Práctica 1. Implementando un visualizador de datos del Catastro.

Objetivo

El objetivo de esta práctica es doble: por una parte el alumno se familiarizará con la librería de soporte que se ha desarrollado para facilitar la implementación de las prácticas. Dicha librería sustituye a GLUT. En segundo lugar, se usará dicha librería para implementar un visualizador de información gráfica catastral.

Material entregado

El código de la práctica está publicado en https://github.com/fjabad/pgupv. Incluye varios ejemplos de uso de la biblioteca de soporte PGUPV. La estructura de directorios es la siguiente:

- **`bin`**: En este directorio se generarán los ejecutables de las prácticas. Tiene los ficheros DLL con las dependencias para Windows (SDL, Assimp y DevIL).
- **`deps`**: Aquí están las dependencias utilizadas (principalmente, los includes y los lib). Se han eliminado algunos ficheros para ahorrar espacio.
- **`PGUPV`**: código fuente de la librería desarrollada para facilitar la programación de las prácticas. Al compilar este proyecto se genera una biblioteca **`PGUPV.lib`**, que tendrás que incluir en tus proyectos.
- **`assets`**
  - **`models`**: tiene algunos ficheros de modelos geométricos que se usarán durante las prácticas
  - **`shaders`**: tiene algunos shaders básicos
- **`examples/ej1-1 a ej1-4`**: ejemplos
- **`exercises/p0`** y **`p1`**: ejercicios entregables

Los ficheros del proyecto se construyen utilizando CMake. En Windows no necesitas instalar ninguna librería adicional. Los requisitos necesarios para poder compilar y ejecutar las prácticas son:

- Windows 7, 32 o 64 bits.
- Visual Studio 2022, cualquier versión.
- Una tarjeta gráfica que soporte OpenGL 4 (NVIDIA GeForce 400 o superior, AMD Radeon HD5000 o superior, Intel HD Graphics 4000 o superior).

¡Cuidado! Asegúrate de tener instalado el driver más reciente de tu tarjeta gráfica. Los últimos drivers corrigen errores y son más estables. También asegúrate de que tu tarjeta soporta OpenGL 4.x. Si no estás seguro de las características de tu tarjeta búscala en la lista correspondiente a su fabricante:

- http://en.wikipedia.org/wiki/Nvidia_gpus
- http://en.wikipedia.org/wiki/Comparison_of_AMD_graphics_processing_units
- http://en.wikipedia.org/wiki/Comparison_of_Intel_graphics_processing_units

Para construir los proyectos, tanto en Windows como en Linux, bájate el repositorio de https://github.com/fjabad/pgupv y sigue las instrucciones del fichero **`leeme.txt`** que se encuentra en el directorio raíz de las prácticas. El código suministrado se ha probado en varias plataformas (NVidia, AMD, Intel/Windows y Linux). Sin embargo, la plataforma de referencia son las máquinas del laboratorio. Las prácticas se evaluarán teniendo en cuenta su comportamiento en dicha plataforma. Por lo tanto, antes de hacer cada entrega, asegúrate de que funciona correctamente en el laboratorio.

¡Cuidado! Mac OS X ha abandonado OpenGL (en favor de Metal). La versión más actual del sistema operativo sólo soporta OpenGL 4.1 (publicada en 2010). Por ello, ya no se soporta Mac OS X para realizar las prácticas de Programación Gráfica. Si tienes un Mac con procesador Intel, tendrás que instalar un segundo sistema operativo para trabajar con versiones recientes de OpenGL (no sirve una máquina virtual, debes instalar el sistema operativo en su partición y arrancar la máquina usando dicho sistema operativo).
Otra opción es usar las máquinas virtuales con GPU disponibles en:

https://polilabsvpn.upv.es (DSIC-GPU)

https://polilabs.upv.es (Aula Gráfica 1 y Aula Gráfica 2)

La biblioteca PGUPV

Se ha desarrollado una biblioteca de apoyo que permite implementar aplicaciones en OpenGL 4.x rápidamente. Dicha biblioteca se encarga de pedir al sistema operativo una ventana que soporte una determinada versión de OpenGL y tenga las características pedidas. También se encarga de gestionar los eventos.
PGUPV también ofrece una serie de clases que abstraen los detalles de implementación de apartados importantes de OpenGL, tal y como *Buffer Objects*, *Vertex Array Objects*, etc. También aporta una implementación software de una pila de matrices para implementar las diferentes transformaciones por las que pasa un vértice (*model-view-projection*).
La biblioteca define el espacio de nombres PGUPV. ¡Recuerda utilizarlo, o el compilador no encontrará las definiciones de las clases!
Las clases principales de la biblioteca son:

- **`App`**: Clase que se encarga de inicializar la aplicación. Es una clase *Singleton* (es decir, no se pueden instanciar nuevos objetos de la misma). Para obtener una referencia a dicho objeto, usa la función **`getInstance`** de la clase **`App`**. Normalmente obtendrás una referencia al objeto **`App`** en la función **`main`** de tu programa, y la usarás para configurar la ventana OpenGL.
- **`Window`**: Clase que abstrae la información relacionada con una ventana del sistema operativo.
- **`Renderer`**: Clase abstracta encargada de dibujar la escena. Deberás derivar una nueva clase de **`Renderer`** e implementar, al menos, las funciones **`render`** y **`reshape`**.

