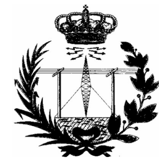
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

Graphical User Interface for the project "dynamic and scalable large scale image reconstruction"



AUTOR: Domingo José Pérez Sánchez
DIRECTOR: José Luis Gómez Tornero
Cartagena, Septiembre 2012

Autor	Domingo José Pérez Sánchez
E-mail del Autor	domingo.jperez@hotmail.com
Director(es)	José Luis Gómez Tornero
E-mail del Director	josel.gomez@upct.es
Codirector(es)	Alexandros Feresidis
Título del PFC	**Graphical User Interface for the project "dynamic and scalable large scale image reconstruction"**

Resumen

En este proyecto se lleva a cabo la implementación de una interfaz gráfica que dé uso al proyecto inicial de escalado de imágenes. La idea principal es que el usuario pueda manejar la información dada en forma de nube de puntos para recrear edificios. En el proyecto se pretende crear una interfaz gráfica lo más fácil y manejable posible para el usuario, haciendo todos los cálculos matemáticos de forma transparente. Esos cálculos consistirán en aproximar de la mejor forma posible los edificios elegidos por el usuario en las imágenes al entorno tridimensional. Por último, también se permitie guardar toda la información y datos de los edificios recreados.

Titulación	Ingeniero técnico de Telecomunicación, especialidad Telemática
Intensificación	
Departamento	
Fecha de Presentación	Julio de 2012

Índice

1. Introducción 6
2. Background 7
3. Desarrollo del proyecto 10
3.1. Introducción a las clases 10
3.2. Funcionamiento y código de las clases implementadas 12
3.2.1. ImageWidget 12
3.2.2. ImageView 25
3.2.3. TrackCloudDrawer 32
3.2.4. Viewer 35
3.2.5. Parameters 44
3.2.6. Clases para la reconstrucción de edificios 45
3.3. Comunicación entre clases (signals y slots) 50
3.4. Manual de usuario 52
4. Resultados 57
4.1. Análisis de resultados obtenidos dependiendo de las muestras elegidas y algoritmo utilizado. 57
4.2. Diferencia visual en el espacio tridimensional entre los planos calculados por los distintos algortimos 67
4.3. Ejemplo de reconstrucción de un edificio 68
5. Conclusiones 72
6. Bibliografía 73

1. Introducción

Se parte de un proyecto desarrollado por el “computer vision lab” de la EPFL. El proyecto se denomina “Dynamic and scalable large scale image reconstruction”[1], siendo su función principal la de reconstruir diferentes áreas de ciudades a partir de una galería amplia de imágenes. Se pretende crear un entorno que permita, a partir de los meta-datos disponibles, crear una descripción de estos datos consistente y única a partir de toda la información disponible.

El objetivo del proyecto es crear una interfaz gráfica para ofrecer un uso sencillo de la información disponible en los meta-datos para cualquier usuario. Los meta-datos son representados como una reconstrucción tridimensional haciendo uso del lenguaje c++, con las funcionalidades de la librería OpenGL, lo que mostrará un entorno navegable donde el usuario podrá observar una nube de puntos representativos de un área concreta.

Se añade la posibilidad de que el usuario pueda observar todas las imágenes que contienen cierto punto de la información mostrada. Además, usando dichas imágenes, el usuario podrá representar planos de edificios, como fachadas o tejados, en el entorno tridimensional anterior o bien guardar esa información para un posterior procesado y reconstrucción.

2. Background

-**NokiaQT**: Espacio de programación

Qt es una biblioteca multiplataforma ampliamente usada para desarrollar aplicaciones con interfaz gráfica de usuario, así como para el desarrollo de programas sin interfaz gráfica, como herramientas para la línea de comandos y consolas para servidores. Qt se desarrolla como un software libre y de código abierto a través de Qt Project, donde participa tanto la comunidad, como desarrolladores de Nokia, Digia y otras empresas.

Utiliza KDE, entorno de escritorio para sistemas como GNU/Linux o FreeBSD, emtre otros y el lenguaje de programación C++ de forma nativa, aunque adicionalmente puede ser utilizada en varios otros lenguajes de programación a través de bindings. Funciona en todas las principales plataformas, y tiene un amplio apoyo. El API de la biblioteca cuenta con métodos para acceder a bases de datos mediante SQL, así como uso de XML, gestión de hilos, soporte de red, una API multiplataforma unificada para la manipulación de archivos y una multitud de otros para el manejo de ficheros, además de estructuras de datos tradicionales.

-**OpenGL:** Librería necesaria para representar la información

Interfaz software para hardware gráfico formada por más de 700 comandos distintos, la mayoría especificados para la version 3.0 de OpenGL y unos pocos destinados a las librerías de utilización de OpenGL. Estos comandos se usan para especificar los objetos y operaciones necesarias al generar aplicaciones interactivas tridimensionales.

OpenGL está diseñado como un proceso simplificado e independiente de la interfaz hardware para poder ser implementado así en muchas plataformas hardware distintas. Para lograr estas cualidades, no sólo existirán comandos que realicen esta tareas de ventanas, así cómo la obtención de la entrada del usuario, incluidas en OpenGL, sino que permitirá que el sistema también pueda trabajar a través de cualquier ventana controlando el hardware en particular que esté utilizando en ese momento.

Con OpenGL se construirá el modelo deseado a partir de un pequeño conjunto de primitivas geométricas de puntos, líneas y polígonos.

-**C++:** lenguaje de programación utilizado

C++ es un lenguaje imperativo orientado a objetos derivado del lenguaje C. En realidad un superconjunto de éste, que surgió para añadir cualidades y características de las que C carecía. El resultado es que, como su antecesor C sigue muy ligado al hardware subyacente, manteniendo una considerable potencia para programación a bajo nivel, se han añadido elementos que le permiten también un estilo de

programación con alto nivel de abstracción. Estrictamente hablando, C no es un subconjunto de C++; de hecho es posible escribir código C que sea ilegal en C++. Pero a efectos prácticos, dado el esfuerzo de compatibilidad desplegado en su diseño, puede considerarse que C++ es una extensión del C clásico. La definición "oficial" del lenguaje nos dice que C++ es un lenguaje de propósito general basado en el C, al que se han añadido nuevos tipos de datos, clases, plantillas, mecanismo de excepciones, sistema de espacios de nombres, sobrecarga de operadores, referencias, operadores para manejo de memoria persistente, y algunas utilidades adicionales de librería (en realidad la librería Estándar C es un subconjunto de la librería C++).

-**Eigen**: Librería

Eigen es una librería de C++ que actúa cómo plantilla para realizar cálculos de álgebra lineal cómo matrices, vectores y algoritmos relacionados. Entre sus mejores atributos encontramos que es muy versátil y compatible con todos los tamaños de matriz, descomposiciones de éstas, características geométricas, así cómo con todo tipo numérico estándar. También es fácilmente extensible a tipos numéricos personalizados, así cómo rápido y confiable, ya que los algoritmos son cuidadosamente seleccionados y probados a través de su banco de pruebas. El API que utiliza es muy cercano a los programadores de C++, gracias a las plantillas de expresión.

Eigen tiene un buen soporte de compilador ya que periódicamente lanza una serie de pruebas para garantizar la fiabilidad y evitar cualquier error de compilación. Dispone de unos los tiempos de compilación muy razonables.

3. Desarrollo del proyecto

3.1. Introducción a las clases

El código del programa está compuesto por un proyecto llamado viewer. Este proyecto tendrá su carpeta de cabeceras y su carpeta de códigos fuente. Dentro de ambas carpetas hay creada una subcarpeta con el nombre de core donde irán los ficheros encargados de obtener y guardar la información sobre la posterior representación tridimensional. Dentro de la carpeta core tendremos las siguientes clases:

- *feature.h ; feature.cpp*

  Una de las clases más importantes del proyecto. Esta clase Feature hace referencia a un punto de la nube de puntos creada y contiene información sobre dicho punto. Su posición en cada momento de la representación mediante dos coordenadas x e y, un factor de escalado scale, el número de cámara al que pertenece camIndex y el índice de punto dentro de ese número de cámara index.

- *track.h ; track.cpp*

  Es la representación de los puntos (Feature) pasados al plano de tres dimensiones. Incluye un método llamado rotate_and_translate que como su nombre indica se dedica a recalcular la posición del punto cuando se porduce un movimiento de rotación o traslación en la imagen de la pantalla donde se está visualizando la aplicación.

- *trackcloud.h ; trackcloud.cpp*

  Es lo que se conoce como la nube de puntos. Contiene todos los puntos o Tracks que componen la representación tridimensional.

- *camera.h ; camera.cpp*

  Contiene la información de cada la cámara en el momento en el que fue tomada cada imagen. Como atributo más usado tendremos un QString que contiene el nombre de la imagen que esa cámara ha fotografiado.

- *cameracloud.h ; cameracloud.cpp*

  Está compuesto por un QVector de la clase Camera, que hace referencia a un array el cual contiene todos los elementos disponibles de la clase Camera.

- *trianglesoup.h ; trianglesoup.cpp*
  Se encarga de la triangulación entre la cámara y los puntos, tan sólo se compone de un vector de números de tipo float.

- *mesh.h ; mesh.cpp*
  Esta clase contiene en su interior toda la información para mostrar por pantalla lo que se desea. Puede contener tanto una nube puntos Trackcloud, como una nube de cámaras Cameracloud o relaciones de triangulación Trianglesoup. Todo ello se puede guardar o cargar a un archivo de extensión ply.

- *ply.h ; ply.cpp*
  Archivo para cargar o guardar información de un archivo de tipo ply.

- *reconstfilemanager.h ; reconstfilemanager.cpp*
  Se encarga del tratamiento de los ficheros que puede manejar el programa. Puede abrir, cerrar, cargar o guardar estos ficheros.

- *calibration.h ; calibration.cpp*
  Contiene todas las funciones y proyecciones que se pueden realizar a partir de la información de las cámaras. Trabaja con los puntos y su misión es proporcionar funciones para el tratamiento en la representación de estos puntos.

De nuevo en la carpeta principal tenemos que comenzar por el fichero principal main.cpp, el cual se encarga de inicializar el programa. Lee los parámetros de entrada, identificando el tipo de archivo que se le ha pasado y dejando que sea la clase Reconstfilemanager la encargada de, a partir del fichero de entrada, obtener si fuera necesario el resto de ficheros para poder dibujar la nube de puntos. Además esta rutina principal, como es lógico, inicializa los valores de las ventanas que dispondrá el usuario para interactuar, y en qué posición del monitor se colocan. Por último creará un atributo de la clase tipo viewer, a partir del cual se seguirá trabajando para darle distintas utilidades de cara al usuario final que lo utilizará.

Descripción del resto de clases:

- *viewer.h ; viewer.cpp*
  Viewer es la clase contenedor general de todo el proyecto. En ella se va a incluir toda la información con la que el usuario va a navegar a través de la interfaz y cuando cambie alguna circunstancia o ejecute alguna opción en esta pantalla principal, la clase viewer también es la encargada de recoger esas acciones y actuar en consecuencia. Para su inicialización se necesita una nube de puntos, una nube de cámaras y opcionalmente una ruta a un archivo de tipo ply.

- *parameters.h ; parameters.cpp*
  Esta clase se encarga de establecer ciertos parámetros sobre la representación gráfica que realiza la clase Viewer. En concreto se comunica con ella para

establecer un centro de rotación y un tamaño del cubo para tomar más o menos muestras alrededor de un punto seleccionado.

- *imageview.h ; imageview.cpp*
  Es la interfaz creada a partir de la elección de un punto de la nube de puntos. Se crea una clase por cada imagen. En esta clase se incluyen todos los tipos de elementos gráficos que se pueden utilizar sobre la imagen para elegir sobre ella distintos planos. También se incluyen señales de eventos para la interacción del usuario con la interfaz donde se encuentra la imagen.

- *imagewidget.h ; imagewidget.cpp*
  Esta clase contiene un vector de Imageview, por tanto es la que se encarga de su manejo externo y de pasar la información que se modifique en una de las clases Imageview al resto. Esta información también debe ser comunicada al viewer para actuar sobre la primera interfaz. Como atributos tiene todas las opciones que le queramos dar a nuestra interfaz, su misión es importantísima para el buen resultado de la aplicación, pues capturar bien las coordenadas y pasarlas de forma correcta asegurará unos datos fiables para todos los calculos que se deben realizar en la reconstrucción de los planos finales.

- *cameraCloudDrawer.h ; cameraCloudDrawer.cpp ; trackCloudDrawer.h ; trackCloudDrawer.cpp ; triangleSoupDrawer.h ; triangleSoupDrawer.cpp*
  Las clases Drawer se han diseñado para implementar métodos que se encarguen de dibujar elementos en el entorno tridimensional de la clase a la que dan nombre, así por ejemplo cameraCloudDrawer se encargará de dibujar las nube de cámaras, trackCloudDrawer se encargará de la nube de puntos y triangleSoupDrawer de las triangulaciones entre elementos. Todos estos métodos son usados por la clase Viewer, por lo que las clases Drawer también implementan otros métodos con información adicional para el manejo de las clases cameraCloud, trackCloud y triangleSoup.

- *polygonData2d.h ; polygonData2d.cpp ; polygon.h ; polygon.cpp ; plane.h ; plane.cpp ; building.h ; building.cpp*
  Estas clases se encargan de componer todos los elementos que el usuario desee representar sobre las imágenes. Partiendo de una clase general, Building, la cual representa un edificio que contiene uno o varios elementos de la clase Plane o planos. Los planos a su vez contendrán uno o varios polígonos, clase Polygon, y por último dichos polígonos estarán formados por puntos, clase PolygonData2d.

3.2. Funcionamiento y código de las clases implementadas

3.2.1. ImageWidget

Esta clase es generada por la clase Viewer para manejar las imágenes seleccionadas por el usuario. Es la encargada de ejecutar una segunda interfaz donde el usuario es capaz de visualizar las imágenes y elegir los planosque desee representar.

Comenzando por la cabecera se incluyen los siguientes métodos:

```
        ImageWidget(CameraCloudPtr camera,TrackCloudPtr trackCl,Track
t,CameraCloudDrawer& ccd,TrackCloudDrawer& tcd,int radius,QWidget *parent = 0);
    float* getCorners();
    void fitPlane();
    QVector<QImage*> getImages2();
    QVector<int> getIdCamsFix();
    QVector<double> getPointsCamFix();
```

Mientras que los atributos se irán viendo conforme se necesite para el correcto desarrollo de estos métodos.

En la inicialización del constructor, se crea una interfaz de tal forma que las imágenes se distribuyan dentro de la pantalla, separadas por etiquetas, y en la parte inferior una serie de botones para realizar las acciones. Por lo tanto, la interfaz queda compuesta por cuatro layouts. El principal, que va a contener al resto, será uno de tipo QTabWidget ,para situar las imágenes bajo distintas etiquetas, y por último dos layouts más del tipo QHBoxLayout, que incluyen los botones inferiores.

Código:

```
    //Create interface
    QString *s=new QString("FIX 2D COORD");
    buttonFix2d=createButton(s);
    s->clear();
    s->append("NEW PLANE");
    buttonPlane=createButton(s);
    s->clear();
    s->append("NEW BUILDING");
    buttonBuilding=createButton(s);
    s->clear();
    s->append("SAVE ALL DATA");
    buttonSave=createButton(s);
    s->clear();
    s->append("NEW POLYGON");
    buttonPolygon=createButton(s);
    s->clear();
    s->append("outside");
    checkOut=new QCheckBox(*s);
    lay=new QGridLayout;
    layT=new QTabWidget;
    layH=new QHBoxLayout;
    layH2=new QHBoxLayout;
    this->setLayout(lay);
    this->setWindowTitle("IMAGES");
    nPol=new QLineEdit("number of sides (3-12)");
    lay->addWidget(layT,0,0);
    lay->addLayout(layH,1,0);
    layH->addWidget(nPol);
    layH->addWidget(checkOut);
    layH->addWidget(buttonPolygon);
    lay->addLayout(layH2,2,0);
    layH2->addWidget(buttonFix2d);
    layH2->addWidget(buttonPlane);
    layH2->addWidget(buttonBuilding);
    layH2->addWidget(buttonSave)
```

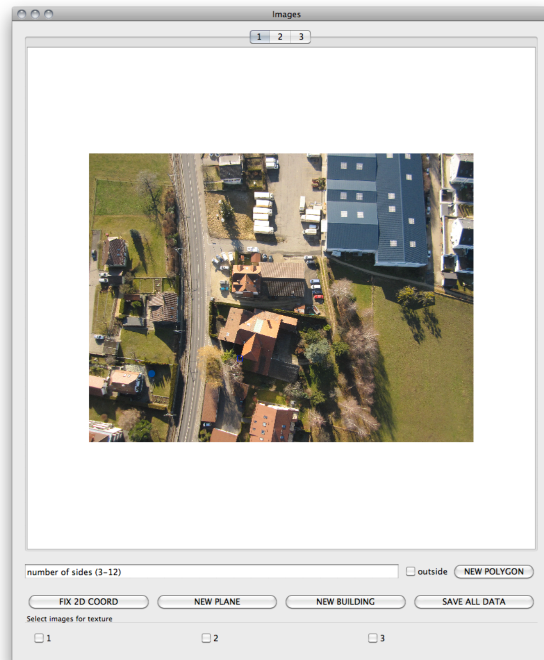


Ilustración 1 Ejemplo de interfaz *ImageWidget*

A continuación se detalla el funcionamiento de cada elemento de la interfaz.

La primera ejecución de forma transparente al usuario es lo que denominamos como "fit plane". Este método pertenece a la propia clase ImageWidget y consiste en seleccionar todos los puntos que se encuentren dentro de cierta área definida por el usuario en la anterior interfaz. Si recordamos, se había definido un cubo con cierto valor de arista, y eso es lo que se utiliza ahora como frontera para discriminar unos puntos de otros y determinar sólo válidos aquellos que se encuentren dentro de dicho cubo. Se ha utilizado un cubo por comodidad a la hora de su representación en OpenGL, pero lo más usado es, sin embargo, una esfera alrededor del punto central para así tener un radio equidistante al resto de puntos si estos se encuentran justo en la frontera. Aún así, ambas figuras ofrecen resultados similares y la eficiencia del resultado dependerá más del tamaño de dicha figura o del plano que después se desee representar que en el tipo de figura en sí.

Una vez seleccionados todos estos puntos o Tracks, hay que usar el método implemementado por la clase TrackCloudDrawer llamado fitplane(), el cual devuelve la ecuación general de un plano a partir de los puntos en 3D.

Código:

```
void ImageWidget::fitPlane()
{
   QVector<Track*> array_t;
   QVector<double> tracksToDouble;
   //POINTS INTO A CUBE WITH the selectPoint as its center,
   //it could be also a sphere.
   for(int i=0;i<trackCl->size();i++)
   {
      Track *tr=&(trackCl->getTrack(i));
      if(tr->x()>t.x()-radius & tr->x()<t.x()+radius)
      {
```

```
        if(tr->y()>t.y()-radius & tr->y()<t.y()+radius)
        {
          if(tr->z()>t.z()-radius & tr->z()<t.z()+radius)
          {
            array_t.append(tr);
            tracksToDouble.append(tr->x());
            tracksToDouble.append(tr->y());
            tracksToDouble.append(tr->z());
          }
        }
      }
    }
    qDebug()<<"there is"<<array_t.size()<<"points inside :)";
    plane=tcd.fitPlane(tracksToDouble);
}
```

Una vez que tenemos el “fit plane” , éste se usará en el desarrollo de algunas de las acciones que el usuario puede realizar.

En lo referente a acciones llevadas a cabo por el usuario, la acción inicial es solicitar por pantalla un número de lados de un polígono, y al mismo tiempo, la opción de decidir si queremos que forme parte o no del plano. El número mínimo de lados debe ser 3 y el máximo 13 lados, una vez que se cumple esta condición se envía una señal a cada clase ImageView para que acepten la selección de puntos sobre ellas. El punto seleccionado sobre cada imagen es proyectado sobre el fit plane calculado anteriormente, y una vez se tiene este punto se proyecta sobre las demás imágenes. Este método no resulta muy preciso, sólo nos da una estimación, pues hay que tener en cuenta que el fitplane es un plano calculado como la media de una nube de puntos.

