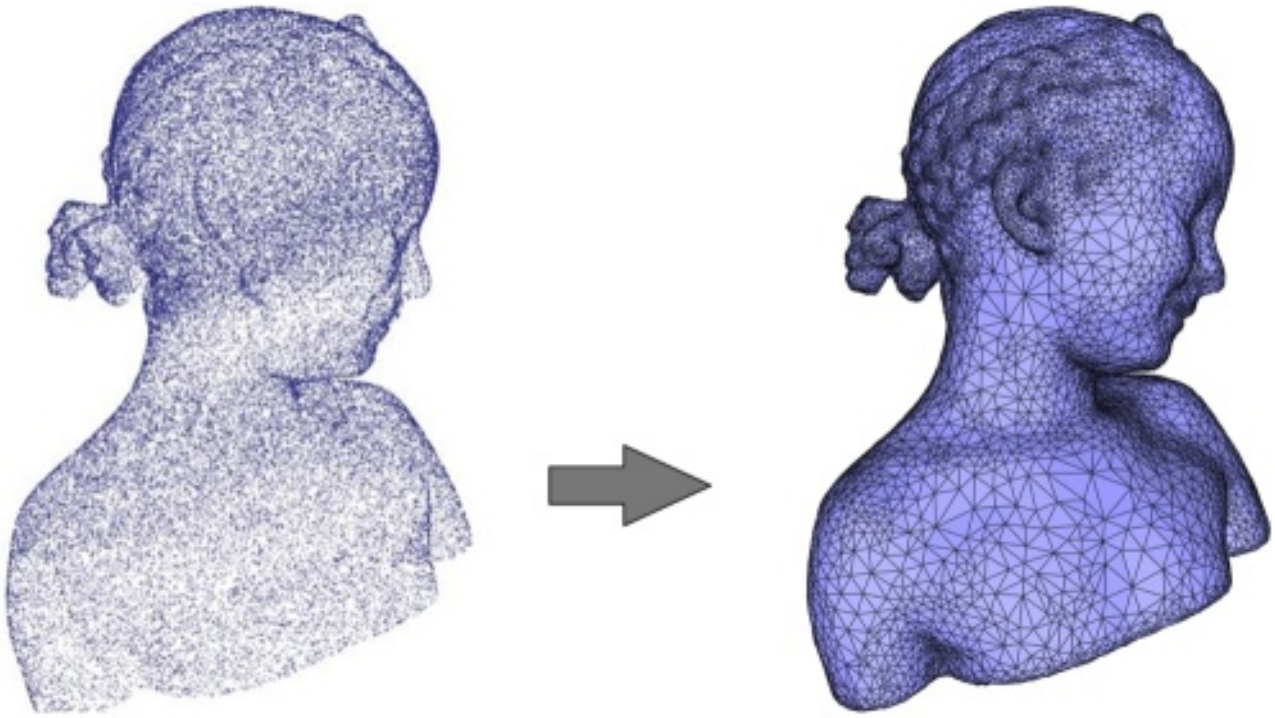
Cálculo del egomotion de una cámara RGBD



Javier Galván Martínez
48564495H

Introducción

En esta segunda práctica, se pretende realizar la construcción de un mapa del entorno a partir de los datos capturados por una cámara RGBD (en nuestro caso, usaremos una Kinect). Para ello usaremos ROS y la librería de procesamiento 3D Point Cloud Library (PCL).

Imaginemos que tenemos un robot que se está moviendo por el entorno. Cada cierto tiempo, el robot realiza una toma de datos de la cámara. Esta cámara nos proporciona una imagen RGB y una nube de puntos 3D. El objetivo de esta práctica es, usando los datos en cada posición del robot, calcular el movimiento (transformación 3D) que ha realizado el robot (le llamaremos egomotion) para cada par de imágenes consecutivas. Una vez calculado este movimiento, podemos transformar todos los datos capturados en un sistema de coordenadas común y así podemos construir un mapa. Como los datos 3D capturados son numerosos, habrá que usar alguna técnica para reducirlos.

Debemos encontrar las partes comunes que comparten las nubes generadas por las imágenes de entrada. La idea consiste en encontrar un conjunto de puntos característicos compartidos por cada par de nubes. Una vez encontrados, estos puntos se utilizan para calcular la transformación. Existen multitud de métodos para extraer esta información. Inicialmente utilizaremos varios e iremos descartando según su rendimiento. Utilizaremos *SIFT, HARRIS, ISS, SUSAN* para extraer keypoints, *PFH, FPFH, SHOT* para descriptores y comentaremos la experimentación con *NARF*.

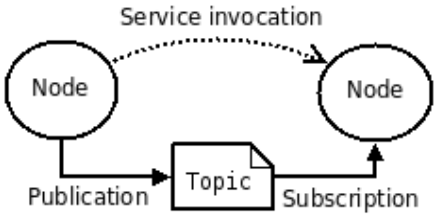
El proceso a seguir es el siguiente:

Entrada: Conjunto de datos X.

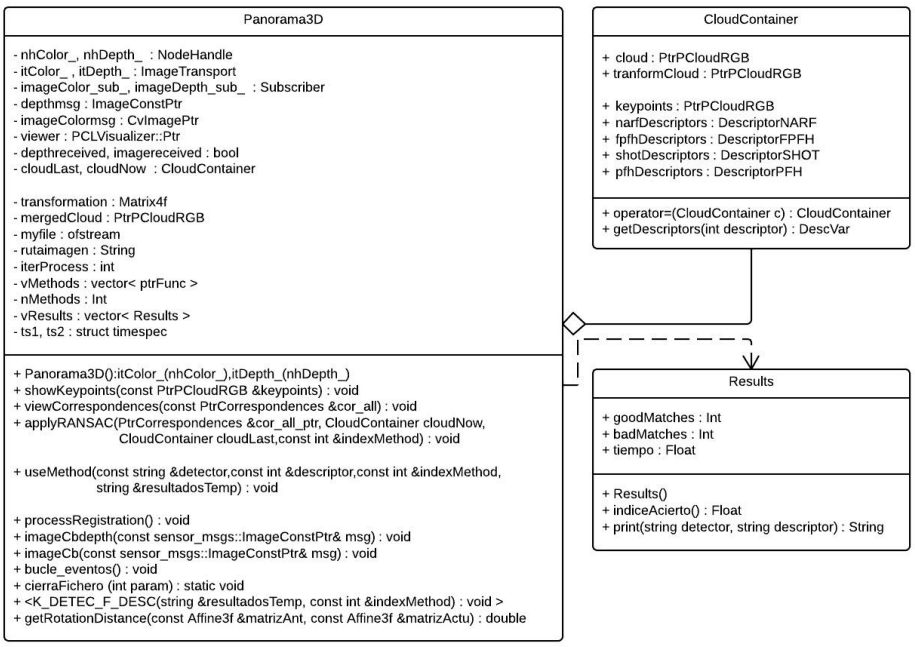
1. Transformación total Tt = I
2. Mapa 3D M
3. para i = 1 hasta t hacer:
    a. Extraer características de los datos xi .
    b. Extraer características de los datos xi+1 .
    c. Encontrar emparejamientos entre las características
    d. Obtener la mejor transformación Ti que explican los emparejamientos.
    e. Obtener la transformación total Tt = Tt * Ti .
    f. Aplicar al conjunto de datos Ci+1 la transformación Tt para colocarlos en el sistema de coordenadas de C1 . Acumular datos transformados en M .
4. fin para
5. devolver M

Estructura

Para la práctica volveremos a utilizar Robot Operating System (ROS), que sigue el sistema de nodos que vimos en la práctica anterior:



En esta ocasión para el *Subscriber* tenemos una división en tres clases, **Panorama3D** , que escucha los topics correspondientes para los datos de entrada; **CloudContainer** que nos será de utilidad para manejar las nubes y la información extraída de las mismas y por último **Results**, una "clase utilidades" para manejar datos de análisis sobre el sistema. Están divididas en varios ficheros donde cabe resaltar ***utilities3D.hpp***, nos servirá de librería, donde encontraremos funciones auxiliares propias de la extracción de características, etc. A continuación describiremos las partes más notorias de estas estructuras.



