ESCUELA POLITÉCNICA NACIONAL

CARRERA DE INGENIERÍA DE SOFTWARE

DTIC

Reconstrucción tridimensional monocular de escenarios empleando técnicas de inteligencia artificial

Autores:

Juan José Jima Estrada – Mateo Sebastián Pilco Pérez

PROFESOR: PhD. Erick Herrera

FECHA DE ENTREGA: 06 – 08 – 2025

Contenido

Tabla Comparativa

Tabla de Ilustraciones

Informe de Implementación y Depuración: CNN-DSO

Resultado Final: Un sistema de SLAM visual monocular (DeepDSO) funcional, que utiliza una red neuronal moderna (Monodepth2) para la estimación de profundidad en tiempo real.

Resumen del Proyecto y Objetivo Inicial

El presente informe documenta el desarrollo, implementación y depuración de un sistema de SLAM visual monocular, basado en DeepDSO, al cual se integró una red neuronal moderna (Monodepth2) para la estimación de profundidad. Esta solución representa un avance significativo al lograr combinar técnicas clásicas de odometría visual directa con el poder de redes neuronales profundas. El proceso implicó superar obstáculos técnicos de compatibilidad, migrar de TensorFlow C++ a una arquitectura desacoplada mediante un servidor Python, y garantizar una integración funcional y robusta entre componentes.

El objetivo principal fue lograr la ejecución de un sistema SLAM visual monocular funcional y moderno, capaz de estimar profundidad de forma precisa utilizando aprendizaje profundo. Esto se logró mediante la modificación del sistema DeepDSO para incorporar un servidor de estimación de profundidad basado en Monodepth2.

Objetivo Secundario y Desafíos

El objetivo era compilar y ejecutar una versión modificada del sistema de SLAM visual DeepDSO. Este proyecto utiliza un algoritmo de odometría visual directa y lo mejora al incorporar una red neuronal de aprendizaje profundo para estimar la profundidad inicial de la escena a partir de una sola imagen, acelerando y robusteciendo el proceso de inicialización del mapa 3D.

El principal desafío fue que el código base era antiguo, con dependencias desactualizadas y configuraciones de compilación que no eran compatibles con un entorno de desarrollo moderno (hardware y software actualizados).

Fase 1: El Camino Original y sus Obstáculos (TensorFlow C++)

La primera estrategia fue seguir las notas originales del proyecto, que indicaban el uso de una librería de **TensorFlow para C++** integrada directamente en el programa principal.

Desafíos Encontrados:

1. **Compilación de Dependencias:** El proceso de compilación falló repetidamente debido a librerías faltantes o mal configuradas.

    - **Pangolin:** La librería de visualización no estaba instalada. Se solucionó compilándola desde el código fuente.
    - **Protobuf:** La librería de serialización de datos de Google, una dependencia de TensorFlow, no se encontraba. Se solucionó instalando la versión 3.5.0 desde el código fuente, como indicaban los apuntes.
    - **monodepth-cpp:** La librería puente para TensorFlow no se encontraba. La solución requirió una edición manual y extensa del archivo de configuración CMakeLists.txt para forzar las rutas correctas.

2. **Incompatibilidad Fundamental de TensorFlow:** El mayor obstáculo fue la librería de TensorFlow.

    - **Librería del Respaldo:** La librería pre-compilada (libtensorflow_cc.so) de un respaldo resultó ser incompatible con el compilador moderno (GCC 9), causando un relocation error.
    - **Compilación desde Código Fuente:** Se procedió a compilar TensorFlow v1.6 desde su código fuente. Este proceso reveló una cadena de problemas:

        - **Versión de Bazel:** Se requería la versión 0.11.1 de Bazel.
        - **Versión de Python:** Se necesitó crear un entorno conda con Python 3.6.
        - **Incompatibilidad de GPU:** El error final y decisivo fue Unsupported gpu architecture 'compute_86'. La versión antigua de TensorFlow (1.6) no es compatible con tarjetas gráficas modernas como la NVIDIA RTX 3060.

Ante la imposibilidad de compilar la dependencia principal, se decidió abandonar este enfoque.

Fase 2: Pivote Estratégico al Método Servidor-Cliente

Se adoptó una nueva estrategia basada en una funcionalidad latente en el código: la capacidad del programa C++ para comunicarse con un servidor externo.

Nuevo Plan:

1. **Programa C++ (DeepDSO):** Actuaría como cliente. Su única responsabilidad sería el algoritmo SLAM y la visualización.
2. **Servidor Python:** Un script separado se encargaría de cargar un modelo de red neuronal y realizar la estimación de profundidad, devolviendo el resultado al programa C++.