Código:

```
//1 - Line edit -> to choose the number of sides of the polygon.
void ImageWidget::polSides(){
  if(checkOut->isChecked()){
    outside=true;
  }
  else{
    outside=false;
  }
  if(nPol->text().toInt()>2 && nPol->text().toInt()<13){
    nSidesPolygon.append(nPol->text().toInt());//Save the sides of the polygons
    sumSidesPolygon+=nSidesPolygon.last();
    outsidePolygon.append(outside);//Save if the polygon ar in or out
    emit nPointsPoly(nSidesPolygon.last());//To imageView
  }
}
```

El segundo paso, que es opcional, permite al usuario añadir al plano creado más información, bien añadiendo otro plano o bien eliminando parte del plano creado por anteriores polígonos. Un ejemplo de ello sería poder dejar en una fachada el hueco de una ventana. Para esta acción disponemos del botón “New Polygon”, cuya ejecución conlleva, en primer lugar, una comprobación de que el polígono anterior se ha completado y, en segundo lugar, la acción de añadir las nuevas coordenadas almacenadas en la variable corners, a la variable global *allcorners*.

Código:

```
//2 - Button "New Polygon" -> If the user wants to add a new polygon.
void ImageWidget::setPolygon(){
   if(corners.size()==sumSidesPolygon*3){
      if(corners.size()!=allCorners.size()){
         int j=0;
         for(int i=0;i<nSidesPolygon.size()-1;i++){
            j+=nSidesPolygon[i];
         }
         for(j=j*3;j<corners.size();j++){
            allCorners.append(corners[j]);
         }
         qDebug()<<"AllCorners.size()="<<allCorners.size();
         activeFix=true;
         //corners.clear();
      }
      else{
         //Step already done
      }
   }
}
```

El tercer paso es, una vez elegido el plano en una fotografía, fijarlo, es decir, comprobar que los puntos elegidos estén bien situados y, a continuación, pulsar el botón “Fix 2D Coord”. Aunque con el cubo exterior y el fix plane ya se obtendría una representación, al no ser ésta muy fiable, se necesitará de al menos dos imágenes distintas de información del plano que se quiere representar.

Por ello, es necesario que el usuario navegue por otra imagen y, en el caso de que los puntos no estén colocados en el lugar correcto, volver a situar éstos para que concuerden con el mismo objeto o lugar de la otra imagen.

Una vez realizado este proceso, se vuelve a presionar el botón “Fix 2D Coord”. A continuación aparecerá un polígono formado por una línea roja, que será la nueva aproximación creada, y que se acercará bastante al resultado buscado. En los casos de más de dos imágenes, el plano deseado se muestra en todas con un alto índice de fiabilidad. En el caso de querer seguir aumentando la precisión , es posible volver a fijar los puntos de otra fotografía para aproximar aún más el plano, pero estas modificaciones, debido al algoritmo utilizado, son más suaves por regla general que la primera adaptación, donde no teníamos puntos de referencia en dos imágenes 2D.

Así el código queda:

```
//3 - Button "Fix 2D Points" -> If the user wants to fix the points of an image.
void ImageWidget::fixPoints()
{
   int cam_index;
   int tab=layT->currentIndex();
   //Check if we have already the N points at the image
   this->setPolygon();
   if(activeFix)
   {
      cam_index=images.at(tab)->getId();
      //add: if the same camera click at FIX
```

```
        cFix.append(camera->at(cam_index));
        idCamsFix.append(cam_index);
        QVector<QPointF> pAux=images.at(tab)->getVertexExternalCoord();
        for(int i=0;i<pAux.size();i++){
            fixQPoints.append(pAux.at(i));
            pointsCamFix.append(pAux.at(i).x());
            pointsCamFix.append(pAux.at(i).y());
        }
        //enable the button for texture
        arrayBox.at(tab)->setCheckable(1);
        emit fixPolygon(tab);//Slot fixIt(int n) [ImageView]
        calculateFixPlane();
    }
}
```

Como último método para calcular el plano marcado en las imágenes, se realiza la llamada al método calculateFixPlane, implementado por esta clase (desarrollado en el 3.2.1.1)

Una vez fijado el plano deseado, en el cuarto paso podemos decidir si queremos seguir dibujando planos que pertenezcan a la misma figura, por ejemplo, a otra parte del edificio. Para realizar esta acción el usuario debe pulsar en el botón "NewPlane". Como se aprecia en el código, se resetean variables locales y también las de la clase ImageView.

```
//4 - Button "New Plane" -> When the user has finished a plane to start another one.
void ImageWidget::calculateNewPlane()
{
    emit newP(idCamsTexture);//Slot newPlane() [Viewer]
    emit clearImageView();//Slot clearRect() [ImageView]
    nSidesPolygon.clear();
    sumSidesPolygon=0;
    cFix.clear();
    fixQPoints.clear();
    corners.clear();
    allCorners.clear();
    activeFix=false;
    idCamsFix.clear();
    idCamsTexture.clear();
    for(int i=0;i<arrayBox.size();i++){
        if(arrayBox.at(i)->isChecked())
            arrayBox.at(i)->toggle();
        arrayBox.at(i)->setCheckable(0);
    }
    outsidePolygon.clear();
}
```

La quinta opción es que el usuario decida crear un nuevo edificio.

Esta información va directamente a la clase Viewer, que será la encargada de manejarla para que, cuando se guarden los datos, se diferencien entre distintos edificios.

```
//5 - Button "New building" -> To start to compound parts of another building
void ImageWidget::newBuilding(){
   emit building();//Slot newBuilding() [Viewer]
}
```

Finalmente el sexto paso consistiría en guardar la información, tarea del Viewer, que almacena los datos en un fichero que contiene la información de todo el proceso y sobre todo la información más importante, los puntos resultantes en formato de 3D.

```
//6 - Button "Save data" -> To save all the information of the planes on a file.
void ImageWidget::saveDat(){
   emit allData();//Slot saveInfo() [Viewer]
}
```

3.2.1.1. Funciones utilizadas

- *fitPlane( )*

```
void ImageWidget::fitPlane()
```

Al inicio de esta clase se llama al método fitplane para calcular un plano obtenido con la media de todos los puntos del interior del cubo definido en Viewer. El proceso consiste en cargar todos los puntos de la nube y guardar sólo aquellos que cumplan la condición de encontrarse dentro del cubo, tal que:

```
    Track *tr=&(trackCl->getTrack(i));
    if(tr->x()>t.x()-radius & tr->x()<t.x()+radius)
    {
       if(tr->y()>t.y()-radius & tr->y()<t.y()+radius)
       {
          if(tr->z()>t.z()-radius & tr->z()<t.z()+radius)
          {
             array_t.append(tr);
             tracksToDouble.append(tr->x());
             tracksToDouble.append(tr->y());
             tracksToDouble.append(tr->z());
          }
       }
    }
```

Una vez obtenidos dichos puntos, hay que calcular su fitplane, esta vez llamando al método que realiza los cálculos matemáticos, perteneciente a la clase TrackCloudDrawer:

```
   plane=tcd.fitPlane(tracksToDouble);
```

(Función desarrollada en el apartado 3.3)

- *calculateFixPlane( )*

El usuario dentro de la interfaz ImageViewer debe fijar al menos un polígono que represente el mismo objeto sobre un plano, en como mínimo dos imágenes. A partir de

aquí es posible proyectar varios puntos que indiquen el mismo lugar en ambas fotografías (esquinas del plano) y a partir de su proyección desde la cámara, pasando por la imagen, hasta la nube de puntos, poder calcular dónde se unen las rectas y finalmente obtener el plano que representan en la vista de tres dimensiones o Viewer.[2]

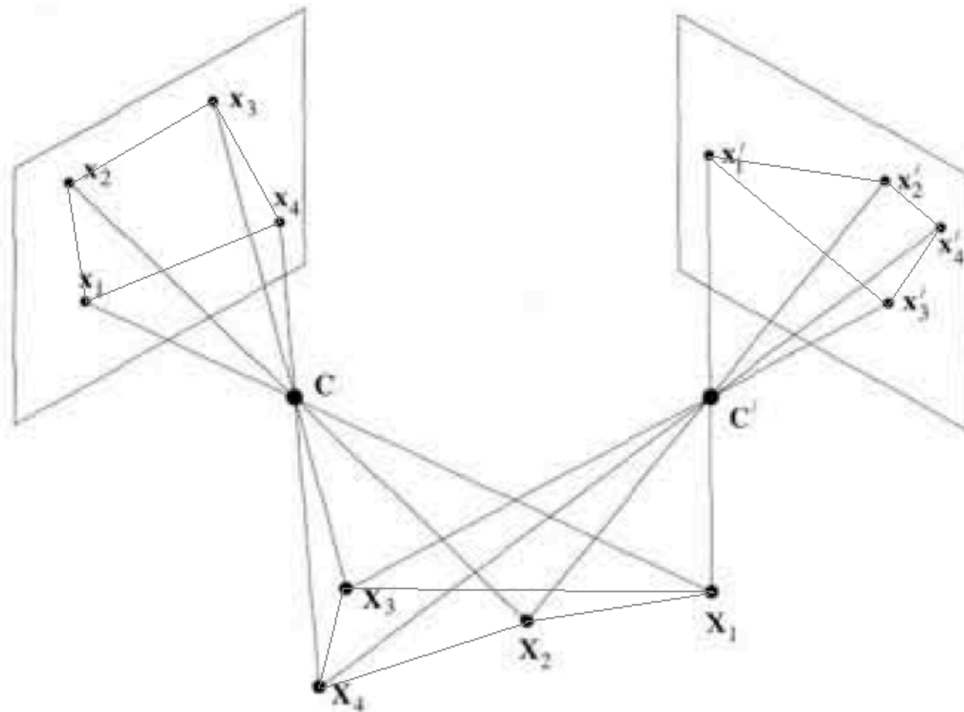


Ilustración 2 Representación de un plano en 3D a partir de dos imágnes en 2D

Para llevar el concepto anterior a código hay que hacer varias consideraciones. La primera es conocer los polígonos que ha definido el usuario, pues no queremos obtener simplemente las coordenadas de un plano, sino que se trata de un plano con unos bordes o fronteras definidos. Una vez conocidos las esquinas de cada polígono vamos a tener esos mismos puntos situados sobre un mismo plano en 3D. El código quedaría como:

```
for(int i=0;i<nSidesPolygon.size();i++)
{
   sum+=nSidesPolygon[i];
}
for(int i=0;i<sum;i++)
{
   for(int j=0;j<cFix.size();j++)
   {
      points2d.append(fixQPoints[j*sum+i]);
   }
   point3d=ccd.DLT(cFix,points2d);
   for(int j=0;j<3;j++)
   {
      allCorners.replace(i*3+j,point3d[j]);
   }
   points2d.clear();
   point3d.clear();
}
```

Para obtener los puntos en 3D de las proyecciones realizadas desde la cámara, se utiliza un método llamado DLT perteneciente a la clase CameraCloudDrawer y que devuelve el punto en el plano de 3D donde se encuentran las proyecciones de los puntos en 2D de las imágenes utilizadas.

Una vez obtenidos todos los puntos, cabe la posibilidad de que no encajen perfectamente dentro de un mismo plano, pues la precisión del usuario a la hora de elegirlos puede no ser exacta.

Esta circunstancia nos lleva a una nube de puntos en 3D más o menos reducida, de la cual podremos calcular su fit plane.

```
//MAKE THE FIT PLANE OF THE N POINTS (CORNERS)
QVector<double> array_d;
for(int i=0;i<allCorners.size();i++){
   array_d.append((double)allCorners.at(i));
}
QVector<double> plane2=tcd.fitPlane(array_d);
```

Una vez que tenemos la ecuación general del plano, el siguiente paso es situar todos los puntos anteriores a los que se le ha calculado el fit plane, y proyectarlos al plano. Esto se hace mediante una función implementada en la clase ImageWidget llamada projection , a la cual se pasa como argumento la ecuación del plano en su forma general, así cómo el punto que queremos aproximar a ese plano.

```
QVector<double> pointAux;
QVector<double> pointP;
for(int i=0;i<sum;i++){
   for(int j=0;j<3;j++){
      pointAux.append(allCorners[i*3+j]);
   }
   pointP=this->projection(plane2,pointAux);
   for(int j=0;j<3;j++){
      allCorners.replace(i*3+j,pointP.at(j));
   }
   pointAux.clear();
   pointP.clear();
}
```

De esta forma todos los puntos quedan almacenados en la variable allCorners, la cual se pasará a la clase Viewer para su representación.

Para la representación de la figura escogida en el resto de las imágenes, se tendrá que proyectar el plano sobre cada una de ellas, lo que también nos permitirá ver el error cometido anteriormente, cuando se calcularon los puntos sin usar una referencia en las imágenes, sólo a partir de los puntos cercanos en la nube de puntos.

```
//Reproject this points over the images to see the error
final_points.clear();
double u,v;
QPointF *p;
for(int i=0;i<id_cams.size();i++)
```

```
    {
      Camera camAux=camera->at(id_cams.at(i));
      cal=camAux.getCalib();
      for(int j=0;j<sum;j++)
      {
        //cal->project(point3d.at(j*3),point3d.at(j*3+1),point3d.at(j*3+2),u,v);
        cal->project(allCorners.at(j*3),allCorners.at(j*3+1),allCorners.at(j*3+2),u,v);
        p=new QPointF(u,v);
        final_points.append(p);
      }
    }
```

Por último en esta clase, se activan los flags determinados y se mandan las señales correspondientes. A la clase ImageView se le envían las coordenadas que debe dibujar en cada una de las imágenes y a la clase Viewer el número de polígonos que hay y si pertenecen o no al plano que se quiere dibujar (los puntos los obtendrá mediante un get del objeto ImageWidget)

```
    //if the last plane has been changed
    if(cFix.size()>2){
      opt1=-3;
      opt2=1;
    }
    //First plane
    else{
      opt1=-2;
      opt2=0;
    }
    emit emitPoint(final_points,opt1);//Slot point(QVector<QPointF*>,int) [ImageView]
    emit changeFitPlane(nSidesPolygon,outsidePolygon,opt2);//Slot
newPointsToFit(QVector<int>,int) [Viewer]
```

- *Projection( )*

Este método se usa para aproximar un punto a un plano. Para resolver este problema se realiza una aproximación lineal, tal que a partir de un punto $P = (x_0, y_0, z_0)$ y un plano $\Pi = Ax + By + Cz + D$, el punto más cercano por aproximación lineal se obtiene como:

$$aux = \frac{-Ax_0 - By_0 - Cz_0 - D}{A^2 + B^2 + C^2}$$, siendo las coordenadas del punto $$\begin{aligned} x &= x_0 + A \cdot aux \\ y &= y_0 + B \cdot aux \\ z &= z_0 + C \cdot aux \end{aligned}$$.

Se implementa de la siguiente manera:

```
QVector<double> ImageWidget::projection(QVector<double> planeCoef, QVector<double> point)
{
  QVector<double> result;

if(planeCoef.at(0)*point.at(0)+planeCoef.at(1)*point.at(1)+planeCoef.at(2)*point.at(2)+planeCoef.at(3)
==0){
    result=point;
  }
```

```
    else{
        double aux=
(-planeCoef.at(0)*point.at(0)-planeCoef.at(1)*point.at(1)-planeCoef.at(2)*point.at(2)-planeCoef.at(3))/
(planeCoef.at(0)*planeCoef.at(0)+planeCoef.at(1)*planeCoef.at(1)+
planeCoef.at(2)*planeCoef.at(2));
        result.append(point.at(0)+planeCoef.at(0)*aux);
        result.append(point.at(1)+planeCoef.at(1)*aux);
        result.append(point.at(2)+planeCoef.at(2)*aux);
    }
    return result;
}
```

3.2.1.2. Signals y slots

- *PointSelected*

Cada vez que el usuario introduce un vértice de un polígono en una imagen, la clase ImageView sobre la que el usuario ha hecho click envía una señal a ImageWidget para que sea esta la encargada de calcular dicho punto, al cual corresponde en la representación en 3D. Una vez obtenido el calculo, le enviará a todas las imágenes que están en la clase ImageView su punto correspondiente.

En primer lugar, para realizar las operaciones anteriores, se obtiene la cámara a la que pertenece el punto que se ha pasado como argumento. Una vez que tenemos la cámara, se accede a su calibración obteniendo un objeto de la clase CalibrationPtr que permitirá, a través del método compute_3d_point ,obtener 2 puntos que formen el vector que se debe cruzar con el fit plane, plano obtenido representativo de la nube de puntos, para así tener el punto correspondiente en 3D.

Los cálculos a realizar son los correspondientes a un vector que corta un plano, obteniendo un punto. En código queda tal que:

```
    c=camera->at(id);
    cal=c.getCalib();
    double *p1=new double[3];
    double *p2=new double[3];
    cal->compute_3d_point((double)point->x(),(double)point->y(),5,p1);
    cal->compute_3d_point((double)point->x(),(double)point->y(),10,p2);
    double *p12=new double[3];
    p12[0]=p2[0]-p1[0];
    p12[1]=p2[1]-p1[1];
    p12[2]=p2[2]-p1[2];
    double x=((plane.at(1)*p12[1]+plane.at(2)*p12[2])*p1[0]/p12[0]-plane.at(1)*p1[1]-
plane.at(2)*p1[2]-plane.at(3))/
        (plane.at(0)+(plane.at(1)*p12[1]+plane.at(2)*p12[2])/p12[0]);
    double y=((x-p1[0])*p12[1]/p12[0])+p1[1];
    double z=((x-p1[0])*p12[2]/p12[0])+p1[2];
```

A continuación se añade dicho punto a la variable que guarda las esquinas (corner) que delimitan los planos, teniendo en cuenta que puede ser un punto nuevo, o que el usuario esté moviendo un punto ya colocado previamente.

```
if(corner==-1)
{
   corners.append(x);
   corners.append(y);
   corners.append(z);
}
else
{
   corners.replace(corner*3,x);
   corners.replace(corner*3+1,y);
   corners.replace(corner*3+2,z);
}
```

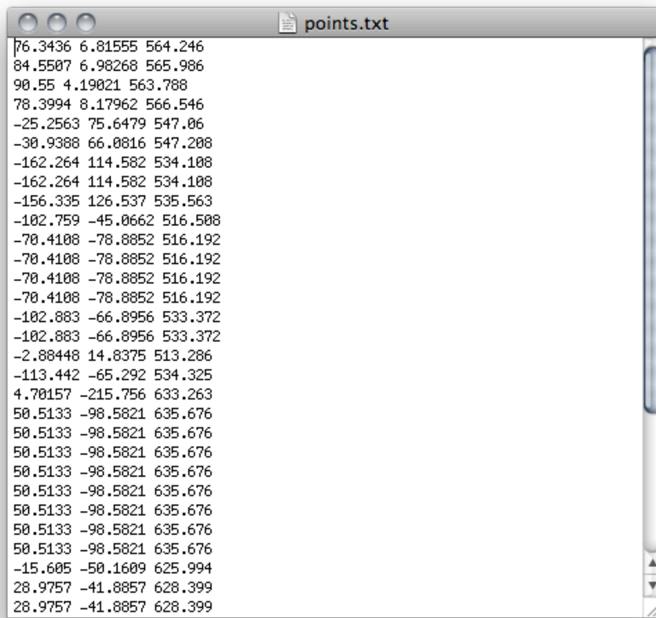
Una vez se tiene el punto en 3D guardado, el siguiente paso será calcular, para cada una de las imágenes que teníamos, donde se encuentra ese punto en ellas.

Para ello se vuelve a usar la clase CalibrationPtr y el método project, el cual si le pasamos el punto en 3D, devuelve su proyección en la imagen correspondiente. Una vez realizado este proceso para todas las cámaras, se envía un array con todos los puntos a todas las clases ImageView, las cuales contienen las imágenes, y dibujarán sobre ella el punto que corresponda.

```
//Calculate all points in 2D to send them to all ImageView objects
double u,v;
final_points.clear();
foreach (Feature f, camSelectedIndex)
{
   int j=f.getCamIndex();
   c=camera->at(j);
   cal=c.getCalib();
   cal->project(x,y,z,u,v);
   QPointF *p=new QPointF(u,v);
   final_points.append(p);
}
emit emitPoint(final_points,corner);
```

- *SavePoint*

Esta señal se recibe cuando el usuario desea guardar los datos de los planos que ha dibujado en la interfaz. Los datos que se guardan son las coordenadas de los puntos en 3D obtenidos. El fichero de salida utilizado va a tener el nombre points.txt y se va a guardar por en el directo local del usuario. Este archivo tendrá un estilo como el de la imagen:



```
points.txt
76.3436 6.81555 564.246
84.5507 6.98268 565.986
90.55 4.19021 563.788
78.3994 8.17962 566.546
-25.2563 75.6479 547.06
-30.9388 66.0816 547.208
-162.264 114.582 534.108
-162.264 114.582 534.108
-156.335 126.537 535.563
-102.759 -45.0662 516.508
-70.4108 -78.8852 516.192
-70.4108 -78.8852 516.192
-70.4108 -78.8852 516.192
-70.4108 -78.8852 516.192
-102.883 -66.8956 533.372
-102.883 -66.8956 533.372
-2.88448 14.8375 513.286
-113.442 -65.292 534.325
4.70157 -215.756 633.263
50.5133 -98.5821 635.676
50.5133 -98.5821 635.676
50.5133 -98.5821 635.676
50.5133 -98.5821 635.676
50.5133 -98.5821 635.676
50.5133 -98.5821 635.676
50.5133 -98.5821 635.676
50.5133 -98.5821 635.676
-15.605 -50.1609 625.994
28.9757 -41.8857 628.399
28.9757 -41.8857 628.399
```

Los puntos se guardan de forma consecutiva según las variables x, y y z dejando un espacio en blanco entre cada coordenada y salto de línea al siguiente punto.

