle por cada nombre de habitación existente para buscar la habitación correspondiente del fichero *datos.js* y añadirla cuando se localice.

Una vez completado el nuevo array de datos con los nuevos nombres de habitación procedemos a su volcado en el fichero *datos.js* e informando de error en caso de que esto no se pudiera realizar.

A partir de aquí, las funciones que se van a explicar corresponden a funciones auxiliares utilizadas por las principales ya descritas anteriormente.

### **Función *formatearDatos***

Esta función recibe por parámetro un array de datos leídos que contienen todas las lecturas de los sensores de una habitación concreta y un parámetro *response*, que será el encargado de enviar una respuesta una vez que haya terminado el tratamiento de los datos.

```
function formatearDatos(datosLeidos, res)
{
    var datosFormateados = [];
    var i;

    for (i = 0; i < datosLeidos.length; i++)
    {
        if(datosLeidos[i] != null)
        {
            let datoFormateado = {};

            datoFormateado["sensor"] = datosLeidos[i].sensor;
            delete datosLeidos[i].sensor;

            datoFormateado["time"] = datosLeidos[i].time;
            delete datosLeidos[i].time;

            if (datosLeidos[i].temp)
            {
```

```
        datoFormateado["Temperatura"]
datosLeidos[i].temp + "° C";
        delete datosLeidos[i].temp;
    }

    if (datosLeidos[i].hum)
    {
        datoFormateado["Humedad"]
datosLeidos[i].hum + "%";
        delete datosLeidos[i].hum;
    }

    if (datosLeidos[i].vbat)
    {
        datoFormateado["Bateria"]
datosLeidos[i].vbat + "V";
        delete datosLeidos[i].vbat;
    }

    if (datosLeidos[i].window)
    {
        datoFormateado["Ventana"]
estadoVentana(datosLeidos[i].window);
        delete datosLeidos[i].window;
    }
    else
    {
        if (datosLeidos[i].window == null)
            delete datosLeidos[i].window;
    }

    if (datosLeidos[i].CO2 == null)
        delete datosLeidos[i].CO2;

    /* Copiamos el resto de los valores */
    for(var indice in datosLeidos[i])
    {
        datoFormateado[indice]
datosLeidos[i][indice];
    }

    datosFormateados.push(datoFormateado);
```

```
        }  
    }  
    res.status(200).send(JSON.stringify(datosFormateados));  
}
```

Como se puede observar, es una función bastante sencilla. Recorre el array `datosLeídos` que contiene la información recibida de la base de datos por cada sensor leído y si encuentra resultados con ciertos patrones les da otros valores que se puedan interpretar de forma más sencilla para el usuario. Si no encuentra ninguno, lo deja tal y como está.

### **Función estadoVentana**

```
function estadoVentana(estado)  
{  
    if (estado == 1)  
        return "Cerrada";  
    return "Abierta";  
}
```

Esta función es usada al formatear datos y devuelve una cadena de texto con los estados abierta o cerrada en función de su estado (0 o 1). De esta forma la función de formatear datos que la usa queda en una línea.

### **Función estaFiltrado**

Esta función hace las veces de función de búsqueda para indicar si un elemento existe o no dentro del array. Solo que, en este caso en lugar de comparar una estructura entera, compara solamente uno de los atributos, que es el nombre nemotécnico de la habitación.

### **Función filtrarTemperaturas**

Esta función recibe por parámetros un array formado por estructuras con 3 atributos creado en la función `obtenerTemperaturas`. Dichos atributos son:

- Habitación: habitación a la que hace referencia utilizando su nombre nemotécnico



- Temperatura: temperatura en formato decimal
- Fecha: fecha y hora a la que se realizó la lectura del sensor

Inicializamos un array vacío donde volcaremos los resultados filtrados. Realizamos una búsqueda en bucle, seleccionando la temperatura correspondiente en la iteración y comprobando si está filtrada o no, y en caso de que no esté filtrado realizamos una búsqueda para quedarnos con la lectura más reciente. Esto se hace por que es posible que una habitación tenga más de un sensor de temperatura y de esta forma evitamos conflictos. El resultado de seleccionar la temperatura más reciente devolverá siempre el nombre nemotécnico de la habitación y un color correspondiente a esa temperatura. Todo esto se detallará en la función `temperturaMasActual` que se explica mas adelante.

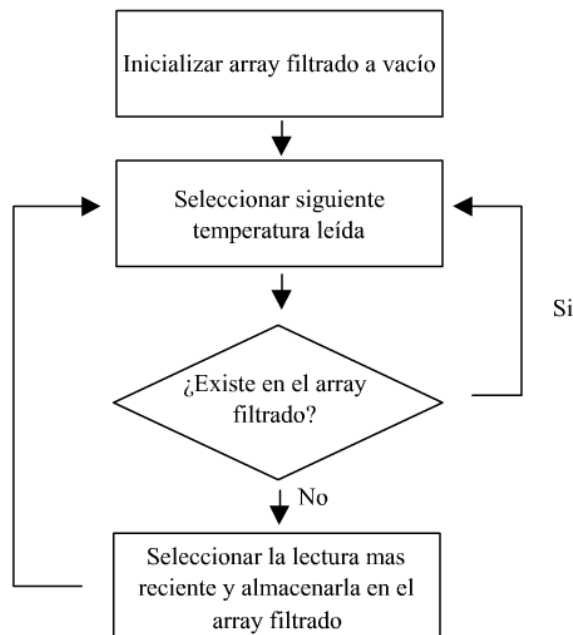


Figura 33: Esquema de funcionamiento de la función `filtrarTemperaturas`

```
function filtrarTemperaturas(arrayTemperaturas)
{
    var temperaturasFiltrado = [];

    for (let i = 0; i < arrayTemperaturas.length; i++)
    {
        /* Comprobamos si existe o no en los resultados
        filtrados */
```

```
        if (!estaFiltrado(arrayTemperaturas[i].habitacion,
temperaturasFiltrado))
        {
            /* En caso de que no esté filtrado, buscamos la
lectura mas actual y la añadimos */
            let datoTemperatura = {};
            datoTemperatura["habitacion"] =
arrayTemperaturas[i].habitacion;
            datoTemperatura["temperatura"] =
temperaturaMasActual(arrayTemperaturas[i], arrayTemperaturas, i);
            temperaturasFiltrado.push(datoTemperatura);
        }
    }
    return temperaturasFiltrado;
}
```

### **Función temperaturaMasActual**

Esta función recibe tres parámetros: Los datos de la habitación (temperatura, nombre nemotécnico y fecha) leídos, el array donde se encuentran todas las lecturas y la posición en la que está el primer parámetro.

Realizamos una búsqueda partiendo desde la posición que nos llega por parámetro, de esta manera nos ahorramos búsquedas ya que vamos reduciendo su número a medida que se van filtrando los resultados. La búsqueda consiste en encontrar la temperatura cuyo valor de fecha sea el mayor (más actual) y una vez que se ha hallado dicho valor retornamos el color correspondiente en formato hexadecimal que corresponde a dicha temperatura, de esta forma ahorramos que la aplicación cliente sea el encargado de seleccionarlo liberándola de esa tarea y haciéndola algo más ligera.

Primera llamada: 10 búsquedas

--	--	--	--	--	--	--	--	--	--

Segunda llamada: 9 búsquedas

--	--	--	--	--	--	--	--	--	--

### Séptima llamada: 3 búsquedas

--	--	--	--	--	--	--	--	--	--

Figura 34: Ejemplo de llamadas con un array de 10 temperaturas

```
function temperaturaMasActual(datosHabitacion, arrayTemperaturas,
origen)
{
    let nombreHabitacion = datosHabitacion.habitacion;
    let masActual = datosHabitacion.fecha;
    let temperaturaActual = datosHabitacion.temperatura;

    /* Realizamos una búsqueda secuencial partiendo desde el
    origen dado por parámetro para quedarnos con el que tenga la fecha
    más alta */
    for (let i = origen; i < arrayTemperaturas.length; i++)
    {
        if (arrayTemperaturas[i].habitacion == nombreHabitacion
        && arrayTemperaturas[i].fecha > masActual)
        {
            temperaturaActual =
arrayTemperaturas[i].temperatura;
            masActual = arrayTemperaturas[i].fecha;
        }
    }

    /* Devolvemos una temperatura redondeada a enteros siempre
    mayor que 0. Este valor será la posición dentro del vector de
    temperaturas en la aplicación, donde están almacenados los tonos de
    colores en función de la temperatura */
    return getColorTemperatura(temperaturaActual);
}
```

### Función getColorTemperatura

Esta función devuelve un color en formato hexadecimal correspondiente a la temperatura. Para ello calculamos la posición que tendría que tener dentro del array realizando una división de la temperatura por 5 y redondeando el resultado a un número entero, y además ese resultado debe ser siempre mayor o igual que 0. De esta forma podremos seleccionar un color diferente por cada 5° de temperatura. El color de

las temperaturas se define cuando se inicializa la aplicación servidor, que en este caso sería:

```
var COLORES = ["#d8d8ff", "#9393ff", "#5a5aff", "#1c1cff", "#3dff3d",  
"#01c101", "#ffaa00", "#ff3f00", "#ff0000", "#840101"];
```

que se compone de 4 tonos azules para temperaturas frías, 2 tonos verdes para temperaturas intermedias y 4 colores rojos para temperaturas altas. Además, este mismo array también es utilizado para los grados de humedad, compartiendo paleta.

Una vez que se ha comprobado que el valor es mayor que 0, hay que comprobar que el resultado tampoco de un índice fuera del array. Realizadas estas operaciones, se retorna el valor que contendrá el color que también será el retorno de la función anterior (*temperaturaMasActual*).

```
function getColorTemperatura(temperatura)  
{  
    let posicionArrayTemperaturas = Math.max(Math.floor(temperatura  
/ 5), 0);  
  
    if (posicionArrayTemperaturas >= COLORES.length)  
    {  
        posicionArrayTemperaturas = COLORES.length - 1;  
    }  
    return COLORES[posicionArrayTemperaturas];  
}
```

### **Funciones filtrarHumedades, humedadsMasActual y getColorHumedad**

Estas funciones están implementadas de la misma forma que sus homólogas de temperatura. Solo se diferencia la función `getColorHumedad` con respecto a temperatura que en lugar de intervalos de 5 grados, se mide en intervalos de 10%. Es decir, en lugar de dividir por 5, realizamos una división por 10. El resto es exactamente igual por lo que no es necesario redundar en sus explicaciones.

Con esto ya tendríamos elaborada la aplicación servidor, que permanece a la escucha de las peticiones de la aplicación cliente. A continuación, detallaremos las funciones de la aplicación cliente.

## ii. Cliente

La aplicación cliente está almacenada dentro de la carpeta public del servidor, que serán los únicos ficheros accesibles al usuario.

El HTML de la aplicación cliente carga los scripts necesarios (jQuery, Bootstrap, Three.js y render de la aplicación) y está dividida en 3 secciones: Una barra de navegación superior, un contenedor ocupando  $\frac{3}{4}$  de la parte visual y otro contenedor ocupando la cuarta parte restante. En el primer contenedor se muestra el modelo 3D elegido para poder interactuar con él y el segundo contenedor muestra los datos leídos de los sensores de una habitación seleccionada en el modelo del primer contenedor, así como la información referente al edificio y planta mostrados.

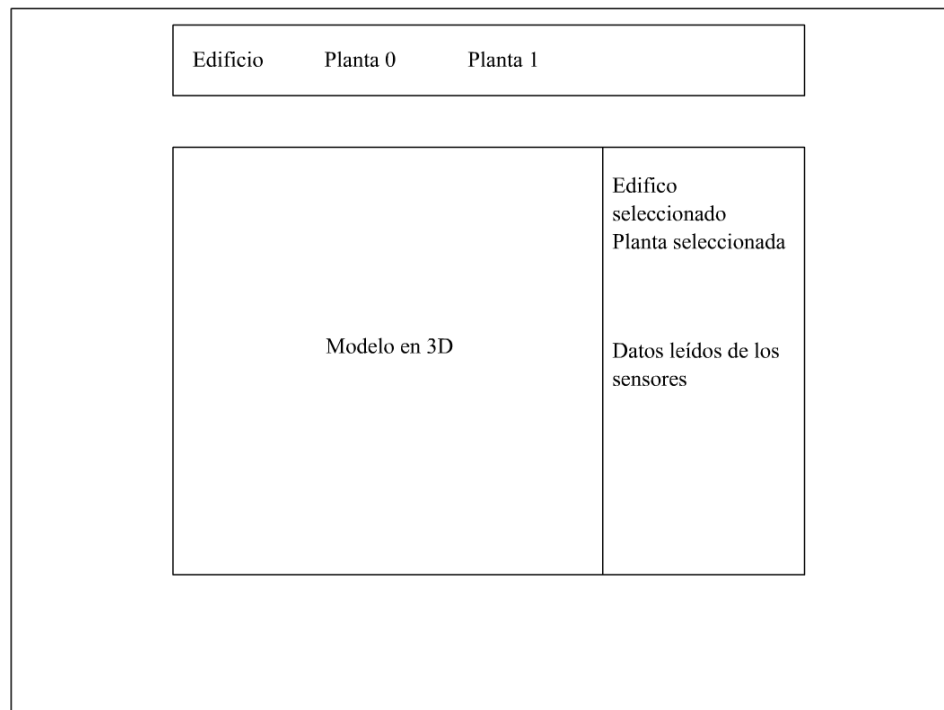


Figura 35: Esquema visual de la aplicación cliente

Los scripts más importantes que se cargan son *three.min.js* que contiene las funciones principales de la librería gráfica *Three.js* y *render.js*, que contiene el motor utilizado en la aplicación. Se puede consultar la documentación referente a *Three.js* en

la bibliografía añadida al final, mientras que aquí ahondaremos en el funcionamiento del script motor de la aplicación.

### **Inicio de la aplicación**

Lo primero que hacemos al inicio de es definir todas las variables de ámbito global que se van a utilizar que se van a utilizar, ya sean las necesarias para la ejecución del 3D como para el control de la aplicación. Vamos a comentar las más importantes:

- Ecenario, teclado, controls, Camara y Render son las utilizadas para el control del 3D de la aplicación, requeridas por Three.js
- Loader: se encarga de cargar los ficheros de modelo en la aplicación
- pabellonActivo y plantaActiva: Indican el edificio y planta con las cuales se está trabajando en ese momento en la aplicación.
- TemperaturasPared y humedadesPared: inicializadas a falso, indican si en ese momento se han sustituido los materiales originales de las habitaciones por el correspondiente a su temperatura o humedad. Si está en falso, el modelo se muestra tal y como es.
- backupDeMaterialesOriginales: Es un array en el que se guarda una copia de seguridad de los materiales que se han sustituido por un color de temperatura o humedad. Inicialmente está vacío y se rellena cuando se procesa el cambio.
- temperaturasPorHabitacion y humedadesPorHabitacion: Cada cierto tiempo se rellenan con los colores que deberían tomar las habitaciones según su nivel de temperatura o grado de humedad. Está formado por estructuras de dos elementos, el nombre nemotécnico de la habitación y el color que debería tomar.
- cambiandoColoresHabitacion: Inicializada a falso, es un FLAG que se activa cuando se está realizando el proceso de cambio de materiales originales por el color correspondiente a temperatura o humedad. Esto es así debido a que si pasamos el ratón por encima de una habitación cambia de color, evitando que el color hover se almacene en el backup de materiales originales. Mientras el FLAG está activo, no es posible realizar ningún cambio visual en el modelo.