La aplicación mínima

A continuación se muestra el código que implementa el programa más sencillo. Dicho programa se encarga de crear una ventana y borrarla (proyecto **`ej1-2`**).

```
#include "PGUPV.h"

using namespace PGUPV;

class MyRender : public Renderer {
public:
  void setup(void) override;
  void render(void) override;
  void reshape(uint w, uint h) override;
};

void MyRender::setup() {
  glClearColor(0.1f, 0.1f, 0.9f, 1.0f);
}

void MyRender::render() {
  glClear(GL_COLOR_BUFFER_BIT);
}

void MyRender::reshape(uint w, uint h) {
  glViewport(0, 0, w, h);
}

int main(int argc, char *argv[]) {
  App &myApp = App::getInstance();
  myApp.initApp(argc, argv, PGUPV::DOUBLE_BUFFER | PGUPV::DEPTH_BUFFER);
  myApp.getWindow().setRenderer(std::make_shared<MyRender>());
  return myApp.run();
}
```

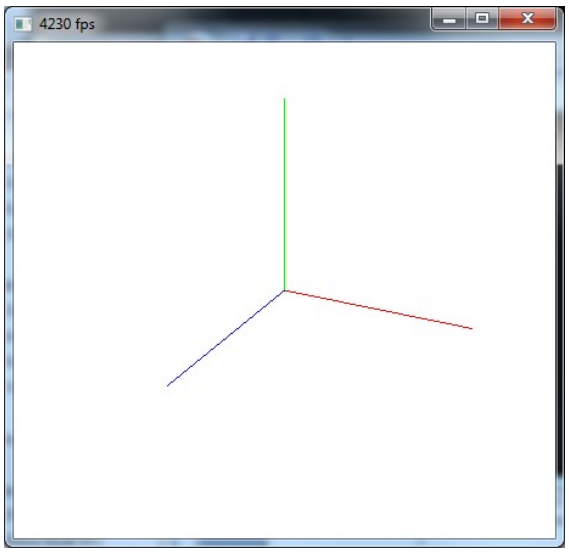
La tarea principal del programador para usar la biblioteca PGUPV es crear una clase que derive de **`PGUPV::Renderer`**, e implementar, al menos, las siguientes funciones:

- **`setup`**: esta función se invocará una vez, al arrancar la aplicación. En ella deberías inicializar variables, cargar o construir modelos, cargar y compilar shaders, etc.
- **`reshape`**: esta función se ejecuta al arrancar el programa y cada vez que el usuario cambie el tamaño de la ventana. Recibe el nuevo tamaño por parámetro y en ella tendrás que ajustar el viewport de OpenGL y, normalmente, redefinir el volumen de la cámara.
- **`render`**: en esta función se dibuja la escena en el estado actual. No hace falta que ejecutes el intercambio de buffers (*swap buffers*).

Dibujando los ejes coordenados

El siguiente ejemplo (ej1-3) dibuja los ejes coordenados (no se muestra la función **`main`**, porque es igual que antes):

```
#include <glm/gtc/matrix_transform.hpp>

#include "PGUPV.h"

using namespace PGUPV;
using glm::vec3;
class MyRender : public Renderer {
public:
  void setup(void) override;
  void render(void) override;
  void reshape(uint w, uint h) override;

private:
  Axes axes;
  Program program;
  std::shared_ptr<GLMatrices> mats;
};

void MyRender::setup() {
  glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
  program.addAttributeLocation(Mesh::VERTICES, "position");
  program.addAttributeLocation(Mesh::COLORS, "vertcolor");
  mats = GLMatrices::build();
  program.connectUniformBlock(mats, UBO_GL_MATRICES_BINDING_INDEX);
  program.loadFiles(App::assetsDir() + "shaders/constantshading");
  program.compile();
}

void MyRender::render() {
  glClear(GL_COLOR_BUFFER_BIT);
  mats->setMatrix(GLMatrices::VIEW_MATRIX,
    glm::lookAt(vec3(1, 1, 2), vec3(0, 0, 0), vec3(0, 1, 0)));
  program.use();
  axes.render();
}

void MyRender::reshape(uint w, uint h) {
  glViewport(0, 0, w, h);
  if (h == 0) h = 1;
  float ar = (float)w / h;
  mats->setMatrix(GLMatrices::PROJ_MATRIX,
    glm::perspective(glm::radians(60.0f), ar, .1f, 10.0f));
}
```



El ejemplo hace uso de un modelo disponible en la biblioteca PGUPV llamado "**`Axes`**". La biblioteca facilita la construcción de modelos abstrayendo los conceptos de *Vertex Array Object* y *Vertex Buffer Object* vistos en teoría. Hay dos clases encargadas de facilitar la construcción de modelos:

- **`Mesh`**: representa una malla constituida por un conjunto de vértices (y sus respectivos atributos) e índices. También contiene una o más llamadas a comandos de dibujo de OpenGL (**`glDrawElements`**, **`glDrawArrays`**...), que indica cómo usar dichos vértices para dibujar la malla.
- **`Model`**: representa un modelo compuesto por múltiples mallas que comparten un sistema de coordenadas común.

La clase **`Axes`** se ha derivado de la clase **`Model`**. Como puedes ver en su constructor, en el fichero **`stockModels.cpp`**, se crea un nuevo **`Mesh`** al que se le añaden los vértices, colores y una orden de dibujado **`DrawArrays`**. Después, se añade dicho **`Mesh`** al modelo.