Según el código quedaría como:

```
QFile file;
QDir::setCurrent(QDir::homePath());
file.setFileName("points.txt");
file.open(QIODevice::Append);
QString *g;
QString msg2;
msg2.clear();
msg2.append(g->number(t.x()));
msg2.append(" ");
msg2.append(g->number(t.y()));
msg2.append(" ");
msg2.append(g->number(t.z()));
msg2.append(" ");
qDebug()<<"Save point in file";
QChar *data=msg2.data();
while(!data->isNull()){
   file.write((char*)data);
   data++;
}
char msg1[2]="\n";
file.write(msg1,strlen(msg1));
file.close();
```

3.2.2. ImageView

Clase que hace referencia a cada imagen obtenida en la interfaz. Se considera como una clase aparte para tratar de forma más sencilla las distintas operaciones y cambios que el usuario puede realizar sobre ella. Por tanto, una vez abierta la segunda interfaz, y después de que el usuario haya elegido un punto de la nube, crea tantas instancias de la clase ImageView como imágenes contengan ese punto. Como parámetros de entrada para la creación de esta clase necesitaremos lógicamente la imagen de tipo Qimage, que se ha de mostrar, el punto elegido por el usuario con el cambio ya dado a 2D, y el índice y el identificador de la imagen. Por tanto, habrá que escalar cada imagen para que no sobrepase las dimensiones de la ventana, aunque exista la opción de modificar el zoom, y sobre esa imagen mostrar un cuadrado azul que indique el punto seleccionado de la nube.

Código:

```
ImageView::ImageView(QSharedPointer<QImage> image,QSharedPointer<QPoint> point,int index,int id,
           QGraphicsView *parent): QGraphicsView(parent),id(id),index(index)
{
  scene = new QGraphicsScene();
  setScene(scene);
  moveTheRect=-1;
  moveThePixmap=false;
  newPoly=false;
  nSidesPol=0;
  contVertex=0;
  sumSidesPolygon=0;
  totalPol=0;
  //set up correct options
  pixmap = scene->addPixmap(QPixmap::fromImage(*image));
  pixmap->acceptDrops();
  pixmap->setScale(0.15);
  QPointF aux=pixmap->mapToScene(*point);
  p=&aux;
  rectangle=new QRectF(QPointF(0,0),QSizeF(8,8));
  alreadyFix=0;
  rectangle2=new QRectF(QPointF(0,0),QSizeF(8,8));
  rect2=scene->addRect(*rectangle2);
  rect2->setPos(p->rx()-rectangle2->width()/2,p->ry()-rectangle2->width()/2);
  rect2->setPen(QPen(QColor(0,0,255)));
}
```

Una vez que tenemos las imágenes mostradas en la interfaz, evaluamos los distintos eventos que podrá suceder.

Algunos deberán seguir un orden pero otros podrán ser invocados por el usuario en cualquier momento haciendo uso del ratón.

El usuario puede realizar la acción de hacer click y mover el ratón para desplazar la imagen, también le se permite hacer zoom usando la rueda del ratón y por último también podrá colocar los vértices de un polígono (si se ha especificado correctamente) así cómo mover dichos vértices, una vez colocados en la imagen.

Código:

```
//---------------------------- ----------
// Mouse events
//---------------------------------------
// MousePressEvent -> Click on the image to move one point.
// MousePressEvent + Shift -> set a poin on the image.
void ImageView::mousePressEvent(QMouseEvent *e)
{
   if(newPoly){
      if(contVertex==sumSidesPolygon)
      {
         for(int i=0;i<rectangles.size();i++)
         {
            if(rectangles.at(i)->isUnderMouse())
            {
               qDebug()<<"Rectangle"<<i;
               moveTheRect=i;
            }
         }
      }
      else{
         if(e->modifiers() == Qt::ShiftModifier)
         {
            QPointF mts=this->mapToScene(e->pos());
            QPointF mts2=pixmap->mapFromScene(mts);
            p=&mts2;
   //         qDebug()<<"Point pressed ("<<e->x()<<","<<e->y()<<")";
   //         qDebug()<<"Point scene("<<mts.x()<<","<<mts.y()<<")";
   //         qDebug()<<"Point item ("<<p->x()<<","<<p->y()<<")";
            emit sendPoint2(p,id,moveTheRect);
         }
      }
   }
   if(moveTheRect<0){
      prevX=e->pos().x();
      prevY=e->pos().y();
      moveThePixmap=true;
   }
}
// MouseReleaseEvent -> If the use has clicked inside a square when
// he/she release the button the info is sended to imageWidget.
void ImageView::mouseReleaseEvent(QMouseEvent *e)
{
   if(moveTheRect>=0)
   {
      QPointF mts=this->mapToScene(e->pos());
      QPointF mts2=pixmap->mapFromScene(mts);
      p=&mts2;
      emit sendPoint2(p,id,moveTheRect);
      moveTheRect=-1;
   }
   scene->update();
}
// WheelEvent -> to make zoom (in/out).
void ImageView::wheelEvent(QWheelEvent *e)
{
   if (e->orientation() == Qt::Vertical){
      if(e->delta()>0){
```

```
        this->scale(1.1,1.1);
       }
       else{
        this->scale(0.9,0.9);
       }
    }
    e->accept();
}
// MouseMoveEvent -> the movement of the mouse is detected only when
// the user has pressed on a square first. This method is updating the
// coordinates of the squares.
void ImageView::mouseMoveEvent(QMouseEvent *e)
{
    if(moveTheRect>=0){
       QPointF mts=this->mapToScene(e->pos());
       QPointF mts2=pixmap->mapFromScene(mts);
       p=&mts2;
       emit sendPoint2(p,id,moveTheRect);
    }
    else if(moveThePixmap){
       qreal rx=prevX-e->pos().x();
       qreal ry=prevY-e->pos().y();
       horizontalScrollBar()->setValue(horizontalScrollBar()->value()+(int)rx);
       verticalScrollBar()->setValue(verticalScrollBar()->value()+(int)ry);
       prevX=e->pos().x();
       prevY=e->pos().y();
    }
}
```

En el teclado sólo se ha añadido una shortcut para aligerar la operación de guardado (save) , así pues tecleando S se salvará la información.

```
//-------------------------------------
// Keyboard events
//-------------------------------------
void ImageView::keyPressEvent(QKeyEvent *e)
{
    if((e->key()==Qt::Key_S) && (e->modifiers()==Qt::NoButton)){
       emit saveP();
    }
}
```

El siguiente paso es analizar las señales que llegan desde la clase ImageWidget. La comunicación entre ambas clases es necesaria para identificar la operaciones que el usuario ha ido realizando y habilitar o deshabilitar dichas operaciones.

3.2.2.1. Signals y slots

- *nPolygon*

Cuando el usuario elige que quiere añadir un nuevo polígono a la imagen, la clase ImageView recibe una señal indicando el número de lados que tiene dicho polígono.

Se añadirán los lados al vector correspondiente, para luego recibir los puntos (esquinas) pertenecientes a dicho polígono.

```
void ImageView::nPolygon(int nSides){
   nSidesPol=nSides;
   arrayNSidesPol.append(nSidesPol);
   sumSidesPolygon+=nSides;
   qDebug()<<"[ImageView] new polygon with"<<nSides<<"sides"<<" Total sides-
>"<<sumSidesPolygon;
   newPoly=true;
}
```

- *point*

Esta señal se recibe cada vez que el usuario ha hecho click sobre un punto para añadir una esquina del polígono, sin embargo no es la señal recibida directamente del click, sino la que viene después de ser tratada por la clase ImageWidget y Viewer , la que se interpretará.

La señal recibida puede tener distintos significados, bien que hay que dibujar un punto nuevo, o bien que hay que mover un punto que ya existía o bien que hay que dibujar un polígono rojo (en este caso el usuario ha pulsado “Fix Plane”).

Estas opciones se eligen según la variable llamada corner, que discriminará entre los distintos casos, nombrados a continuación:

- *Corner == -1*

  Si corner toma el valor -1 quiere decir que se trata de un nuevo punto a añadir. En este caso, lo primero que se hará es obtener el punto según el número de índice de la clase ImageView. Se transforman las coordenadas para hacerlas coincidir según el tamaño que tenga la imagen y una vez hecho esto y actualizado las variables donde se almacena el punto, se coloca un cuadrado a su alrededor para hacer más fácil la identificación por parte del usuario, y la posible modificación del lugar del punto.

```
if(corner==-1)
   {
     QPointF *p2=array_p.at(index);
     QPointF aux=pixmap->mapToScene(*p2);
     p=&aux;
     vertexExternalCoord.append(*p2);
     vertex.append(aux);
     contVertex++;
     vertex2.append(*array_p.at(index));
     rect = scene->addRect(*rectangle);
     rect->setPos(p->rx()-rectangle->width()/2,p->ry()-rectangle->width()/2);
     rect->acceptDrops();
     rectangles.append(rect);
```

Conforme se van añadiendo puntos, hay que ir uniéndolos mediante líneas, hasta llegar a completar el polígono. Ambos casos se contemplan en el siguiente código:

```
if(contVertex==sumSidesPolygon)
{
   //I have a polygon
   polygon=new QPolygonF(vertex);
   poly=scene->addPolygon(*polygon);
   arrayPolygon.append(polygon);
   arrayPoly.append(poly);
   //Find the topLeft & bottomRight
   QPointF tLeft=vertex2.at(0);
   QPointF bRight=vertex2.at(0);
   for(int i=0;i<vertex2.size();i++)
   {
      if(tLeft.x()>vertex2.at(i).x() && tLeft.y()>vertex2.at(i).y())
      {
         tLeft=vertex2.at(i);
      }
      if(bRight.x()<vertex2.at(i).x() && bRight.y()<vertex2.at(i).y())
      {
         bRight=vertex2.at(i);
      }
   }
   QRect *aux=new QRect(tLeft.toPoint(),bRight.toPoint());
   QPixmap pm=pixmap->pixmap();
   pm=pm.copy(*aux);
   QImage aux_im=pm.toImage();
   imRect=new QImage(aux_im);
   for(int i=0;i<arrayLine.size();i++)
      scene->removeItem(arrayLine[i]);
   arrayLine.clear();
   vertex.clear();
}
else if(vertex.size()>1){
   //I haven't got all the points yet, so join the points that I have at the moment
   int n=vertex.size();
   QLineF *line=new QLineF(vertex[n-2],vertex[n-1]);
   l=scene->addLine(*line);
   arrayLine.append(l);
}
```

En el primer caso, en el cual el polígono está establecido por completo, se añade una clase polígono identificando qué punto se encuentra en la parte superior a la izquierda y abajo a la derecha.

Por otra parte, en el caso de que no se haya completado el polígono, se irán añadiendo líneas entre un punto y otro para que el usuario pueda ir viendo como se forma lo que finalmente será el polígono.

- *Corner >= 0*

Si la variable corner toma un valor mayor que 0 quiere decir que el usuario está moviendo una de las esquinas del polígono, lo que hará que cambie ese punto

en todas las imágenes, excepto en aquellas donde ya se haya fijado el plano a la imagen y el usuario haya indicado explícitamente su posición.

El movimiento simplemente lo simulamos dibujando y borrando el punto y el cuadrado situado alrededor del punto que se está moviendo. Como puede que haya varios polígonos en un misma figura, se realizan las comprobaciones oportunas en el código para comprobar el punto de qué polígono exactamente se está moviendo.

```
else if (corner>=0)
{
   if(!alreadyFix){
      QPointF *p2=array_p.at(index);
      QPointF aux=pixmap->mapToScene(*p2);
      p=&aux;
      rectangles.at(corner)->setPos(p->rx()-rectangle->width()/2,p->ry()-rectangle->width()/2);
      vertexExternalCoord.replace(corner,*p2);
      int cont=0;
      for(int i=0;i<arrayNSidesPol.size();i++){
         corner-=arrayNSidesPol[i];
         if(corner<0){
            corner+=arrayNSidesPol[i];
            break;
         }
         else{
            cont++;
         }
      }
      arrayPolygon[cont]->replace(corner,aux);
      arrayPoly[cont]->setPolygon(*arrayPolygon[cont]);
   }
}
```

- Corner < -1

Si la variable corner toma un valor menor a -1 quiere decir que se han realizado los calculos necesarios para dibujar un nuevo polígono, con menos error, este nuevo polígono se dibuja en rojo y viene dado por la proyección del plano calculado a partir de al menos dos imágenes proyectadas en 3D. Puede ser que ya esté dibujado, en ese caso habrá que borrarlo para volver a dibujarlo, de ahí vendrá la primera comprobación del valor de corner igual a -3.

```
else if (corner<-1){
   if(corner==-3)
   {
//         for(int i=0;i<polItem.size();i++)
//            scene->removeItem(polItem[i]);
//         polItem.clear();
      for(int i=polItem.size();i>totalPol;i--){
         scene->removeItem(polItem[i-1]);
         polItem.pop_back();
      }
   }
   QPolygonF *pol;
   QVector<QPointF> cornersPol;
   int aux=0;
   for(int i=0;i<arrayNSidesPol.size();i++)
```

```
        {
          for(int j=0;j<arrayNSidesPol[i];j++)
          {
            QPointF *p2;
            p2=array_p[index*sumSidesPolygon+aux+j];
            QPointF aux=pixmap->mapToScene(*p2);
            cornersPol.append(aux);
          }
          pol=new QPolygonF(cornersPol);
          polItem.append(scene->addPolygon(*pol));
          QPen *q=new QPen(QBrush(Qt::SolidPattern),0.4);
          q->setColor(QColor(255,0,0));
          polItem[i+totalPol]->setPen(*q);
          aux+=arrayNSidesPol[i];
          cornersPol.clear();
        }
    }
    this->repaint();
}
```

- *fixIt*

Recibe la señal enviada por ImageWidget cuando el usuario pulsa “Fix 2D Coord” e indica que el usuario ha dado por bueno el polígono elegido, por lo tanto se marcará la línea más gruesa y se impide que se pueda seguir modificando.

```
void ImageView::fixIt(int n){
    if(n==index){
        for(int i=0;i<arrayPoly.size();i++){
            arrayPoly[i]->setPen(QPen(QBrush(Qt::SolidPattern),0.2));
        }
        for(int i=0;i<rectangles.size();i++){
            rectangles[i]->hide();
        }
        alreadyFix=1;
        contVertex=0;
    }
}
```

- *clearRect*

Una vez que sede formar el polígono y se elige nuevo plano, hay que resetear toda la información de la clase, excepto el polígono final, que estará marcado en rojo.

```
void ImageView::clearRect()
{
    moveTheRect=-1;
    vertex.clear();
    vertex2.clear();
    vertexExternalCoord.clear();
    for(int i=0;i<rectangles.size();i++){
        scene->removeItem(rectangles.at(i));
    }
    rectangles.clear();
    alreadyFix=0;
    newPoly=false;
```

```
    nSidesPol=0;
    arrayNSidesPol.clear();
    for(int i=0;i<arrayPoly.size();i++){
        scene->removeItem(arrayPoly[i]);
    }
    arrayPoly.clear();
    arrayPolygon.clear();
    sumSidesPolygon=0;
    totalPol=polItem.size();
    contVertex=0;
}
```

3.2.3. TrackCloudDrawer

Esta clase auxiliar de TrackCloud se utiliza para inicializar la nube de puntos, es decir, para dibujarla. Para la inicialización se podrán utilizar cómo argumentos una nube puntos TrackCloud o información a través de un fichero de extensión ply, como TriangleSoup:

```
void TrackCloudDrawer::init(TrackCloud& t)
```

```
void TrackCloudDrawer::initFromPly(TriangleSoup &ts)
```

Al obtener en esta clase la nube de puntos, se ejecuta una función importante para nuestro proyecto, la llamada fitPlane() , cuya misión es proporcionar un plano que sea media de todos los puntos definidos en cierta área.

Analizamos la función fitPlane():

```
QVector<double> TrackCloudDrawer::fitPlane(QVector<double> array_d)
```

Para su cálculo se usa el método de “Principal Components Analysis”, PCA[3].

Dicho método es una forma de identificar patrones en los datos y expresar esos patrones de forma que se realcen sus similitudes y diferencias. Aquí nos centraremos en encontrar esos patrones para usarlos de forma que podamos reducir las dimensiones de los puntos, de tres componentes a dos componentes, perdiendo la mínima información posible y obteniendo el plano en 2D deseado.

Se sigue el método PCA según los siguientes pasos:

1. **Datos iniciales**
   Son los puntos que se le pasan a la función por los argumentos.

```
QVector<double> array_d
```

2. **Obtener la media**
   Hay que obtener la media de cada dimensión.

```
    //mean
```

```
Vector3d mean=Vector3d::Zero(3);
for(int i=0;i<array_d.size()/3;i++)
{
   mean(0)+=array_d.at(i*3);
   mean(1)+=array_d.at(i*3+1);
   mean(2)+=array_d.at(i*3+2);
   if(i==array_d.size()/3-1)
   {
      mean=mean/(array_d.size()/3);
   }
}
```

Una vez obtenida hay que normalizar la media para que sea igual a 0.

```
//Normalized the points with mean=0
MatrixXd m_points(array_d.size()/3,3);
m_points.setZero();
for(int i=0;i<array_d.size()/3;i++)
{
   m_points(i,0)=array_d.at(i*3)-mean(0);
   m_points(i,1)=array_d.at(i*3+1)-mean(1);
   m_points(i,2)=array_d.at(i*3+2)-mean(2);
}
```

3. Calcular la matriz de covarianzas

Este cálculo para tres dimensiones es necesario calcular cov(x,y), cov(x,z) y cov(y,z). Quedando la matriz de covarianzas tal que:

$$C = \begin{pmatrix} \operatorname{cov}(x,x) & \operatorname{cov}(x,y) & \operatorname{cov}(x,z) \\ \operatorname{cov}(y,x) & \operatorname{cov}(y,y) & \operatorname{cov}(y,z) \\ \operatorname{cov}(z,x) & \operatorname{cov}(z,y) & \operatorname{cov}(z,z) \end{pmatrix}$$

La matriz de covarianzas es simétrica siendo cov(a,b)=cov((b,a).

En código:

```
//Covariance
Matrix3d cov=Matrix3d::Zero();
for(int i=0;i<cov.rows();i++)
{
  for(int j=0;j<cov.cols();j++)
  {
     cov(i,j)=m_points.col(i).transpose()*m_points.col(j);
  }
}
```

4. Calcular los vectores y valores propios de la matriz de covarianzas

Sabiendo que la matriz de covarianzas es cuadrada se pueden calcular los vectores propios y valores propios para esta matriz. Estos valores son importantes porque contienen información útil de nuestros datos.

```
//Calculate eigenvector and eigenvalue
SelfAdjointEigenSolver<Matrix3d> eigensolver(cov);
```

```
Vector3d e_values=eigensolver.eigenvalues();
Matrix3d e_vectors=eigensolver.eigenvectors();
```

5. Elegir los componentes correctos para crear un vector de las características deseadas

En este paso se produce la comprensión de datos y se reducirá una dimensión, partir de los vectores propios y de los valores propios de la anterior sección. Si se elige el vector propio con el mayor valor propio se obtiene el principal componente del conjunto de datos. Por regla general, una vez que se calculan los vectores propios a partir de la matriz de covarianzas, el próximo paso es ordenarlos según sus valores propios de mayor a menor. Esto nos ofrece los componentes por orden de importancia y se puede, por tanto, eliminar los componentes menos significantes. Haciendo ésto se pierde información pero si los valores propios son pequeños la pérdida no será muy relevante. Así el conjunto de datos final tendrá menos dimensiones que el inicial. De forma teórica, si inicialmente se tenían n dimensiones en el conjunto de datos y se calculan n vectores propios y valores propios y entre ellos se eligen p vectores propios, entonces el conjunto de datos final que se obtiene será de p dimensiones.