- Projector, raycaster, INTERSECTED, INTERSECTEDMATERIAL y TARGET: Variables utilizadas para la función de hover (cambiar color cuando pasamos el ratón por encima de una habitación y que sea fácil de seleccionar).
  - INTERSECTED es un puntero que marca el objeto por el cual se ha pasado el ratón por encima.
  - INTERSECTEDMATERIAL guarda un backup del material original de la habitación mientras se realiza el hover.
  - TARGET es un puntero que almacena la habitación seleccionada al hacer click en ella, que será la almacenada en INTERSECTED en ese momento. Utilizada para saber que habitación se necesita pedir datos al hacer click en ella.
- INTERVALO\_TIEMPO: Indica cada cuanto tiempo se realiza una petición al servidor para guardar las temperaturas o grados de humedad en tiempo real y estar siempre actualizado.

A continuación de definir las variables, se definen los listeners encargados de los eventos, ya sea para cambiar el edificio seleccionado, planta seleccionada o botones de temperatura/humedad en tiempo real.

```
// Render
var Render = new THREE.WebGLRenderer();

var ancho = $("#render").width();
var alto = $("#render").height();

// Escenario
var Escenario;

//Teclado
var teclado;

//Posiciones buenas de la cámara
var CAMX = 33, CAMY = 47, CAMZ = 28;
var CAMRX = -0.7408537006636182, CAMRY = 0.29919445016969104, CAMRZ
= 0.2633346990493383;
```

```
// Controles y cámara
var controls, Camara;

// Cargador de modelos
var loader = new THREE.JSONLoader();

/* Variables para mostrar el modelo */
var pabellonActivo, plantaActiva;

var temperaturasPared = false;
var humedadesPared = false;
var backupDeMaterialesOriginales = [];

var temperaturasPorHabitacion = [];
var humedadesPorHabitacion = [];

/* Esta variable permanecerá verdadera mientras dura el proceso de
cambiar los materiales de las temperaturas a tiempo real. Se hace de
esta manera para evitar errores a la hora de seleccionar una
habitacion mientras esta está cambiando sus materiales */
var cambiandoColoresHabitacion = false;

/* Hover */
var projector, mouse = { x: 0, y: 0 }, INTERSECTED,
INTERSECTEDMATERIAL, TARGET;
var raycaster = new THREE.Raycaster();

/* Colores */
var SELECCIONADO; /* Color cuando pasa el ratón por encima de una
habitación */

/* Si está vacío, es por que el servidor de lectura está en la misma
IP */
var URL_SERVIDOR = "";

/* Intervalo de tiempo por el que se hacen peticiones para averiguar
la temperatura y humedad */
var INTERVALO_TIEMPO;

/* URL para la lectura de datos */
SERVIDOR_LECTURA_DATOS = URL_SERVIDOR + "/api/obtenerDatos";
SERVIDOR_LECTURA_TEMPERATURAS = URL_SERVIDOR +
"/api/obtenerTemperaturas";
```



```
SERVIDOR_LECTURA_HUMEDADES = URL_SERVIDOR + "/api/obtenerHumedades";
SERVIDOR_LECTURA_NOMBRES = URL_SERVIDOR +
"api/obtenerNombresHabitacion";

var cargados = 0;

/* Botones de la interfaz */
$("#pabellonInformatica").click(function(){ pabellonActivo = "INF";
cambiarPiso();});
$("#pabellonCentral").click(function(){ pabellonActivo = "SCO";
cambiarPiso();});
$("#pabellonObrasPublicas").click(function(){ pabellonActivo =
"OPU"; cambiarPiso();});
$("#pabellonArquitectura").click(function(){ pabellonActivo = "ATE";
cambiarPiso();});
$("#pabellonTeleco").click(function(){ pabellonActivo = "TEL";
cambiarPiso();});
$("#pabellonInvestigacion").click(function(){ pabellonActivo =
"INV"; cambiarPiso();});
$("#pabellonComunicaciones").click(function(){ pabellonActivo =
"AIN"; cambiarPiso();});
$("#planta0").click(function(){ plantaActiva = "P00";
cambiarPiso();});
$("#planta1").click(function(){ plantaActiva = "P01";
cambiarPiso();});
$("#planta2").click(function(){ plantaActiva = "P02";
cambiarPiso();});
$("#sotano").click(function(){ plantaActiva = "PS1";
cambiarPiso();});
$("#temperaturasPared").click(activacionTemperaturas);
$("#humedadesPared").click(activacionHumedades);
// función que se ejecuta cuando se mueve el ratón, y que sirve para
el hover
document.addEventListener( 'mousemove', onDocumentMouseMove, false
);
```

## **Función Inicio**

Esta función no recibe ningún parámetro y es la encargada de inicializar todas las variables necesarias para el funcionamiento. Lo primero es asignar un objeto del HTML como renderizador, que en este caso es la división de  $\frac{3}{4}$  partes de la parte visual. A continuación, inicializamos el escenario, teclado y cámara, añadiéndola al escenario. Después inicializamos las luces dentro y establecemos que el Edificio de Informática Planta 0 será el inicial, mostrándolo. Por último, terminamos de cargar el resto de

variables y objetos necesarios para el funcionamiento como son los controles de ratón (Orbit controls), proyector, color de la habitación seleccionada y el intervalo de tiempo, establecido en 1 minuto.

```
function inicio()
{
    // Cambiamos el tamaño del render y lo agregamos
    Render.setSize(ancho, alto);
    document.getElementById("render").appendChild(Render.domElement);

    Camara = new THREE.PerspectiveCamera(45, ancho / alto, 1, 1000);

    // Inicialización del Escenario
    Escenario = new THREE.Scene();

    //Teclado
    teclado = new THREE.KeyboardState();

    Camara.useTarget = false;
    Camara.position.set(parseFloat(CAMX), parseFloat(CAMY),
parseFloat(CAMZ));
    Camara.rotation.set(CAMRX, CAMRY, 0);
    console.log("Todo Cargado");

    // Agregamos la cámara al escenario
    Escenario.add(Camara);

    // Cargamos la luz
    cargarLuz();

    /* Establecemos por defecto el pabellón de informática planta
0*/
    pabellonActivo = "INF";
    plantaActiva = "P00";

    // Cargamos el modelo de uno de los pabellones
    mostrarPiso();

    /* Controlador de cámara para el ratón */
    controls = new THREE.OrbitControls(Camara, Render.domElement);
```

```
controls.update();

// Para impedir que se puedan mover los modelos con las teclas
controls.enableKeys = false;

// initialize object to perform world/screen calculations
projector = new THREE.Projector();

/* Color para el hover al ir seleccionando */
SELECCIONADO = new THREE.MeshLambertMaterial({ color: 0x668899}
);

/* Definimos el intervalo de tiempo en milisegundos. 60000 = 1
minuto */
INTERVALO_TIEMPO = 60000;
}
```

## Función onDocumentMouseMove

Esta función es la encargada de cambiar los colores de una habitación cuando pasamos el ratón por encima. El algoritmo de funcionamiento es el siguiente:

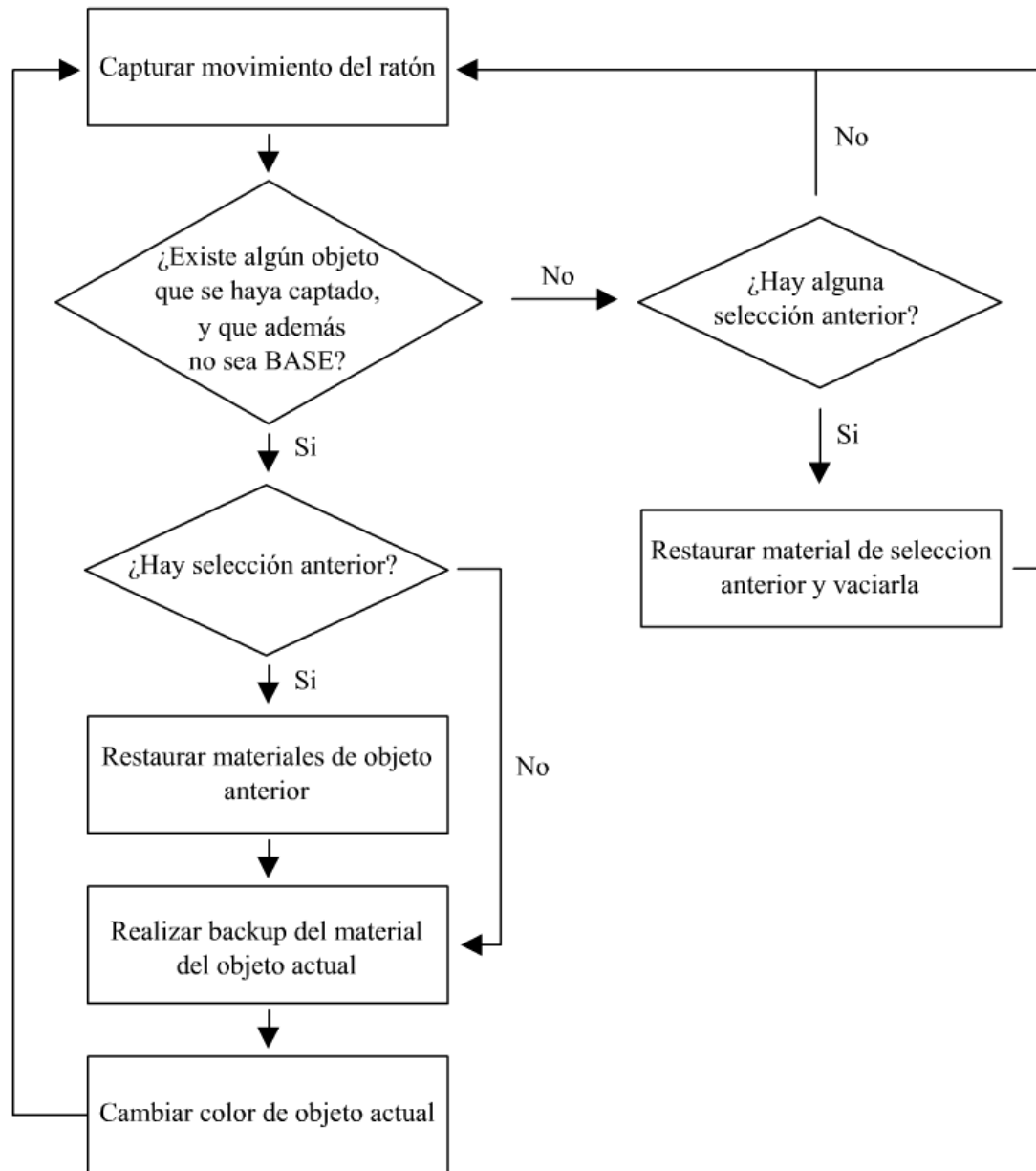


Figura 36: Algoritmo de funcionamiento de la función Hover

```

function onDocumentMouseMove( event )
{
    /* FLAG DE SEGURIDAD */
    if (!cambiandoColoresHabitacion)
    {
        // Obtenemos primero la posición del div que tiene el
        renderizador
        var rect = Render.domElement.getBoundingClientRect();
    }
}
    
```

```
// Calculamos la posición del ratón
mouse.x = ( ( event.clientX - rect.left ) / rect.width )
* 2 - 1;
mouse.y = - ( ( event.clientY - rect.top ) / rect.height
) * 2 + 1;

raycaster.setFromCamera( mouse, Camara );

intersects = raycaster.intersectObjects(
Escenario.children );

if ( intersects.length > 0)
{
    /* Preguntamos si el objeto nuevo que se ha pasado
    el ratón por encima es diferente al anterior y que además no
    corresponda al modelo BASE

    BASE es la parte de modelo de la cual no hay
    datos, por lo que no es necesario interactuar con el */
    if ( intersects[0].object != INTERSECTED &&
intersects[0].object.habitacion != "BASE")
    {
        // Restauramos el color original del
anterior (Si existe)
        if ( INTERSECTED )
        {
            if
(Array.isArray(INTERSECTEDMATERIAL))
            {
                INTERSECTED.material =
INTERSECTEDMATERIAL.slice();
            }
            else
            {
                INTERSECTED.material =
INTERSECTEDMATERIAL;
            }
        }

        // Almacenamos en INTERSECTED el nuevo
objeto al cual se le ha pasado el ratón por encima
        INTERSECTED = intersects[0].object;

        // Guardamos el material del nuevo
seleccionado. El cual puede ser un array de materiales o un color
```

sólido (Como es el caso de las temperaturas o humedades en tiempo real)

```
        if (Array.isArray(INTERSECTED.material))
        {
            INTERSECTEDMATERIAL =
INTERSECTED.material.slice();
        }
        else
        {
            INTERSECTEDMATERIAL =
INTERSECTED.material;
        }

        INTERSECTED.material = SELECCIONADO;
    }
}
else // En caso de no encontrar nada
{
    // Restauramos el material de la intersección
    previa (en caso de que exista)
    if ( INTERSECTED )
    {
        if (Array.isArray(INTERSECTEDMATERIAL))
        {
            INTERSECTED.material =
INTERSECTEDMATERIAL.slice();
        }
        else
        {
            INTERSECTED.material =
INTERSECTEDMATERIAL;
        }
    }
    // Eliminamos la intersección previa
    INTERSECTED = null;
}
}
}
```

Lo primero que hay que tener en cuenta que Three.js está pensado para aplicaciones que ocupen el 100% de la ventana, por lo que en caso de no ser así es necesario obtener la posición del div encargado de renderizar y escalar el click donde se ha producido el evento de ratón en función de esa posición. Esas operaciones se

realizan al principio estableciendo `mouse.x` y `mouse.y` con el resultado de escalar y trazando un rayo virtual entre el click y el Escenario de la aplicación, la cual nos devolverá los objetos que se encuentran presentes en ese rayo.

Una vez que tenemos esos objetos (dentro del array *intersects*) seleccionamos el primero en caso de que existiera alguno y comprobamos que además de que no coincida con el objeto seleccionado anteriormente no sea el modelo BASE (del cual no hay datos). Si es el mismo que la selección anterior no hacemos nada, pero si se trata de un objeto diferente restauramos el material de la selección anterior preguntando primero si lo que vamos a copiar es un color único (un objeto sencillo, o que se haya cambiado el color por temperaturas/humedad) o un array.

Restaurado ya el material original, procedemos a actualizar el objeto seleccionado con el ratón asignando el puntero *INTERSECTED* y realizando un backup de sus materiales en *INTERSECTEDMATERIAL*. A continuación, cambiamos el color del material seleccionado por el establecido en la función de inicio para color de hover.

En caso de que no se haya encontrado ningún nuevo objeto, como es por ejemplo al estar el ratón dentro de los menús, datos de sensores o espacio vacío del modelo preguntamos si había una selección anterior para así restaurar su material original y vaciar el contenido del puntero *INTERSECTED*.