Utilities.hpp

```
#define DescriptorByType boost::get<typename
pcl::PointCloud<Type>::Ptr>(cloudNow.getDescriptors(descriptor))

typedef pcl::PointCloud<pcl::PointXYZ> PCloud;
typedef pcl::PointCloud<pcl::PointXYZ>::Ptr PtrPCloud;
typedef pcl::PointCloud <pcl::PointXYZRGB> PCloudRGB;
typedef pcl::PointCloud<pcl::PointXYZRGB>::Ptr PtrPCloudRGB;
typedef pcl::PointCloud<pcl::Normal>::Ptr PtrNormal;
typedef pcl::PointCloud<pcl::SHOT352>::Ptr DescriptorSHOT;
typedef pcl::PointCloud<pcl::FPFHSignature33>::Ptr DescriptorFPFH;
typedef pcl::PointCloud<pcl::Narf36>::Ptr DescriptorNARF;
typedef pcl::PointCloud<pcl::PFHSignature125>::Ptr DescriptorPFH;
typedef boost::variant<DescriptorFPFH, DescriptorSHOT, DescriptorNARF, DescriptorPFH> DescVar;
typedef boost::shared_ptr<Correspondences> PtrCorrespondences;

enum PCL_DESCRIPTORS{ F_FPFH = 0, F_PFH = 1,   F_SHOT =2, F_NARF =3 };
+ getDescriptors(int descriptor) : DescVar

+ removeNaN(PtrPCloudRGB &cloud) : void
+ getCloudColorDepth(const Mat imageColor, const float* depthImage) : PtrPCloudRGB
+ computeCloudResolution (const PtrPCloudRGB &cloud) : double
+ getNormal(PtrPCloudRGB keypoints, PtrNormal &cloud_normals) : void

+ applyHarris(CloudContainer &cloudCont) : void
+ applySIFT(CloudContainer &cloudCont) : void
+ applyISS(CloudContainer &cloudCont) : void
+ applyFPFH(CloudContainer &cloudCont) : void
+ applySHOT(CloudContainer &cloudCont) : void
+ applyPFH(CloudContainer &cloudCont) : void

+ getRangeImage(RangeImage& range_image, const PtrPCloudRGB &cloud) : void
+ applyNARF_Key(CloudContainer &cloudCont, RangeImage& range_image) : pcl::PointCloud<int>
+ applyNARF_DESC(pcl::PointCloud<int> keypoint_indices,const RangeImage& range_image,
CloudContainer &cloudCont) : void
```

Librería Utilities3D

Para esta práctica ha sido necesario desarrollar una librería que contuviese definiciones de tipos de datos con signatura compleja y funciones auxiliares requeridas por las clases. Estas funciones son en su gran mayoría las encargadas de extraer keypoints y descriptores dada una nube de puntos. También encontramos el enum que define los métodos de extracción de características que nos servirá para la función *getDescriptors()* la cual devuelve un tipo variable ***DescVar*** que será tipo de descriptor concreto.

Clase CloudContainer

```
class CloudContainer{

        public:

        PtrPCloudRGB cloud;
        PtrPCloudRGB keypoints;
        //Caracteristicas segun el metodo
        DescriptorNARF narfDescriptors;
        DescriptorFPFH fpfhDescriptors;
        DescriptorSHOT shotDescriptors;
        DescriptorPFH pfhDescriptors;

        CloudContainer & operator=(const CloudContainer &c){
                if(&c==this)
                        return *this;
                else{

                        keypoints = c.keypoints;
                        narfDescriptors = c.narfDescriptors;
                        fpfhDescriptors = c.fpfhDescriptors;
                        cloud = c.cloud;
                }
                return *this;
        }
        DescVar getDescriptors(int descriptor){

                switch (descriptor){
                        case F_FPFH:
                                return fpfhDescriptors;
                        case F_SHOT:
                                return shotDescriptors;
                        case F_NARF:
                                return narfDescriptors;
                        case F_PFH:
                                return pfhDescriptors;
                }
        }
};
```

Como vemos esta clase cuenta con un tipo *PtrPCloudRGB* que contiene la nube de puntos, otro para keypoints y estructuras con tipos de datos distintos para guardar los descriptores según el método usado.

Clase Results

```
class Results{

    public:
```

```
    int goodMatches;
    int badMatches;
    float tiempo;

    Results(){
        goodMatches = 0;
        badMatches = 0;
        tiempo = 0;
    }
    float indiceAcierto(){
        float total  = goodMatches+badMatches;
        if(total!=0)
               return (goodMatches*100)/(total);
        return 0;
    }


    //Imprime el resultado del metodo detector y descriptor usados
    string print(string detector, string descriptor){
        stringstream ss;
        float porcentaje = floorf(indiceAcierto() * 100 + 0.5) / 100;
        ss << "---------------"<<endl;
        ss <<"| " << detector << " + " << descriptor << " | ";
        ss << "Rating: " << porcentaje/tiempo << endl;
        ss << "---------------"<<endl;
        ss <<"| " << "GoodMatches: "<< goodMatches << " | ";
        ss << "BadMatches: "<< badMatches << " | ";

        ss << "Accuracy: " << porcentaje <<"%" << " | ";
        ss << "Time: "<< tiempo << " s"<<" | "<< endl<<endl;

        return ss.str();
    }

};
```

Se trata de una clase almacén de resultados, veremos que una vez tratados unos métodos y otros hacemos uso de sus utilidades para guardar tiempos de ejecución, buenas y malas correspondencias obtenidas con RANSAC, un porcentaje en base a esos aciertos y un "rating" que relaciona aciertos-tiempo. También construye la string que se escribirá en un fichero de resultados.

Clase Panorama3D

La clase principal encargada de escuchar los topics correspondientes y tratar las imágenes de color y profundidad, utilizar los métodos de extracción de características, matching, filtrado filtrado de correspondencias, acumular la transformada, etc.

```
typedef void (Panorama3D::*ptrFunc)(string &,const int &);
vector<ptrFunc> vMethods;
```

Debemos definir primero un tipo de dato, puntero a función, que será la que aplique cada combinación de métodos, incluyendo estas funciones en un vector podremos recorrerlo y ejecutarlas a través de un bucle y así automatizar la experimentación, como veremos más adelante.

```
Panorama3D():itColor_(nhColor_),itDepth_(nhDepth_){

        depthreceived = false;
        imagereceived = false;
        iterProcess = 0;

        viewer= pcl::visualization::PCLVisualizer::Ptr(new
        pcl::visualization::PCLVisualizer ("3D Viewer"));
        viewer->setBackgroundColor (0, 0, 0);

        viewer->initCameraParameters ();
        viewer->setSize (800, 600);

        imageColor_sub_=itColor_.subscribe("/camera/rgb/image_color", 1,
        &Panorama3D::imageCb, this);
                        imageDepth_sub_ = itDepth_.subscribe("/camera/depth/image", 1,
        &Panorama3D::imageCbdepth, this);

        //La señal de corte de ejecucion para cerrar el fichero de resultados
        void (*handler)(int);
        handler = signal(SIGINT, cierraFichero);

        FICHERO.open ("/home/javi/resultados.txt");
        RESULTADOSFIN="";

        //Anyadimos las combinaciones de metodos a probar

        vMethods.push_back(&Panorama3D::K_ISS_F_FPFH);
        vMethods.push_back(&Panorama3D::K_HARRYS_F_FPFH);
        //.... otros metodos a probar

        nMethods = vMethods.size();
        vResults.resize(nMethods);

}
```

En el constructor de la clase nos suscribirnos a los topics donde se publican las imágenes de color y de profundidad. Cuando algo se publique se llamará automáticamente a la función *imageCb* (imageColor CallBack) que se ocupará de manejar la imagen de color entrante y del mismo modo *imageDepthCb* con la imagen de profundidad.

Añadimos los punteros a métodos (detector, descriptor) al vector que se ocupa de almacenarlos *vMethods* y guardamos el número de métodos, lo utilizaremos en el bucle de experimentación y definirá el tamaño del vector de resultados, donde cada posición se corresponde con una combinación de métodos.