6. Obtener el plano

Se guardan los datos de los vectores propios en una variable denominada normal, que será la que represente la normal del plano. A partir de ese valor y de un punto que debe estar en el plano, podemos obtener la ecuación general tal que:

```
 QVector<double> normal;
for(int i=0;i<3;i++)
{
   normal.append(e_vectors(i,0));
}
//Calculate plane ecuation.
//With normal vector and a point
//how e_vectors are perpendicular between them, then the two e_vectors has to take part of the plane.
//Or the same thing...I can take one of them as a point inside the plane
double d=-(normal[0]*mean(0)+normal[1]*mean(1)+normal[2]*mean(2));
qDebug()<<"PLANE 1:"<<normal[0]<<"x+"<<normal[1]<<"y+"<<normal[2]<<"z+"<<d<<"=0";
```

Otra forma de calcular el plano podría haber sido a partir de los vectores propios directamente:

```
//Other calculus for the plane (to check the plane)
double A=e_vectors(1,1)*e_vectors(2,2)-e_vectors(2,1)*e_vectors(1,2);
double B=e_vectors(2,1)*e_vectors(0,2)-e_vectors(0,1)*e_vectors(2,2);
double C=e_vectors(0,1)*e_vectors(1,2)-e_vectors(1,1)*e_vectors(0,2);
double D=((e_vectors(2,1)*e_vectors(1,2)-e_vectors(1,1)*e_vectors(2,2))*mean(0)+
        (e_vectors(0,1)*e_vectors(2,2)-e_vectors(2,1)*e_vectors(0,2))*mean(1)+
        (e_vectors(1,1)*e_vectors(0,2)-e_vectors(0,1)*e_vectors(1,2))*mean(2));
qDebug()<<"PLANE 2:"<<A<<"x+"<<B<<"y+"<<C<<"z+"<<D<<"=0";
```

Con esto finaliza el método PCA, donde se ha perdido cierta información pero he conseguido un plano inicial para poder usarlo en las primeras proyecciones de los puntos que el usuario elija en las imágenes.

3.2.4. Viewer

En esta clase las funciones adicionales que se añadieron son:

- Creación de una interfaz auxiliar donde se permite establecer el centro de rotación manualmente y elegir el tamaño del cubo que rodea al punto en 3D elegido, definida como clase *Parameters.*
- Iluminar las cámaras que contienen en sus imágenes el punto seleccionado.
- Permitir un movimiento ágil del punto que se desea seleccionar.
- Dibujar los planos de los edificios que el usuario elija para reconstruir.

Se comienza por el constructor de la clase Viewer donde ya en dicho constructor se crea la interfaz auxiliar que es un elemento de la clase Parameters. La ventana se mostrará en la parte inferior izquierda de la pantalla y se encargará de manejar 3 señales: la primera se envía al hacer click sobre el botón “set center”, la segunda es la de actualizar las variables del centro si el usuario hace doble click sobre un punto de la imagen y la tercera señal consiste en enviar una señal si el usuario cambia el tamaño del cubo en la barra configurada para ello.

```
//Auxiliary window of parameters
paramWindow =new parameters();
auxLayout = new QHBoxLayout;
auxLayout->addWidget(paramWindow);
int desktopWidth=QApplication::desktop()->width();
int desktopHeight=QApplication::desktop()->height();
paramWindow->resize(desktopWidth/2,desktopHeight/4);
paramWindow->move(0,desktopHeight/4*3);
paramWindow->setWindowTitle("Parameters");
paramWindow->show();
connect(paramWindow,SIGNAL(sendCubeValue(int)),this,SLOT(changeCubeValue(int)));
connect(this,SIGNAL(newCenter(int,int,int)),paramWindow,SLOT(changeCenter(int,int,int)));
connect(paramWindow,SIGNAL(setCenter(int,int,int)),this,SLOT(changeCenter(int,int,int)));
```

- **Método draw ()**

A la hora de representar en la librería OpenGL los determinados elementos que se muestran en la interfaz, como son la nube de puntos o la nube de cámaras, se comprueba si dichas variables tienen un valor asignado y en caso de que así sea, se llama a los métodos draw de las distintas clases, que son los encargados de mostrar la información en forma gráfica sobre la interfaz.

```
glPointSize(1.f);
glColor3f(1,0,0);
if(trackCloud && showCloud){
   ccd.draw();
}

glColor3f(0,1,0);

if(trackCloud && showCloud){
   tcd.draw();
```

```
}
glColor3f(0,0,1);
if(trackCloud && showCloud && showSoup){
   tsd.draw();
}
```

La representación del punto elegido con el ratón se simula utilizando tres planos rellenos cruzándose entre sí, uno por cada eje, tal que:

```
glColor3f(0,1,1);

float *c = currentPoint.v_;

glBegin(GL_QUADS);
float d = 0.4;

glVertex3f(c[0]-d,c[1]+d,c[2]);
glVertex3f(c[0]-d,c[1]-d,c[2]);
glVertex3f(c[0]+d,c[1]-d,c[2]);
glVertex3f(c[0]+d,c[1]+d,c[2]);

glVertex3f(c[0],c[1]+d,c[2]-d);
glVertex3f(c[0],c[1]-d,c[2]-d);
glVertex3f(c[0],c[1]-d,c[2]+d);
glVertex3f(c[0],c[1]+d,c[2]+d);

glVertex3f(c[0]-d,c[1],c[2]+d);
glVertex3f(c[0]-d,c[1],c[2]-d);
glVertex3f(c[0]+d,c[1],c[2]-d);
glVertex3f(c[0]+d,c[1],c[2]+d);

glEnd();
```

Alrededor de dicho punto se traza un cubo, del que se dibujan sólo las aristas:

```
//Draw the cube
if(qb){
glBegin(GL_LINES);
glVertex3d(c[0]-rad,c[1]+rad,c[2]-rad);
glVertex3d(c[0]-rad,c[1]-rad,c[2]-rad);
glVertex3d(c[0]+rad,c[1]-rad,c[2]-rad);
glVertex3d(c[0]+rad,c[1]+rad,c[2]-rad);
glVertex3d(c[0]-rad,c[1]+rad,c[2]+rad);
glVertex3d(c[0]-rad,c[1]-rad,c[2]+rad);
glVertex3d(c[0]+rad,c[1]-rad,c[2]+rad);
glVertex3d(c[0]+rad,c[1]+rad,c[2]+rad);
glVertex3d(c[0]-rad,c[1]+rad,c[2]-rad);
glVertex3d(c[0]-rad,c[1]+rad,c[2]+rad);
glVertex3d(c[0]-rad,c[1]-rad,c[2]-rad);
glVertex3d(c[0]-rad,c[1]-rad,c[2]+rad);
glVertex3d(c[0]+rad,c[1]+rad,c[2]-rad);
glVertex3d(c[0]+rad,c[1]+rad,c[2]+rad);
glVertex3d(c[0]+rad,c[1]-rad,c[2]-rad);
glVertex3d(c[0]+rad,c[1]-rad,c[2]+rad);
glVertex3d(c[0]-rad,c[1]-rad,c[2]+rad);
glVertex3d(c[0]+rad,c[1]-rad,c[2]+rad);
glVertex3d(c[0]-rad,c[1]-rad,c[2]-rad);
```

```
    glVertex3d(c[0]+rad,c[1]-rad,c[2]-rad);
    glVertex3d(c[0]-rad,c[1]+rad,c[2]+rad);
    glVertex3d(c[0]+rad,c[1]+rad,c[2]+rad);
    glVertex3d(c[0]+rad,c[1]+rad,c[2]-rad);
    glVertex3d(c[0]-rad,c[1]+rad,c[2]-rad);
    glEnd();
    }
```

Después, opcionalmente, se pueden dibujar los planos que el usuario ha ido seleccionando (en el caso de que lo haya hecho):

```
    for(int i=0;i<arrayPlane.size();i++){
        arrayPlane.at(i)->draw();
    }
```

Y también si está activo y hay seleccionado algún punto, las cámaras que incluyen a ese punto se iluminan pintándose de azul claro:

```
    if(colorCamera){
        QList<Feature> camSelectedIndex = ccd.getSelected();
        foreach (Feature f, camSelectedIndex)
        {
            glColor3f(0,1,1);
            ccd.drawCamera((*cameraCloud)[f.getCamIndex()],7.0,false);
        }
    }
```

Con esto concluye el método draw y todos los elementos que son representables por la clase Viewer.

- **Método *postSelection ()***

Este método es ejecutado cada vez que se produce alguna modificación en la interfaz, comprobándose siempre si hay algún punto seleccionado mediante el método selectedName. En caso de que haya algún punto se calculan sus componentes x, y, z para dibujarlo. Además, hay un flag que indica si el usuario quiere mostrar las imágenes que contienen a ese punto. Este flag se llama display, y en caso de estar activo, se creará la clase ImageWidget , que mostrará una nueva ventana para la correcta visualización. Después de crear la interfaz se añaden las señales que conectarán la clase ImageViewer con la clase actual, Viewer.

```
void Viewer::postSelection(const QPoint &point)
{
    // Find the selectedPoint coordinates, using camera()->pointUnderPixel().
    bool found;

    qglviewer::Vec selectedPoint = camera()->pointUnderPixel(point, found);
    secondaryPoint=selectedPoint;

    if(selectedName() == -1){
    }
    else
    {
        Track t = trackCloud->value(selectedName());
        currentPoint.setValue(t.x(),t.y(),t.z());
```

```
        ccd.setSelected(t);
        updateGL();
        if(display)
        {
           window = new ImageWidget(cameraCloud,trackCloud,t,ccd,tcd,rad);
           int desktopWidth=QApplication::desktop()->width();
           int desktopHeight=QApplication::desktop()->height();
           window->setWindowTitle("Images");
           window->resize(desktopWidth/2,desktopHeight);
           window->move(desktopWidth/2,0);
           window->show();
           display=false;

connect(window,SIGNAL(changeFitPlane(QVector<int>,QVector<bool>,int)),this,SLOT(newPointsTo
Fit(QVector<int>,QVector<bool>,int)));
           connect(window,SIGNAL(newP(QVector<int>)),this,SLOT(newPlane(QVector<int>)));
           connect(window,SIGNAL(building()),this,SLOT(newBuilding()));
           connect(window,SIGNAL(allData()),this,SLOT(saveInfo()));
        }
    }
}
```

A continuación se analizan la forma de interactuar del usuario con la interfaz a través del teclado y del ratón.

Se pueden llevar a cabo las acciones siguientes:

- ***Click de ratón***

  Si el usuario hace un click de ratón, el evento se pasa al manejador de la librería OpenGL, y a su vez a la clase QGLViewer, excepto si se tiene presionado Shift o Alt. En el caso de Alt, el punto seleccionado se marca de azul, y en el de Shift, la tecla presionada entonces, además de seleccionarse el punto, se activa el flag que lanza la interfaz para mostrar las imágenes que contienen ese punto.

```
void Viewer::mousePressEvent(QMouseEvent *e)
{
   if(e->modifiers() == Qt::ShiftModifier){

      //cameraCloud->loadSelected();
      display=true;
      select(e);
      updateGL();
   }
   else if(e->modifiers() == Qt::AltModifier && e->modifiers() != Qt::ShiftModifier){
      select(e);
   }
   else{
      QGLViewer::mousePressEvent(e);
   }
}
```

- ***Doble click de ratón***

  Si se hace doble click izquierdo de ratón se selecciona el punto sobre el que se ha hecho el click y el centro de rotación de la interfaz cambia, enviando a su vez

una señal a la interfaz auxiliar que indica que ha habido un cambio, para su posterior actualización.

```
void Viewer::mouseDoubleClickEvent(QMouseEvent *e)
{
   if(e->button() == Qt::LeftButton)
   {
      select(e);
      setSceneCenter(currentPoint);
      emit newCenter(currentPoint.x,currentPoint.y,currentPoint.z);
      updateGL();
   }
}
```

- ***Movimiento del ratón***

El movimiento del ratón es manejado por la clase QGLViewer, salvo en el caso de que se mantenga presionada la tecla Shift, que se navegará de forma rápida seleccionando los puntos por los que pasa el ratón.

```
void Viewer::mouseMoveEvent(QMouseEvent *e)
{
   if(e->modifiers() == Qt::ShiftModifier){
      select(e);
   }
   QGLViewer::mouseMoveEvent(e);
}
```

- ***Presionar una tecla***

Aquí se analizan todas las teclas que producen cambios en la interfaz. Para permitir la combinación de más una tecla se añade una línea con una constante que almacena dicho identificador, en caso de existir:

```
   const Qt::KeyboardModifiers modifiers = e->modifiers();
```

La función de cada botón se especifica en la guía de usuario.

```
void Viewer::keyPressEvent(QKeyEvent *e)
{
   const Qt::KeyboardModifiers modifiers = e->modifiers();
   // A simple switch on e->key() is not sufficient if we want to take state key into account.
   // With a switch, it would have been impossible to separate 'F' from 'CTRL+F'.
   // That's why we use imbricated if...else and a "handled" boolean.
   bool handled = false;
   if ((e->key()==Qt::Key_C) && (modifiers==Qt::NoButton))
   {
      showCamera= ! showCamera;
      handled = true;
      updateGL();
   }
   else if ((e->key()==Qt::Key_V) && (modifiers==Qt::NoButton))
   {
      //       posFromCamera = !posFromCamera;
      handled = true;
      updateGL();
   }
```

```
else
   if ((e->key()==Qt::Key_F) && (modifiers==Qt::NoButton))
   {
      handled = true;
      updateGL();
   }
   else
      if ((e->key()==Qt::Key_L) && (modifiers==Qt::NoButton))
      {
         showCloud = !showCloud;
         handled = true;
         updateGL();
      }
      else
         if ((e->key()==Qt::Key_P) && (modifiers==Qt::NoButton)){
            QFile file;
            QDir::setCurrent(QDir::homePath());
            file.setFileName("points.txt");
            file.open(QIODevice::Append);
            QString *g;
            QString msg2;
            msg2.clear();
            msg2.append(g->number(currentPoint.x));
            msg2.append(" ");
            msg2.append(g->number(currentPoint.y));
            msg2.append(" ");
            msg2.append(g->number(currentPoint.z));
            msg2.append(" ");
            qDebug()<<"Save point in file";
            QChar *data=msg2.data();
            while(!data->isNull()){
               file.write((char*)data);
               data++;
            }
            char msg1[2]="\n";
            file.write(msg1,strlen(msg1));
            file.close();

         }
         else
            if((e->key()==Qt::Key_D) && (modifiers==Qt::NoButton))
            {
               //show sphere
               quadratic=gluNewQuadric();
               gluQuadricNormals(quadratic, GLU_SMOOTH);
               gluQuadricTexture(quadratic, GL_TRUE);
               gluSphere(quadratic,1.3f,32,32);
               updateGL();
            }
            else
               if((e->key()==Qt::Key_Q) && (modifiers==Qt::NoButton))
               {
                  rad=1;
                  qb=!qb;
                  updateGL();
               }
               else
                  if((e->key()==Qt::Key_W) && (modifiers==Qt::NoButton))
                  {
```

```
                if(window)
                {
                  ctrl=!ctrl;
                  float* aux=window->getCorners();
                  if(aux)
                  {
                    vectorV2.clear();
                    for(int i=0;i<12;i=i+3)
                    {
                      Vec *point=new Vec();
                      point->setValue(aux[i],aux[i+1],aux[i+2]);
                      vectorV2.append(*point);
                    }
                  }
                  updateGL();
                }
              }
              else if((e->key()==Qt::Key_M) && (modifiers==Qt::NoButton))
                {
                  colorCamera= !colorCamera;
                  handled = true;
                  updateGL();
                }
  if (!handled)
    QGLViewer::keyPressEvent(e);
}
```

Señales que puede recibir la clase Viewer procedentes de la interfaz auxiliar:

- ***changeCubeValue***
- ***changeCenter***

La primera de ellas cambia el valor del cubo que rodea al punto seleccionado y la segunda cambia el centro de rotación de la interfaz.

```
void Viewer::changeCubeValue(int v)
{
  rad=v;
  updateGL();
}
void Viewer::changeCenter(int x, int y, int z)
{
  QPoint *p=new QPoint(x,y);
  select(*p);
  currentPoint.setValue(x,y,z);
  setSceneCenter(currentPoint);
  postSelection(*p);
  updateGL();
}
```

Señales que puede recibir la clase Viewer procedentes de la clase ImageWidget:

- ***newPointsToFit***

Señal recibida de ImageWidget cuando el usuario ha seleccionado un plano que está bien colocado y ha hecho click en “Fix 2D coord”. Se almacenan los datos del plano en las variables correspondientes para su representación.

```
void Viewer::newPointsToFit(QVector<int> polySides,QVector<bool> inOut,int option)
{
   if(window){
      float* aux=window->getCorners();
      if(aux)
      {
         //Show coordinates of corners
//          qDebug()<<"[Viewer] Corners";
//          qDebug()<<"("<<aux[0]<<aux[1]<<aux[2]<<")";
//          qDebug()<<"("<<aux[3]<<aux[4]<<aux[5]<<")";
//          qDebug()<<"("<<aux[6]<<aux[7]<<aux[8]<<")";
//          qDebug()<<"("<<aux[9]<<aux[10]<<aux[11]<<")";
         //if the last plane have to be modified, for another tab, option=1
         if(option==1){
            for(int i=0;i<polySides.size();i++) {
               arrayPolygon.pop_back();
            }
         }
         //FOR QVector<int> polySides
         int auxSum=0;
         for(int k=0;k<polySides.size();k++){
            vectorV.clear();
            if(k>0)
               auxSum=polySides[k]*3;
            for(int i=0;i<polySides[k]*3;i=i+3){
               Vec *point=new Vec();
               point->setValue(aux[i+auxSum],aux[i+1+auxSum],aux[i+2+auxSum]);
               vectorV.append(*point);
            }
            Polygon *poly=new Polygon(vectorV,inOut[k]);
            arrayPolygon.append(poly);
         }
         updateGL();
      }
   }
}
```

- ***newPlane***

Cuando el usuario termina con un plano, se añade al correspondiente array y se limpia el array de los polígonos para dibujar el siguiente plano.

```
void Viewer::newPlane(QVector<int> idCamsText)
{
   //NEW THINGS
   //If I have polygons that I have to save in a class Plane "pl"
   qDebug()<<"elements in polygon array:"<<arrayPolygon.size();
   Plane *pl=new Plane(arrayPolygon);
   arrayPlane.append(pl);
   //reset arrayPolygon
   arrayPolygon.clear();
}
```

- ***newBuilding***

Se resetean los planos anteriores y se guarda la información en un nuevo array

```
void Viewer::newBuilding(){
   //create a new building and call write function
```

```
    Building *build=new Building(arrayPlane);
    arrayBuilding.append(build);
    arrayPlane.clear();
}
```

- ***saveInfo***

El fichero para guardar los datos se realiza en forma de cadena entre varias clases. Aquí es donde se crea el fichero, pasándole la información de cuántos edificios ha creado el usuario, con el array de nombre arrayBuilding. A continuación se irán enlazando el resto de clases, cada una llamando a un método write definido dentro de ellas, hasta completar el fichero en su totalidad. El orden sería el siguiente: Empezaría en el método write de la clase Building, para llamar al método write de la clase Plane, quien por último llamaría al método write de la clase Polygon, el cual se encarga de completar el fichero añadiendo los puntos finales.

El código de la clase Viewer es el siguiente (el resto de código se encuentra en la definición de cada clase):

```
void Viewer::saveInfo(){
//    WRITE in a file :)
      QChar *data;
      QFile file;
      QDir::setCurrent(QDir::homePath());
      file.setFileName("save_data.txt");
      file.open(QIODevice::Append);
      QString *g;
      QString msg;
      msg.append("BUILDINGS -> ");
      msg.append(g->number(arrayBuilding.size()));
      msg.append("\n");
      data=msg.data();
      while(!data->isNull()){
         file.write((char*)data);
         data++;
      }
//       Rectangle *r=arrayRect.last();
      for(int i=0;i<arrayBuilding.size();i++){
         Building *building=arrayBuilding.at(i);
         building->write(file);
      }
      file.close();
}
```

Como resultado final de que cada método write se ejecute, se crea un fichero donde primero se indica el número de edificios que el usuario ha construido, seguido por las características del primer edificio, el número de planos. Una vez llegamos a este punto, se muestran el número de polígonos que pertenecen a un plano y, por último, el número de vértices de ese plano, indicando cada número de vértices un polígono distinto. Lo último en representar son los puntos, a los que precederá un asterisco en caso de que se quiera indicar que el polígono es excluyente del plano indicado. Una vez hecho esto, si hay algún polígono más que representar, se escribe con todos sus datos a continuación hasta que no haya más y se vuelve a realizar el mismo proceso con los planos, hasta que se terminen de escribir todos por completo.