### **Función cargarLuz**

```
function cargarLuz()  
{  
    var luz = new THREE.PointLight(0xffffff);  
    luz.position.set(-100, 200, 100);  
    Escenario.add(luz);  
  
    var luz2 = new THREE.PointLight(0xffffff);  
    luz2.position.set(100, 0, 0);  
    Escenario.add(luz2);  
}
```

Función muy sencilla que establece dos puntos de luz blancos dentro del escenario, necesario para visualizar el modelo en 3D

### **Función animación**

Función importante y necesaria de callback que se encarga de gestionar el motor gráfico de Three.js. Esta función se ejecuta de forma ininterrumpida para visualizar el 3D.

```
function animacion()  
{  
    requestAnimationFrame(animacion);  
    // Agregamos todo el escenario y la cámara al render  
    Render.render(Escenario, Camara);  
}
```

### **Función eliminarPiso**

Esta función se encarga de eliminar todos los objetos presentes en la escena activa, para posteriormente cargar un nuevo modelo de planta o edificio

```
function eliminarPiso()  
{  
    while(Escenario.children.length > 0)  
    {  
        Escenario.remove(Escenario.children[0]);  
    }  
}
```

*Escenario.children* es un array donde se almacenan todos los objetos presentes en la escena, por lo que vamos eliminando con la función *remove* cada objeto presente en ella y de esa forma vaciarla.

### **Función mostrarPiso**

Función de carga en la aplicación cliente el modelo elegido definido por el edificio y planta actuales. Realiza una petición AJAX para obtener los datos y las URLs donde están almacenados todos los submodelos de habitaciones y BASE del edificio y planta escogidos.

```
function mostrarPiso()  
{  
    let urlCarpetaModelo = "modelos/" + pabellonActivo + "/" +  
    plantaActiva + "/";  
    let urlModelo = urlCarpetaModelo + "datos.js";
```



```
$.ajax({
    dataType: "text",
    url: urlModelo,

    success: function(respuesta)
    {
        ficheros = JSON.parse(respuesta);

        for (i = 0; i < ficheros.Edificio.length; i++)
        {
            let habitacion =
ficheros.Edificio[i].habitacion;

            let nombre = (ficheros.Edificio[i].nombre) ?
ficheros.Edificio[i].nombre : "";

            loader.load(urlCarpetaModelo +
ficheros.Edificio[i].nombreFichero, function (geometry, materials)
            {
                let object = new THREE.Mesh(geometry,
materials);

                object.habitacion = habitacion;

                /* Esta variable se usa para saber si
se le ha cambiado el color con las temperaturas o humedad, ya que si
está en verdadero existe un backup de los colores originales */
                object.colorCambiado = false;

                /* Nombre no nemotécnico en caso de
que exista */

                object.nombre = nombre;
                Escenario.add(object);

            });
        }
    }
});
```

La primera tarea que realiza es calcular la URL que debería tener el fichero datos.js del modelo escogido para luego realizar una lectura del mismo. Una vez leído convertimos la respuesta en un array de JavaScript, el cual en cada iteración realizaremos una carga del submodelo a la escena, así como asignar sus propiedades pertinentes: nombre nemotécnico, nombre no nemotécnico si hubiera o una cadena vacía en caso de no existir y una propiedad que indica si este objeto ha cambiado su

material, con valor inicial falso. Esta propiedad estará con valor true en caso de que su material haya sido cambiado ya sea por temperaturas o por humedad.

### **Función segundaPlanta**

Esta función comprobará que el edificio escogido posea una segunda planta, como es el caso del Edificio de Usos Comunes o el Edificio de Investigación. Si es cierto, mostramos el botón segunda planta en la barra de menú y si no lo es, ocultamos el botón y además establecemos la planta activa a planta baja, para evitar una carga de un modelo inexistente. Este caso puede darse debido a tener previamente seleccionada una segunda planta y cambiar el modelo de edificio a mostrar.

```
function segundaPlanta()
{
    if (pabellonActivo == "SCO" || pabellonActivo == "INV")
    {
        $("#botonSegundaPlanta").show();
    }
    else
    {
        $("#botonSegundaPlanta").hide();

        if (plantaActiva == "P02")
            plantaActiva = "P00";
    }
}
```

### **Función sotano**

La acción es similar a la función anterior, pero comprobando si el edificio escogido posee sótano o no, y en caso de no tenerlo establece la planta activa a planta baja si estuviera activa la planta 2.

```
function sotano()
{
    if (pabellonActivo == "INF" || pabellonActivo == "TEL" ||
pabellonActivo == "INV")
    {
        $("#botonSotano").show();
    }
}
```

```
    }  
    else  
    {  
        $("#botonSotano").hide();  
  
        if (plantaActiva == "PS1")  
            plantaActiva = "P00";  
    }  
}
```

### **Función click**

Aquí añadimos un listener para el click dentro del objeto render. De esta manera, si hacemos click dentro de él comprobará si hay alguna intersección realizada con el ratón. De ser así, activamos el puntero TARGET a la habitación intersecada por el ratón de la cual mostramos el nombre de la habitación en la interfaz de usuario y posteriormente enviamos una petición para obtener los datos de los sensores asociados a esa habitación.

```
$("#render").click(function()  
{  
    if(!cambiandoColoresHabitacion)  
    {  
        if (INTERSECTED !=null && INTERSECTED.habitacion !=  
"BASE")  
        {  
            TARGET = INTERSECTED;  
            if (TARGET.nombre != "")  
            {  
                $("#nombreHabitacion").html("<strong>" +  
TARGET.nombre + "</strong> (" + TARGET.habitacion + ")");  
            }  
            else  
            {  
  
                $("#nombreHabitacion").html(TARGET.habitacion);  
            }  
            obtenerDatos();  
        }  
    }  
});
```

### **Función obtenerDatos**

Función que envía una petición POST al servidor para obtener los datos de lectura de sensores correspondientes al edificio, planta y habitación seleccionado. Una vez obtenida la respuesta, la convertimos a un formato de JavaScript y ejecutamos la función actualizarDatosSensores que se encarga de mostrar en la interfaz los datos recibidos.

```
function obtenerDatos()
{
    /* Primero, comprobamos si hay alguna sala targeteada */
    if (TARGET)
    {
        $.ajax({
            type: "POST",
            url: SERVIDOR_LECTURA_DATOS,
            data: {
                'habitacion': TARGET.habitacion,
                'planta': plantaActiva,
                'edificio': pabellonActivo
            },

            success: function(respuesta)
            {
                var informacionSala = JSON.parse(respuesta);
                actualizarDatosSensores(informacionSala);
            }
        });
    }
}
```

### **Función actualizarDatosSensores**

Se encarga de mostrar los datos recibidos por los sensores de una habitación en pantalla, ya sean datos correctos o un mensaje de error.

```
function actualizarDatosSensores(datos)
{
    /* Vaciamos la tabla para añadir nuevos elementos */
```

```
$("#tablaDatos tr").remove();
$("#tablaDatos").append("<tr></tr>");

/* Si hay algún mensaje de error, lo mostramos y si no,
mostramos el contenido de los datos recibidos */
if (datos.error)
{
    $('#tablaDatos tr:last').after('<tr><td colspan="2"
class="alert alert-danger">' + datos.error + '</td></tr>');
}
else
{
    for (var i = 0; i < datos.length; i++)
    {
        /* Mostramos primero los datos mas relevantes*/
        $('#tablaDatos tr:last').after('<tr
class="nombreSensor" ><td>Sensor</td><td>' + datos[i].sensor +
'</td></tr>');

        $('#tablaDatos tr:last').after('<tr
class="fechaSensor" ><td colspan="2">' + datos[i].time +
'</td></tr>');

        for(var indice in datos[i])
        {
            if (indice != "sensor" && indice != "time")
                $('#tablaDatos tr:last').after('<tr><td>' +
indice + '</td><td>' + datos[i][indice] + '</td></tr>');
        }
    }
}
```

La primera tarea que hay que hacer es vaciar la tabla donde se van a mostrar los datos, para evitar que se vayan superponiendo. Una vez vacía, comprobamos si la respuesta recibida corresponde a un mensaje de error y si es así lo mostramos por pantalla.

En caso de no recibir un mensaje de error, realizamos un bucle por cada sensor recibido donde mostramos primero el nombre del mismo, luego la fecha de lectura y posteriormente los datos de lectura. Estos datos pueden estar formateados o no, según se haya establecido en la aplicación servidor.

## **Función actualizarInformacionEdificioInterfaz**

Su tarea es muy sencilla y únicamente muestra en la interfaz de usuario el edificio y planta escogidos, para que en cualquier momento pueda saberse en cual estás.

```
function actualizarInformacionEdificiodInterfaz()
{
    let tituloEdificio, plantaEdificio;

    switch(pabellonActivo)
    {
        case "ATE":
            tituloEdificio = "Pabellón de Arquitectura";
            break;
        case "INF":
            tituloEdificio = "Pabellon de Informática";
            break;
        case "INV":
            tituloEdificio = "Edificio de Investigación";
            break;
        case "OPU":
            tituloEdificio = "Pabellón de Obras Públicas";
            break;
        case "SCO":
            tituloEdificio = "Edificio de Usos Comunes";
            default:
        case "TEL":
            tituloEdificio = "Pabellón de Telecomunicaciones";
            break;
    }

    switch(plantaActiva)
    {
        case "P00":
            plantaEdificio = "Planta Baja";
            break;
        case "P01":
            plantaEdificio = "Primera Planta";
            break;
        case "P02":
```

```
        plantaEdificio = "Segunda Planta";
    break;
    case "PS1":
        plantaEdificio = "Planta Sótano 1";
    break;
}

$("#tituloEdificio").html("<h2>" + tituloEdificio + "</h2>");
$("#plantaEdificio").html("<h3>" + plantaEdificio + "</h3>");
}
```

### **Función obtenerTemperaturas**

Función que realiza una petición POST al servidor con edificio y planta como parámetros y recibe una respuesta que puede ser:

- Error, en caso de no encontrarse sensores de temperatura
- Un array en formato JSON donde cada elemento tiene dos atributos, el nombre nemotécnico de la habitación y el color que debería tener.

Una vez que se han recibido, se almacena la respuesta en un array global llamado `temperaturasPorHabitacion` para poder alternar su muestra o no sin tener que realizar nuevas peticiones. En caso de recibir un error, vaciamos el array para evitar problemas.

```
function obtenerTemperaturas()
{
    $.ajax({
        type: "POST",
        url: SERVIDOR_LECTURA_TEMPERATURAS,
        data: {
            'planta': plantaActiva,
            'edificio': pabellonActivo
        },

        success: function(respuesta)
        {
            /* Vaciamos primero el array por si acaso */
            temperaturasPorHabitacion = [];

            /* Se rellena el Array con los nuevos colores
            respecto a la temperatura */
        }
    });
}
```

```
temperaturasPorHabitacion = JSON.parse(respuesta);

/* Comprobamos antes si hay algún problema con los
datos recibidos. En caso de no encontrar nada, devuevle un error que
se muestra y vaciamos el array */
if (temperaturasPorHabitacion.error)
{
    console.info("ADVERTENCIA: " +
temperaturasPorHabitacion.error);
    temperaturasPorHabitacion = [];
}

});
}
```

### **Función obtenerHumedades**

Realiza la misma tarea que la función anterior, pero obteniendo el color referente al grado de humedad en lugar de temperatura.

```
function obtenerHumedades()
{
    $.ajax({
        type: "POST",
        url: SERVIDOR_LECTURA_HUMEDADES,
        data: {
            'planta': plantaActiva,
            'edificio': pabellonActivo
        },

        success: function(respuesta)
        {
            /* Vaciamos primero el array por si acaso */
            humedadesPorHabitacion = [];

            /* Se rellena el Array con los nuevos colores con
            respecto al grado de humedad */
            humedadesPorHabitacion = JSON.parse(respuesta);

            /* Comprobamos antes si hay algún problema con los
            datos recibidos. En caso de no encontrar nada, devuevle un error que
            se muestra y vaciamos el array */
            if (humedadesPorHabitacion.error)
            {

```



```
                console.info("ADVERTENCIA: " +
humedadesPorHabitacion.error);
                humedadesPorHabitacion = [];
            }
        }
    });
}
```

### **Función cambiarColorHabitacion**

Esta función recibe 3 parámetros:

- habitacion: nombre de la habitación en nemotécnico
- colorHabitacion: color en hexadecimal para colorear dicha habitación
- backupDeMateriales: flag booleano que indica si hay que realizar una copia de seguridad de los materiales originales antes de cambiar el color. Útil por si queremos cambiar el color la primera vez o sustituir un color ya cambiado.

El objetivo de la función es como bien dice su nombre sustituir los materiales originales de una habitación por el color indicado por parámetro y realizando una copia de seguridad del material original si fuera necesario.

```
function cambiarColorHabitacion(habitacion, colorHabitacion,
backupDeMateriales)
{
    let objeto = new Object();

    /* Primero buscamos el objeto en la escena que tenga el mismo
nombre de habitación */
    for (let i = 0; i < Escenario.children.length; i++)
    {
        if (Escenario.children[i].habitacion &&
Escenario.children[i].habitacion == habitacion)
        {
            if (backupDeMateriales)
            {
                objeto.habitacion =
Escenario.children[i].habitacion;

                /* Hay que comprobar si el material está
formado por varios (array) o color único, para que se pueda copiar
sin problemas */
                if
(Array.isArray(Escenario.children[i].material))
```

```
        {
            objeto.material =
Escenario.children[i].material.slice();
        }
        else
        {
            objeto.material =
Escenario.children[i].material;
        }
        backupDeMaterialesOriginales.push(objeto);
    }
    Escenario.children[i].material = new
THREE.MeshLambertMaterial({ color: colorHabitacion} );
    Escenario.children[i].colorCambiado = true;
}
}
}
```

Lo primero que hacemos es buscar la habitación que coincida por la introducida por parámetro dentro de los objetos de la escena. Una vez localizada realizamos un backup de su material original si el flag estuviera activo para posteriormente asignarle el nuevo color y activar el flag de color cambiado del objeto.

### **Función restaurarMaterialesHabitacion**

El propósito de esta función es restaurar el material original de la habitación introducida por parámetro en nombre nemotécnico.

```
function restaurarMaterialesHabitacion(habitacionCambiada)
{
    for (let i = 0; i < backupDeMaterialesOriginales.length; i++)
    {
        /* Encontrados sus materiales originales y procedemos a
sustituirlos */
        if (backupDeMaterialesOriginales[i].habitacion ==
habitacionCambiada.habitacion)
        {
            if
(Array.isArray(backupDeMaterialesOriginales[i].material))
            {
                habitacionCambiada.material =
backupDeMaterialesOriginales[i].material.slice();
            }
        }
    }
}
```

```
        else
        {
            habitacionCambiada.material =
            backupDeMaterialesOriginales[i].material;
        }
        habitacionCambiada.colorCambiado = false;
    }
}
}
```

Para ello recorreremos el array global con el backup de los materiales originales y cuando localiza la habitación que se desea restaurar, copia sus materiales en ella. A continuación, establecemos el flag *colorCambiado* de la habitación a falso, para indicar que ya posee su material original.