Explorando un modelo

El último ejemplo (**`ej1-5`**), se basa en la aplicación anterior, a la que se le ha añadido una cámara. La cámara implementada en la clase **`OrbitCameraHandler`** permite estudiar un modelo con el ratón: con el botón izquierdo pulsado se puede rotar la cámara alrededor del punto de interés. Con la ruleta del ratón se puede acercar o alejar la cámara y pulsando el botón central se puede mover la cámara en dirección perpendicular a la dirección de la vista (*pan*). Los únicos cambios introducidos son:

```
using namespace PGUPV;

void MyRender::setup() {

  […]
  setCameraHandler(new OrbitCameraHandler());
}

void MyRender::render() {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  mats->setMatrix(GLMatrices::VIEW_MATRIX, getCamera().getViewMatrix());
  program.use();
  axes.render();
  if (!models.empty()) {
    models[modelSelector->get()]->render();
  }
}

void MyRender::reshape(uint w, uint h) {
  glViewport(0, 0, w, h);
  mats->setMatrix(GLMatrices::PROJ_MATRIX, getCamera().getProjMatrix());
}
```

El método **`getCamera()`** devuelve la cámara gestionada por el manejador de cámara que instalamos en el **`setup`**.
En dicho ejemplo también se muestra cómo atender a las pulsaciones de teclado (en la función **`MyRender::update`**) para llevar la cámara a su posición original al pulsar el espacio.

Tu trabajo

Tu trabajo de esta semana consiste en estudiar los ejemplos suministrados y familiarizarte con la biblioteca PGUPV. Para ello, tendrás que resolver dos ejercicios, cuyas plantillas tienes en los proyectos **`p0`** y **`p1`** de las prácticas.

Primera parte

Se parte del proyecto p0, cuya salida se muestra en la Figura 1, que dibuja un triángulo rojo.
El primer ejercicio consiste en implementar un programa que dibuje un cuadrado 2D dividido en cuatro partes iguales, con un círculo inscrito. El cuadro deberá ser negro y el

círculo rojo, dibujados con líneas. Ambos elementos estarán en el plano XY, centrados en el origen. El resultado esperado se muestra en la Figura 2.
Como hay varios vértices que se repiten al dibujar el cuadrado, deberás dibujarlo usando el modo indexado (la malla sólo debe definir 9 vértices). Como en el caso del círculo no se repiten vértices, es mejor no usar el modo indexado de dibujo.
Cada figura debe dibujarse con una única llamada de dibujo (*draw-call*).

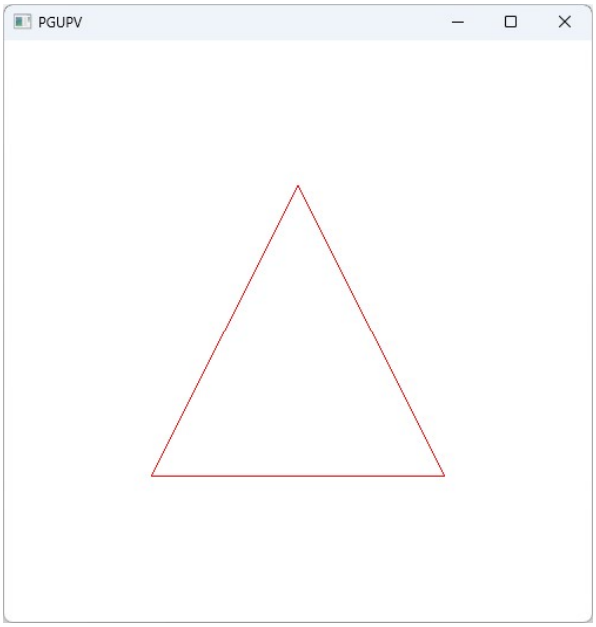


Figura 1. Ejemplo en p0

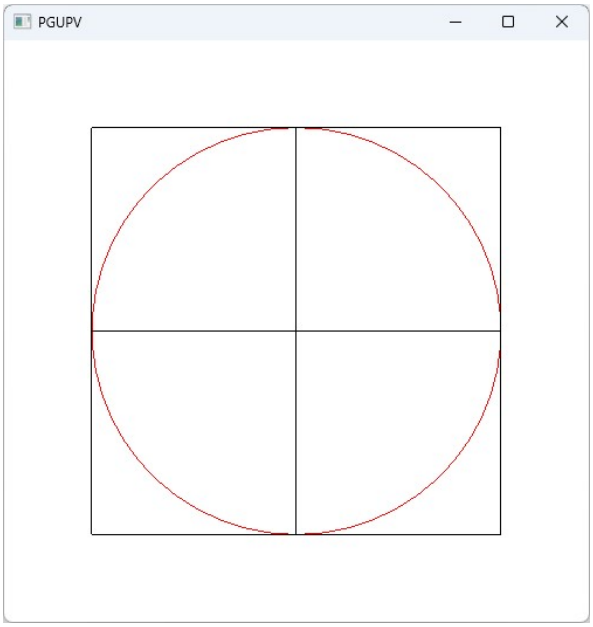


Figura 2. Resultado esperado

Segunda parte

La segunda parte de la práctica consiste en implementar un visualizador de la información catastral a nivel de partes de edificios, tal y como está publicada en la web del catastro:

https://www.catastro.hacienda.gob.es/webinspire/index.html

La directiva INSPIRE europea (Directiva 2007/2/CE, Infrastructure for Spatial Information in Europe) establece una serie de requisitos para asegurar que las fuentes de datos espaciales de los países de la UE sean compatibles e interoperables. Entre los datos geográficos que cubre la directiva, se encuentra:

- Parcelas catastrales,
- direcciones, y
- edificios

Esta información debe estar disponible en un formato predefinido, y debe ser accesible mediante protocolos de red estándar. En la página anterior se pueden encontrar enlaces a la información de todos los municipios de España.
En esta práctica nos centraremos en el conjunto de datos de edificios, y más concretamente en las partes de edificios, que ofrece la forma geométrica de los distintos

volúmenes de un edificio. El fichero correspondiente al municipio de Valencia se puede descargar desde la ruta:

http://www.catastro.minhap.es/INSPIRE/Buildings/46/46900-VALENCIA/A.ES.SDGC.BU.46900.zip