El diagrama de flujo:

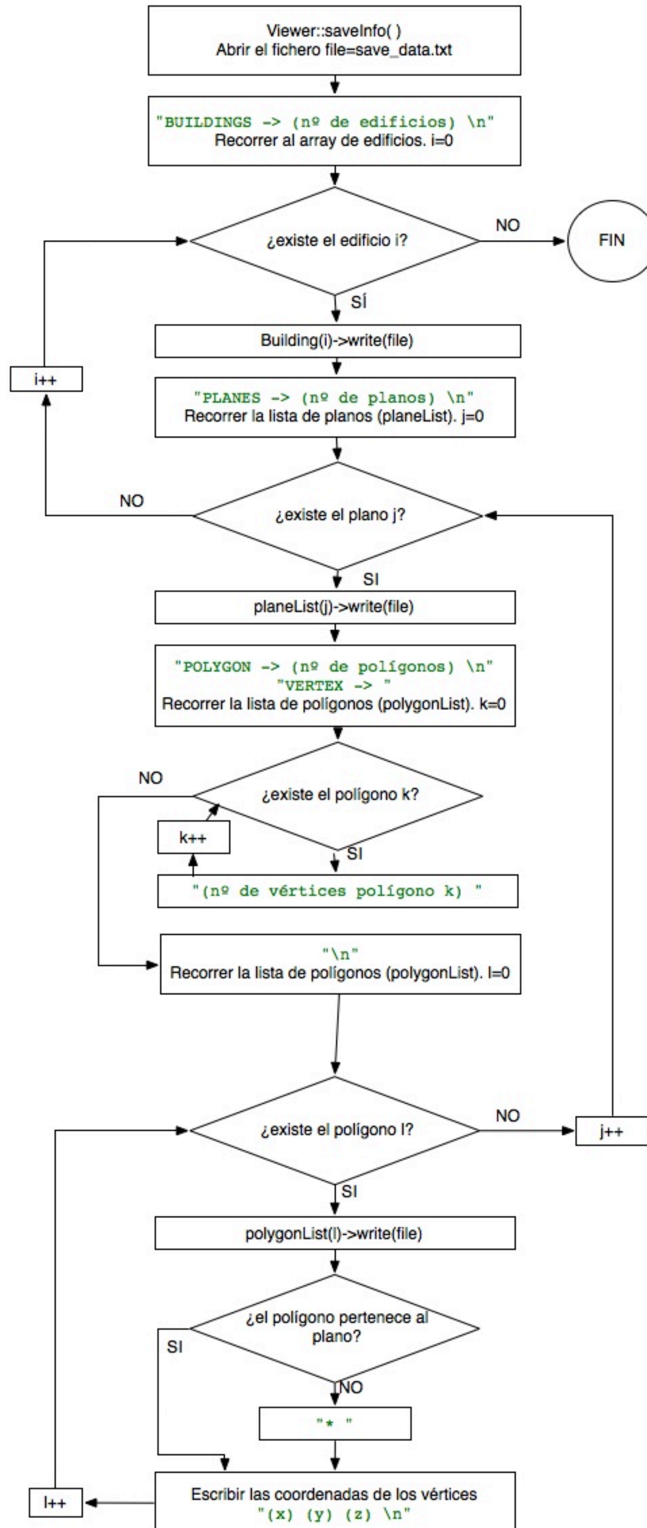


Ilustración 3 Diagrama de flujo qu representa el proceso de guardado de la información

3.2.5. Parameters

Clase que implementa una interfaz auxiliar de la clase Viewer para manejar ciertas opciones útiles para el usuario. Estas opciones son: la elección de un centro en

la representación gráfca, las coordenadas de dicho centro y la posibilidad de elegir el número de muestras alrededor de un punto, dándole tamaño a la imagen de un cubo que se situará alrededor del punto seleccionado.

Código:

```
parameters::parameters(QWidget *parent) :
   QWidget(parent)
{
   cubeSlider=createSlider();
   button=createButton();
   leX=new QLineEdit;
   leY=new QLineEdit;
   leZ=new QLineEdit;
   lab1=new QLabel("Center (x,y,z) ->");
   lab2=new QLabel("Cube's radius");
//    button=new QPushButton("Set Center");
   QGridLayout *lg = new QGridLayout;
   QHBoxLayout *lh = new QHBoxLayout;
   lg->addLayout(lh,0,0);
   lh->addWidget(lab1);
   lh->addWidget(leX);
   lh->addWidget(leY);
   lh->addWidget(leZ);
   lh->addWidget(button);
   lg->addWidget(lab2,1,0);
   lg->addWidget(cubeSlider,2,0);
   setLayout(lg);
   connect(cubeSlider,SIGNAL(valueChanged(int)),this,SLOT(cubeValue(int)));
   connect(button,SIGNAL(clicked()),this,SLOT(pressCenter()));
}
```

Resultado gráfico:

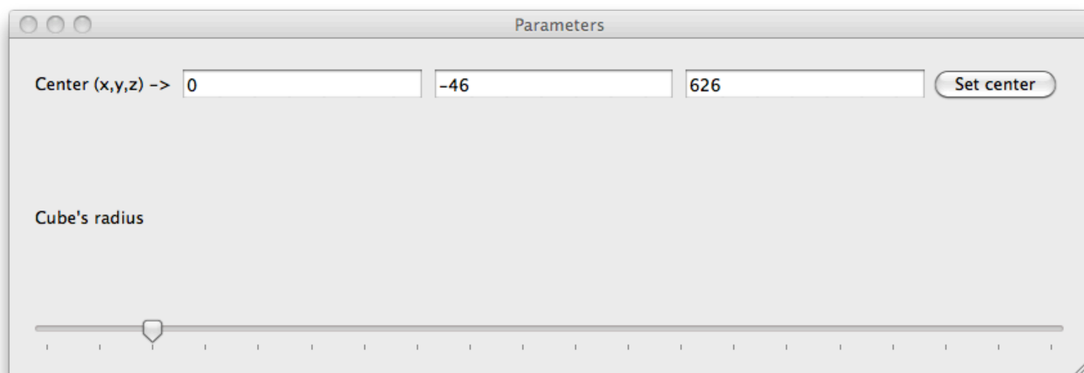


Ilustración 4 Venta de la clase *Parameters*

3.2.6. Clases para la reconstrucción de edificios

Partiendo del concepto general un edificio, (clase Building), se va a componer de planos, al menos de un plano (clase Plane). Dicho planos se van a componer a su vez de polígonos (clase Polygon). Un polígono está compuesto por un número de vértices,

por las coordenadas de esos vértices sobre la imagen de la cámara, que serán de la clase PolygonData2d y por una variable que indique si el polígono pertenece al plano o no. Por último, las coordenadas del polígono sobre la imagen de la cámara(clase PolygonData2d),contienen la información sobre el punto de la imágen, el índice de la cámara y un flag denominado manual. Estudiando las clases más detalladamente, tenemos:

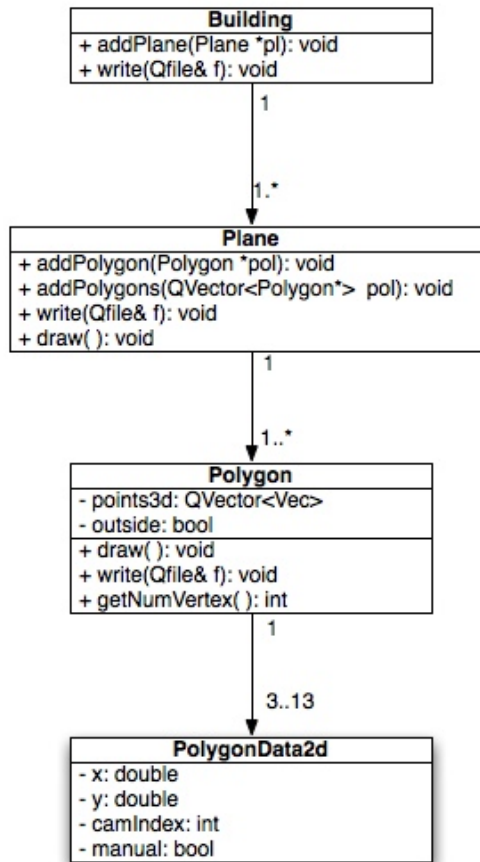


Ilustración 5 Relación entre las clases encargadas de reconstruir un edificio

3.2.6.1. PolygonData2D

Clase que almacena los puntos dentro de las imágenes, por tanto sus atributos serán dos coordenadas, un índice de cámara y una variable de tipo booleana para indicar si el punto ha sido calculado de forma manual o de forma estimada mediante algún algoritmo.

```
class PolygonData2d
{
public:
   PolygonData2d();
   PolygonData2d(double coordX,double coordY,int id,bool m);
private:
   double x;
   double y;
   int camIndex;
   bool manual;
};
```

3.2.6.2. *Polygon*

Esta clase se encarga de crear los polígonos. Cada polígono se define por sus esquinas, es decir, por puntos de la clase PolygonData2d, siendo cada uno de ellos una esquina del polígono. Estas esquinas tienen su representación correspondiente en el plano tridimensional,por ello también existe un array de tipo Vec, que contiene las coordenadas de los puntos. Como último atributo disponemos de una variable booleana llamada outsider, que indica si el polígono pertenece o elimina cierta parte de un plano.

```
class Polygon
{
public:
   Polygon();
   Polygon(QVector<Vec> qv,bool inOut);
   void draw();
   void write(QFile &f);
   int getNumVertex();
private:
   QVector<PolygonData2d> staff;//format: (x,y,camIndex,pointManual/pointAutomatic)
   QVector<Vec> points3d;
   bool outside;
};
```

Esta clase tendrá disponible el método dibujar, para la acción de representar el polígono en el entorno tridimensional. Si pertenece al plano, el color de la representación será rosa y si está fuera de él, se mostrará de color azul.

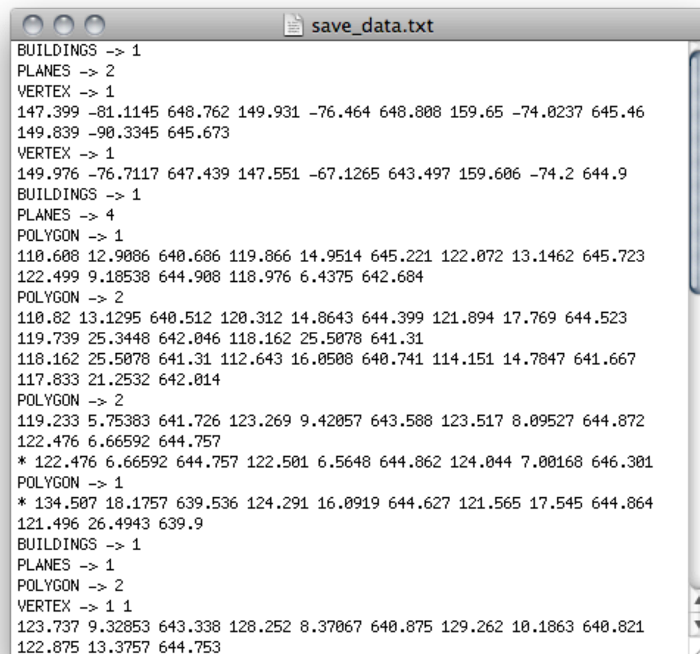
```
void Polygon::draw(){
   if(!outside)
      glColor3f(1,0,1);
   else
      glColor3f(0,0,1);
   glEnable(GL_TEXTURE_2D);
   glBegin(GL_POLYGON);
   for(int i=0;i<points3d.size();i++){
      glVertex3d(points3d.at(i).x,points3d.at(i).y,points3d.at(i).z);
   }
   glEnd();
   glDisable(GL_TEXTURE_2D);
}
```

El siguiente método utilizado es el write(), el cual se encarga de escribir la información en un fichero cuyo identificador se le pasa por argumento, y será la información que se guarda son las coordenadas tridimensionales que forman las esquinas del polígono. En caso de que el polígono se encuentre fuera del plano se imprimirá un asterisco antes de las coordenadas.

```
void Polygon::write(QFile &f){
   QChar *data;
   QString *g;
   QString msg;
   for(int i=0;i<points3d.size();i++){
      msg.clear();
      if(i==0){
```

```
            if(outside)
                msg.append("* ");
        }
        msg.append(g->number(points3d.at(i).x));
        msg.append(" ");
        msg.append(g->number(points3d.at(i).y));
        msg.append(" ");
        msg.append(g->number(points3d.at(i).z));
        msg.append(" ");
        data=msg.data();
        while(!data->isNull()){
            f.write((char*)data);
            data++;
        }
    }
    msg.clear();
    msg.append("\n");
    data=msg.data();
    while(!data->isNull()){
        f.write((char*)data);
        data++;
    }
}
```

Ejemplo método write


save_data.txt

```
BUILDINGS -> 1
PLANES -> 2
VERTEX -> 1
147.399 -81.1145 648.762 149.931 -76.464 648.808 159.65 -74.0237 645.46
149.839 -90.3345 645.673
VERTEX -> 1
149.976 -76.7117 647.439 147.551 -67.1265 643.497 159.606 -74.2 644.9
BUILDINGS -> 1
PLANES -> 4
POLYGON -> 1
110.608 12.9086 640.686 119.866 14.9514 645.221 122.072 13.1462 645.723
122.499 9.18538 644.908 118.976 6.4375 642.684
POLYGON -> 2
110.82 13.1295 640.512 120.312 14.8643 644.399 121.894 17.769 644.523
119.739 25.3448 642.046 118.162 25.5078 641.31
118.162 25.5078 641.31 112.643 16.0508 640.741 114.151 14.7847 641.667
117.833 21.2532 642.014
POLYGON -> 2
119.233 5.75383 641.726 123.269 9.42057 643.588 123.517 8.09527 644.872
122.476 6.66592 644.757
* 122.476 6.66592 644.757 122.501 6.5648 644.862 124.044 7.00168 646.301
POLYGON -> 1
* 134.507 18.1757 639.536 124.291 16.0919 644.627 121.565 17.545 644.864
121.496 26.4943 639.9
BUILDINGS -> 1
PLANES -> 1
POLYGON -> 2
VERTEX -> 1 1
123.737 9.32853 643.338 128.252 8.37067 640.875 129.262 10.1863 640.821
122.875 13.3757 644.753
```

Ilustración 6 Ejemplo de un fichero con información sobre varios edificios

Por último existe un método get que devuelve el número de vértices que tiene el polígono.

```
int Polygon::getNumVertex(){
    return points3d.size();
```

```
}
```

3.2.6.3. Plane

Esta clase se encarga de agrupar polígonos, es decir, un plano estará formado al menos por un polígono.

```
class Plane
{
public:
   Plane();
   Plane(QVector<Polygon*> poly);
   void addPolygon(Polygon *pol){polygonList.append(pol);}
   void addPolygons(QVector<Polygon*> pol){polygonList=pol;}
   void write(QFile& f);
   void draw();
private:
   QVector<Polygon*> polygonList;
};
```

Como métodos, se observa que, a un plano se le puede agregar un polígono o una lista de éstos. Además se implementa el método draw, cuya única función es la de llamar al método del mismo nombre perteneciente a los polígonos que contiene.

```
void Plane::draw(){
   for(int i=0;i<polygonList.size();i++){
      polygonList.at(i)->draw();
   }
}
```

El método write se encarga de escribir, en el fichero que se le ha pasado como argumento, la información relativa al número de polígonos que contiene y los vértices que tienen cada uno de esos polígonos , para seguidamente llamar al método write de cada uno de los polígonos.

```
void Plane::write(QFile &f){
   QChar *data;
   QString *g;
   QString msg;
   msg.append("POLYGON -> ");
   msg.append(g->number(polygonList.size()));
   msg.append("\n");
   msg.append("VERTEX -> ");
   for(int i=0;i<polygonList.size();i++){
      msg.append(g->number(polygonList[i]->getNumVertex()));
      msg.append(" ");
   }
   msg.append("\n");
   data=msg.data();
   while(!data->isNull()){
      f.write((char*)data);
      data++;
   }
   for(int i=0;i<polygonList.size();i++){
      polygonList.at(i)->write(f);
```

```
    }
}
```

3.2.6.4. Building

Como elemento superior, que contiene al resto de los elementos que conforman las figuras, se encuentra la clase que forma los edificios, llamada Building. El único atributo de esta clase es una lista de planos, con un método para poder añadirlos.

```
class Building
{
public:
    Building();
    Building(QVector<Plane*> p);
    void addPlane(Plane *pl){planeList.append(pl);}
    void write(QFile& f);
private:
    QVector<Plane*> planeList;
};
```

El método write se usa para escribir la información relativa al número de planos que forman parte del edificio. Seguídamente se llamará a la función de cada uno de estos planos, para que cada uno escriba sus características.

```
void Building::write(QFile& f){
    QChar *data;
    QString *g;
    QString msg;
    msg.append("PLANES -> ");
    msg.append(g->number(planeList.size()));
    msg.append("\n");
    data=msg.data();
    while(!data->isNull()){
        f.write((char*)data);
        data++;
    }
    for(int i=0;i<planeList.size();i++){
        planeList.at(i)->write(f);
    }
}
```

3.3. Comunicación entre clases (signals y slots)

En este apartado se analiza toda la información que se transmite entre las distintas clases para el funcionamiento del programa.

Empezando por la clase Viewer y sus conexiones con la ventana auxiliar para su manejo, el intercambio de información se basa en que el usuario puede elegir cambiar el tamaño del cubo visible en la interfaz Viewer, cambiar el centro de rotación o visualizar donde está el centro actual en la imagen.

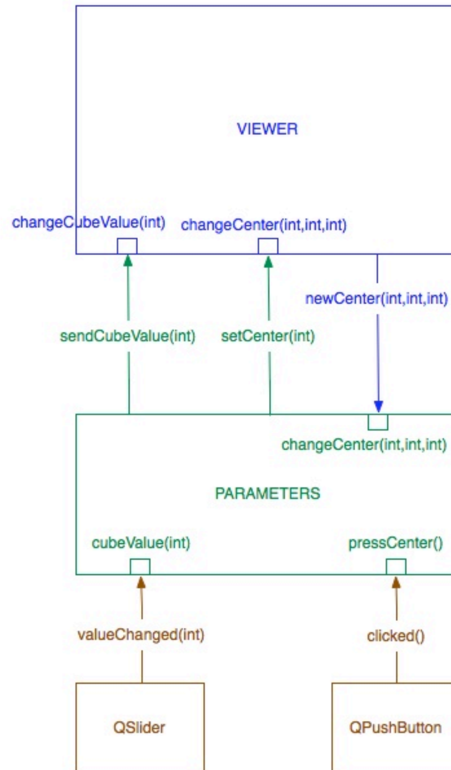


Ilustración 7 Señales y slots entre la clase *Viewer, Parameters* y el manejo de botones de la interfaz

Por otro lado la clase Viewer, cuando el usuario lo indique, crea una clase ImageWidget. La comunicación es unidireccional, de ImageWidget a Viewer, ya que ha sido la clase Viewer la creadora de la otra. Por tanto, ImageWidget tiene toda la información necesaria, pero no al revés, ya que será el usuario el que añada información sobre nuevos puntos, planos, edificios, etc, o indique una acción.

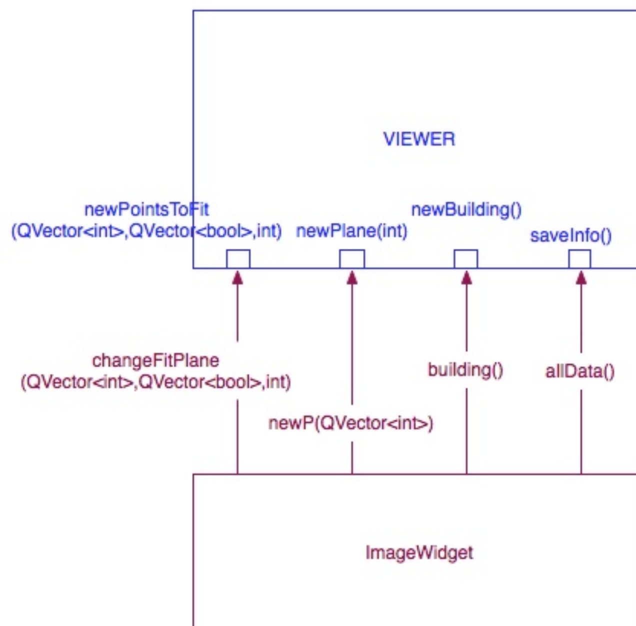


Ilustración 8 Señales y slots de la clase *Viewer* e *ImageWidget*

A su vez la clase ImageWidget necesita crear otra clase para el manejo de las imágenes llamada ImageView. Esta última clase se encarga de dibujar sobre las imágenes los puntos o figuras que indique el usuario o la clase ImageWidget y de presentar los puntos o figuras en sus respectivas coordenadas independientemente del tamaño de la imagen.

También la clase ImageWidget provee al usuario de ciertos botones para ir realizando las acciones, que también son manejados en forma de señales para realizar las acciones determinadas si el usuario pulsa alguno de ellos.