### **Función activarBotonTemperaturasEnTiempoReal**

Función sencilla que sustituye el color todas las habitaciones de las cuales haya alguna temperatura medida y almacenada en el array *temperaturasPorHabitacion* (el cual se rellena con el método *obtenerTemperaturas*) y también crea una copia de seguridad de los materiales originales. Además, cambia la imagen del botón *temperaturasPared* de la interfaz de usuario por uno más oscuro (para simular una pulsación) y activa el flag *temperaturasPared* para indicar que las temperaturas en tiempo real están activas.

```
function activarBotonTemperaturasEnTiempoReal()
{
    for (let i = 0; i < temperaturasPorHabitacion.length; i++)
    {
        cambiarColorHabitacion(temperaturasPorHabitacion[i].habitacion
        , temperaturasPorHabitacion[i].temperatura, true);
    }

    $("#temperaturasPared").attr("src",
    "img/botones/botonTemperaturaActivo.png");
    temperaturasPared = true;
}
```

### **Función desactivarBotonTemperaturasEnTiempoReal**

Realiza la operación inversa a la función anterior que es restaurar las habitaciones cuyos materiales han sido cambiados por un color en función de su temperatura. Además, restaura la imagen original del botón, desactiva el flag de *temperaturasPared* y vacía el array backup de materiales.

```
function desactivarBotonTemperaturasEnTiempoReal()
{
    for (let i = 0; i < Escenario.children.length; i++)
    {
        /* Preguntamos si se ha cambiado su material original,
        por el de una temperatura a tiempo real */
        if (Escenario.children[i].colorCambiado == true)
        {
            restaurarMaterialesHabitacion(Escenario.children[i]);
        }
    }

    $("#temperaturasPared").attr("src",
    "img/botones/botonTemperatura.png");
    temperaturasPared = false;
    backupDeMaterialesOriginales = [];
}
```

### **Función activarBotonHumedadesEnTiempoReal**

Realiza la misma función que su homóloga de temperaturas, pero tratando con las humedades.

```
function activarBotonHumedadesEnTiempoReal()
{
    for (let i = 0; i < humedadesPorHabitacion.length; i++)
    {
        cambiarColorHabitacion(humedadesPorHabitacion[i].habitacion,
        humedadesPorHabitacion[i].humedad, true);
    }
    $("#humedadesPared").attr("src",
    "img/botones/botonHumedadActivo.png");
    humedadesPared = true;
}
```

## **Función desactivarBotonHumedadesEnTiempoReal**

Realiza la misma función que su homóloga de temperaturas, pero tratando con las humedades

```
function desactivarBotonHumedadesEnTiempoReal()
{
    for (let i = 0; i < Escenario.children.length; i++)
    {
        /* Preguntamos si se ha cambiado su material original,
        por el de una temperatura o humedad a tiempo real */
        if (Escenario.children[i].colorCambiado == true)
        {
            restaurarMaterialesHabitacion(Escenario.children[i]);
        }
    }

    $("#humedadesPared").attr("src",
    "img/botones/botonHumedad.png");
    humedadesPared = false;
    backupDeMaterialesOriginales = [];
}
```

## **Función cambiarPiso**

Función que se ejecuta al pulsar algún elemento de la barra de menú superior. Entonces realiza las operaciones necesarias para sustituir un piso por otro:

1. Eliminar el piso actual
2. Desactivar los colores en tiempo real
3. Obtener los nuevos valores de temperatura y humedad del nuevo piso
4. Cargar luces
5. Comprobar si el nuevo piso posee segunda planta o sótano
6. Cargar el nuevo piso y actualizar la selección en la interfaz

```
function cambiarPiso()
{
    eliminarPiso();
```

```
        /* Establecemos las temperaturas a tiempo real en falso de
nuevo */
        if (temperaturasPared)
        {
            desactivarBotonTemperaturasEnTiempoReal();
        }

        if (humedadesPared)
        {
            desactivarBotonHumedadesEnTiempoReal();
        }

        backupDeMaterialesOriginales = [];
        obtenerTemperaturas();
        obtenerHumedades();

        cargarLuz();
        segundaPlanta();
        sotano();
        mostrarPiso();
        actualizarInformacionEdificiodInterfaz();
    }
```

### **Función activacionTemperaturas**

Se ejecuta al pulsar el botón de temperaturas en tiempo real de la interfaz de usuario.

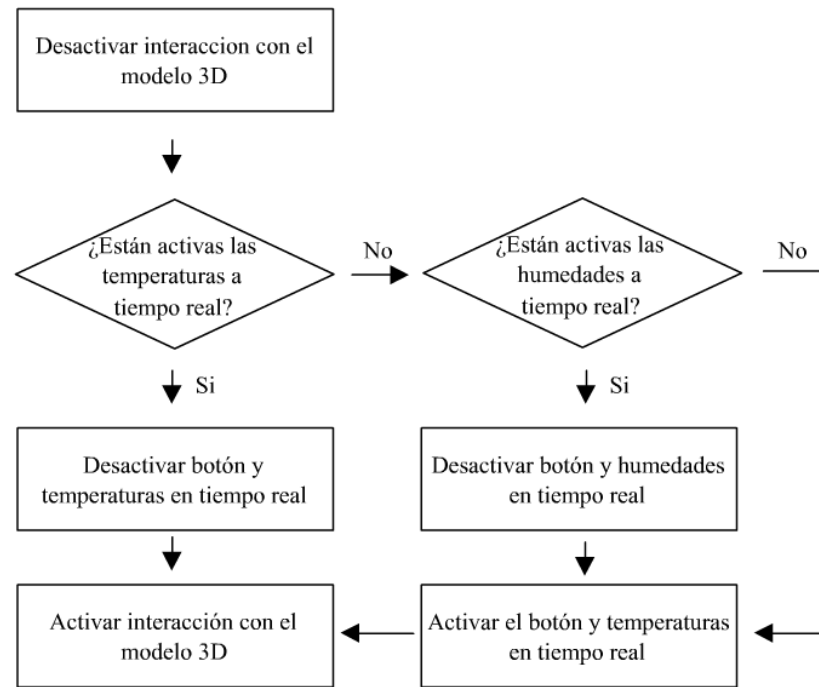


Figura 37: Algoritmo switch del botón temperaturas

```

function activacionTemperaturas()
{
    desactivarTarget();

    // Si estaba desactivada, la activamos y hacemos una copia de
    los materiales utilizados
    if (temperaturasPared == false)
    {
        if (humedadesPared)
        {
            desactivarBotonHumedadesEnTiempoReal();
        }

        activarBotonTemperaturasEnTiempoReal();
    }
    else
    {
        desactivarBotonTemperaturasEnTiempoReal();
    }

    activarTarget();
}
    
```

Nada más ejecutarse, desactiva por seguridad la capacidad de interactuar con el modelo para evitar que interfiera con el cambio de colores y comprobamos si el flag de temperaturas en tiempo real está activo. En caso de no estarlo comprobamos si está activo el flag de humedades para desactivarlo antes de proceder con la activación del flag de temperaturas en tiempo real. En caso de que ya estuviera activo, desactivamos el botón de temperaturas en tiempo real.

De esta forma, simulamos que los dos botones de la interfaz (temperaturas y humedad) se comportan como un switch o conmutador, donde pueden estar activo solamente 1 de ellos o ninguno.

### **Función activacionHumedades**

Esta función trabaja de la misma forma que la función anterior, pero con humedad en lugar de temperatura.

```
function activacionHumedades()
{
    desactivarTarget();

    /* En caso de que no se estén mostrando las habitaciones según
    su grado de humedad */
    if (humedadesPared == false)
    {
        /* Comprobamos si está activo el cambio de color de las
        paredes con respecto a la temperatura. Si lo está, no realizamos un
        backup de los materiales originales y activamos temperaturasPared */
        if (temperaturasPared)
        {
            desactivarBotonTemperaturasEnTiempoReal();
        }

        activarBotonHumedadesEnTiempoReal();

    } /* Y aquí en caso de que si se estén mostrando las habitaciones
    según su grado de humedad */
    else
    {
        desactivarBotonHumedadesEnTiempoReal();
    }
}
```



```
    activarTarget();  
}
```

### **Funcion actualizarEnTiempoReal**

Se ejecuta de forma periódica (por defecto, el intervalo es de 1 minuto) donde recarga los valores y colores de temperaturas y humedades en tiempo real. Si además, están activos alguno de los flags, realiza el cambio de los colores pero sin guardar copia de seguridad (ya que solo realiza una actualización).

Se ha implementado de esta manera para que los valores se actualicen automáticamente cada cierto intervalo de tiempo guardando las respuestas en memoria local, para no tener que enviar peticiones cada vez que se pulsa el switch evitando muchas peticiones si el usuario hiciera clicks ininterrumpidos.

```
function actualizarEnTiempoReal()  
{  
    /* Función que obtiene los valores actuales de temperaturas y  
    humedad para las paredes y si está activa alguna de las dos las muestra  
    */  
  
    obtenerTemperaturas();  
    obtenerHumedades();  
  
    if (temperaturasPared == true)  
    {  
        for (let i = 0; i < temperaturasPorHabitacion.length; i++)  
        {  
  
            cambiarColorHabitacion(temperaturasPorHabitacion[i].habitacion  
            , temperaturasPorHabitacion[i].temperatura, false);  
  
        }  
    }  
  
    if (humedadesPared == true)  
    {  
        for (let i = 0; i < humedadesPorHabitacion.length; i++)  
        {  
  
            cambiarColorHabitacion(humedadesPorHabitacion[i].habitacion,  
            humedadesPorHabitacion[i].humedad, false);  
  
        }  
    }  
}
```

```
    }  
}
```

### **Función activarTarget**

Función que activa el flag que permite interactuar con el modelo. Se ha implementado de esta forma para que la lectura del código fuera más fácil. Además, en un futuro puede aprovecharse la función para realizar otras tareas cuando se activa o desactiva la interacción con el modelo 3D como, por ejemplo, algún mensaje de carga.

```
function activarTarget()  
{  
    cambiandoColoresHabitacion = false;  
}
```

### **Función desactivarTarget**

Inversa a la anterior, impide que se pueda interactuar con el modelo 3D para evitar errores durante los cambios de colores

```
function desactivarTarget()  
{  
    cambiandoColoresHabitacion = true;  
}
```

### **Resto de ejecución**

Una vez definidas todas las funciones, procedemos a ejecutar las necesarias para iniciar la aplicación:

1. Inicializar variables (incluyendo la carga del modelo por defecto)
2. Obtener temperaturas y humedades del modelo por defecto
3. Comprobar segunda planta y sótano
4. Establecer el intervalo de tiempo por el cual se irán actualizando las temperaturas y humedades en tiempo real
5. Establecer la función de callback que se ejecutará de forma constante para que funcione la parte gráfica

```
inicio();  
obtenerTemperaturas();  
obtenerHumedades();  
segundaPlanta();  
sotano();  
  
setInterval(actualizarEnTiempoReal, INTERVALO_TIEMPO);  
  
animacion();
```

### iii. Conversor de ficheros

Para convertir los ficheros al formato .js desde el sitio web oficial está disponible un script en Python que se encarga de la conversión de OBJ a JS. Dicho script se ejecuta de la siguiente manera:

```
python convert_obj_three_for_python3.py -i origen.obj -o  
destino.js
```

Como podemos ver, tener que ejecutar el comando cambiando los nombres por cada submodelo, y además tener que hacer el fichero de metadatos datos.js a mano es una tarea bastante tediosa. Por eso se ha desarrollado una aplicación en C++ que se encarga de buscar todos los ficheros OBJ existentes en el mismo directorio donde se está ejecutando y crea una carpeta Convertido donde almacena el resultado de las conversiones a JS de todos los OBJ existentes, además de crear el fichero datos.js con los valores necesarios utilizando el script de conversión. De esta forma, se facilita mucho la tarea de conversión.

Para que funcione, es necesario que estén en el mismo directorio los ficheros a convertir, el script conversor de Python y la aplicación que lo ejecuta. Veamos su código fuente completo:

```
#include <iostream>  
#include <vector>  
#include <string>  
#include <windows.h>  
#include <dirent.h>  
#include <io.h>  
#include <stdio.h>  
#include <fstream>  
  
using namespace std;  
  
/* VARIABLES GLOBALES */
```

```
ofstream FICHERO_JSON("datos.js");

/* Indica si el fichero es un OBJ */
bool esFicheroObjeto(string nombreFichero)
{
    if (nombreFichero.substr(nombreFichero.find_last_of(".") + 1) ==
"obj")
    {
        return true;
    }
    return false;
}

/* Establece la primera línea del fichero de datos */
void iniciarJson()
{
    system("md Convertido");
    FICHERO_JSON<<"{\n\"Edificio\": [\"<<endl;
}

/* Cada línea que escribe contiene los metadatos de cada submodelo */
void escribirEnJson(string nombreFichero, bool &esPrimero)
{
    string habitacion = nombreFichero.substr(0,
nombreFichero.find("."));
    /* Siempre escribe una "," y salto de línea primero, antes de
escribir la línea con la información.
De esta forma no es necesario hacer ninguna operación extra al
escribir la última (que no lleva una coma al final)
En caso de ser la primera línea, no escribe la coma para evitar que
salga al principio. El resultado total es un fichero en formato JSON */
    if (!esPrimero)
    {
        FICHERO_JSON<<","<<endl;
    }
    else
    {
        esPrimero = false;
    }
    FICHERO_JSON<<"\t{ \"nombreFichero\": \""<<habitacion<<".js\" ,
\"habitacion\": \""<<habitacion<<"\" }";
}

/* Cierra el fichero JSON escribiendo los últimos caracteres necesarios
para completar la estructura */
void finalizarJson()
{
    FICHERO_JSON<<endl<<"]\n}";
    FICHERO_JSON.close();
    string comandoMoveJS = "MOVE datos.js Convertido\\datos.js";
    system(comandoMoveJS.c_str());
}

/* Ejecuta la orden para convertir el fichero, con el script de Python que
debe estar en la misma ruta de ejecución */
void lanzarConversor(string nombreFichero)
{

```

```
/* python convert_obj_three_for_python3.py -i
PabellonInformatica2.obj -o pabellonInformatica.js */
string nombreFicheroSinExtension = nombreFichero.substr(0,
nombreFichero.find("."));
string ficheroSalida = nombreFicheroSinExtension + ".js";

string comando = "python convert_obj_three_for_python3.py -i " +
nombreFichero + " -o " + ficheroSalida;
string comandoMoveJS = "MOVE " + ficheroSalida + " Convertido\\" +
ficheroSalida;
string comandoMove = "COPY " + nombreFicheroSinExtension + "\\*.*
Convertido\\*.* /Y";

/* Para mostrar durante la conversión */
cout<<"Convirtiendo "<<nombreFichero<<endl<<"-----
-----"<<endl;
system(comando.c_str());
system(comandoMoveJS.c_str());
system(comandoMove.c_str());
cout<<"-----"lt;<endl;
}

int main() {

    vector<string> cadenas;

    DIR *dir;
    struct dirent *ent;
    bool esPrimero = true;

    iniciarJson();
    if ((dir = opendir(".")) != NULL)
    {
        /* Recorremos todos los archivos del directorio y comprobamos
si es un fichero que contenga un modelo para ser convertido a .js */
        while ((ent = readdir (dir)) != NULL)
        {
            if (esFicheroObjeto(ent->d_name))
            {
                escribirEnJson(ent->d_name, esPrimero);
                lanzarConversor(ent->d_name);
            }
        }
        closedir (dir);

        finalizarJson();

        system("pause");
    }
    else
    {
        /* could not open directory */
        perror ("");
        return EXIT_FAILURE;
    }
}
```

Dicho código se encuentra también en la carpeta Conversor de modelos.