Tendremos un manejador de **señales**, cuando el programa se pare utilizando el comando **Ctrl+C** los resultados obtenidos hasta entonces (almacenados mediante la clase Results e impresos sobre la variable *RESULTADOSFIN*) se volcarán en el fichero "resultados.txt". Establecemos valores iniciales para indicar el número de iteraciones realizadas, si se tratan de las primeras imágenes entrantes para desbloquear los *callbacks,* el visor de 3D, ruta de capturas de pantalla, etc.

A continuación debemos procesar el tipo de dato de la imagen que recibimos, hasta que no tengamos al menos tanto una imagen de color como una de profundidad, no se desbloquea la llamada a *procressRegistration()*

```
//Procesamos ImagenProfundidad y activamos su recepcion
void imageCbdepth(const sensor_msgs::ImageConstPtr& msg){

	depthmsg = msg;
	depthreceived=true;
	if(imagereceived && depthreceived)
		processRegistration();

}
//Procesamos ImagenColor y activamos su recepcion
void imageCb(const sensor_msgs::ImageConstPtr& msg){
	try{
		imageColormsg = cv_bridge::toCvCopy(msg,
		sensor_msgs::image_encodings::BGR8);
	}
		catch (cv_bridge::Exception& e){
		ROS_ERROR("cv_bridge exception: %s", e.what());
		return;
	}

	imagereceived=true;
	if(imagereceived && depthreceived)
		processRegistration();
}
```

A continuación comentaremos la función *processRegistration*, se explicará de forma fragmentada según el flujo de llamadas durante la ejecución:

```
void processRegistration(){

	//Limpiamos para poder visualizar los Keypoints
	viewer->removeAllPointClouds();

	const float* depthImageFloat = reinterpret_cast<const
	float*>(&depthmsg->data[0]);

	cloudNow.cloud = getCloudColorDepth(imageColormsg->image, depthImageFloat);
```

```
        removeNaN(cloudNow.cloud);

        //------ VISUALIZACION DE LA NUBE --------//

        visualization::PointCloudColorHandlerRGBField<pcl::PointXYZRGB>
        rgb(cloudNow.cloud);

        //Actualiza la nube y si no existe la creo.
        if (!viewer->updatePointCloud (cloudNow.cloud,rgb, "cloud")){
                viewer->addPointCloud(cloudNow.cloud,rgb,"cloud");
        }
```

Inicialmente transforma extraemos una nube de puntos de la imagen de color y la de profundidad y la almacenamos en el *CloudContainer.cloud* y quitamos los NaN de la nube.

```
        //----- APLICAR METODOS -----//

        string resultadosTemp = "";

        for(short i=0; i<nMethods;++i){
                clock_gettime( CLOCK_REALTIME, &ts1 );
                (this->*vMethods[i])(resultadosTemp, i);
        }

        //----- VOLCAR RESULTADOS -----//
        RESULTADOSFIN = resultadosTemp;

        cloudLast = cloudNow;
        ++iterProcess;
}
```

Para evaluar las combinaciones de métodos de forma automática declaramos una string de resultados temporales, ya que no escribiremos los resultados a menos que acabe el ciclo completo de la iteración, extracción de características, Ransac, etc.

Comenzamos un **bucle** que se ejecutará **tantas veces** como número de **combinaciones de métodos** tengamos. Marcamos el momento de inicio de prueba del par de métodos y se llama a la función correspondiente de la iteración. A continuación seguiremos la traza utilizando como ejemplo ISS junto con FPFH, llamaría pues a la función del vector *K_ISS_F_FPFH* :

```
void K_ISS_F_FPFH(string &resultadosTemp, const int &indexMethod){
    applyISS(cloudNow);
    removeNaN(cloudNow.keypoints);
```

```
    applyFPFH(cloudNow);
    useMethod("K_ISS",F_FPFH, indexMethod, resultadosTemp);
}
```

Estos prototipos de método utilizan el extractor de keypoints y el de descriptores definido en su nombre, en este caso ISS para keypoints, se limpian los NaN de la nube de kaypoints y se utiliza FPFH para descriptores. Con estos datos almacenados en la variable de contenedor actual, *cloudNow*, llamamos a *useMethod(detector, descriptor, indice del metodo en el vector, resultados)* con la combinación de métodos, su índice y el acumulador de resultados temporales.

Si se tratase de la primera iteración simplemente guardaríamos la información del contenedor en *cloudLast* y pondríamos el indicador de la iteración se incrementaría.

```
void useMethod(const string &detector,const int &descriptor,const int &indexMethod,
string &resultadosTemp){

        if(iterProcess!=0){

                rutaimagen = "/home/javi/ImagenesPanorama/";
                string sdescriptor="";
                PtrCorrespondences correspondences;

                //------ APLICAR MATCHING -----//
                switch (descriptor){
                   case F_FPFH:
                       correspondences = matching<pcl::FPFHSignature33, F_FPFH>();
                       sdescriptor="F_FPFH";
                       break;
                   case F_NARF:
                       correspondences = matching<pcl::Narf36, F_NARF>();
                       sdescriptor = "F_NARF";
                       break;
                    case F_SHOT:
                       correspondences = matching<pcl::SHOT352, F_SHOT>();
                       sdescriptor = "F_SHOT";
                       break;
                     case F_PFH:
                       correspondences = matching<pcl::PFHSignature125, F_PFH>();
                       sdescriptor = "F_PFH";
                }

                //----- APLICAR RANSAC ----//
                applyRANSAC(correspondences, cloudNow, cloudLast, indexMethod);
                viewCorrespondences(correspondences);

                stringstream ssNumber;
                ssNumber << iterProcess;
                rutaimagen+= ssNumber.str();
                rutaimagen+=detector+"+"+sdescriptor;
                //viewer->saveScreenshot (rutaimagen);

               //------ RESULTADOS -----//
              clock_gettime( CLOCK_REALTIME, &ts2 );
```

```
            vResults[indexMethod].tiempo +=(float) ( 1.0*(1.0*ts2.tv_nsec -
            ts1.tv_nsec*1.0)*1e-9+ 1.0*ts2.tv_sec - 1.0*ts1.tv_sec );
            resultadosTemp += vResults[indexMethod].print(detector, sdescriptor);
        }
    }
```

No siendo la primera iteración, suponiendo que ya tenemos una nube anterior en memoria, procedemos a apartado de *Matching.* De esta manera obtenemos las correspondencias entre los dos conjuntos de características de la nube anterior *cloudLast* y la actual *cloudNow*. Para la función de matching se han utilizado un **template** de C++:

```
template<typename Type, int descriptor>PtrCorrespondences matching(void)
```

De esta forma definimos una función genérica que varía según el tipo de dato que estemos usando, si es *FPFH* el tipo de dato será `pcl::FPFHSignature33` y por tanto la llamada a *getDescriptorByType* nos dará el descriptor correspondiente. Una vez obtenidas las correspondencias, se devuelven con el tipo *PtrCorrespondences,* ya compartido por todos.

Habiendo obtenido las correspondencias llamamos a *applyRANSAC()* función que aplica el algoritmo y nos da las buenas y malas (*goodMatches, badMatches)* las cuales almacenamos en los resultados del índice correspondiente a la combinación de métodos usada:

```
vResults[indexMethod].badMatches += cor_all_ptr->size()-cor_inliers->size();
vResults[indexMethod].goodMatches += cor_inliers->size();
```

Con estas correspondencias podemos llamar a la función que las pinte, mostrar el resultado por pantalla, guardar la imagen obtenida en la ruta determinada… Y finalmente medimos el tiempo, guardamos los resultados que correspondan a los métodos usados y actualizamos la nube, el número de la iteración general, etc.

ProcessRegistration() se llamará por cada lectura de imágenes que se haga, realizará el bucle con las sentencias señaladas tantas veces como métodos queramos probar y así hará hasta que recibe una señal Ctrl+C de corte de ejecución, en ese momento se volcarán los resultados sobre un fichero y se cerrará el programa.