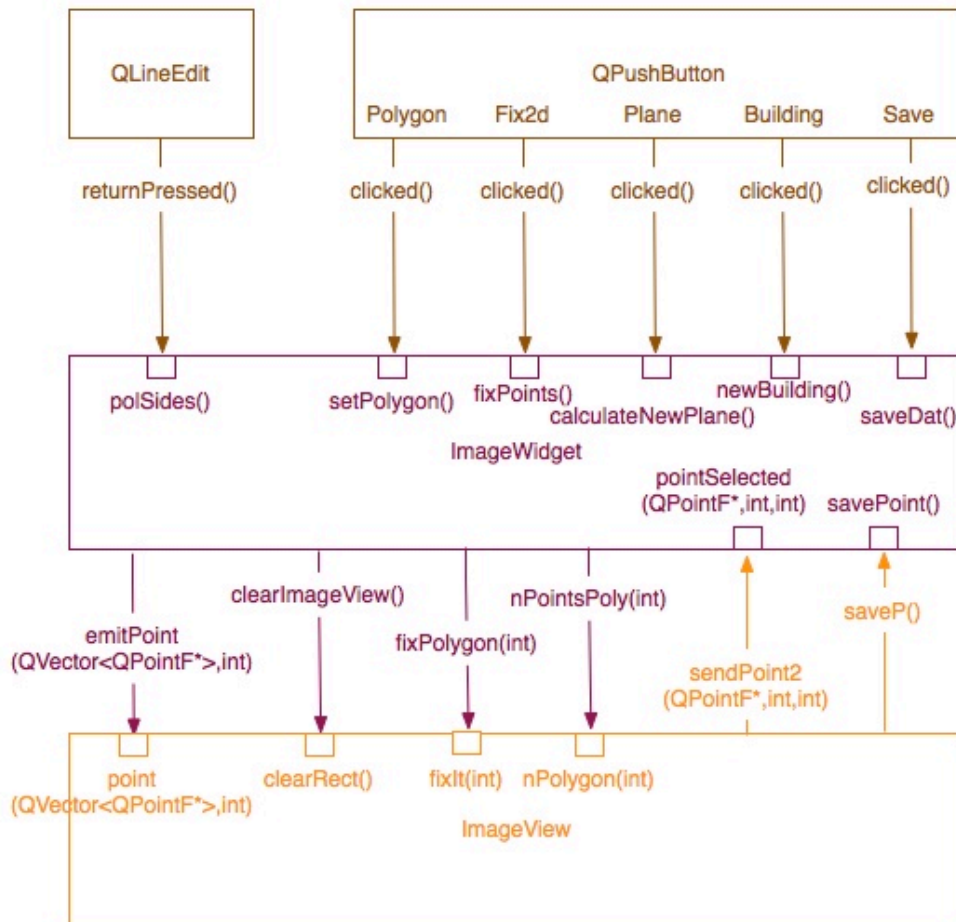


Ilustración 9 Señales y slots de la clase ImageWidget, ImageView y los botones de la interfaz de imágenes

3.4. Manual de usuario

En la pantalla principal, por defecto, se dibuja una nube de puntos (color amarillo) que representa el área geográfica recreada a partir de las fotografías. La posición de la cámara donde se ha tomado cada fotografía saldrá representada por una pirámide de color rojo. Hay una ventana auxiliar, con información acerca de la pantalla principal, que aparece en la esquina inferior izquierda. A continuación se ven las opciones que se pueden realizar en cada ventana.

1. Ventana principal: se puede navegar a través de ella mediante el ratón y activar ciertas opciones usando el teclado.

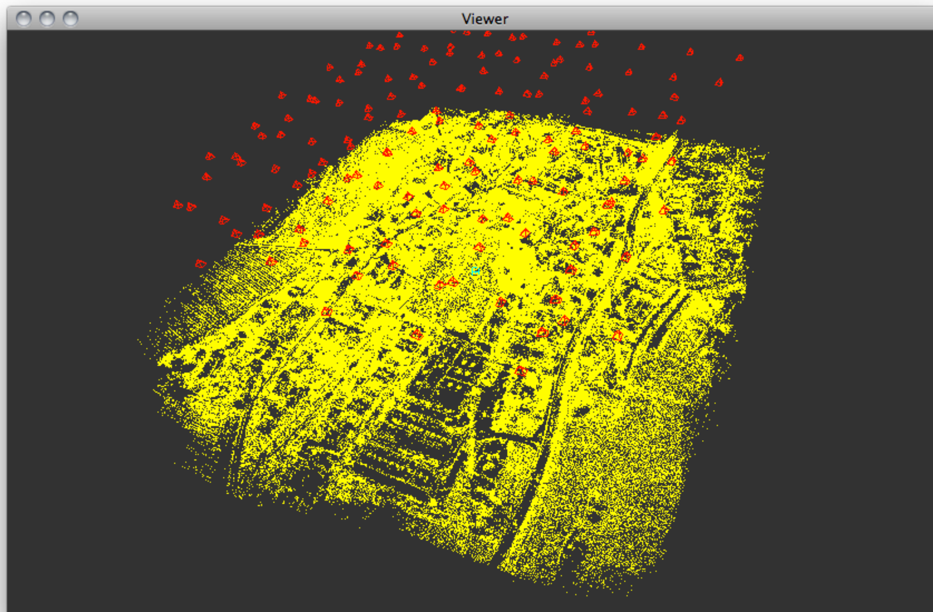


Ilustración 10 Interfaz inicial del programa. Se observa la nube de puntos en amarillo y la nube de cámaras en rojo

- Click izquierdo + movimiento del ratón → Permite rotar la imagen alrededor del centro fijado, inicialmente el punto azul.

- Ruleta del ratón → Añade o quita zoom a la imagen.

- Doble Click izquierdo → Sitúa un nuevo centro de rotación de la imagen además de seleccionar un punto de la nube (el punto más cercano donde se ha clickado).

- Shift + movimiento del ratón → Navega de forma rápida por los distintos puntos, cambiando el punto azul, pero sin cambiar el centro de rotación.

- Shift + click izquierdo → Selecciona el punto más cercano y lanza una segunda interfaz donde se pueden ver todas las imágenes que contiene ese punto.

Teclado:

- A → Muestra/elimina el eje "xyz"
- G → Muestra/elimina una malla en el fondo de la imagen
- H → Ayuda
- L → Muestra/elimina la nube de puntos y la nube de cámaras
- M → Muestra/elimina que se iluminen las cámaras que contienen en sus imágenes el punto azul.

- Q → Muestra/elimina el cubo representado al lado del punto azul
- W → Muestra/elimina, en caso de que existan, los planos de edificios que el usuario ha marcado.

• Ventana auxiliar

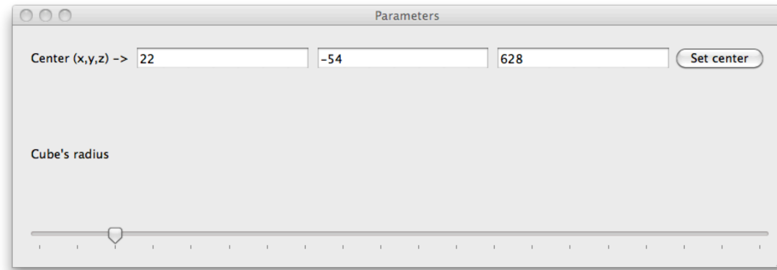


Ilustración 11 Ejemplo de ventana auxiliar para el manejo de la ventana inicial

En esta ventana se puede modificar el radio del cubo, lo que permite escoger nubes de puntos más grandes o más pequeñas a la hora de analizar las imágenes. El tamaño se cambia haciendo click en el marcador que se encuentra en la barra inferior y desplazándolo hacia la derecha para aumentar el radio y a la izquierda para disminuirlo. También se incluye la opción de poder escoger un centro de la imagen introduciendo sus coordenadas y pulsando "Set center", así como ver las coordenadas de los puntos que se eligen en la imagen como centro.

2. Ventana de imágenes (al pulsar shift+click en la principal)

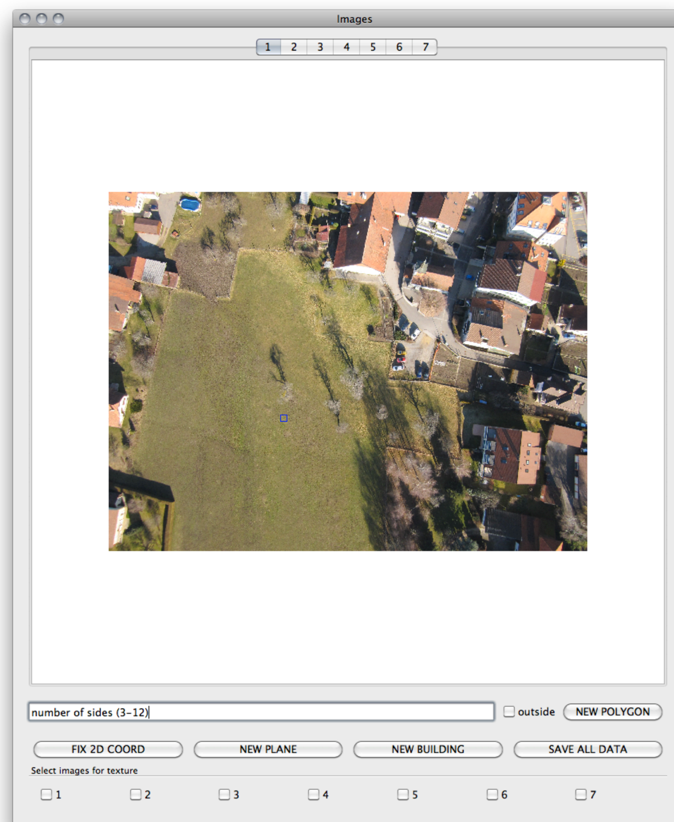


Ilustración 12 Ventana de imágenes

Esta ventana se abre al seleccionar cierto punto de la nube de puntos. En esta ventana podemos navegar por las imágenes que contienen el punto elegido, representado por un cuadrado azul, en cada una de ellas. Además podemos seleccionar cualquier figura geométrica para formar un edificio que luego puede reconstruirse en el plano tridimensional.

Opciones para navegar por las imágenes:

- Click izquierdo + movimiento del ratón → Permite navegar por la imagen según en la dirección que la movamos (siempre que la imagen sea mayor que el tamaño de ventana). También sirve para mover un punto de sitio si se ha hecho click sobre su área respectiva

- Ruleta del ratón → Añade o quita zoom a la imagen.

- Shift + Click izquierdo → Si ya se ha introducido un número de esquinas del polígono, esta combinación añade una esquina en el sitio donde se indique.

Pasos a seguir en esta interfaz:
1. Marcar si el polígono que se va a elegir va a pertenecer o no al plano, con la casilla de “outside”.
2. Escribir un número de lados en el lado en el espacio reservado para ello y pulsar intro.
3. Ir a una imagen de las disponibles y dibujar el polígono haciendo shift+click para introducir cada una de las esquinas. Una vez introducidas, ajustar las esquinas de forma que el polígono coincida lo mejor posible con el que se desea.
4. Si se desea que el plano quede formado por otro polígono, hacer click en “New polygon” y volver al paso 1.
5. Una vez ajustados todos los polígonos en una imagen, hacer click en “Fix 2D coord”.
6. Ir a otra imagen de las disponibles y ajustar en los puntos correspondientes a los de la otra imágenes los polígonos. Una vez hecho esto, hacer click en “Fix 2D coord”. Aparecerán uno o varios polígonos, depende de los que hayamos elegido, en rojo, que marcará los polígonos definitivos.
7. Si deseamos seguir ajustando el polígono rojo, volver al punto 6, para usar otra imagen hasta que ya no haya más o hasta que se ajuste como deseamos.
8. Para finalizar el plano introducido pulsar “New Plane”. Si se desea seguir trabajando con planos que se encuentren en el mismo edificio volver al paso 1.
9. Hacer click en “New Building” para terminar con el edificio actual. Si se desea seguir haciendo edificios, volver al paso 1.
10. Si se desea guardar lo que se ha creado, pulsar “Save all data”.

En forma de diagrama de flujo:

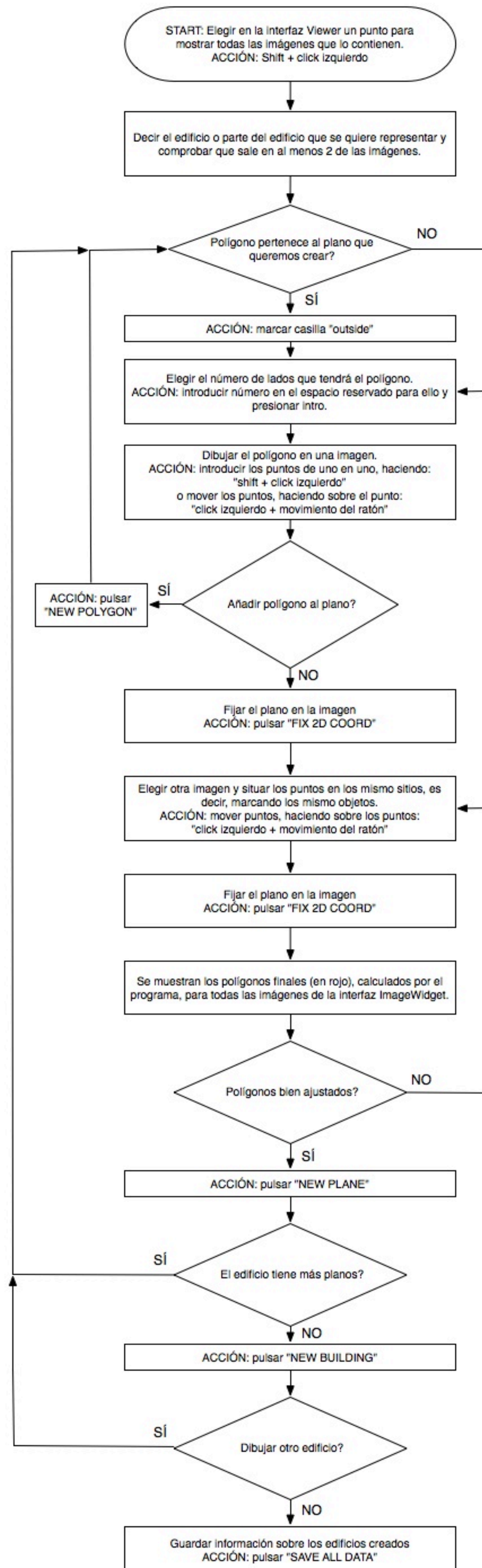


Ilustración 13 Diagrama de flujo para el uso de la ventana de imágenes

4. Resultados

4.1. Análisis de resultados obtenidos dependiendo de las muestras elegidas y algoritmo utilizado.

En este apartado se muestra la exactitud de los resultados obtenidos. Los métodos para el cálculo de puntos y planos han sido dos. El primero de ellos mediante las muestras en el interior de un cubo tridimensional, método PCA, y el segundo mediante triangulación con la selección de una misma figura en al menos dos imágenes. Se va a simular cómo responden dichos cálculos ante distintas opciones de entrada que pueden ser introducidas por el usuario.

Caso 1: cubo de área pequeña

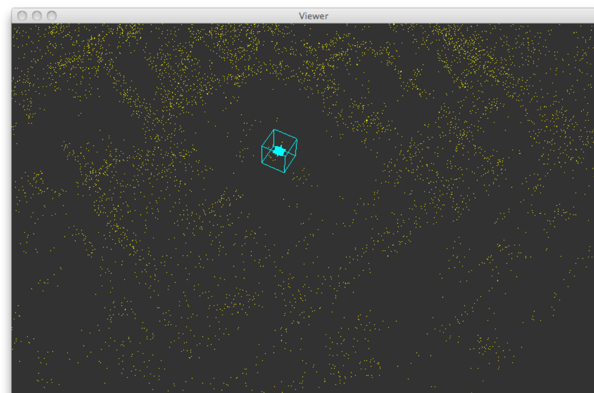


Ilustración 14 Interfaz principal con cubo de área pequeña

En este caso el área que rodea al punto no incluye muchos puntos de la nube. Sobre esos puntos internos se calcula un plano medio con el método PCA. Se van a analizar dos casos distintos, en cada uno se elegirá un plano. En el primer caso el plano se encontrará cerca del punto seleccionado, donde la nube de puntos forma un conjunto de datos representativo del plano que se elige. Para el segundo caso, sin embargo, se elige un plano alejado del punto seleccionado por lo que el conjunto de datos ya no es representativo de esa zona.

- Plano elegido cercano al punto escogido.

Se selecciona un plano que como se aprecia incluye al punto seleccionado (marcado en azul en la imagen).

Ilustración 15 Ejemplo de polígono, seleccionado manualmente, con el punto seleccionado en el interior

Una vez seleccionado en una imagen el polígono, se observa en el resto de imágenes cómo se ha calculado para comprobar el resultado de los cálculos. En este observando el resto de imágenes se comprueba cómo el plano calculado por el PCA da una muy buena aproximación en la mayoría de ellas.

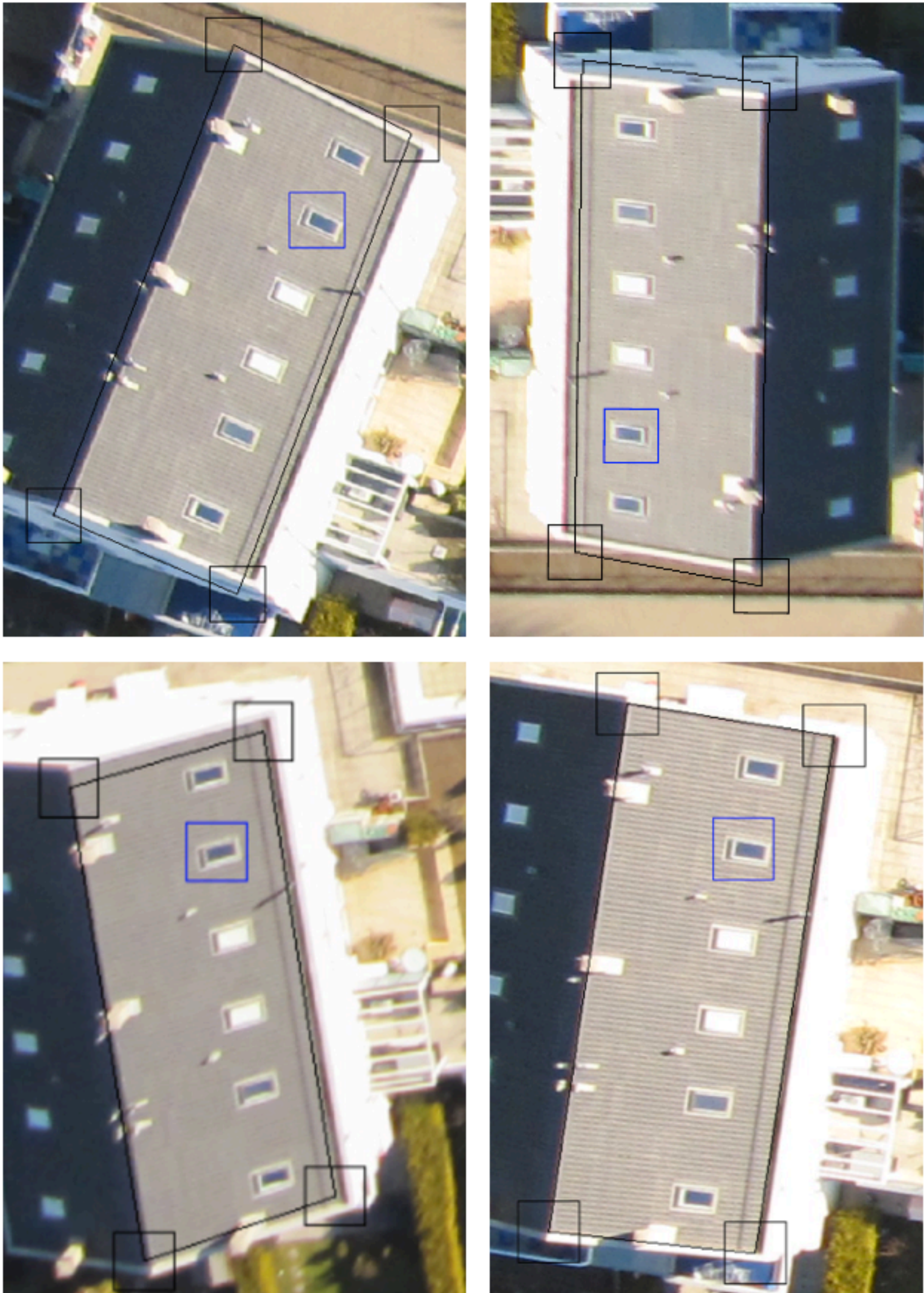


Ilustración 16 Polígonos calculados automáticamente por el método PCA

Si ahora se eligen 2 de las imágenes superiores y se fijan los puntos a ellas, obtendremos la resolución mediante triangulación en forma de rectángulo rojo. Las siguientes representaciones corresponden a 2 imágenes en los que todos los puntos se han calculado automáticamente.