## 4. FUNCIONAMIENTO. MANUAL DE USUARIO

En esta sección vamos a explicar el funcionamiento de la aplicación cliente de cara al usuario.

### a. Inicio

Lo primero que nos encontramos nada más entrar es el modelo del Edificio de Informática cargado por defecto, en la planta baja. Arriba disponemos de un menú para seleccionar otro edificio, así como botones a su derecha para seleccionar la planta que queramos visitar.

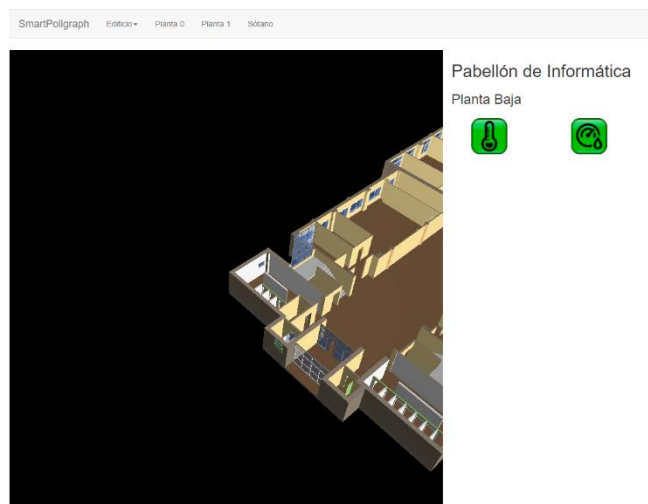


Figura 38: Visual de la aplicación cliente

Debajo de la barra de menús tenemos a la izquierda el modelo en 3D interactivo el cual al pasar el ratón por encima de las habitaciones estas cambiarán de color, indicando que es posible averiguar información sobre la misma. Si no se cambia el color, significa que no existe información que pueda estar almacenada.

A la derecha del modelo, tenemos el título del edificio y planta seleccionados para saber en todo momento donde nos encontramos. Debajo de los títulos, nos encontramos con dos botones que nos permiten cambiar el color de las habitaciones según el filtro correspondiente al botón (Temperaturas o humedad). Inferior a esos dos botones se encuentra un espacio vacío que se rellenará con la información de los sensores disponibles de alguna habitación que hayamos seleccionado, o un mensaje de error en caso de que no se disponga de ningún sensor.

## b. Proceso

### i. Cambiar edificio y planta

Cambiar el edificio y planta que se desea visualizar es muy sencillo. Tan solo hay que pulsar en el botón “Edificio” de la parte superior y se abrirá un menú donde podemos seleccionar el edificio que queramos ver. Si además queremos cambiar la planta, pulsamos en cualquiera de los botones Planta 0, Planta 1, Planta 2 o Sótano.

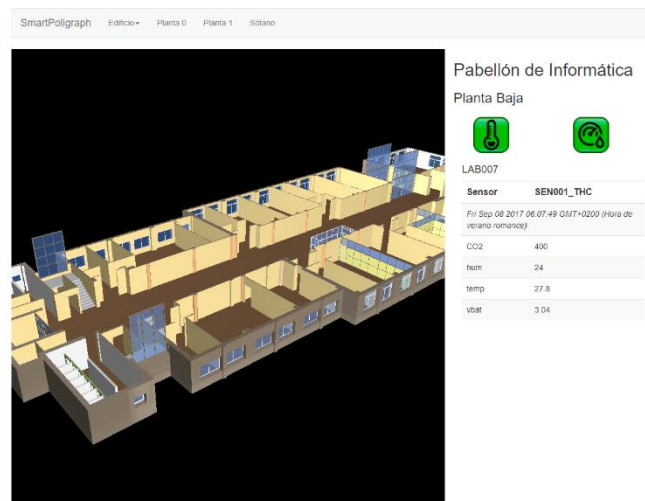


Figura 39: Menú de selección de edificios

### ii. Mover y girar el modelo

Utilizando el ratón podemos girar, desplazar y realizar zoom sobre el modelo que estamos visualizando:

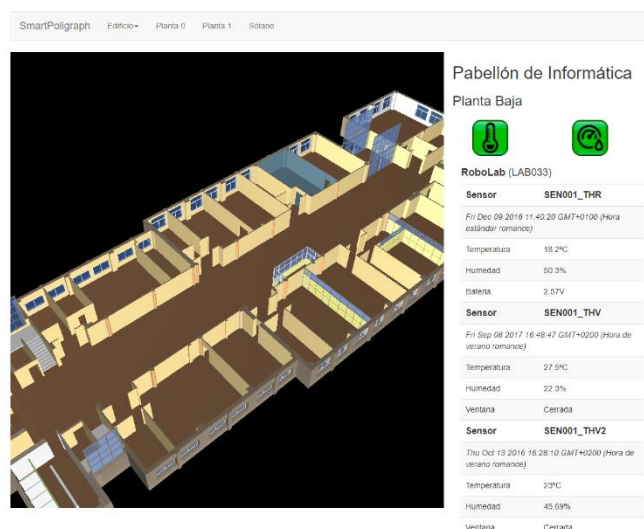
- Girar: Para girar el modelo hay que mantener pulsado el botón izquierdo del ratón y arrastrar para controlar el ángulo de giro
- Desplazar: Para desplazar el modelo de posición, hay que mantener pulsado el botón derecho del ratón y arrastrar para controlar el desplazamiento
- Zoom: Para realizar zoom debemos utilizar la rueda de desplazamiento integrada en el ratón, donde al subirla acercaremos la cámara y al bajarla alejaremos la cámara.



*Figura 40: Ejemplo de giro y desplazamiento de modelo 3D*

### **iii. Seleccionar una habitación y obtener información de los sensores**

Seleccionar una habitación para obtener los datos de todos los sensores referentes a ella es muy sencillo. Tan solo debemos mover el ratón sobre la habitación que queramos seleccionar y hacemos click izquierdo en ella. A continuación, debajo de los botones de temperatura y humedad nos aparecerá la información de dicha habitación, así como todas las últimas lecturas de sus sensores asociados.



*Figura 41: Ejemplo de lectura de sensores*



#### iv. Mostrar las temperaturas en tiempo real

Para ello hacemos click en el botón verde con icono de temperatura y el color del botón cambiará a un tono más oscuro indicando que ha sido pulsado. Entonces, si alguna habitación tuviera sensor de temperaturas, el color de la misma cambiará según el valor de la misma como se puede ver en la imagen:

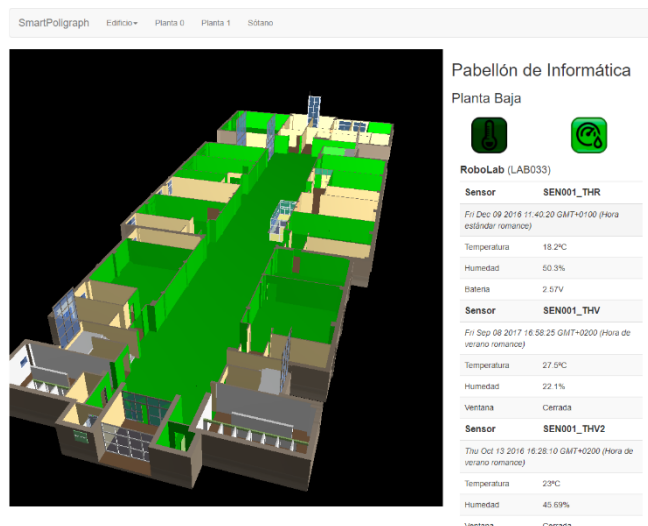


Figura 42: Ejemplo de temperaturas en tiempo real

La paleta de colores utilizada es la siguiente:

Tabla 20: Paleta de colores para temperaturas

Hasta 4,9° C	
De 5 a 9,9° C	
De 10 a 14,9° C	
De 15 a 19,9° C	
De 20 a 24,9° C	
De 25 a 29,9° C	
De 30 a 34,9° C	
De 35 a 39,9° C	
De 40 a 44,9° C	

Superior a 45° C

## v. Mostrar las humedades en tiempo real

Para ello hacemos click en el botón verde con icono de humedad, situado a la derecha y el color del botón cambiará a un tono más oscuro indicando que ha sido pulsado. Entonces, si alguna habitación tuviera sensores de humedad, el color de la misma cambiará según el valor de la misma como se puede ver en la imagen:

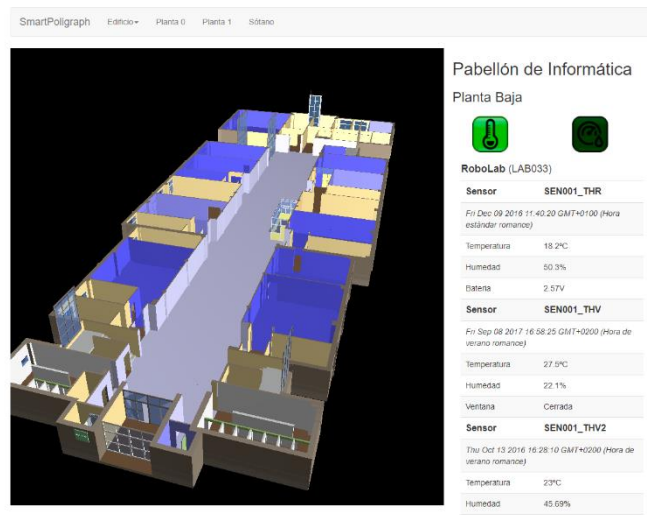


Figura 43: Ejemplo de humedades en tiempo real

La paleta de colores utilizada es la misma que la utilizada para las temperaturas, pero cambiando los intervalos:

Tabla 21: Paleta de colores para los grados de humedad

0 a 9,9 %	
10 a 19,9 %	
20 a 29,9 %	
30 a 39,9 %	
40 a 49,9 %	
50 a 59,9 %	
60 a 69,9 %	
70 a 79,9 %	

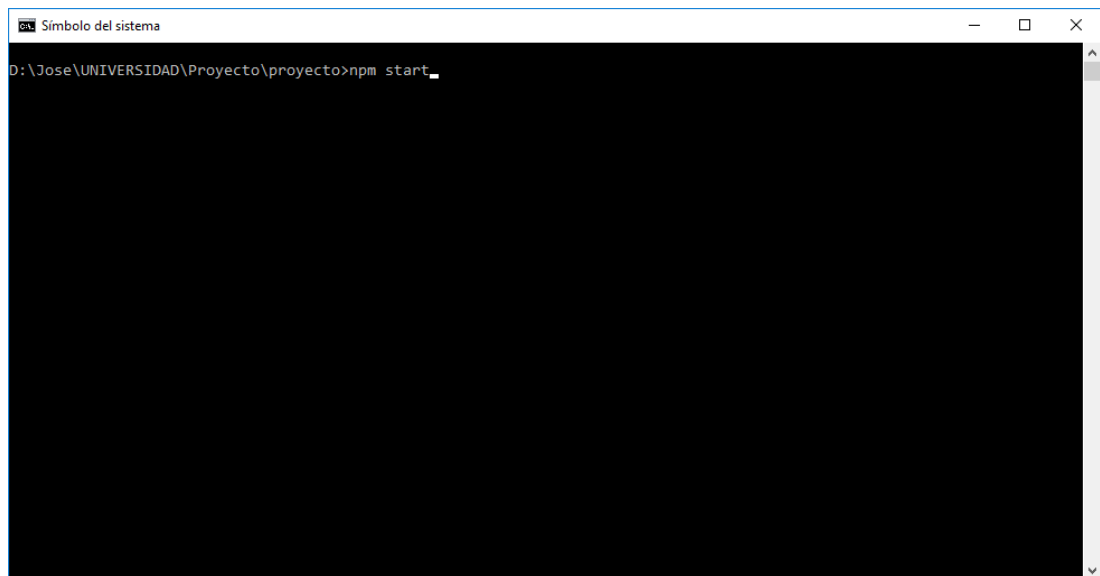
80 a 89,9 %	
90 a 100%	

Hay que indicar que no es posible tener activos al mismo tiempo los botones de temperaturas y grados de humedad en tiempo real. Solo será posible 1 o ninguno.

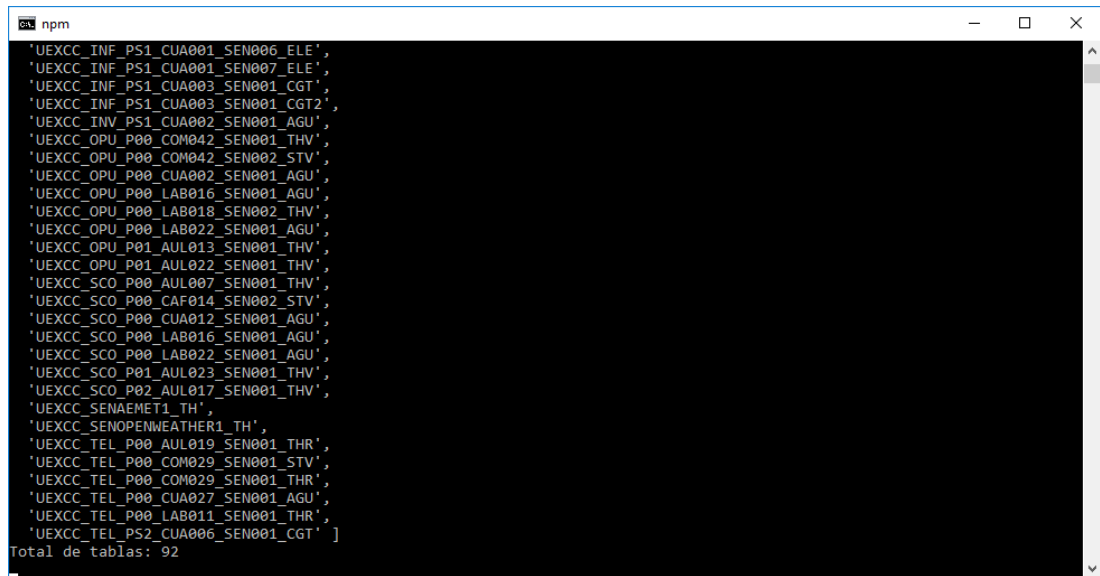
## 5. MANTENIMIENTO

### a. Puesta en marcha de la aplicación servidor

Arrancar el servidor de escucha es muy sencillo. Tan solo debemos irnos a la ruta donde se encuentra el fichero `index.js` y ejecutar el comando `npm start`. A continuación, el servidor se ejecutará e imprimirá por la consola las tablas con sensores encontradas y entonces estará listo para atender las peticiones.