Experimentación

Inicialmente probamos una batería de combinaciones de métodos bastante amplia para determinar cuales son los más eficientes, así realizar la transformación y ccumular los datos transformados en M para reconstruir la el mapa3D utilizando la combinación más eficiente. La experimentación se realizará sobre varios conjuntos de imágenes, *datasets*.

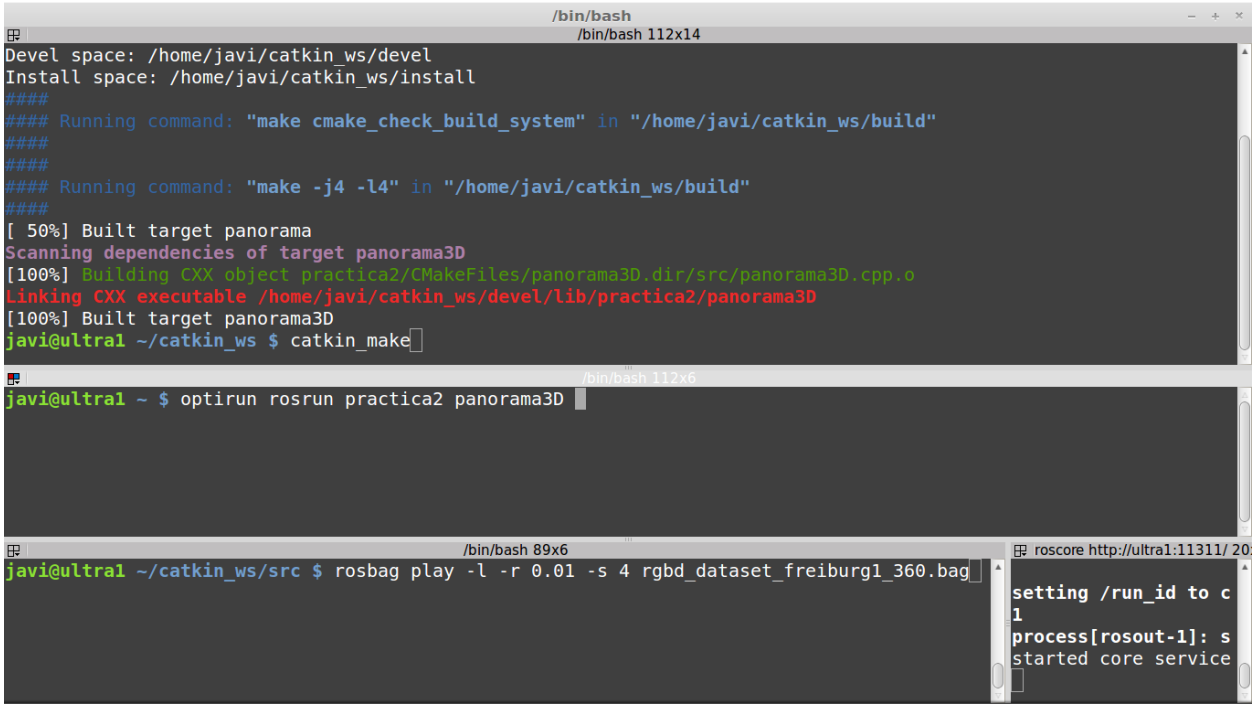
```
        vMethods.push_back(&Panorama3D::K_ISS_F_FPFH);
        vMethods.push_back(&Panorama3D::K_HARRYS_F_FPFH);
        vMethods.push_back(&Panorama3D::K_SIFT_F_FPFH);

        vMethods.push_back(&Panorama3D::K_ISS_F_SHOT);
        vMethods.push_back(&Panorama3D::K_HARRYS_F_SHOT);
        vMethods.push_back(&Panorama3D::K_SIFT_F_SHOT);

        vMethods.push_back(&Panorama3D::K_ISS_F_PFH);
        vMethods.push_back(&Panorama3D::K_SIFT_F_PFH);
        vMethods.push_back(&Panorama3D::K_HARRYS_F_PFH);

        //OPCIONAL
        vMethods.push_back(&Panorama3D::K_NARF_F_NARF);
```

Para lanzar la práctica debemos **compilar** con *catkin_make*, lanzar *roscore* para poder ejecutar las **funciones** de **ROS**, lanzar ***Panorama3D*** con el comando [1]*optirun rosrun practica2 panorama3D* y la entrada que leeremos vendrá dada por el **dataset** "rgbd_dataset_freiburg1_360.bag" para lanzar esta consecución de imágenes ejecutamos la orden

```
rosbag play -l -r 0.01 -s 4 rgbd_dataset_freiburg1_360.bag
```

Al aplicar tantos métodos por cada frame indicamos una publicación de bastante más lenta con el comando **-r 0.01** y saltamos los primeros 4 segundo con **-s 4** ya que no ocurre nada en ese periodo.

[1]*Nota: Ejecutamos el comando optirun para utilizar la gráfica Nvidia mediante los drivers no oficiales Bumblebee que dan soporte a estas GPUs sobre Linux.*