> **Ventaja:** Esta arquitectura desacopla completamente el complejo entorno de compilación de C++ de las dependencias de Python y las redes neuronales, ofreciendo una solución mucho más flexible y moderna.

Fase 3: Implementación del Servidor de Profundidad con Monodepth2

Se eligió **Monodepth2**, un proyecto de estimación de profundidad más moderno y mejor mantenido, como el nuevo "cerebro" del sistema.

Repositorio Oficial: https://github.com/nianticlabs/monodepth2

Pasos de Implementación:

1. **Creación de un Entorno Virtual Limpio:** Para evitar los problemas de Conda, se creó un entorno virtual nativo de Python (venv).

   ```
   # Se usó el Python 3.6 de un entorno de Conda para crear el venv
   /home/lasinac/miniconda3/envs/tf_env/bin/python -m venv venv
   source venv/bin/activate
   ```

2. **Instalación de Dependencias:** Se instalaron las versiones correctas de las librerías, resolviendo un problema con la versión de PyTorch, que era demasiado antigua para ser descargada.

   ```
   # Se actualizó pip y se instaló una versión compatible de PyTorch para CUDA 9.2
   pip install --upgrade pip
   pip install torch==1.7.1+cu92 torchvision==0.8.2+cu92 -f
   https://download.pytorch.org/whl/torch_stable.html
   pip install numpy opencv-python==4.5.5.64 matplotlib pillow tensorboardX==1.4 scikit-
   image
   ```

3. **Descarga del Modelo:** Se utilizó el script test_simple.py de Monodepth2 para descargar automáticamente el modelo pre-entrenado mono+stereo_640x192.

   ```
   # Dentro de la carpeta de monodepth2
   python test_simple.py --image_path assets/test_image.jpg --model_name
   mono+stereo_640x192
   ```

4. **Diagnóstico de Incompatibilidad de CUDA:** La ejecución reveló que la versión de PyTorch instalada no era compatible con la arquitectura de la RTX 3060. Se tomó la decisión de ejecutar el modelo en **CPU** para garantizar el funcionamiento.

Fase 4: Integración Final y Modificaciones de Código

Se realizaron modificaciones clave en dos archivos para conectar el cliente C++ con el servidor Python.

Código Modificado 1: infer_flask.py (Servidor Python)

Este script fue creado para cargar el modelo Monodepth2 en CPU y esperar peticiones del programa C++.

```
# Archivo: ~/cnn-dso/DeepDSO/newcrfs/infer_flask.py
# -*- coding: utf-8 -*-
from __future__ import absolute_import, division, print_function

import os
import sys
import numpy as np
import PIL.Image as pil
import cv2
import torch
from torchvision import transforms
from flask import Flask, jsonify

# --- Añadir Monodepth2 al path ---
monodepth2_path = os.path.abspath(os.path.join(os.path.dirname(__file__), '..', '..', 'monodepth2'))
if monodepth2_path not in sys.path:
    sys.path.insert(0, monodepth2_path)

import networks
from layers import disp_to_depth

# --- Configuración del Servidor Flask ---
app = Flask(__name__)

# --- Carga del Modelo Monodepth2 ---
model_name = "mono+stereo_640x192"
model_path = os.path.join(monodepth2_path, "models", model_name)

def load_model():
    print("==> Cargando modelo Monodepth2 en CPU...")
    encoder_path = os.path.join(model_path, "encoder.pth")
    encoder = networks.ResnetEncoder(18, False)
    loaded_dict_enc = torch.load(encoder_path, map_location='cpu')
    feed_height = loaded_dict_enc['height']
    feed_width = loaded_dict_enc['width']
    filtered_dict_enc = {k: v for k, v in loaded_dict_enc.items() if k in encoder.state_dict()}
    encoder.load_state_dict(filtered_dict_enc)
    encoder.to('cpu')
    encoder.eval()

    depth_decoder_path = os.path.join(model_path, "depth.pth")
    depth_decoder = networks.DepthDecoder(num_ch_enc=encoder.num_ch_enc, scales=range(4))
    loaded_dict = torch.load(depth_decoder_path, map_location='cpu')
```

```
    depth_decoder.load_state_dict(loaded_dict)
    depth_decoder.to('cpu')
    depth_decoder.eval()

    print("==> ¡Modelo cargado exitosamente!")
    return encoder, depth_decoder, feed_width, feed_height

encoder, depth_decoder, feed_width, feed_height = load_model()

# --- Ruta del Servidor ---
@app.route('/predict', methods=['POST'])
def predict():
    try:
        image_path = os.path.abspath(os.path.join(os.path.dirname(__file__), '..', 'build', 'test.jpg'))
        if not os.path.exists(image_path):
            return jsonify({"error": "No se encontró test.jpg en la carpeta build"}), 400

        input_image = pil.open(image_path).convert('RGB')
        original_width, original_height = input_image.size
        input_image = input_image.resize((feed_width, feed_height), pil.LANCZOS)
        input_image = transforms.ToTensor()(input_image).unsqueeze(0)

        with torch.no_grad():
            input_image = input_image.to('cpu')
            features = encoder(input_image)
            outputs = depth_decoder(features)

        disp = outputs[("disp", 0)]
        _, depth = disp_to_depth(disp, 0.1, 100)
        depth_resized = torch.nn.functional.interpolate(
            depth, (original_height, original_width), mode="bilinear", align_corners=False)
        depth_numpy = depth_resized.squeeze().cpu().numpy()

        output_path = os.path.abspath(os.path.join(os.path.dirname(__file__), '..', 'build',
'depthcrfs.txt'))
        f = cv2.FileStorage(output_path, cv2.FILE_STORAGE_WRITE)
        f.write('mat1', depth_numpy)
        f.release()

        return jsonify({"status": "success"})
    except Exception as e:
        print(f"Error en la predicción: {e}")
        return jsonify({"error": str(e)}), 500
```