*Figura 44: Puesta en marcha de la aplicación servidor*



```
npm
'UEXCC_INF_PS1_CUA001_SEN006_ELE',
'UEXCC_INF_PS1_CUA001_SEN007_ELE',
'UEXCC_INF_PS1_CUA003_SEN001_CGT',
'UEXCC_INF_PS1_CUA003_SEN001_CGT2',
'UEXCC_INV_PS1_CUA002_SEN001_AGU',
'UEXCC_OPU_P00_COM042_SEN001_THV',
'UEXCC_OPU_P00_COM042_SEN002_STV',
'UEXCC_OPU_P00_CUA002_SEN001_AGU',
'UEXCC_OPU_P00_LAB016_SEN001_AGU',
'UEXCC_OPU_P00_LAB018_SEN002_THV',
'UEXCC_OPU_P00_LAB022_SEN001_AGU',
'UEXCC_OPU_P01_AUL013_SEN001_THV',
'UEXCC_OPU_P01_AUL022_SEN001_THV',
'UEXCC_SCO_P00_AUL007_SEN001_THV',
'UEXCC_SCO_P00_CAF014_SEN002_STV',
'UEXCC_SCO_P00_CUA012_SEN001_AGU',
'UEXCC_SCO_P00_LAB016_SEN001_AGU',
'UEXCC_SCO_P00_LAB022_SEN001_AGU',
'UEXCC_SCO_P01_AUL023_SEN001_THV',
'UEXCC_SCO_P02_AUL017_SEN001_THV',
'UEXCC_SENAEMET1_TH',
'UEXCC_SENOPENWEATHER1_TH',
'UEXCC_TEL_P00_AUL019_SEN001_THR',
'UEXCC_TEL_P00_COM029_SEN001_STV',
'UEXCC_TEL_P00_COM029_SEN001_THR',
'UEXCC_TEL_P00_CUA027_SEN001_AGU',
'UEXCC_TEL_P00_LAB011_SEN001_THR',
'UEXCC_TEL_PS2_CUA006_SEN001_CGT' ]
Total de tablas: 92
```

Figura 45: Aplicación servidor en ejecución

## b. Puesta en marcha de la aplicación cliente

Una vez que la aplicación servidor se está ejecutando accedemos en nuestro navegador web a la dirección <http://localhost:3977/> y podremos visualizar la aplicación cliente. En caso de estar ejecutándose el servidor en otra máquina, en lugar de localhost habría que indicar la IP de la misma.

La aplicación funciona con los navegadores Mozilla Firefox y Google Chrome, siendo en Google Chrome donde se obtienen mejores resultados en la ejecución.

## c. Adición o edición de elementos gráficos

Se entregan los modelos de cada piso, edificio y facultad completa dentro de la carpeta Modelos en el directorio raíz en forma de proyecto Sketchup. Si queremos modificar un modelo existente debemos editarlo con Sketchup y luego exportar a OBJ los cambios en los submodelos realizados. Posteriormente, aplicamos el conversor y ya estará disponible para ser usado por nuestra aplicación. Veamos un ejemplo de modificación del aula AUL030 del Edificio de Informática Planta 0 a dos nuevas aulas, que llamaremos AUL049 y AUL050.

### i. Cargar el modelo original

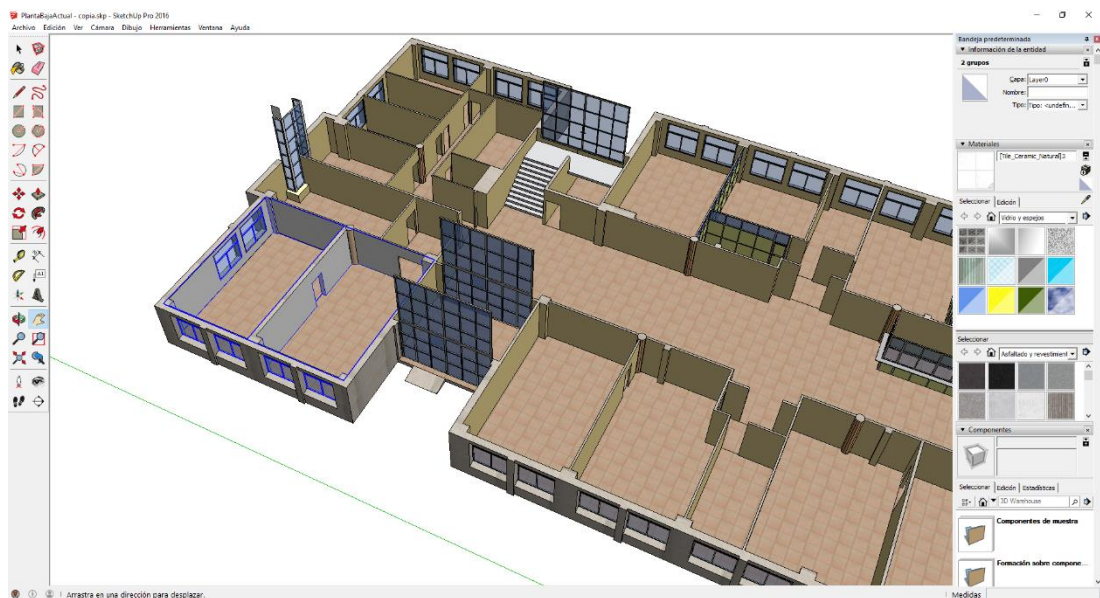
Lo primero es abrir el modelo que vamos a editar con Sketchup, el cual podemos ver seleccionado en la imagen



*Figura 46: Modelo antes de editar*

## **ii. Editar el modelo**

Una vez abierto, procedemos a desagrupar la selección y construir lo que necesitamos utilizando las herramientas que dispone Sketchup. Cuando hemos terminado de editar, agrupamos el interior de las nuevas habitaciones de forma independiente tal de la misma forma que la imagen



*Figura 47: Modelo después de editar*

En este caso hemos dividido el aula en 2, por lo que agrupamos la primera habitación y luego la segunda habitación, de forma que sean elementos independientes

### iii. Exportar modelos

Una vez que hemos terminado y agrupado la edición, eliminamos toda figura existente en el modelo salvo la que vamos a exportar, para que solo nos quede la habitación y sea exportada como submodelo independiente. Veamos un ejemplo en la imagen:

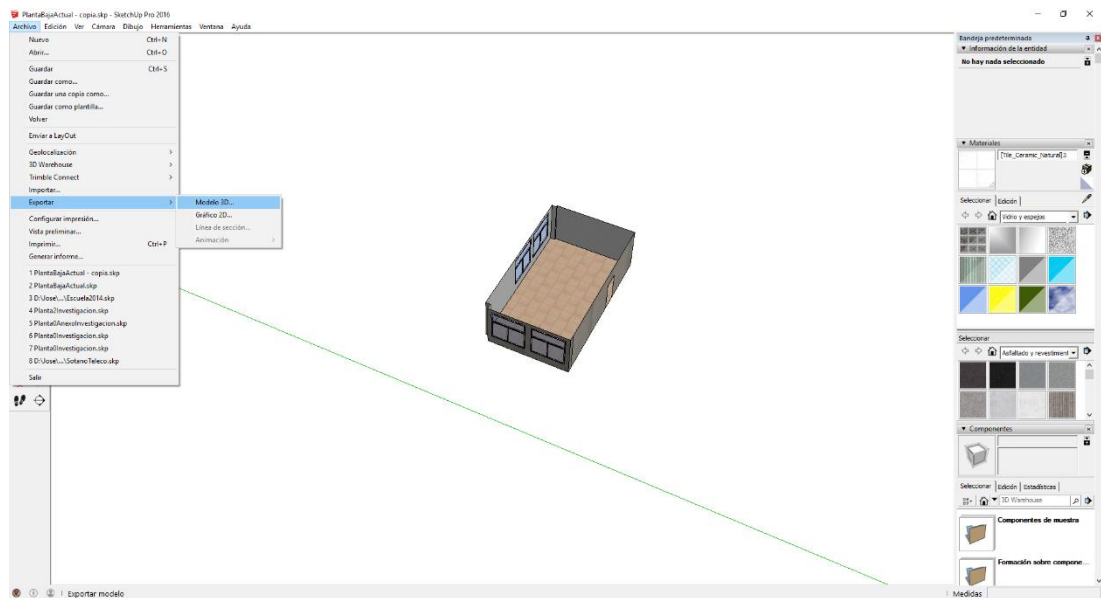


Figura 48: Submodelo AUL049

Seleccionamos formato OBJ, indicamos la ruta donde queremos exportarlo y además es necesario marcar todas las casillas del apartado opciones, para evitar errores gráficos durante la exportación

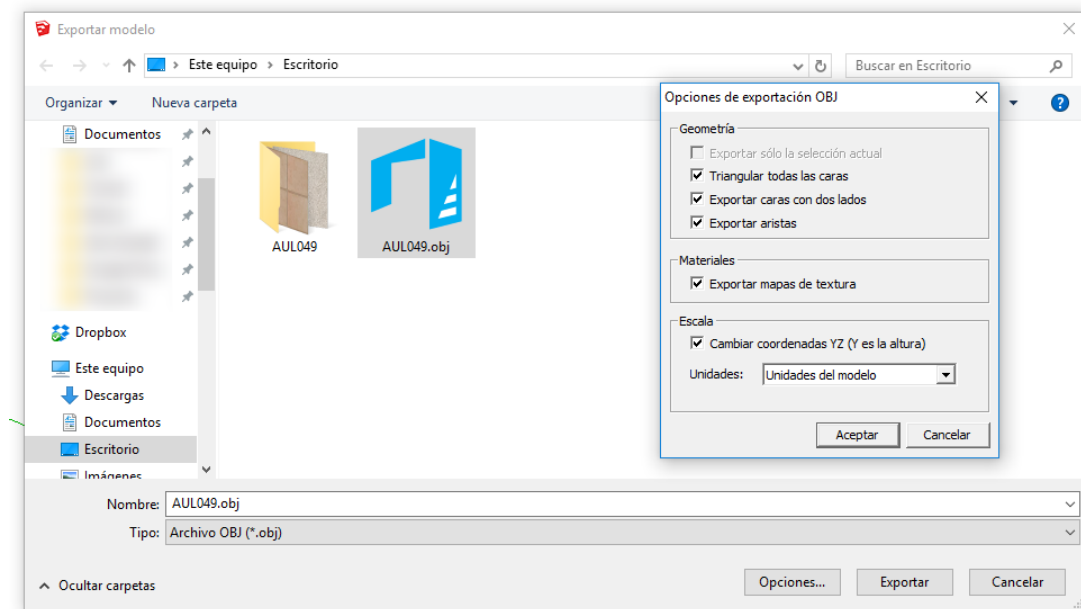


Figura 49: Opciones de exportación

Después repetimos las acciones para el otro submodelo

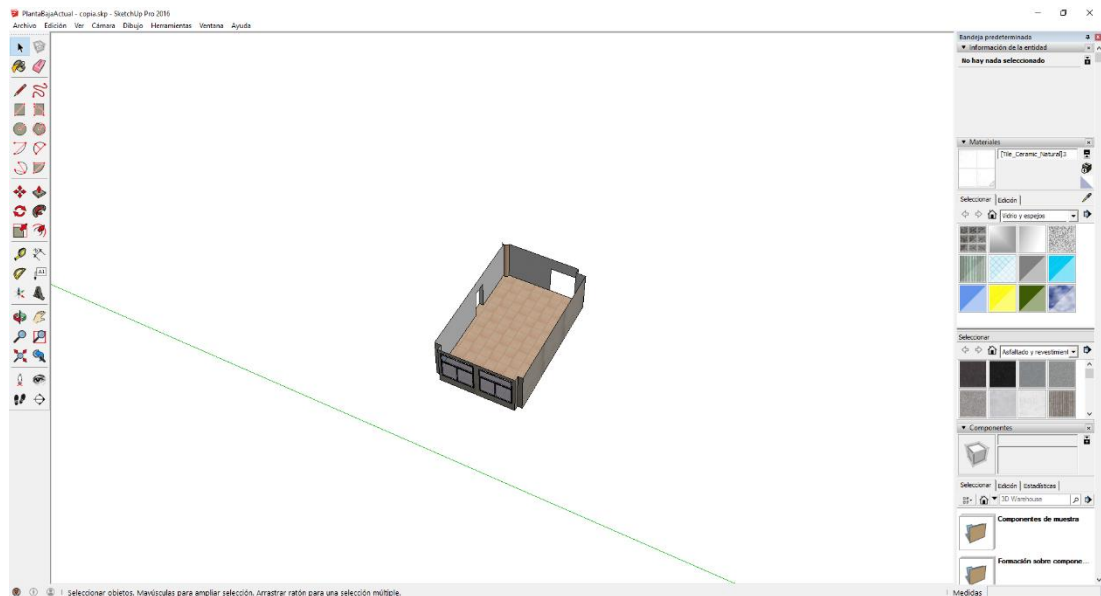


Figura 50: Submodelo AUL050

Una vez exportados todos los submodelos, los guardamos todos en una misma carpeta, en la que también copiaremos los dos ficheros para la conversión de modelos (conversorModelos.exe y convert\_obj\_three\_for\_python.py) ubicados en la carpeta Conversor de Modelos. Debemos de tener algo como en la siguiente imagen



AUL028	LAB008	COM034.mtl	DES025.obj
AUL049	LAB012	COM034.obj	DES027.mtl
AUL050	LAB016	COM038.mtl	DES027.obj
BASE	LAB018	COM038.obj	DES035.mtl
COM001	LAB021	COM044.mtl	DES035.obj
COM003	LAB032	COM044.obj	DES039.mtl
COM006	LAB033	COM046.mtl	DES039.obj
COM009	LAB036	COM046.obj	DES042.mtl
COM013	LAB037	COM048.mtl	DES042.obj
COM014	LAB040	COM048.obj	DES043.mtl
COM017	LAB041	conversorModelos.exe	DES043.obj
COM026	AUL028.mtl	convert_obj_three_for_python3.py	LAB007.mtl
COM029	AUL028.obj	CUA002.mtl	LAB007.obj
COM031	AUL049.mtl	CUA002.obj	LAB008.mtl
COM034	AUL049.obj	CUA004.mtl	LAB008.obj
COM038	AUL050.mtl	CUA004.obj	LAB012.mtl
COM044	AUL050.obj	CUA019.mtl	LAB012.obj
COM046	BASE.mtl	CUA019.obj	LAB016.mtl
COM048	BASE.obj	CUA020.mtl	LAB016.obj
CUA002	COM001.mtl	CUA020.obj	LAB018.mtl
CUA004	COM001.obj	CUA045.mtl	LAB018.obj
CUA019	COM003.mtl	CUA045.obj	LAB021.mtl
CUA020	COM003.obj	CUA047.mtl	LAB021.obj
CUA045	COM006.mtl	CUA047.obj	LAB032.mtl
CUA047	COM006.obj	DES005.mtl	LAB032.obj
DES005	COM009.mtl	DES005.obj	LAB033.mtl
DES010	COM009.obj	DES010.mtl	LAB033.obj
DES011	COM013.mtl	DES010.obj	LAB036.mtl
DES015	COM013.obj	DES011.mtl	LAB036.obj
DES022	COM014.mtl	DES011.obj	LAB037.mtl
DES023	COM014.obj	DES015.mtl	LAB037.obj
DES024	COM017.mtl	DES015.obj	LAB040.mtl
DES025	COM017.obj	DES022.mtl	LAB040.obj
DES027	COM026.mtl	DES022.obj	LAB041.mtl
DES035	COM026.obj	DES023.mtl	LAB041.obj
DES039	COM029.mtl	DES023.obj	
DES042	COM029.obj	DES024.mtl	
DES043	COM031.mtl	DES024.obj	
LAB007	COM031.obj	DES025.mtl	

Figura 51: Submodelos para Informática Planta 0



Como se puede observar, se ha eliminado el submodelo AUL030, ya que este ha sido modificado. Todos los demás submodelos OBJ se entregan en el CD, por lo que no es necesario volver a exportarlos si estos no se han modificado.