El fichero comprimido tiene varios archivos, y con el que vamos a trabajar se llama

A.ES.SDGC.BU.46900.buildingpart.gml

El formato GML (Geography Markup Language) es un estándar abierto desarrollado por el Consorcio de Recursos de Información Geoespacial (OGC) que se utiliza para representar datos geográficos de manera estructurada y legible por máquina. Se basa en el formato XML (Extensible Markup Language).
GML permite describir geometrías, características y relaciones espaciales de manera precisa y detallada, y se usa en aplicaciones de sistemas de información geográfica (GIS), cartografía digital, análisis ambiental, y otras.
Una de las características clave del formato GML es su capacidad para representar datos geoespaciales de forma semántica, es decir, proporcionando información contextual sobre la naturaleza y el significado de los datos. Esto se logra mediante el uso de esquemas XML, que especifican la estructura y el contenido de los datos GML, incluyendo la información sobre la geometría, las propiedades de los objetos geográficos y las relaciones espaciales entre ellos.
A continuación se muestra las primeras líneas del fichero con el que vamos a trabajar:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--Edificios de la D.G. del Catastro.-->
<gml:FeatureCollection gml:id="ES.SDGC.BU"  xmlns:ad="urn:x-
inspire:specification:gmlas:Addresses:3.0" […] >
 <gml:featureMember>
  <bu-ext2d:BuildingPart gml:id="ES.SDGC.BU.000100100YJ27F_part1">
   <bu-core2d:beginLifespanVersion>2008-10-20T00:00:00</bu-
core2d:beginLifespanVersion>
    <bu-core2d:conditionOfConstruction xsi:nil="true"
nilReason="other:unpopulated"></bu-core2d:conditionOfConstruction>
    <bu-core2d:inspireId>
     <base:Identifier>
      <base:localId>000100100YJ27F_part1</base:localId>
      <base:namespace>ES.SDGC.BU</base:namespace>
     </base:Identifier>
    </bu-core2d:inspireId>
    <bu-core2d:addresses
xlink:href="http://ovc.catastro.meh.es/INSPIRE/wfsAD.aspx?service=wfs&amp;versio
n=2&amp;request=GetFeature&amp;STOREDQUERIE_ID=GetadByRefcat&amp;refcat=00010010
0YJ27F&amp;srsname=EPSG::25830"></bu-core2d:addresses>
    <bu-core2d:cadastralParcels
xlink:href="http://ovc.catastro.meh.es/INSPIRE/wfsCP.aspx?service=wfs&amp;versio
n=2&amp;request=GetFeature&amp;STOREDQUERIE_ID=GetParcel&amp;refcat=000100100YJ2
7F&amp;srsname=EPSG::25830"></bu-core2d:cadastralParcels>
    <bu-ext2d:geometry>
     <bu-core2d:BuildingGeometry>
      <bu-core2d:geometry>
       <gml:Surface gml:id="Surface_ES.SDGC.BU.000100100YJ27F_part1"
srsName="urn:ogc:def:crs:EPSG::25830">
        <gml:patches>
         <gml:PolygonPatch>
          <gml:exterior>
           <gml:LinearRing>
```