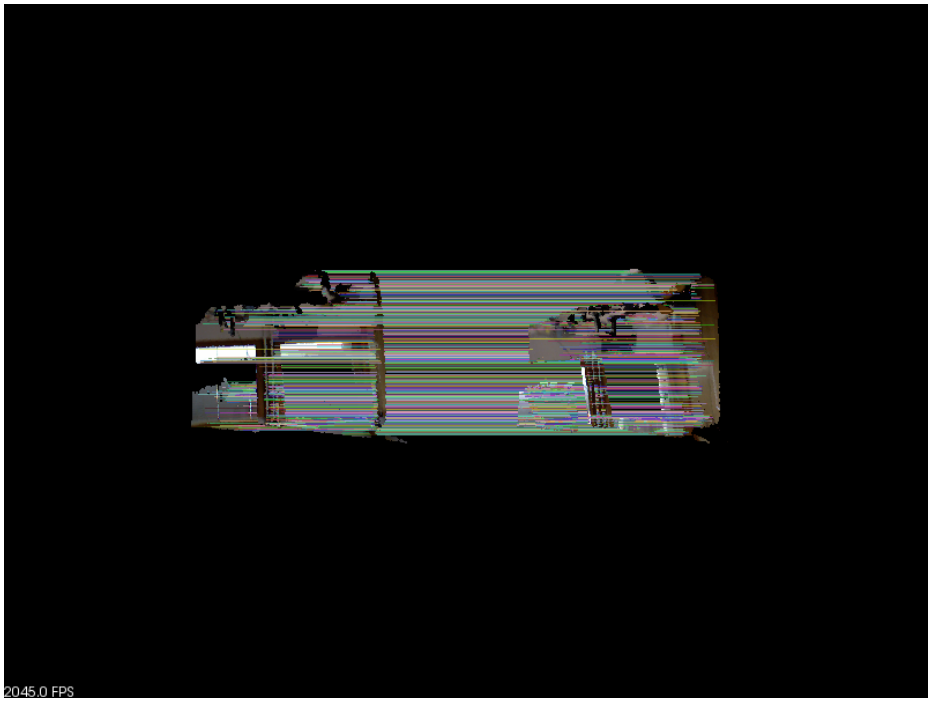


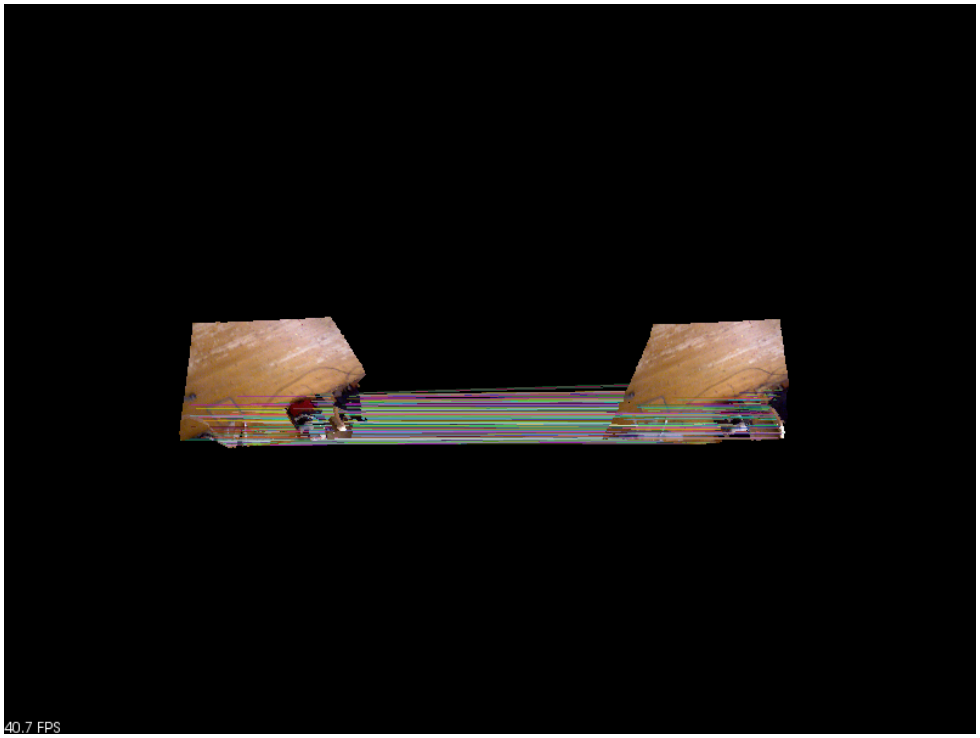
Resaltamos los métodos con mejor resultado fijándonos en el **rating**, una medida de precisión con respecto al tiempo, siendo la precisión la relación existente entre aciertos buenos y malos. No prestaremos mucha atención a los resultados arrojados por *NARF* ya que no es un verdadero método de 3D y se ha relegado a la parte optativa.

Feature Matching Time 415.431 s:

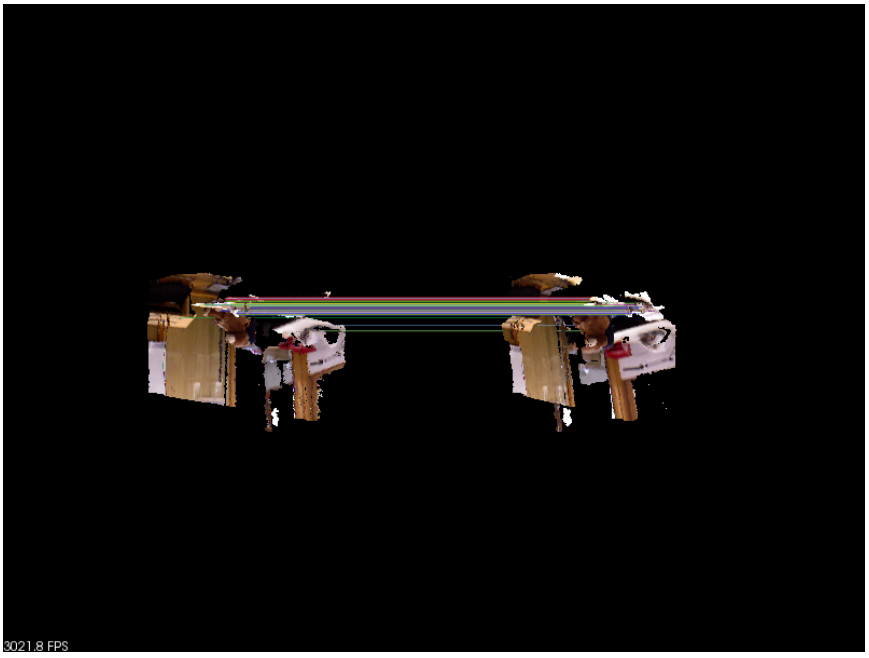
Métodos	Rating	GoodMatches	BadMatches	Accuracy	Time
ISS + FPFH	1,01700411	23274	20372	53.32%	52.4285 s
HARRYS + FPFH	2,927102132	19354	7641	71.69%	24.4918 s
SIFT + FPFH	0,5617013429	4692	16700	21.93%	39.0421 s
ISS + SHOT	0,8957321195	23308	20018	53.8%	60.0626 s
HARRYS + SHOT	2,911694695	27626	9037	75.35%	25.8784 s
SIFT + SHOT	0,4219078851	1961	9416	17.24%	40.862 s
ISS + PFH	1,027804071	12203	10185	54.51%	53.0354 s
SIFT + PFH	0,463866943	820	3605	18.53%	39.9468 s
HARRYS + PFH	2,67667747	15355	8092	65.49%	24.4669 s
NARF + NARF	31,5538	223	25	89.92%	2.84974 s