#### iv. Conversión

Ejecutamos conversorModelos.exe y comenzará automáticamente toda la conversión de cada fichero OBJ que encuentre. Los resultados aparecerán dentro del nuevo directorio Convertido.

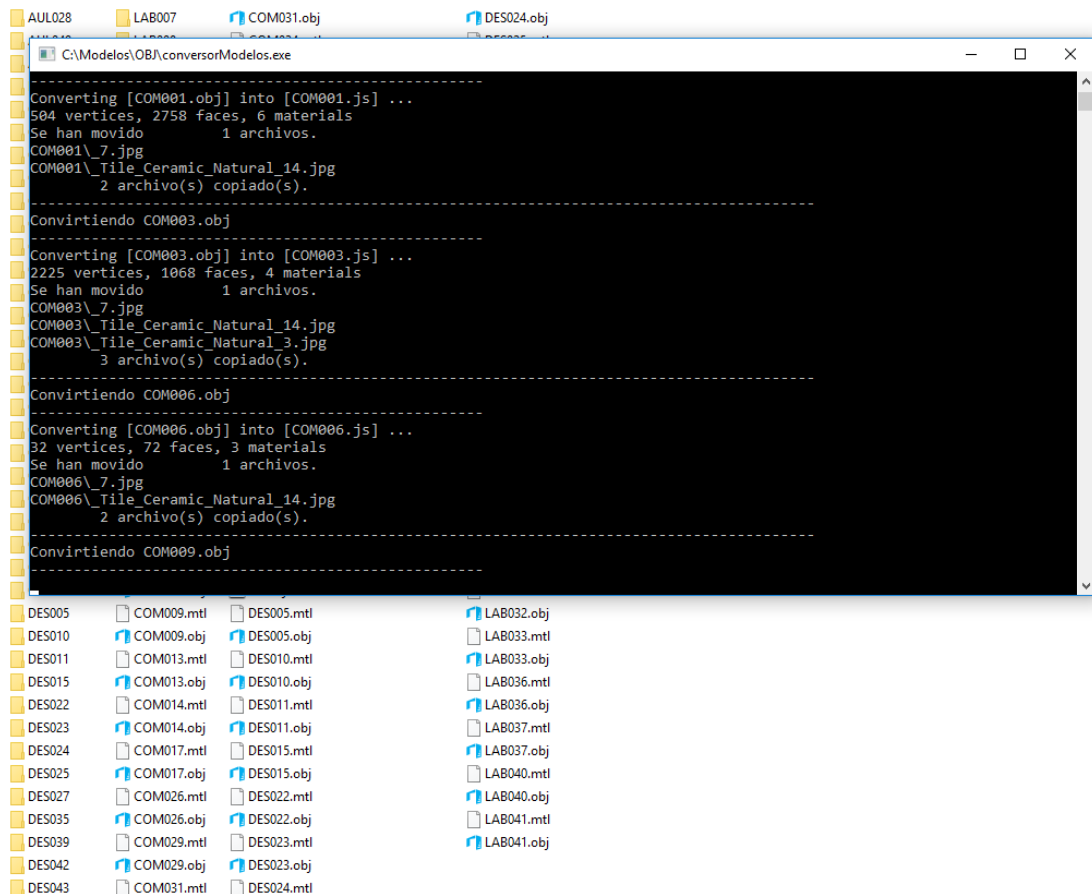


Figura 52: Proceso de conversión

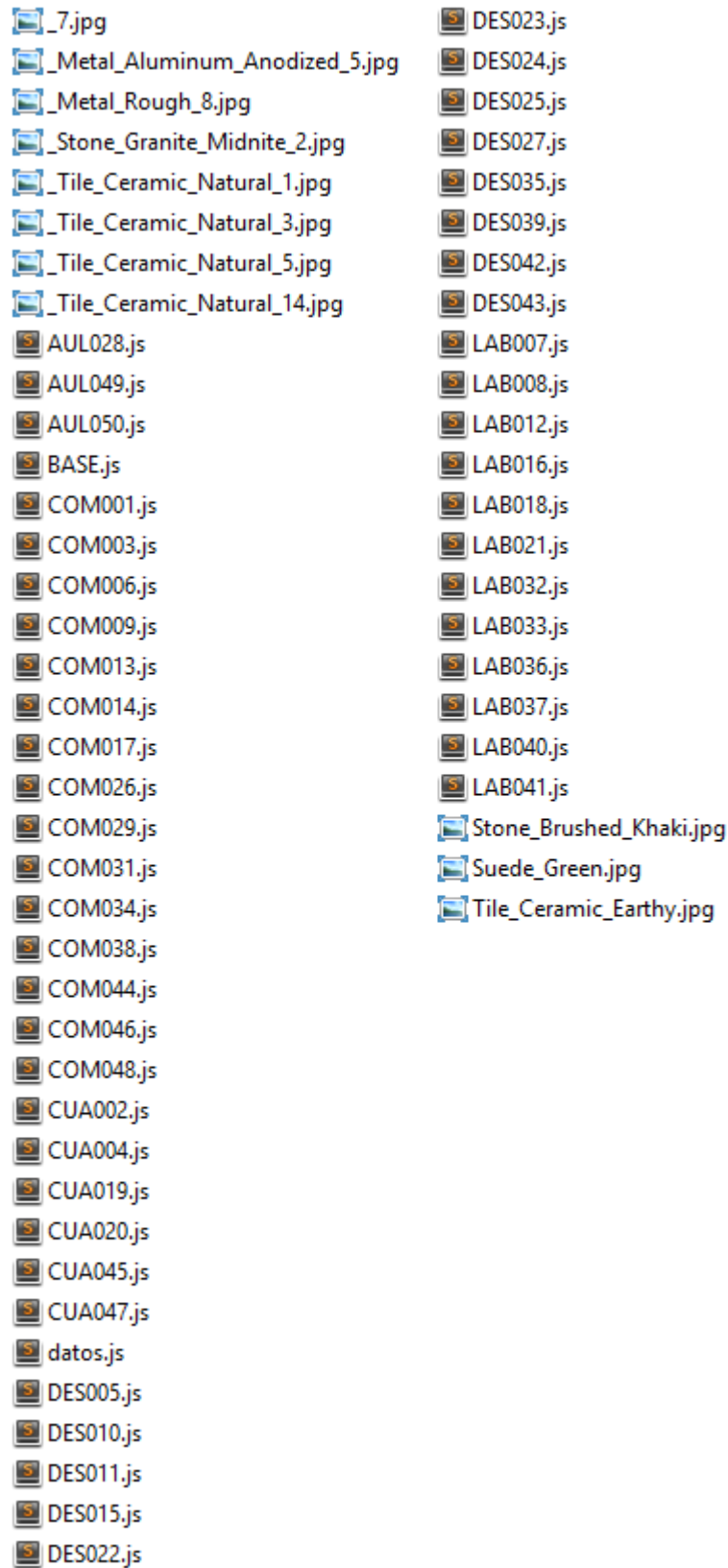


Figura 53: Resultado de la conversión

Cuando la conversión ha finalizado, podemos observar que se ha creado también el fichero de datos `datos.js` con todos los metadatos necesarios para su carga en la aplicación cliente.

## v. Carga en la aplicación

Una vez finalizada la conversión, movemos todos los ficheros resultantes a la carpeta `public/modelos/INF/P00/` eliminando todos los anteriores. De esta forma la próxima vez que la aplicación cargue el modelo de Informática Planta 0 cargará el modelo actualizado

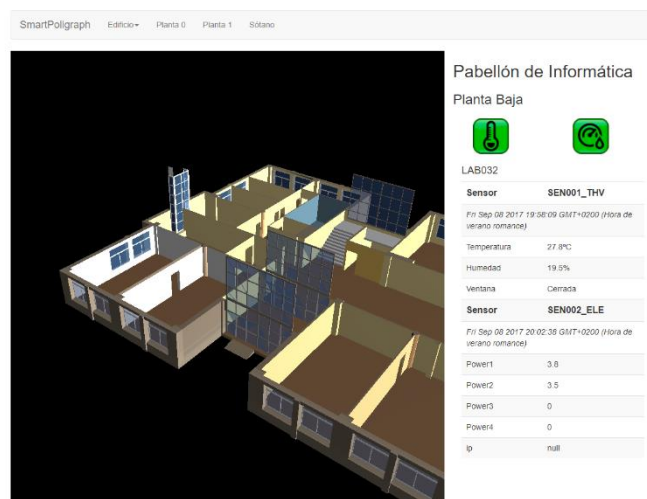


Figura 54: Modelo actualizado en la aplicación

En este ejemplo hemos utilizado el Edificio de Informática Planta 0, pero los pasos son iguales para cualquier modelo que se quiera actualizar.

## d. Adición de sensores

La aplicación, tanto cliente como servidor, no almacena datos de los sensores que hay actualmente disponibles y cada vez que se realiza una petición, pregunta a la base de datos cuales tiene disponibles para una habitación concreta. De esta manera, solo con dar de alta un sensor en la base de datos su tabla es captada automáticamente sin tener que realizar modificación alguna. La única tarea a realizar es reiniciar el servidor si se añade un sensor nuevo a la base de datos, para que este recargue las tablas disponibles y permita buscar en ellas.

### e. Adición de filtros en tiempo real

La aplicación cliente tanto como la aplicación servidor se ha diseñado para que sea fácil añadir más botones al switch en tiempo real que actualmente trabaja con temperaturas y humedad.

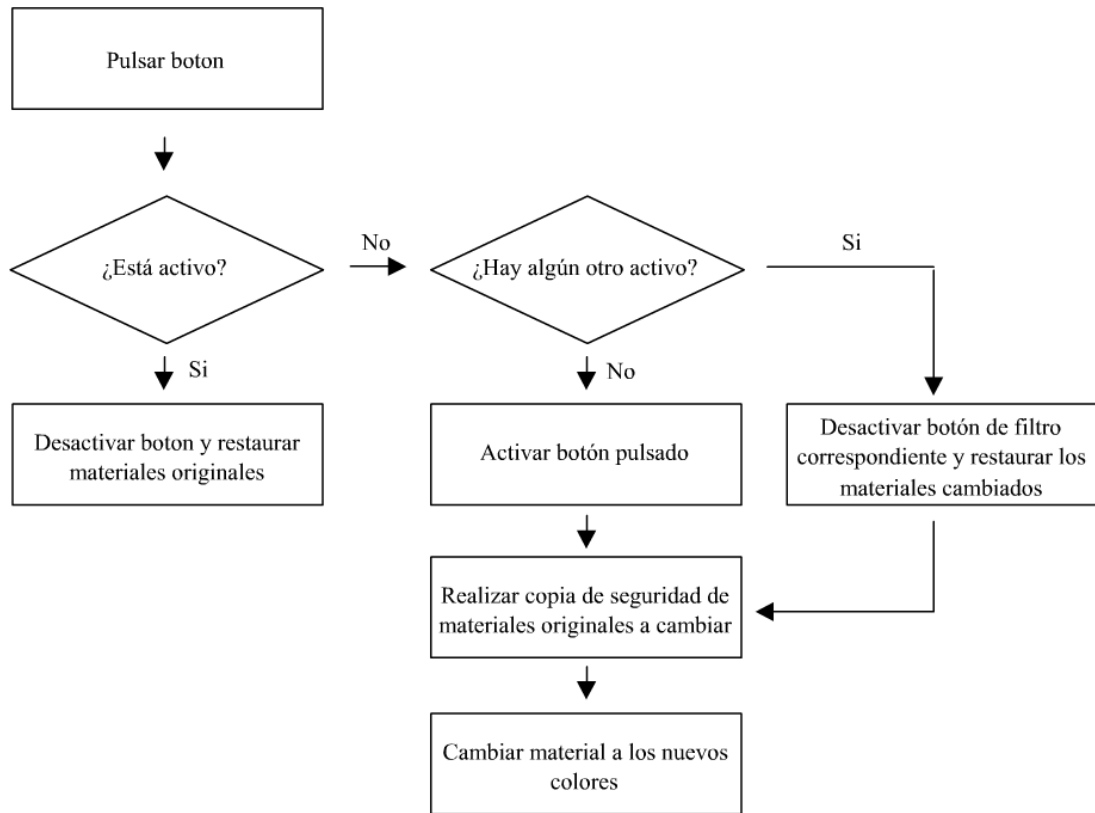


Figura 55: Esquema de funcionamiento swtich de filtros

Pongamos de ejemplo la función `activacionTemperaturas`, encargada de gestionar el botón de las temperaturas en tiempo real

```
function activacionTemperaturas()
{
    desactivarTarget();
    if (temperaturasPared == false)
    {
        if (humedadesPared)
        {
            desactivarBotonHumedadesEnTiempoReal();
        }
    }
}
```

```
        activarBotonTemperaturasEnTiempoReal();
    }
    else
    {
        desactivarBotonTemperaturasEnTiempoReal();
    }
    activarTarget();
}
```

Obviando las funciones para activar y desactivar el target, lo primero que comprobamos es si está activo el mismo botón que se acaba de pulsar. Si estuviera activo lo desactivamos y ejecutamos la función *desactivarBotonTemperaturasEnTiempoReal*, que se encarga de cambiar el icono al botón y de restaurar los materiales originales guardados en la copia de seguridad. En caso de estar desactivado preguntaríamos por cada filtro que pudiera estar activo, para realizar las operaciones de desactivación y restauración.

El motivo por el cual se vuelve a realizar otra copia de seguridad después de ser restaurada, es porque las habitaciones que van a cambiar pueden ser distintas al filtro anterior y de esta forma nos aseguramos que no se produzca ningún fallo.

Para añadir nuevos filtros tendríamos que crearnos las funciones equivalentes a las usadas tanto en la parte de cliente como en la parte de servidor que se encargue de recolectarlas, y añadirlas en la función que se ejecuta periódicamente en la aplicación cliente para actualizarlas (*obtenerX* y *actualizarEnTiempoReal*)

## **f. Actualizar nombres no nemotécnicos**

Para actualizar los nombres no nemotécnicos la API REST del servidor posee una función que actualiza los nombres. Dicha función se puede ejecutar con la url *SERVIDOR/api/actualizarNombresHabitación* a través de una petición POST con 3 parámetros requeridos: edificio, planta y clave.