Ilustración 17 Polígonos calculados automáticamente.
Polígono rojo por triangulación y polígono negro por el método PCA

Se aprecia una mejora para aquellos casos donde el primer ajuste no era del todo exacto, como en la imagen de la izquierda.

- Plano elegido alejado del punto escogido

Para este caso se elige un plano de un tejado que se encuentre lejos del punto seleccionado. Se parte de una imagen inicial, con el plano elegido manualmente:

Ilustración 18 Ejemplo de polígono seleccionado manualmente

Resto de figuras calculadas automáticamente a partir del polígono anterior:

Ilustración 19 Polígonos calculados automáticamente por el método PCA

Comparándolo con el primer caso, hay alguna imagen que se ajusta correctamente pero por regla general la mayoría de planos se alejan más del original que en el caso anterior. Esto se debe a que el plano PCA está calculado con los puntos situados en el interior del cubo y por tanto cercanos al punto elegido. Aquí al estar alejados y usar ese mismo plano, las aproximaciones no son tan buenas.

A continuación se fijan los puntos correctamente en dos imágenes y se recalcula un plano nuevo para este edificio. Los resultados en este caso para los nuevos planos calculados automaticamente son:

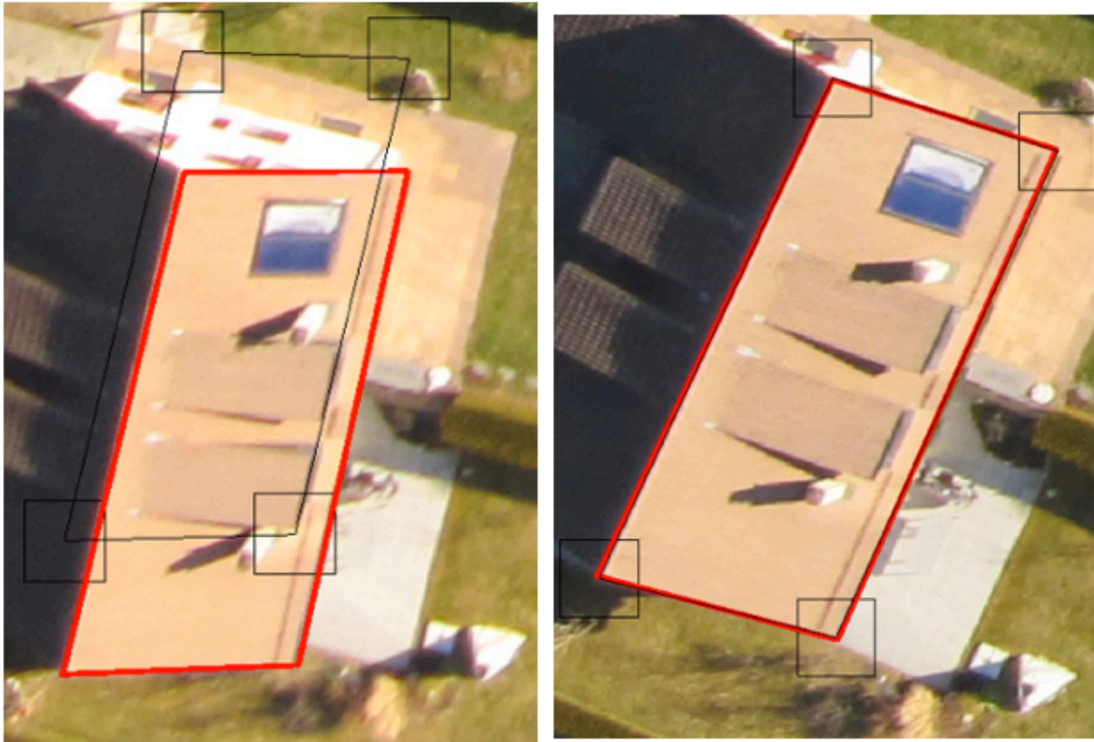


Ilustración 20 Polígonos calculados automáticamente.
Polígono rojo por triangulación y polígono negro por el método PCA

El plano calculado automáticamente sigue siendo igual de exacto en esta imagen que en la anterior pues no depende de dónde esté situado el punto inicial sino sólo de las imágenes escogidas.

Caso 2: cubo de área extensa

Para este apartado el cubo alrededor del punto seleccionado es de un área mayor que en el primer caso y las muestras presentarán distintas configuraciones, es decir, las muestras en este caso están agrupadas como conjuntos de edificios completos. por lo que el plano resultante dependerá mucho de la zona en la cual se tengan más muestras, si son de tejado será un plano más horizontal y si se tiene más información (puntos) de las fachadas el plano será más horizontal.

Se estudia el mismo entorno que en el apartado anterior, el punto seleccionado se encuentra en el mismo sitio, tal que:

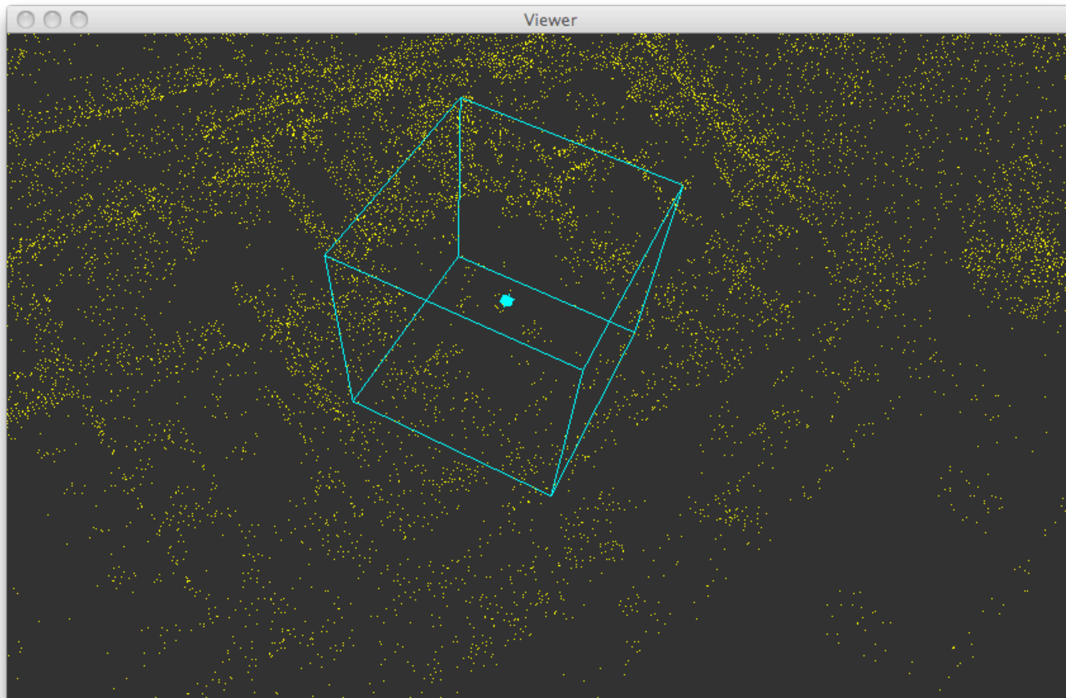


Ilustración 21 Interfaz principal con cubo de área grande

- Plano elegido cercano al punto escogido.

Se elige un plano que incluya en su interior al punto seleccionado (mismo plano que en caso 1).

Ilustración 22 Ejemplo de polígono, seleccionado manualmente, con el punto seleccionado en el interior

El resto de polígonos calculados automáticamente:

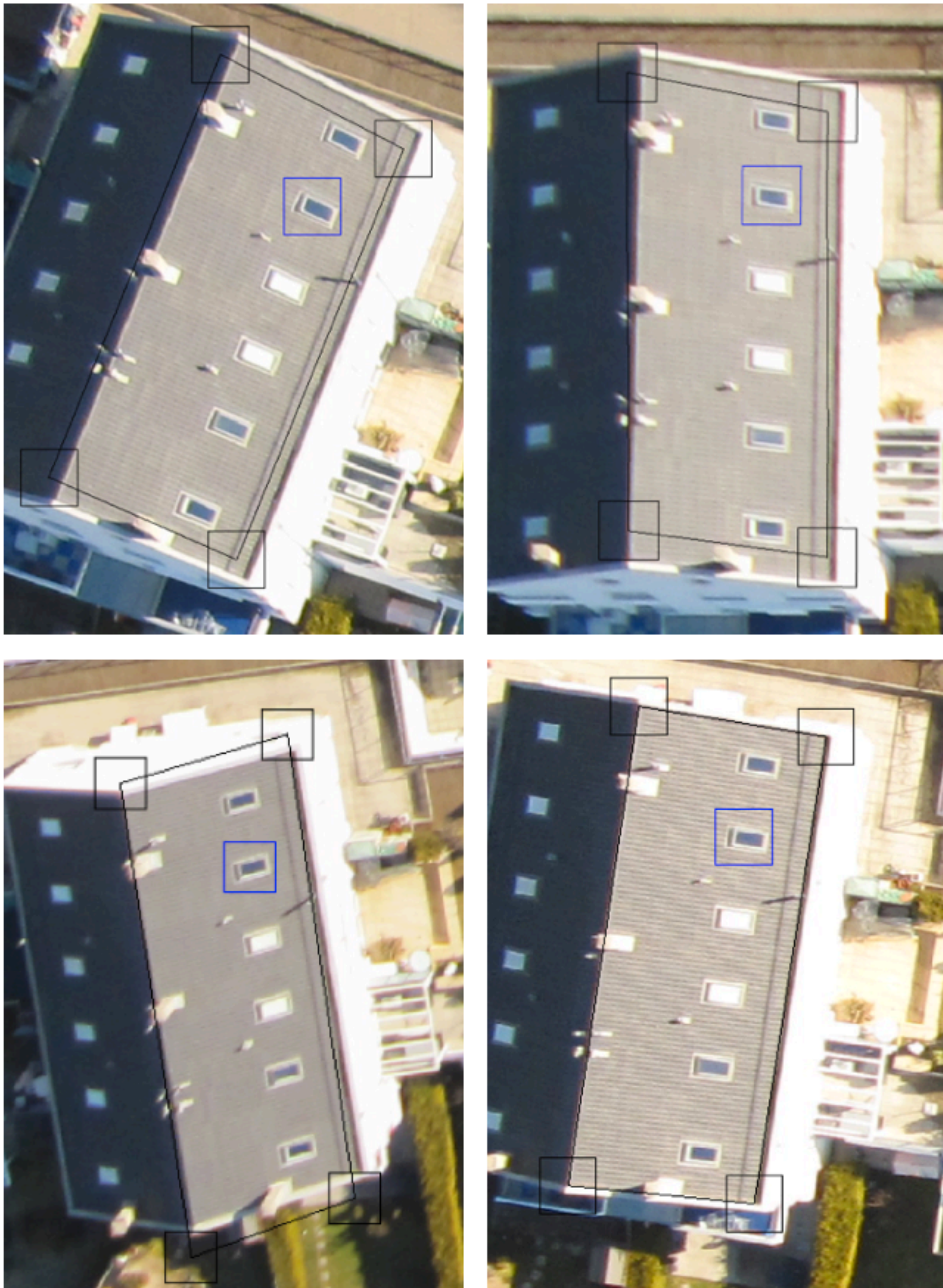


Ilustración 23 Polígonos calculados automáticamente por el método PCA

La aproximación es algo peor que en el primer caso, ya que ahora el plano calculado por el algoritmo PCA no tenía puntos sólo del techo seleccionado sino también de todo el edificio más algunos de alrededor.

Sin embargo se obtienen los mismos resultados para el plano calculado mediante triangulación de dos imágenes.

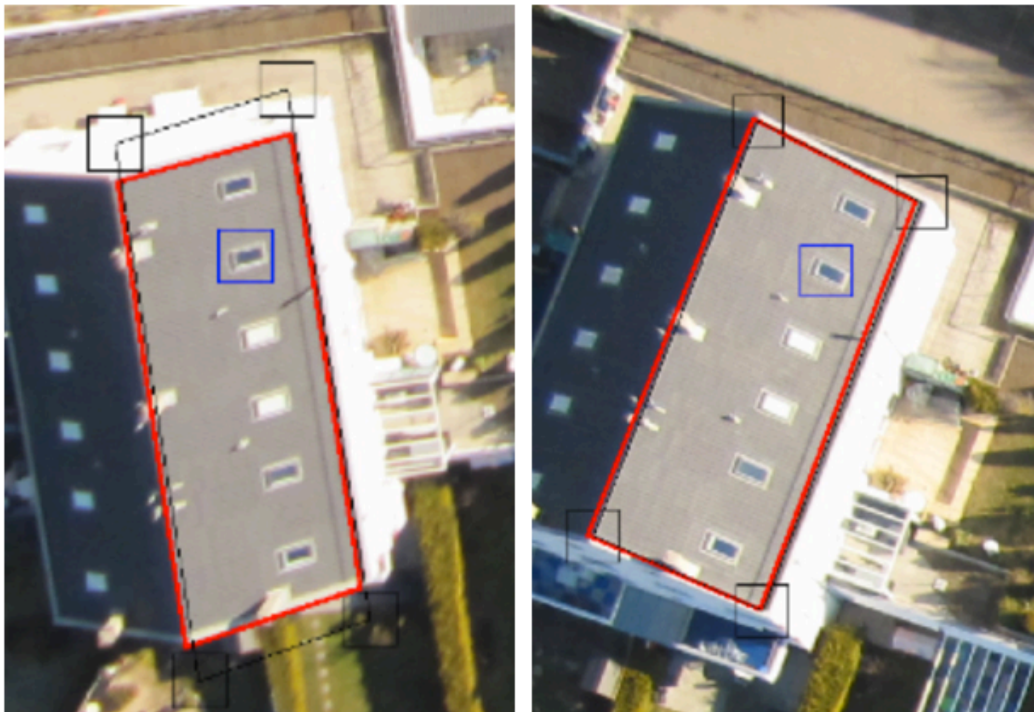


Ilustración 24 Polígonos calculados automáticamente.
Polígono rojo por triangulación y polígono negro por el método PCA

- Plano elegido alejado al punto escogido

Si se escoge un edificio alejado del punto seleccionado, lo más probable es que la aproximación calculada mediante el plano PCA sea mejor que en el primer caso, esto es debido a que ahora al haber un cubo de más área el plano será más general respecto a cualquiera que se elija. Volvemos a partir igual que en el caso 1 de una imagen inicial seleccionada manualmente:

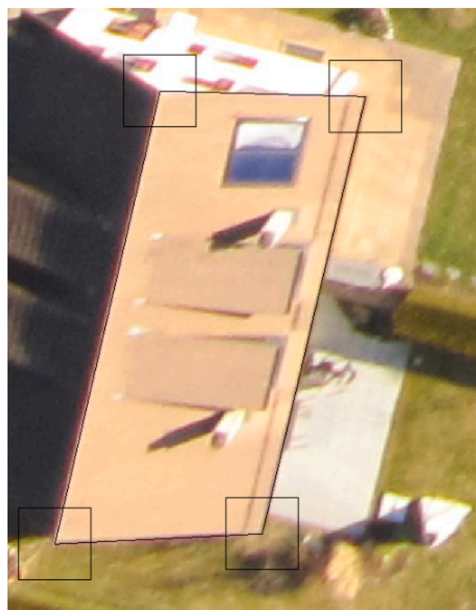


Ilustración 25 Ejemplo de polígono seleccionado manualmente

Imágenes calculadas automáticamente mediante el plano PCA:

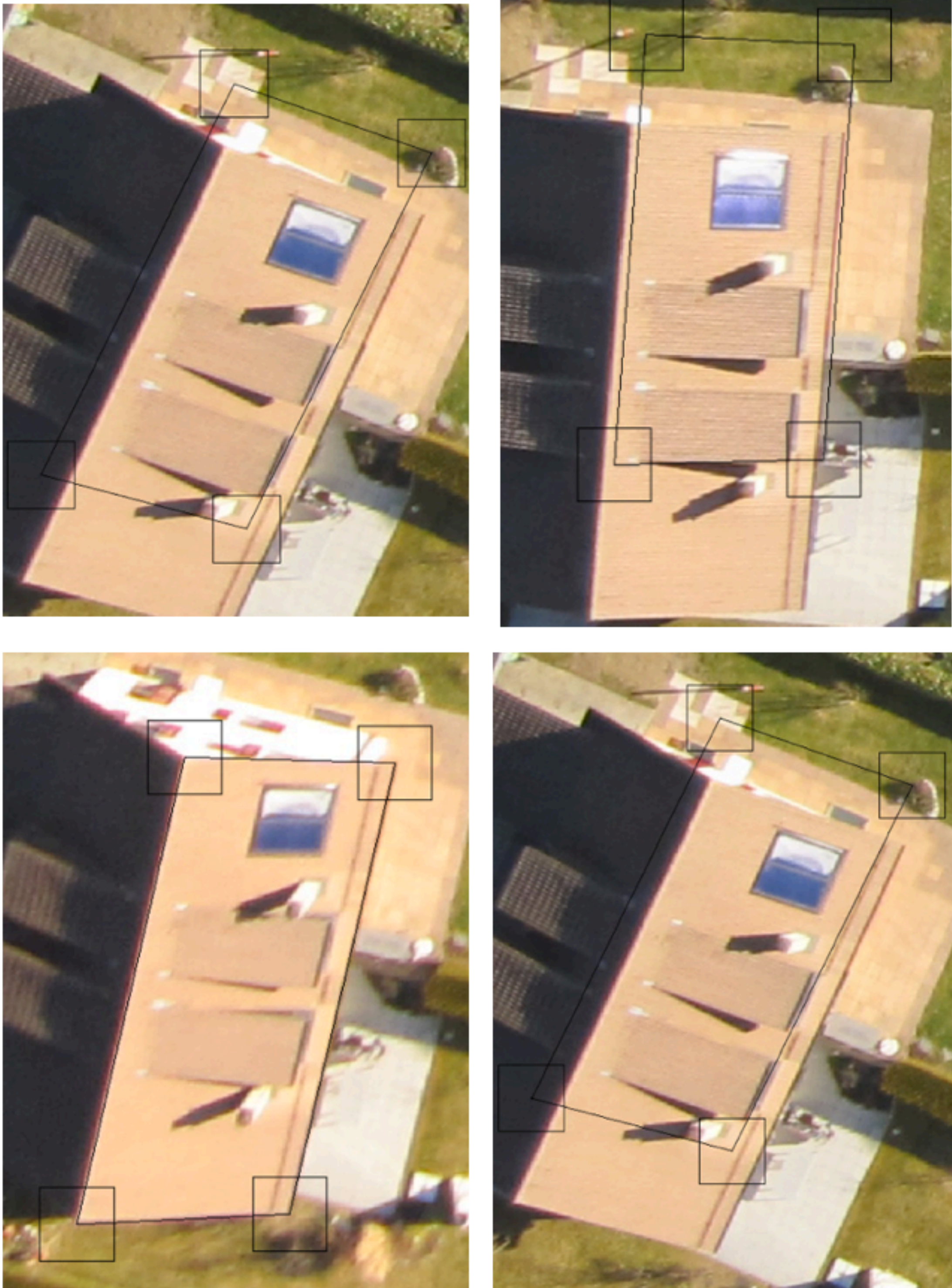


Ilustración 26 Polígonos calculados automáticamente por el método PCA

Una vez que se escojan dos imágenes y se fijen los planos a ellas, el cálculo dará el mismo resultado que el obtenido en el caso 1.

4.2. Diferencia visual en el espacio tridimensional entre los planos calculados por los distintos algortimos

Para el estudio de este caso partimos de un punto en la nube de puntos que se encuentra en el tejado de una casa. Por lo tanto el plano PCA como reasultado a partir del conjunto de datos, será un plano que tenga la inclinación de ese tejado.

Por tanto si se intenta representar cualquier techo de otro edificio usando ese plano dará como resultado una inclinación y unas dimensiones equivocadas, por regla general (como se ve en la Ilustración 27).

Para el caso de la triangulación mediante imágenes, el plano que se obtiene como solución es mucho más fiable aunque sólo sea a partir de dos imágenes, sin embargo tendrá el inconveniente de necesitar que el usuario identifique al menos un polígono en al menos dos fotografías.