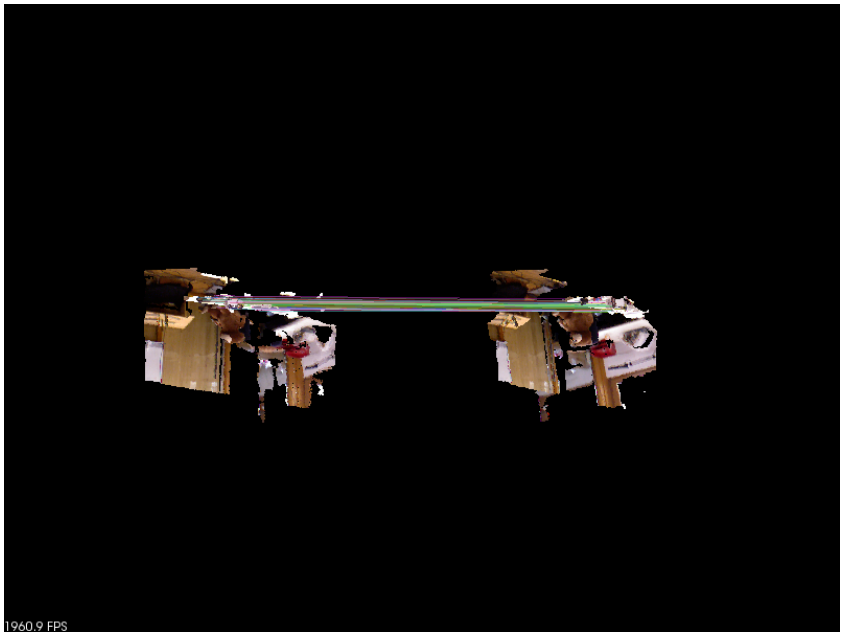
ISS + FPFH



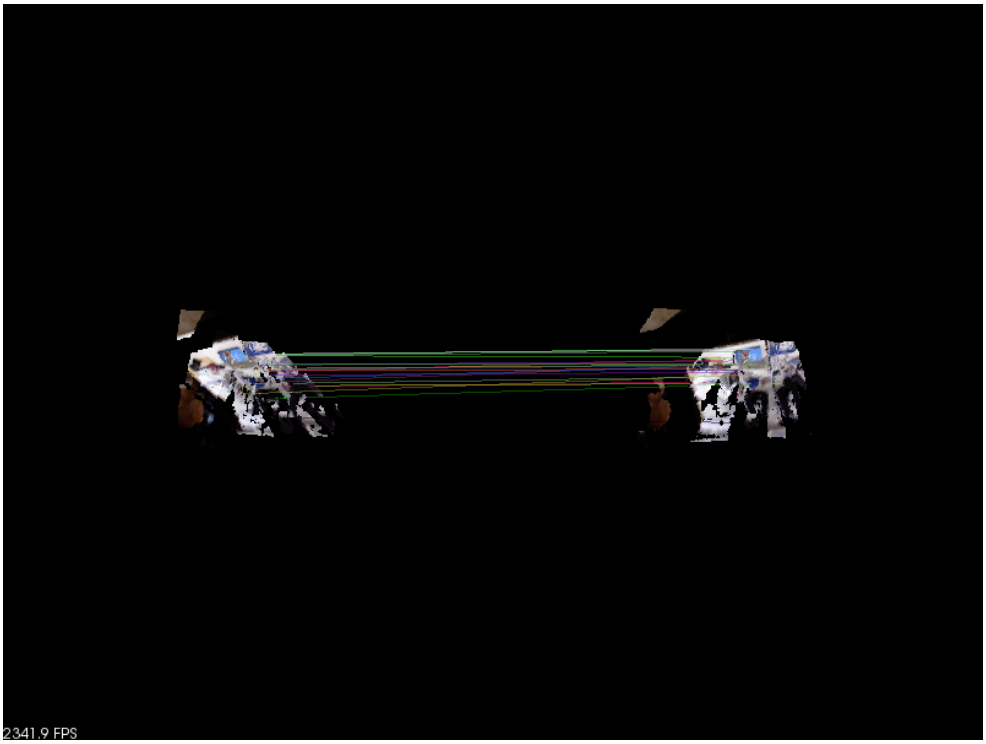


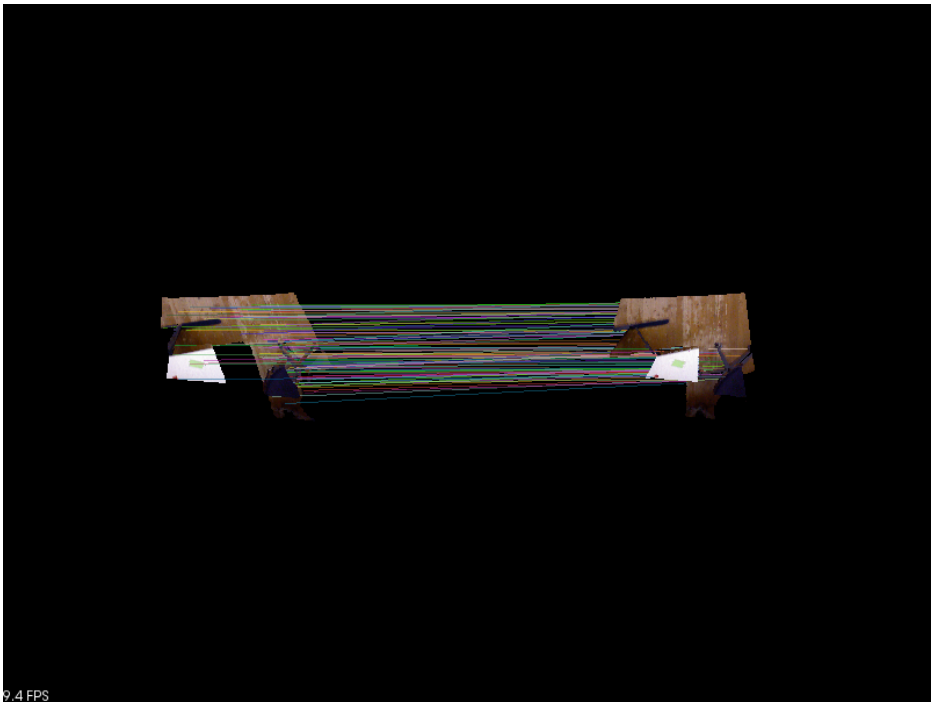
HARRYS + FPFH



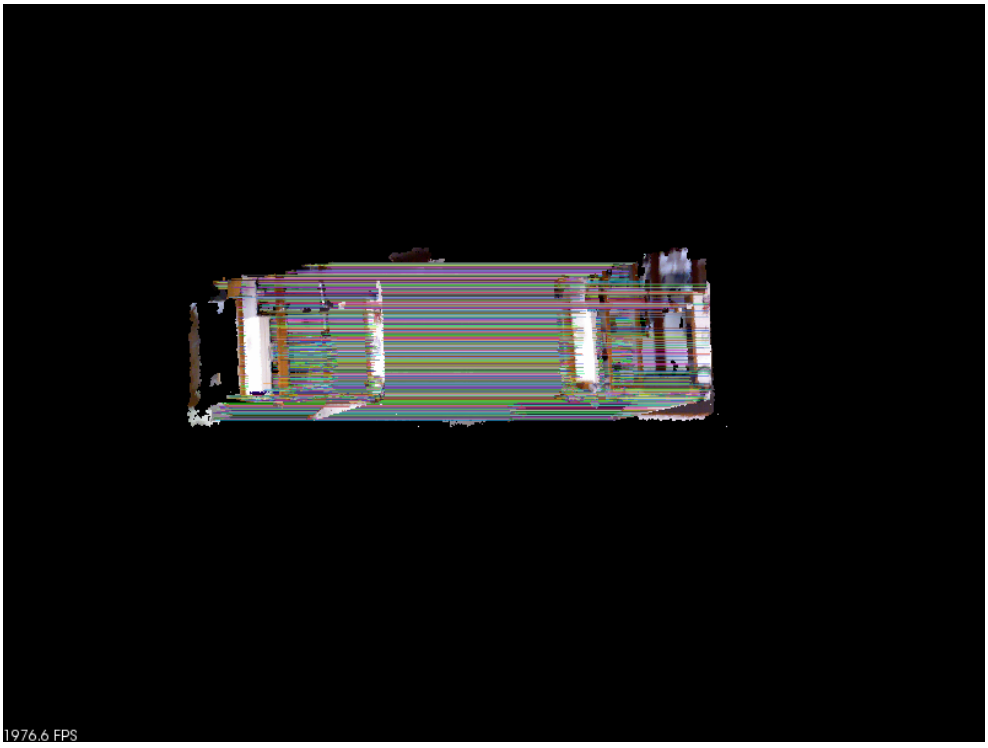


SIFT + FPFH

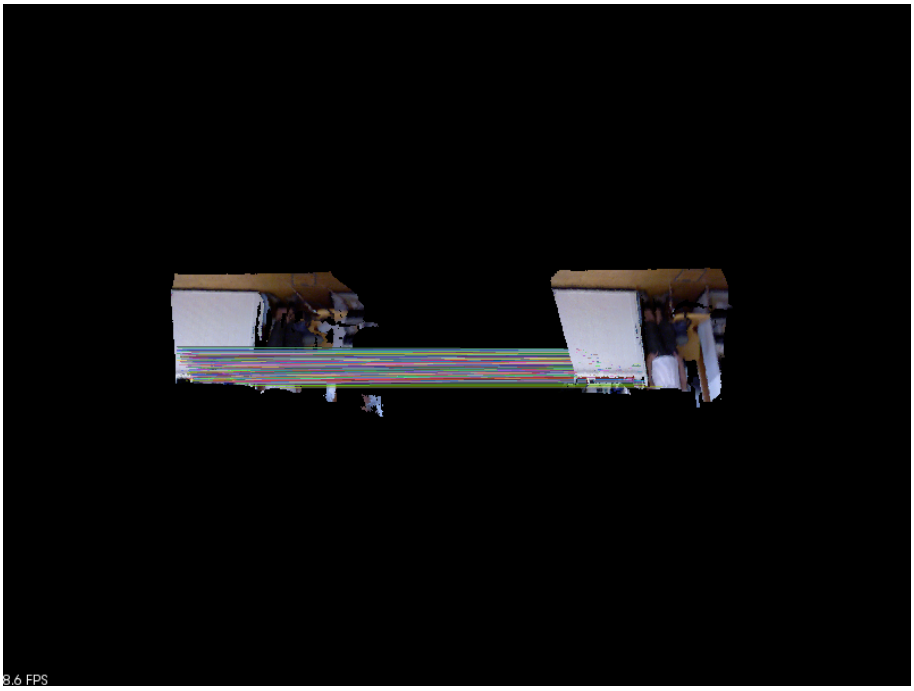




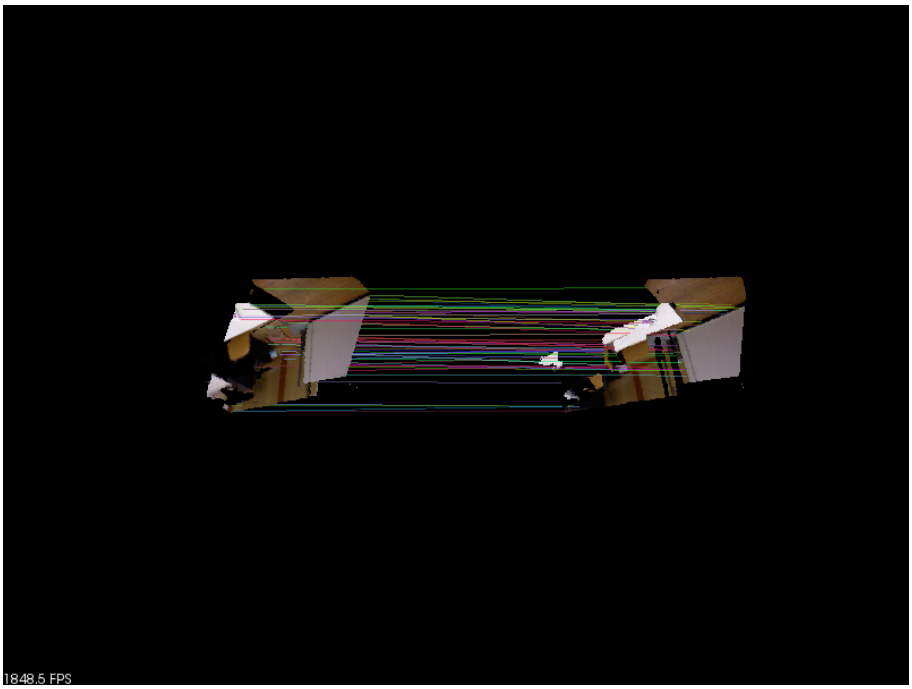
ISS + SHOT



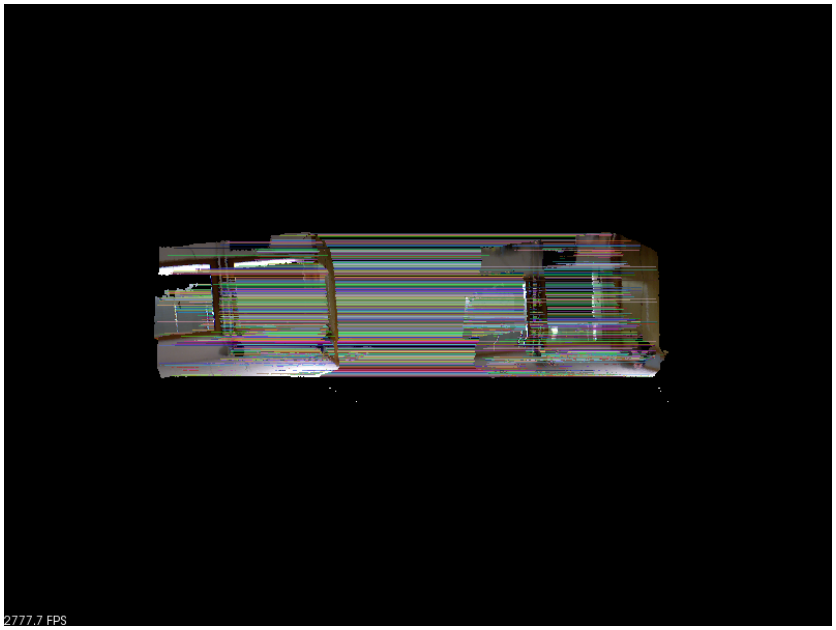
HARRYS + SHOT