```
            <gml:posList srsDimension="2" count="8"> 725590.2005 4377136.1625
725589.5805 4377142.0325 725594.8705 4377140.2825 725594.7705 4377139.8825
725603.3505 4377137.3925 725601.9705 4377132.9425 725599.7505 4377133.5525
725590.2005 4377136.1625</gml:posList>
            </gml:LinearRing>
           </gml:exterior>
          </gml:PolygonPatch>
         </gml:patches>
         </gml:Surface>
       </bu-core2d:geometry>
       <bu-core2d:horizontalGeometryEstimatedAccuracy uom="m">0.1</bu-
core2d:horizontalGeometryEstimatedAccuracy>
       <bu-core2d:horizontalGeometryReference>footPrint</bu-
core2d:horizontalGeometryReference>
       <bu-core2d:referenceGeometry>true</bu-core2d:referenceGeometry>
      </bu-core2d:BuildingGeometry>
     </bu-ext2d:geometry>
     <bu-ext2d:numberOfFloorsAboveGround>1</bu-ext2d:numberOfFloorsAboveGround>
     <bu-ext2d:heightBelowGround uom="m">0</bu-ext2d:heightBelowGround>
     <bu-ext2d:numberOfFloorsBelowGround>0</bu-ext2d:numberOfFloorsBelowGround>
    </bu-ext2d:BuildingPart>
   </gml:featureMember>
   <gml:featureMember>
    <bu-ext2d:BuildingPart gml:id="ES.SDGC.BU.000100100YJ27F_part2">
      <bu-core2d:beginLifespanVersion>2008-10-20T00:00:00</bu-
core2d:beginLifespanVersion>
      <bu-core2d:conditionOfConstruction xsi:nil="true"
nilReason="other:unpopulated"></bu-core2d:conditionOfConstruction>
      <bu-core2d:inspireId>
       <base:Identifier>
         <base:localId>000100100YJ27F_part2</base:localId>
         <base:namespace>ES.SDGC.BU</base:namespace>
       </base:Identifier>
     </bu-core2d:inspireId>
[…]
```

Un *buldingpart* es cada una de las construcciones de una parcela catastral con un volumen homogéneo (tiene el mismo número de plantas), y pueden ser sobre y bajo rasante. Alguno de los atributos son:

- `gml:featureMember`: Estructura que contiene cada parte de edificio.
- `bu-ext2d:BuildingPart`: Estructura de cada parte de un edificio con un id.
- `bu-core2d:beginLifespanVersion`: Fecha de cuándo se ha dado de alta en la base de datos catastral.
- `bu-core2d:inspireId`: Es el identificador único para todos los conjuntos de datos de INSPIRE. Está compuesto por una estructura “base:Identifier” que contiene 2 valores:
  - `base:localId`: Son los 14 primeros caracteres de la referencia catastral en la que se encuentra el edificio, más un sufijo secuencial "_partX"
- `bu-core2d:addresses`: Objeto dirección, mediante un “xlink:href” se accede al servicio WFS de la dirección/s asociados al edificio.
- `bu-core2d:cadastralParcels`: Objeto parcela catastral, mediante un “xlink:href” se accede al servicio WFS de la parcela catastral asociados al edificio.
- `bu-ext2d:geometry`: Geometría de parte de edificio en GML. Tiene una estructura GML “`gml:Surface`”. La geometría se define por las coordenadas de los vértices en un anillo exterior y pueden existir huecos que se definen en una estructura de anillo interior. La lista de coordenadas de los anillos (`gml:postList`) duplican el primer y último vértice, en el anillo exterior.

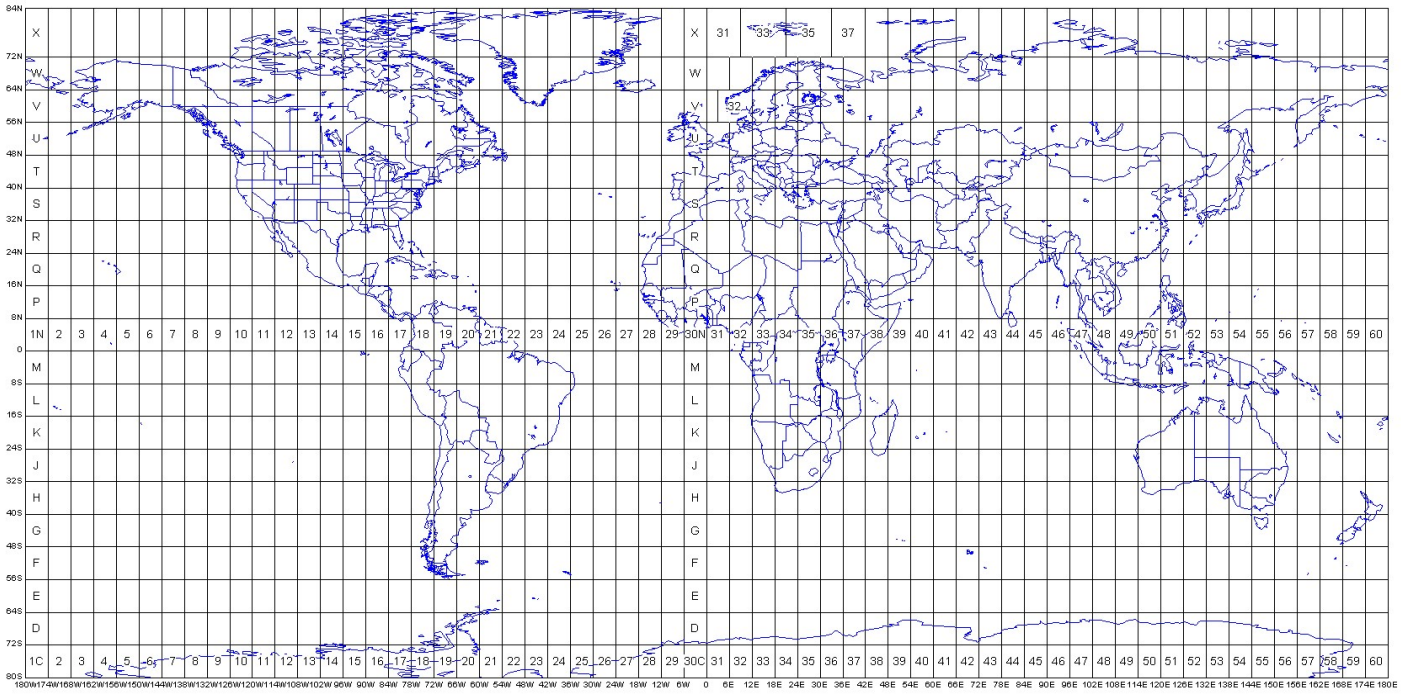
  - el orden es el de las agujas del reloj y en los interiores es el contrario, el sistema de referencia es el definido en "`srsName`".
- `bu-ext2d:numberOfFloorsAboveGround`: Cantidad de plantas sobre rasante.
- `bu-ext2d:heightBelowGround`: Altura de plantas bajo rasante en metros. Es una altura estimada de 3 metros por planta.
- `bu-ext2d:numberOfFloorsBelowGround`: Cantidad de plantas bajo rasante.

En la Figura 3 se muestra un ejemplo de varios edificios, y cómo se representan en el fichero del catastro. Fíjate cómo se define un polígono (anillo), por cada volumen del edificio que tenga la misma altura.

Figura 3. Un conjunto de edificios y su representación en el catastro

Las coordenadas vienen definidas en el sistema de coordenadas indicado en el fichero GML. En el ejemplo, es `EPSG::25830`. EPSG hace referencia al *European Petroleum Survey Group*, una entidad que entre 1986 y 2005 publicó distintos estándares orientados a la geodesia, la topografía y la cartografía. Los sistemas de coordenadas geográficos especifican la forma de asociar unas coordenadas matemáticas a cualquier punto del planeta. Un ejemplo es el uso de las coordenadas geográficas latitud, longitud, normalmente definidas en el sistema WGS 84, que es el sistema de referencia del GPS. Dicho sistema representa la Tierra mediante un elipsoide con una longitud de semiejes de 6378137m y 6356752.314 m. El meridiano de longitud 0 está situado a 102 metros del meridiano de Greenwich.
A pesar de ser muy utilizado, el sistema de coordenadas WGS 84 es engorroso de utilizar, ya que la distancia en metros que representa un grado varía dependiendo de la posición de la Tierra. Por ello, normalmente se usan otros tipos de sistemas de coordenadas, en los que las operaciones son más sencillas. El sistema de coordenadas EPSG::25830 se corresponde al conjunto de ETRS89 / UTM zone 30N. ETRS89 es el elipsoide oficial europeo, de uso en España. Es muy parecido a WGS 84.
El Sistema de Coordenadas Universal Transversal de Mercator (UTM) es un sistema de coordenadas utilizado para representar posiciones en la superficie terrestre en forma plana, lo que facilita su manipulación y análisis en sistemas informáticos y aplicaciones geoespaciales.
UTM divide la Tierra en 60 zonas longitudinales, numeradas del 1 al 60, cada una de las cuales abarca 6 grados de longitud. Estas zonas se extienden desde los 80 grados de latitud sur hasta los 84 grados de latitud norte. Cada zona está orientada de manera que su eje central coincide con el meridiano central de la zona, lo que minimiza la distorsión de la proyección cartográfica (ver la siguiente imagen):

https://www.dmap.co.uk/utmworld.htm



Dentro de cada huso UTM, las coordenadas se expresan en metros, utilizando un sistema de coordenadas cartesianas bidimensional. En el hemisferio norte, la coordenada Y mide la distancia al ecuador. Para la coordenada X, el valor 500 km

(500000 m) está en el meridiano central del huso. Por ello, todas las coordenadas dentro de cada huso son positivas.
No es necesario que leas el fichero GML. La práctica se acompaña de una clase de utilidad que cargar en memoria un fichero GML y ofrece una clase para obtener la información del mismo.

La función:

```
City readBuildings(const std::string& fname, bool loadInteriors);
```

del fichero glmReader.h se encarga de leer el fichero GML que se le pasa en el primer parámetro. Dependiendo del parámetro `loadInteriors`, lee los huecos de los edificios o no.
La estructura City contiene un vector con todos los edificios, e información adicional. Cada edificio tiene como identificador la parcela catastral y un vector de partes de edificio. Cada parte de edificio tiene una secuencia de coordenadas exteriores (2D, en el sistema de coordenadas indicado), en sentido horario, y una o más secuencias de coordenadas interiores, en sentido antihorario.

En esta parte de la práctica se pide:

1. Carga el fichero de partes de edificio que te has bajado desde la web del Catastro, usando el método `readBuildings`.
2. Construye uno o más modelos para representar gráficamente la información cargada. Ten en cuenta:
    a. Las coordenadas almacenadas en GML están en formato double (64 bits). Para ahorrar espacio, asegurar la compatibilidad con sistemas limitados y acelerar el render, deberás usar glm::vec2 (cuyas coordenadas son float de 32 bits).
    b. Piensa la manera de utilizar al máximo la resolución del float, porque puedes tener problemas en la conversión.
    c. En un ordenador actual, la aplicación debería moverse interactivamente (por tener una referencia, una Nvidia RTX 3070, dibuja todos los polígonos a más de 500 FPS). Intenta reducir el número de draw-calls.
3. Haz que los huecos sólo se visualicen a partir de cierto nivel de zoom (no serán visibles cuando haya mucha información en pantalla).
    a. Con
    `std::static_pointer_cast<XYPanZoomCamera>(getCameraHandler())`
    `puedes obtener el puntero al manejador de cámara,`
4. Implementa el inspector de coordenadas que se muestra en la Figura 5. Para ello, introduce tu código en el método `MyRender::mouse_move`. Para ello, tendrás que trabajar con distintos sistemas de coordenadas:
    a. El de la ventana, en píxeles. La coordenada del ratón está en `me.x, me.y`. Cuidado con la coordenada y, ya que viene dada con respecto a la esquina superior izquierda, mientras que en UTM, las coordenadas verticales crecen hacia arriba. El tamaño de la ventana está disponible en `windowSize.x, windowSize.y`
    b. El de la cámara, en las unidades en que trabajes. El punto central de la ventana está en `cam->getCenter()` y, el tamaño de la ventana es `cam->getWidth(), cam->getHeight()`, de nuevo, en las unidades en que trabajes. La configuración inicial de la cámara se especifica en la llamada:

```
setCameraHandler(std::make_shared<XYPanZoomCamera>(<ancho>, <centro>));
```

Partes del ejemplo p1, que implementa una cámara ortográfica que se puede mover con el botón izquierdo del ratón y hacer zoom con la ruleta. A continuación se muestran algunas capturas del resultado esperado. También puedes contrastar tu solución con la aplicación SignA del Instituto Geográfico Nacional:
https://signa.ign.es/signa/

Asegúrate de seleccionar el sistema de coordenadas adecuado en la esquina inferior derecha de esa herramienta:

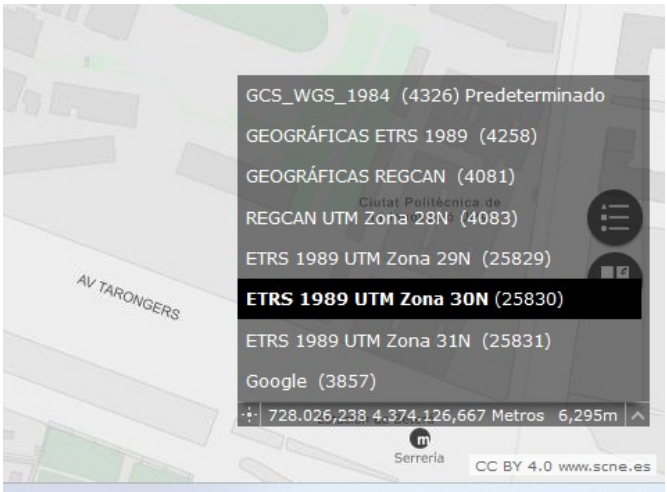


Figura 4. Selecciona el sistema de coordenadas correcto.

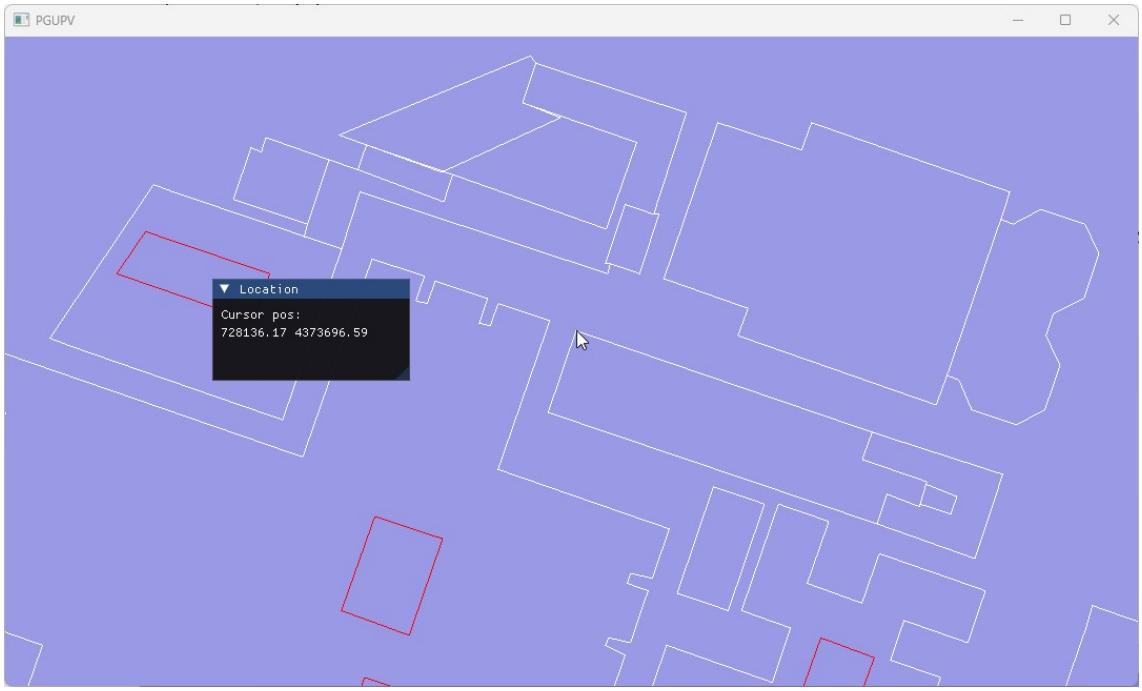


Figura 5. Ejemplo del resultado esperado. La ventana Location da la posición del cursor en tiempo real

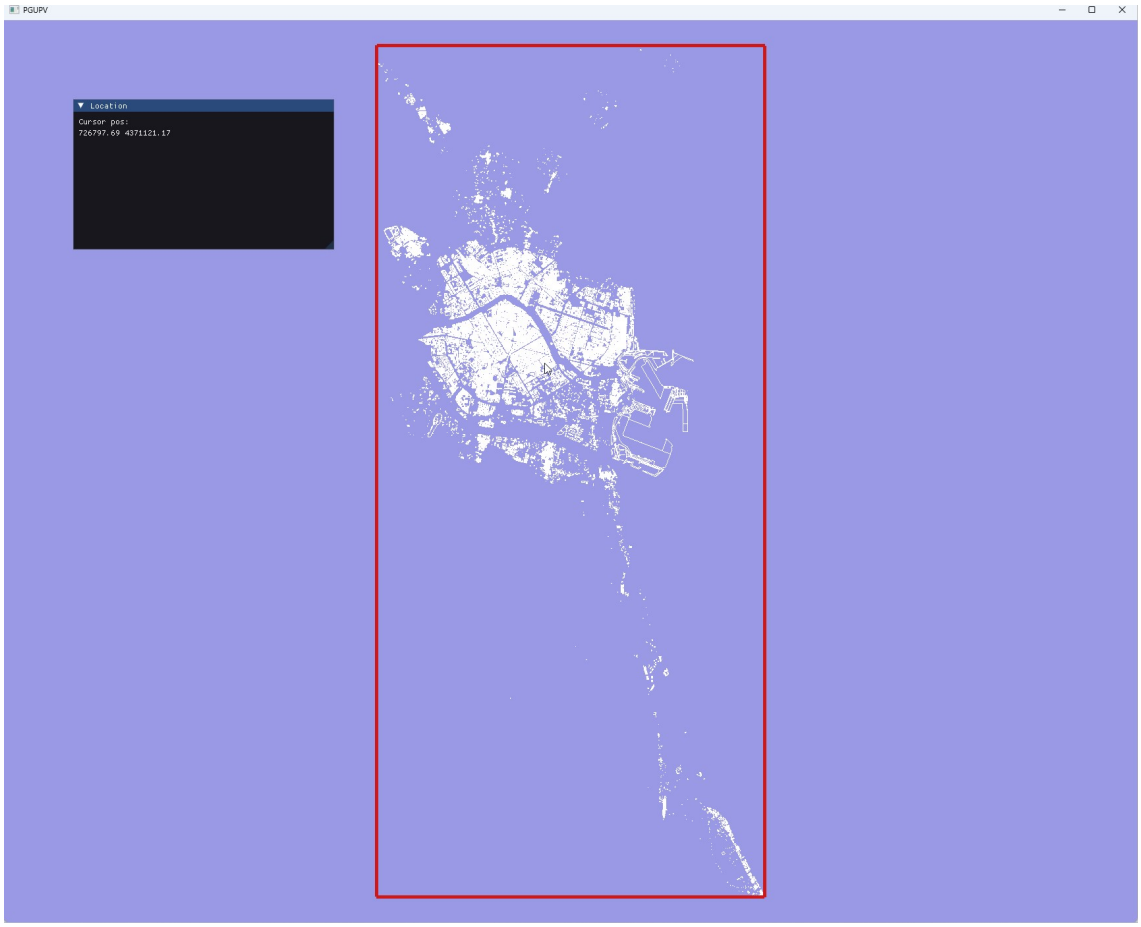


Figura 6. Vista completa de todo el municipio

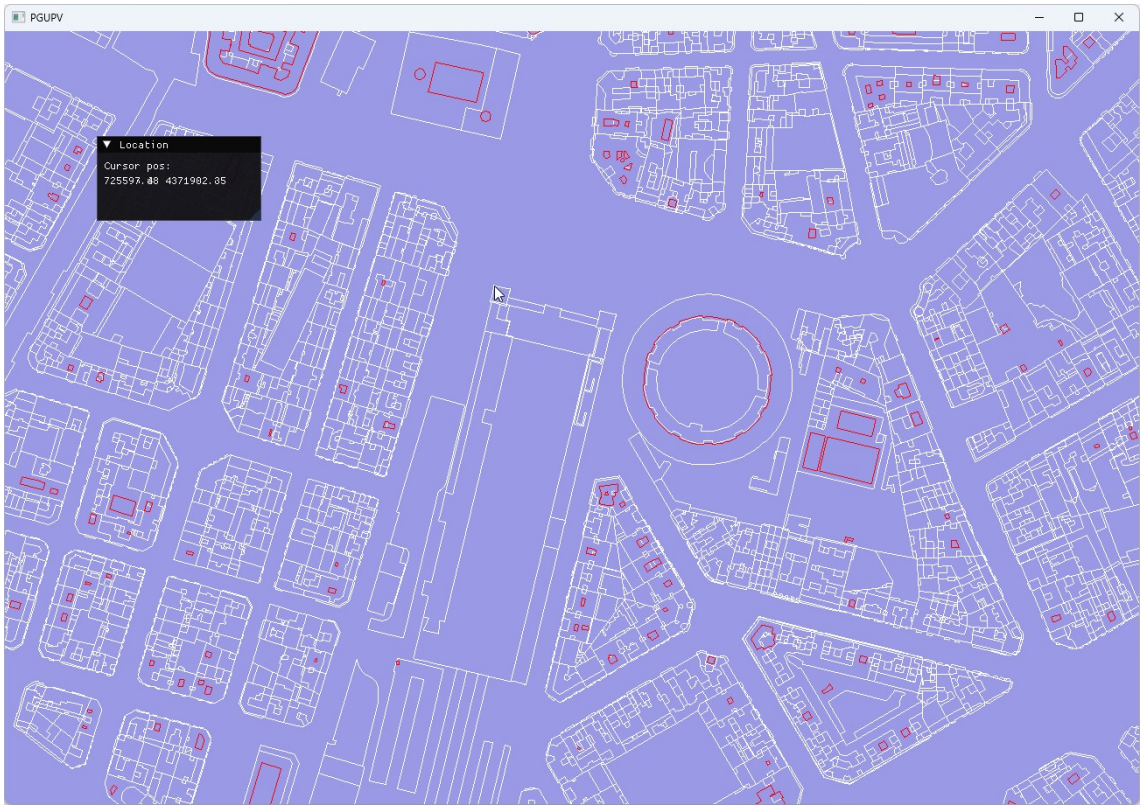


Figura 7. Estación del Norte

Cualquier proyecto que use la biblioteca PGUPV tiene predefinidos una serie de atajos de teclado que iremos introduciendo durante el curso. Aquí tienes algunos:

- Ctrl+M: cambiar el modo de dibujo entre polígonos, aristas o vértices
- Ctrl+S: guardar una imagen con el contenido de la ventana
- Ctrl+F: mostrar la aplicación a pantalla completa
- Ctrl+I: mostrar por consola información sobre OpenGL

Extensión opcional
Implementa un sistema de "tiles", subdividiendo la geometría de los edificios en bloques de tamaño uniforme. Cada tile estará asociado a un modelo de PGUPV. A continuación, antes de dibujar cada frame, se calculará qué tiles están dentro del volumen de la cámara, y sólo se dibujarán éstos. De esta forma, se reduce la cantidad de información que se manda a la GPU que realmente no es visible porque está fuera de la ventana.