En el siguiente ejemplo se puede apreciar esta diferencia:

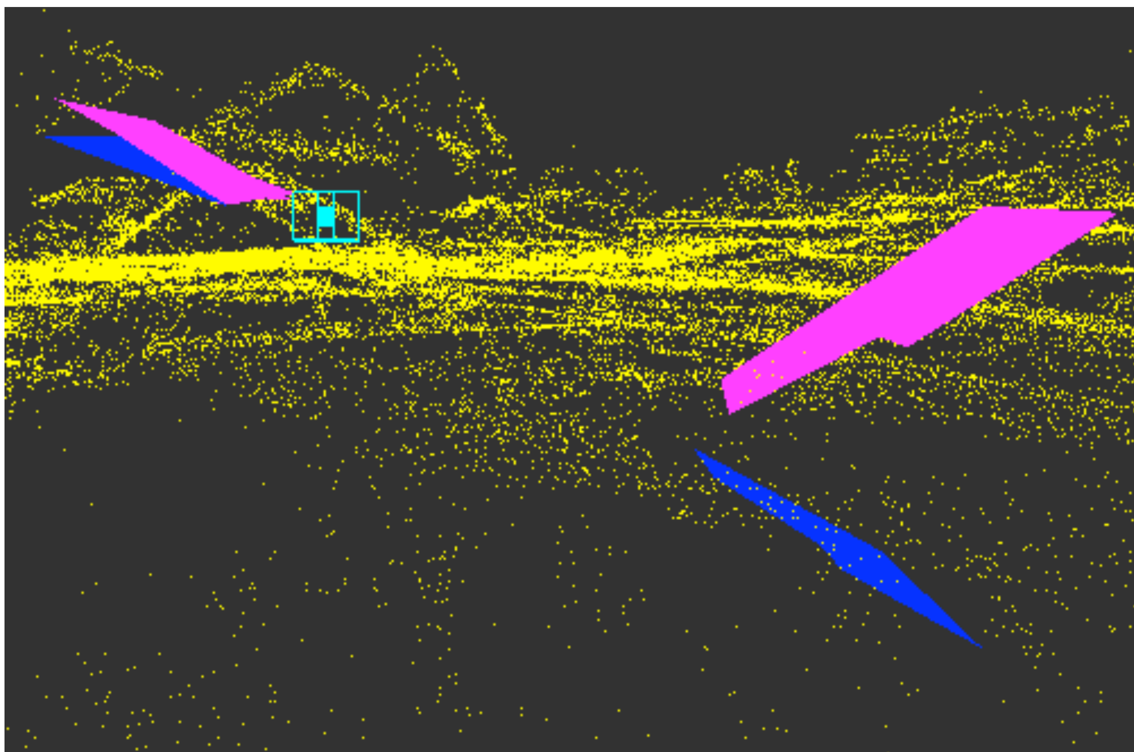


Ilustración 27 Comparación entre métodos de cálculo de distintos planos. Planos azules por el método PCA y planos rosas mediante triangulación en 2 imágenes.

Los planos azules coinciden con la inclinación del plano mediante el método PCA. Por el contrario los planos rosas se han calculado por triangulación, por eso también se percibe que pertenecen a planos distinto. Se puede comprobar que en la parte izquierda no hay una gran diferencia entre el plano correcto y el incorrecto y eso es porque la inclinación es parecida al tejado donde está el punto seleccionado, y el sitio es cercano a dicho punto. Sin embargo en el segundo caso, a la derecha, como el tejado tiene inclinación contraria, se percibe claramente el error entre uno y otro. El

plano azul sigue la misma inclinación mientras el rosa se ha generado haciendo nuevos cálculos.

4.3. Ejemplo de reconstrucción de un edificio

1. Se elige el punto en el plano 3D.

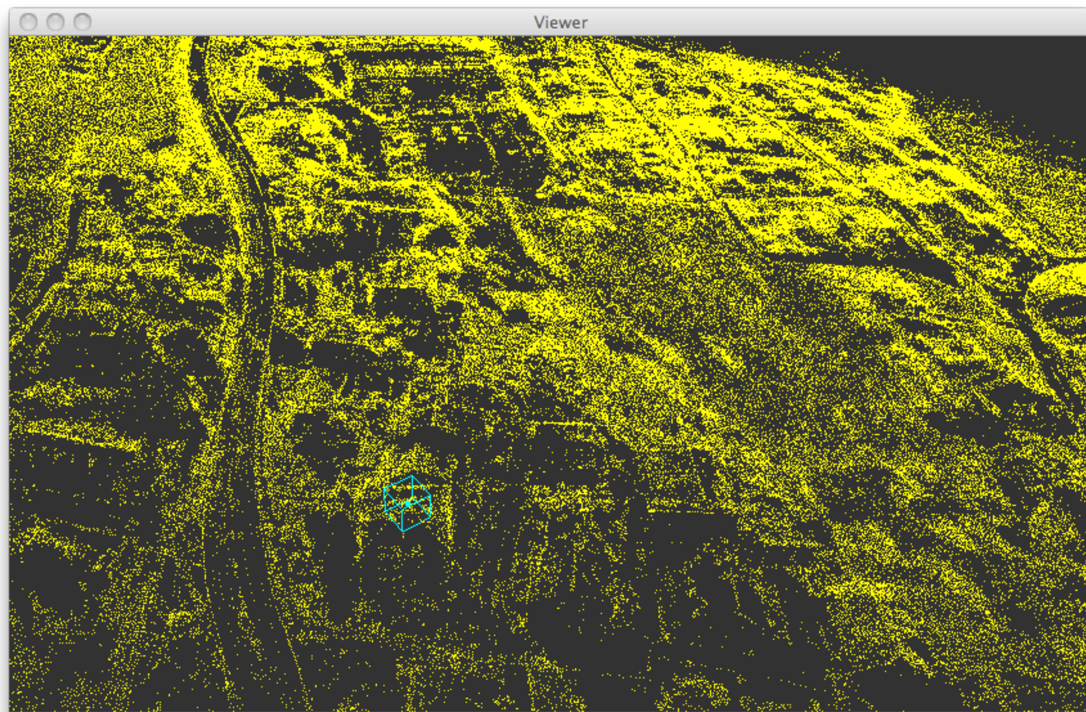


Ilustración 28 Ventana principal con un punto elegido

2. Se seleccionan los planos pertenecientes al edificio

Una vez en la ventana de imágenes seguimos los pasos necesarios para ir seleccionando los distintos planos de nuestro edificio. En cada plano hay que fijar los puntos en al menos dos imágenes para ir obteniendo los planos definitivos.

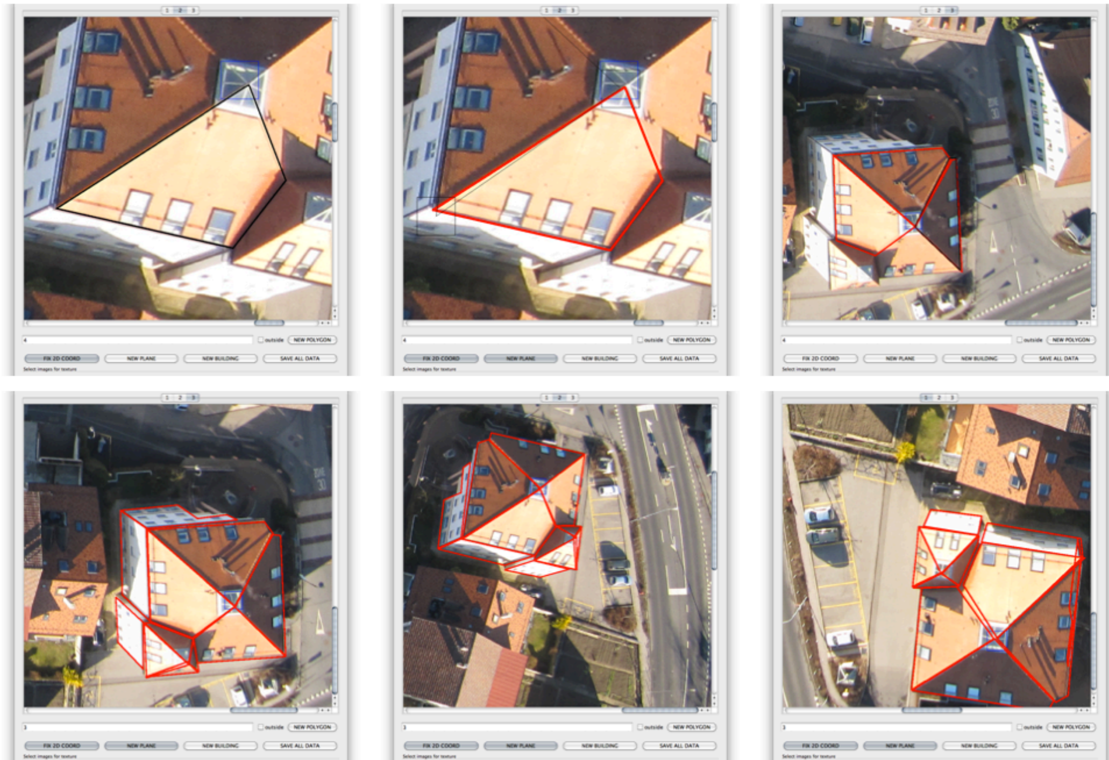


Ilustración 29 Varias capturas en el proceso de elección de planos de un edificio

3. Se guarda la información

Se hace click en el botón “SAVE ALL DATA”. Para guardar toda la información en un fichero de texto.

4. Se comprueban los resultados

En el espacio tridimensional se puede comprobar el resultado obtenido de los datos introducidos en las imágenes:

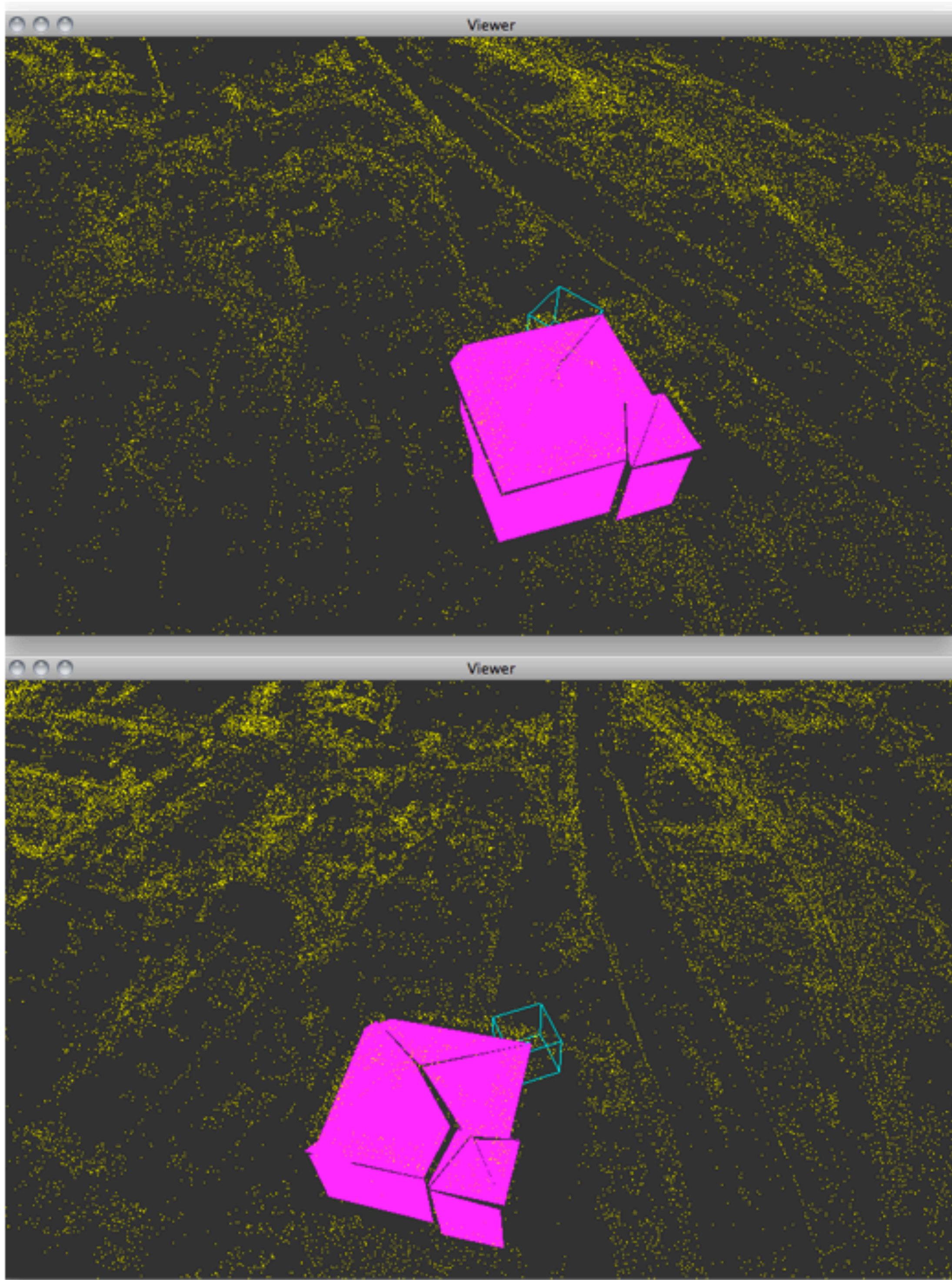
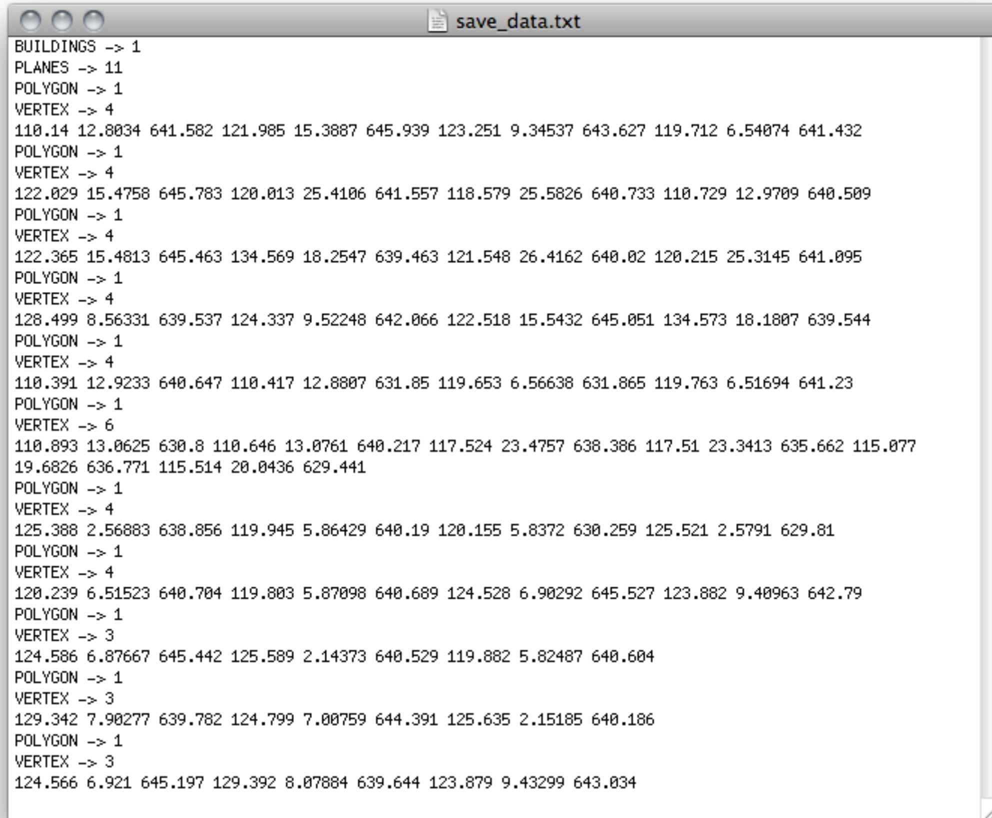


Ilustración 30 Representación tridimensional de una casa

El fichero de texto donde se guarda la información será el siguiente:



save_data.txt

```
BUILDINGS -> 1
PLANES -> 11
POLYGON -> 1
VERTEX -> 4
110.14 12.8034 641.582 121.985 15.3887 645.939 123.251 9.34537 643.627 119.712 6.54074 641.432
POLYGON -> 1
VERTEX -> 4
122.029 15.4758 645.783 120.013 25.4106 641.557 118.579 25.5826 640.733 110.729 12.9709 640.509
POLYGON -> 1
VERTEX -> 4
122.365 15.4813 645.463 134.569 18.2547 639.463 121.548 26.4162 640.02 120.215 25.3145 641.095
POLYGON -> 1
VERTEX -> 4
128.499 8.56331 639.537 124.337 9.52248 642.066 122.518 15.5432 645.051 134.573 18.1807 639.544
POLYGON -> 1
VERTEX -> 4
110.391 12.9233 640.647 110.417 12.8807 631.85 119.653 6.56638 631.865 119.763 6.51694 641.23
POLYGON -> 1
VERTEX -> 6
110.893 13.0625 630.8 110.646 13.0761 640.217 117.524 23.4757 638.386 117.51 23.3413 635.662 115.077
19.6826 636.771 115.514 20.0436 629.441
POLYGON -> 1
VERTEX -> 4
125.388 2.56883 638.856 119.945 5.86429 640.19 120.155 5.8372 630.259 125.521 2.5791 629.81
POLYGON -> 1
VERTEX -> 4
120.239 6.51523 640.704 119.803 5.87098 640.689 124.528 6.90292 645.527 123.882 9.40963 642.79
POLYGON -> 1
VERTEX -> 3
124.586 6.87667 645.442 125.589 2.14373 640.529 119.882 5.82487 640.604
POLYGON -> 1
VERTEX -> 3
129.342 7.90277 639.782 124.799 7.00759 644.391 125.635 2.15185 640.186
POLYGON -> 1
VERTEX -> 3
124.566 6.921 645.197 129.392 8.07884 639.644 123.879 9.43299 643.034
```

Ilustración 31 Fichero de texto de un edificio formado por 11 planos

5. Conclusiones

En el proyecto se ha trabajado por hacer una interfaz útil al usuario. Se ha tenido en cuenta sobre todo que se pueda observar de manera clara las imágenes y se navegue fácilmente de una a otra para poder seleccionar los edificios a reconstruir.

En lo que se refiere a la teoría de imágenes y proyección de puntos en el espacio, se han implementado en código C++ métodos matemáticos ya estudiados para realizar las representaciones. En la mayoría de los casos se obtienen los resultados esperados, excepto para algunas excepciones, con dos imágenes es suficiente para reconstruir partes de un edificio, y en las que la aproximación no es del todo buena, con una tercera imagen se resuelve el problema.

Por tanto se ha obtenido un buen resultado final, donde el usuario ya puede interactuar con el sistema y reconstruir edificios u obtener información acerca de las coordenadas de los polígonos que lo forman.

Como mejoras se proponen varios temas. Uno de ellos es el de seguir mejorando la interfaz de usuario, poder hacer pruebas reales de cómo los usuarios interactúan con dicha interfaz para así habilitar o deshabilitar ciertas opciones y poder introducir mejoras que ayudaran al usuario. Otra posible mejora es permitir al usuario guardar los datos en un archivo que él mismo defina y también poder elegir el destino donde guardarlo.

En líneas futuras respecto a las reconstrucciones de edificios se podría tratar de mejorar en los siguientes aspectos:

- Mostrar los planos en el espacio tridimensional con las texturas escogidas en la interfaz.
- Escribir un script para transformar el archivo guardado y poder representar los edificios con algún software dedicado a ello, por ejemplo google sketch up.
- Conseguir que los planos de un mismo edificio corten unos con otros de forma suave para conseguir una representación más real.

6. Bibliografía

[1] Christop Strecha, Timo Pylvanainen, Pascual Fua. Dynamic and scalable large scale image reconstruction [en línea]. 2010. Disponible en internet: <http://infoscience.epfl.ch/record/148247/files/cvpr_2010.pdf>

[2] Richard Hartley and Andrew Zisserman, Camera projections. View Geometry in Computer Vision. United Kingdom: Cambridge, 2003. p.6-10.

[3] Lindsay I Smith, Principal Component Analysis. A tutorial on Principal Component Analysis [en línea]. 26 de febrero de 2002. Disponible en internet: <http://www.cs.otago.ac.nz/cosc453/student_tutorials/principal_components.pdf>

Como parte de implementación del código se han usado los siguiente libros:

- Dave Shreiner et all., OpenGL programming guide. United States of America. 2010.
- Qt Help → Qt Reference Documentaton. Qt Developer guide: tutorials and examples. Qt API.

Como soporte en cuanto a teoría sobre proyección de imágenes:

- Richard Hartley and Andrew Zisserman. View Geometry in Computer Vision. United Kingdom: Cambridge, 2003. p.6-10