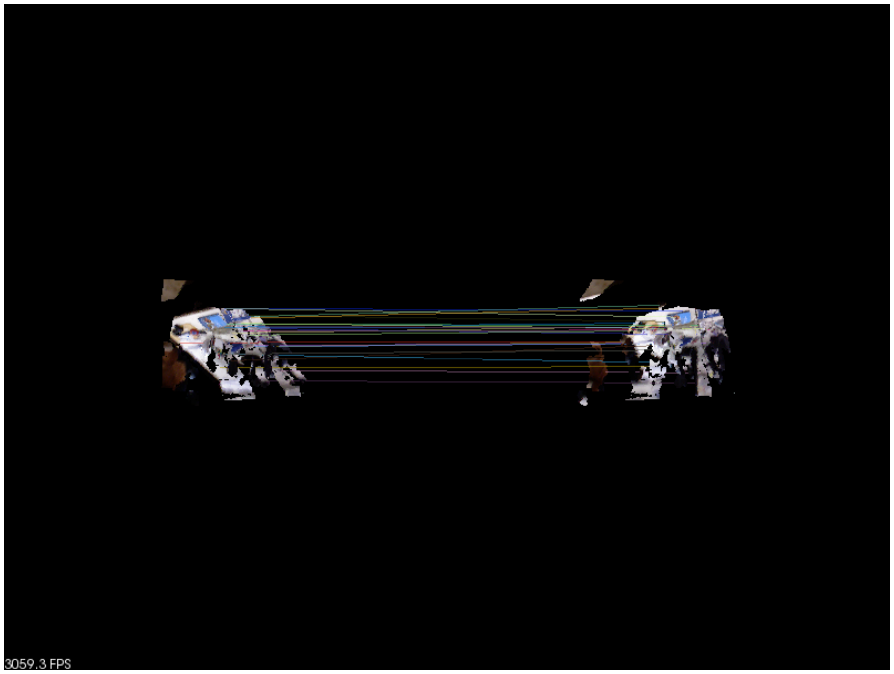
SIFT + SHOT

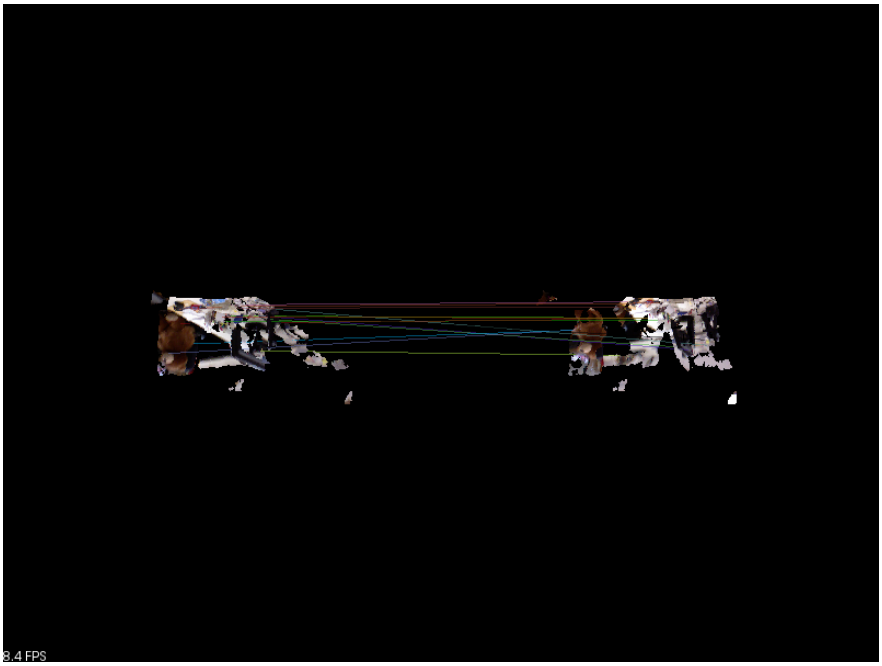


ISS + PFH

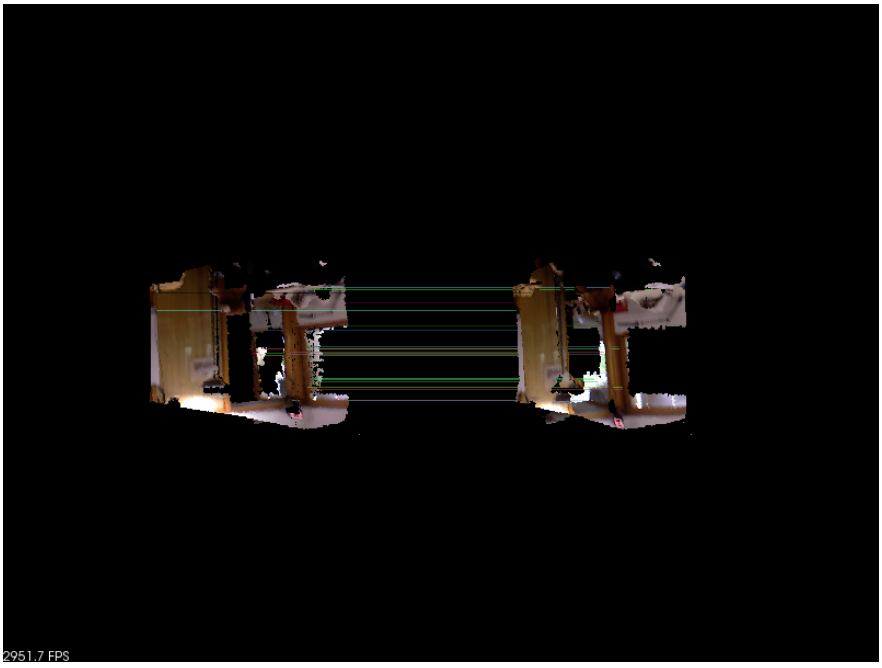


SIFT + PFH





HARRYS + PFH



Se aprecia claramente como unos métodos dan una gran cantidad de *matches* con respecto a otros, pero de nada nos sirve si la cantidad de malas correspondencias aumenta

también.