Lo primero que hay que tener preparado es el fichero que contendrá los nombres que recibirán las habitaciones. Ese fichero debe llamarse *nombresHabitacion.js* y seguir la siguiente estructura JSON:

```
{
```

```
"nombres": [  
  {"habitacion":"XXXXYY", "nombre":"Nombre 1"},  
  {"habitacion":"XXXXYY", "nombre":"Nombre 2"}  
]  
}
```

Un ejemplo para el Edificio de Informática Planta 0 es el siguiente:

```
{  
  "nombres": [  
    {"habitacion":"LAB033", "nombre":"RoboLab"},  
    {"habitacion":"COM003", "nombre":"Cuarto de baño  
mujeres"},  
    {"habitacion":"LAB041", "nombre":"Laboratorio de  
Física"}  
  ]  
}
```

El fichero debe estar ubicado en la misma carpeta que el fichero de metadatos datos.js y submodelos. En la siguiente imagen se muestra el contenido de la carpeta *public/modelos/INF/P00*

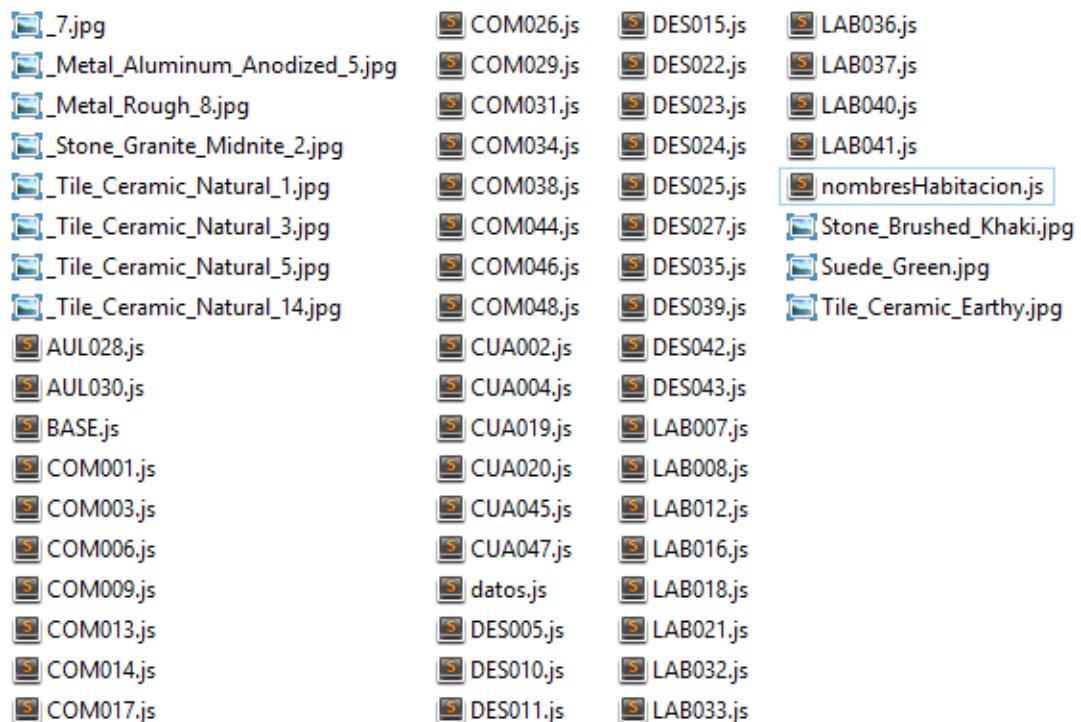


Figura 56: Ejemplo de localización de fichero nombresFichero.js

Para enviar la petición podemos usar cualquier aplicación que nos permita realizar peticiones a una URL enviando datos. En este caso vamos a utilizar como

ejemplo Postman. Si los parámetros no fueran correctos o válidos recibiríamos mensajes de error explicando el motivo. Si todo fuera correcto, nos devolvería el JSON resultante de fusionar los dos ficheros como respuesta. No obstante, el fichero datos.js ya se ha actualizado en el servidor por lo que no es necesario modificarlo manualmente.

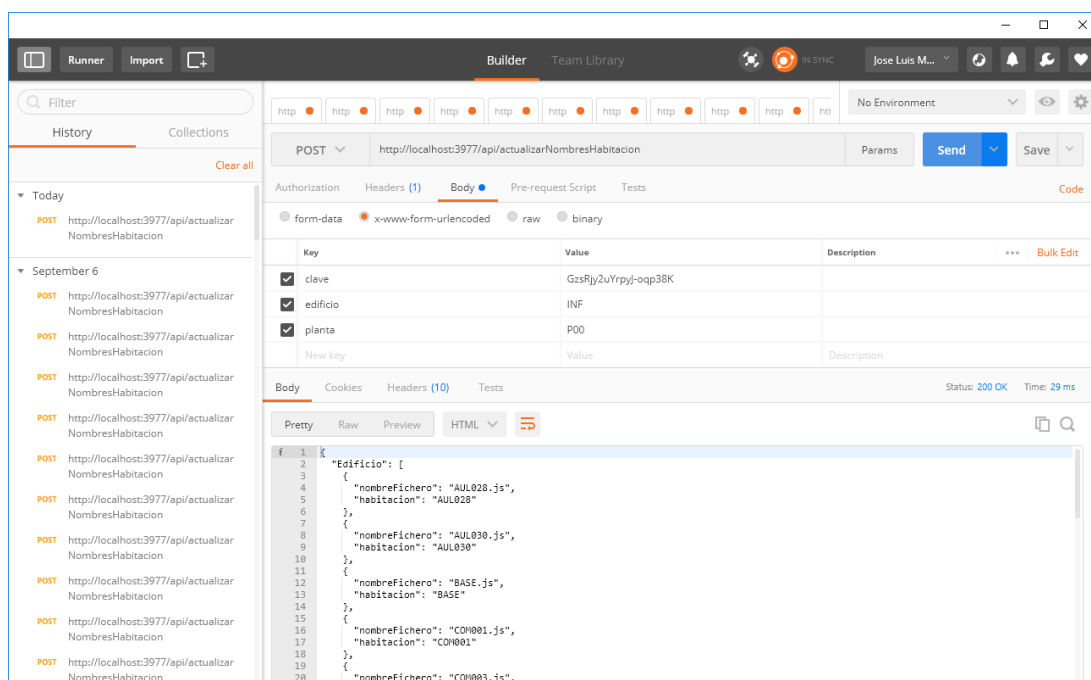


Figura 57: Ejecución de la función actualizar nombres

Los parámetros edificio y planta deben enviarse con su código nemotécnico al que hacen referencia y el parámetro clave se utiliza para que ningún usuario autorizado pueda ejecutar esta función. La clave que está activa en este momento es GzsRjy2uYrpyJ-oqp38K, la cual puede ser cambiada en el código fuente del servidor en cualquier momento si fuera necesario.

Se hace de esta manera para evitar que el conversor añada los nombres si no queremos que estos se muestren en la interfaz y únicamente se dedique a convertir los ficheros.

## 6. CONCLUSIONES Y TRABAJOS FUTUROS

### a. Conocimientos previos

La elección de este PFC es debida a que cursar la asignatura Informática Gráfica me gustó bastante lo aprendido y quería desarrollar alguna aplicación que

utilizara los conocimientos adquiridos allí. La idea de la WebGL y SmartPoliTech me gustó bastante, sin embargo, había un problema: No tenía ningún conocimiento de cómo desarrollar aplicaciones web ya que carecía de alguna asignatura optativa que la impartiera. Tendría que desarrollar una aplicación web sin saber aún HTML, CSS, PHP ni JavaScript.

Sin embargo, esto no es ningún impedimento, sino más bien una motivación para aprender el uso de estos lenguajes tan demandados hoy en día.

## **b. Experiencia y conocimientos adquiridos**

Si te dedicas a la informática, y más aún al desarrollo de software nunca se deja de aprender. Siempre salen lenguajes nuevos, frameworks basados en lenguajes anteriores nuevos, nuevas herramientas y nuevas metodologías.

No iba a ser menos con este proyecto en el cual he aprendido bastante. Empecé realizando cursos sencillos de HTML y JavaScript sencillos a la par que miraba ejemplos de cómo funcionaban algunos aspectos Three.js. Sin embargo, es una mala técnica de aprendizaje puesto que antes de empezar con algo más gordo es necesario tener unos buenos cimientos y yo en este caso no los tenía. No puedes empezar de primeras programando funciones de Three si antes no manejas bien JavaScript y HTML, porque entonces tendrás bastantes lagunas que no has rellenado.

Me llegó una oferta para un curso gratuito de PHP-HTML-JavaScript-MySQL impartido por la Fundación Telefónica al cual me apunté y quedé seleccionado. Una vez finalizado los conocimientos de PHP, HTML y JavaScript eran notablemente superiores ya podía enfrentarme sin problemas a cualquier desarrollo web. Aprovechando esos conocimientos y un curso básico de Three.js (el cual incluyo en la bibliografía) no he tenido problemas para desarrollar esta aplicación.

Sin embargo, el servidor donde debería ejecutarse la aplicación no es apache, por lo cual no podía escribir funciones en PHP para el tratamiento de los datos. Me indicaron que funcionaba con node.js por lo cual tuve que aprender su funcionamiento y como programar aplicaciones para él. No fue nada difícil utilizando el curso que indico en la bibliografía.



Con el desarrollo de este Proyecto considero que he aprendido cosas bastante útiles y demandadas hoy en día, así como aprender a resolver nuevos problemas que pueden surgir al utilizar estas tecnologías.

### **c. Conclusiones**

Para concluir, creo que esta aplicación ayuda a mostrar lo que el proyecto SmartPoliTech quiere aportar. Cualquier usuario puede visualizar toda la información recopilada y además interpretarla de una forma sencilla. No solo con el acceso de los mismos, sino con la navegación a través de un modelo 3D que permite interactuar de una forma más humana y visual.

Además, la utilización de los filtros de temperatura y humedad ayuda a monitorizar en todo momento el estado de la facultad en tiempo real sin necesidad únicamente de tablas y datos estadísticos.

### **d. Posibles ampliaciones futuras**

#### **i. Mejorar eficiencia**

Como toda aplicación, su eficiencia puede mejorar con futuras versiones. Revisión y optimización de algoritmos, revisión de código o cambios en los modelos pueden acelerar la ejecución de la aplicación.

#### **ii. Nuevos filtros en tiempo real**

Además de las temperaturas y grados de humedad, se podrían añadir nuevos filtros en base a otros datos. Por ejemplo, las habitaciones más pobladas, habitaciones con ventanas abiertas o cerradas, las más silenciosas o ruidosas, las más activas, las mejor iluminadas o en función del agua y electricidad consumidos. Al fin y al cabo, los filtros se utilizan para monitorizar la actividad de las mismas.

#### **iii. Kinect**

Teniendo en cuenta que hay zonas donde están trabajando dispositivos Kinect, podrían utilizarse para añadir nuevas herramientas a la aplicación

#### **iv. Estadísticas**

De igual forma que la Escuela Politécnica posee televisiones ubicados en puntos estratégicos mostrando datos estadísticos sobre las temperaturas y consumos, estos podrían mostrarse aquí también. Además de mostrar los valores de la última lectura, añadir funciones para que muestren gráficos estadísticos en función de las temperaturas de un intervalo de tiempo, o consumos eléctricos o de agua. Sería algo similar a lo mostrado en la siguiente imagen

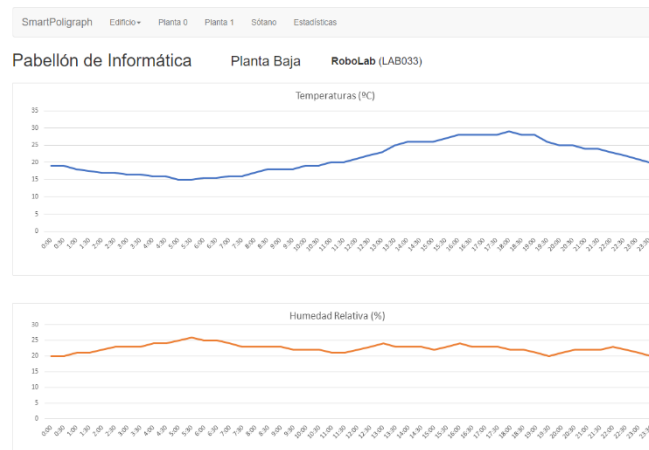


Figura 58: Ejemplo con datos estadísticos

Esto no se restringe a una única habitación, sino que si se implementara podrían evaluarse más de una para que se pudieran comparar fácilmente.

## v. Panel de administración

Se podría desarrollar un panel de administración para controlar ciertos aspectos sin necesidad de cambiar el código fuente del servidor. Por ejemplo, cambiar la paleta de colores o el nombre que se le aplicaría a los datos formateados. También se podría lanzar la función de agregación de nombres de habitación para poder ejecutarla sin necesidad de programas externos o lanzar el conversor de archivos desde el panel. Todo ello protegido con usuario y contraseña.

## **vi. Ampliar a otros centros de la Universidad de Extremadura**

De igual manera que esta aplicación solo trabaja con la Escuela Politécnica, podría implementarse para otros centros o edificios públicos de Extremadura si existieran sensores para ello.

## **REFERENCIAS BIBLIOGRÁFICAS** (Según norma ISO690)

1. BIBLIOTECA UNIVERSIDAD DE CANTABRIA. 2016. Cómo citar bibliografía según ISO 690 (1ª). Disponible en:  
<http://www.buc.unican.es/formacion/citarbibliografiaseguniso690>
2. DEVELOTECA. 2013. WebGL y THREE.JS. Disponible en  
<https://www.youtube.com/watch?v=yW0qHwXHEZs>
3. INFLUXDB. 2017. Querying Data. Disponible en:  
[https://docs.influxdata.com/influxdb/v0.9/guides/querying\\_data/](https://docs.influxdata.com/influxdb/v0.9/guides/querying_data/)
4. KRIS ROOFE. 2013. THREE.js Ray Intersect fails by adding div. Disponible en <https://stackoverflow.com/questions/13542175/three-js-ray-intersect-fails-by-adding-div/13544277#13544277>
5. NODE-INFLUX. 2017a. Starting Guide. Disponible en: <https://node-influx.github.io/class/src/index.js~InfluxDB.html>
6. NODE-INFLUX. 2017b. Tutorial. Disponible en <https://node-influx.github.io/manual/tutorial.html>
7. PRISONER849. 2017. Mouse hover with multimaterial three.js doesnt work. Disponible en <https://stackoverflow.com/questions/44459909/mouse-hover-with-multimaterial-three-js-doesnt-work>
8. ROBLES, V. 2017. Desarrollo web con JavaScript, Angular, NodeJS y MongoDB. Disponible en: <https://www.udemy.com/desarrollo-web-con-javascript-angular-nodejs-y-mongodb/>
9. STEMKOSKI. 2017. Examples. Disponible en:  
<http://stemkoski.github.io/Three.js/>
10. THREE.JS. 2017a. Creating a Scene. Disponible en:  
<https://threejs.org/docs/index.html#manual/introduction/Creating-a-scene>
11. THREE.JS. 2017b. Examples. Disponible en: <https://threejs.org/examples/>