HARRYS + Descriptor: Quizás no alcanza a obtener tantas correspondencias pero la mayoría son buenas y lo hace en un corto espacio de tiempo.

ISS + Descriptor: Funciona bastante bien en general pero se obtienen demasiadas malas correspondencias. En zonas de mayor incertidumbre como el suelo, obtiene menos y sobre todo se ha podido comprobar que es bastante más lento que los demás.

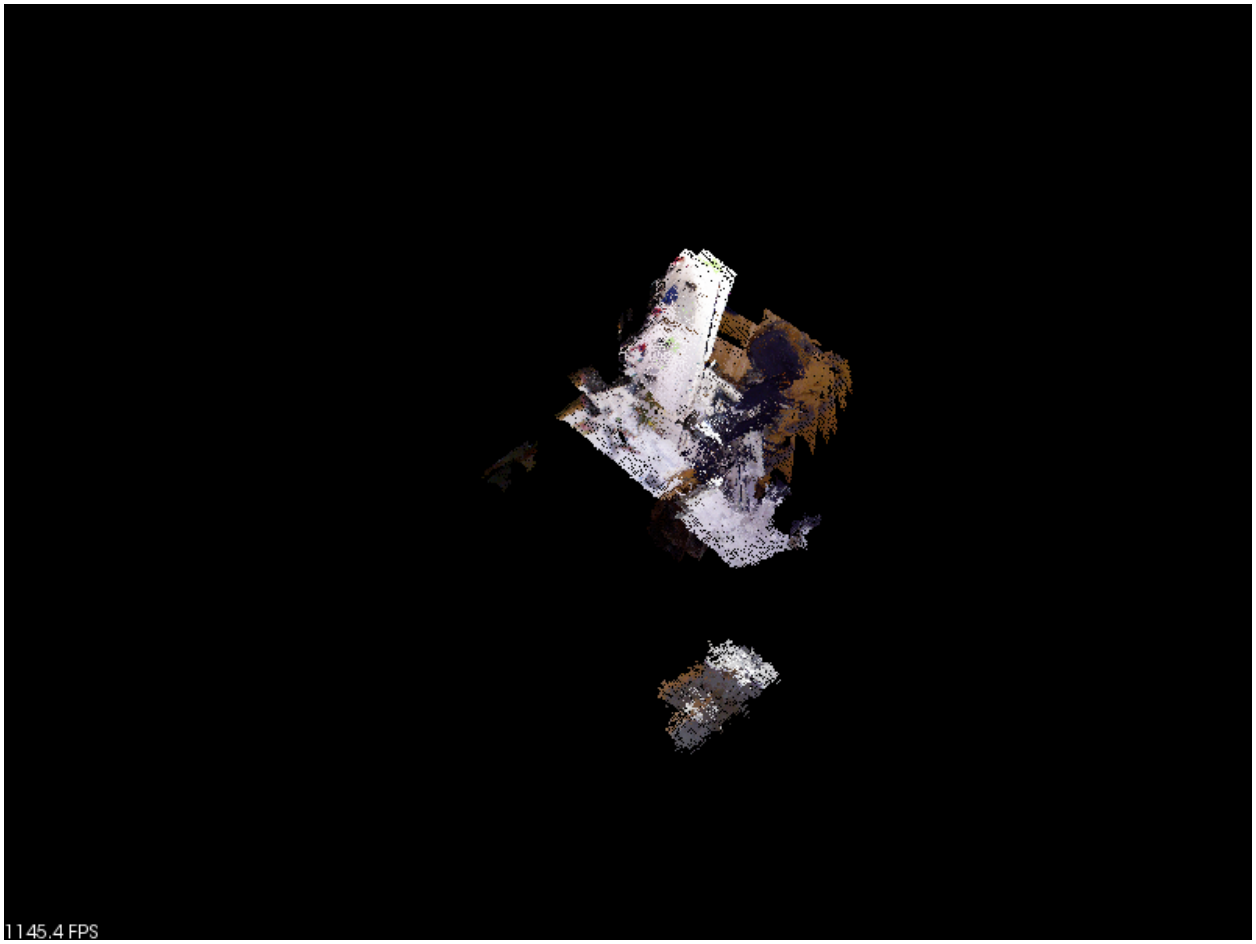
SIFT + Descriptor: Aunque deja atrás en tiempo a los que usan ISS, podemos apreciar claramente que el porcentaje de malas correspondencias que se obtienen en estas combinaciones es significativamente mayor a las buenas.

NARF + NARF: Los resultados obtenidos son bastante desconcertantes con respecto a los demás, esto es debido a que como ya comentamos no se trata de un método realmente aplicable al problema que nos atañe ya que realmente trabaja sobre 2D, aún así lo hemos desarrollado de forma **optativa**.

Podríamos decir que la combinación que mejor resultados ha obtenido es ***HARRYS + FPFH*** ya que alcanza la mayor cantidad de aciertos con respecto a errores en un tiempo bastante ajustado.

Reconstrucción

Tras aplicar RANSAC tratamos de obtener la transformación para ir acumulándola. Comparando las dos nubes, apoyándonos en las correspondencias, se pretende alcanzar la reconstrucción de la sala en 3D. Ya sea por los *outlayers* que pueden surgir, por una mala depuración y descarte de transformaciones, quizás parámetros equivocados en los métodos, etc. pero me ha sido imposible obtener una reconstrucción correcta. A continuación algunas imágenes del mapa generado.



Aunque existen funciones y parámetros definidos en las clases que ocupan de esta tarea, se han omitido anteriormente estos puntos debido a su inefectividad.

Conclusiones y problemas encontrados

Tras la experimentación realizada hemos visto varias diferencias que existen entre los métodos de extracción de características 3D y su posterior búsqueda de correspondencias. Al igual que en la práctica anterior algunos son capaces de encontrar una gran cantidad de *características* de forma veloz pero no dan buen resultado final tras hallar las correspondencias entre dos conjuntos. En general se aprecia la lentitud de estas operaciones lo que ha dificultado la experimentación, se entiende que el tratamiento de 3D es mucho más complejo que el 2D visto con anterioridad.

Como parte **optativa** hemos desarrollado y experimentado los métodos NARF para *keypoints* y *features.* Se ha logrado paralelizar la ejecución de la extracción de características

gracias a directivas de PCL que utilizan OpenMP.

En cuanto a los **problemas** encontrados a lo largo del desarrollo:

- En ocasiones SHOT corta la ejecución lanzando el error:
  ```
  [pcl::KdTreeFLANN::setInputCloud] Cannot create a KDTree with an empty input
  cloud!
  ```

- La librería PCL peca de falta de *typedefs*, tipos de datos ya definidos y nos obliga a crear los nuestros o a lidiar con varios operadores de ámbito seguidos tornando el código bastante enrevesado.

- Los diferentes tipos de datos que utilizan cada método dificultan la genericidad de los mismos y al no compartir una clase padre, ha sido necesario recurrir a templates y otras estructuras para tratar de automatizar la experimentación.