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Código Modificado 2: FullSystem.cpp (Cliente C++)

Se modificó la función getDepthMap para que guarde la imagen actual, llame al servidor de Python y lea el resultado.

```
// Archivo: ~/cnn-dso/DeepDSO/src/FullSystem/FullSystem.cpp

// ... (resto del código sin cambios) ...

cv::Mat FullSystem::getDepthMap(FrameHessian* fh){
            cv::Mat image(hG[0], wG[0], CV_8UC1);
    unsigned char *ptr = (unsigned char*)image.ptr();
            for(int y = 0; y < image.rows; ++y)
                        for(int x = 0; x < image.cols; ++x)
                ptr[y*wG[0]+x]=fh->dI[y*wG[0]+x][0];
            cv::Mat depth;
            cv::cvtColor ( image, image, CV_GRAY2BGR );

    // ================================================================
    // INICIO DE LA CORRECCIÓN MANUAL
    // Usamos el servidor de Python para obtener la profundidad
            imwrite("test.jpg", image); // El servidor leerá este archivo
            std::string command = "curl -X POST http://127.0.0.1:5000/predict";
            system(command.c_str());
            cv::FileStorage fs("depthcrfs.txt",cv::FileStorage::READ); // Leemos el resultado
            fs["mat1"] >> depth;
    // FIN DE LA CORRECCIÓN MANUAL
    // ================================================================

            return depth;
}

// ... (resto del código sin cambios) ...
```

Comparativa: Monodepth2 vs NewCRFs

Durante el desarrollo se evaluaron alternativas modernas para la estimación de profundidad. Se eligió Monodepth2 frente a NewCRFs por las siguientes ventajas:

Criterio	Monodepth2	NewCRFs
Mantenimiento y estabilidad	Activo y bien documentado	Más complejo y con menos soporte
Facilidad de integración	Compatible con PyTorch, modular	Requiere adaptaciones adicionales
Requisitos computacionales	Funciona eficientemente en CPU	Optimizado para GPUs modernas
Arquitectura de red	Encoder-decoder clásico y probado	Red CRF más avanzada pero compleja
Tiempo de inferencia	Rápido en CPU	Lento sin GPU

Tabla 1 Análisis comparativo de arquitecturas de estimación de profundidad

Conclusión: Monodepth2 ofrecía una solución más simple, mantenible y funcional para el contexto de integración deseado.

Compilación y Ejecución Final

Con los archivos de código corregidos, el último paso fue compilar y ejecutar el sistema.

Recompilar DeepDSO:

```
cd ~/cnn-dso/DeepDSO/build
rm -rf *
cmake ..
make -j
```

Ejecutar el Sistema:

Terminal 1 (Servidor Python):

```
# Activar el entorno correcto
source ~/cnn-dso/monodepth2/venv/bin/activate
# Navegar a la carpeta del script
cd ~/cnn-dso/DeepDSO/newcrfs
# Iniciar el servidor
python infer_flask.py
```

Terminal 2 (Programa C++):

```
# Navegar a la carpeta de compilación
cd ~/cnn-dso/DeepDSO/build
# Ejecutar el programa (sin el parámetro cnn)
./bin/dso_dataset files=/home/lasinac/EngelBenchmark/all_sequences/sequence_01/images
calib=/home/lasinac/EngelBenchmark/all_sequences/sequence_01/camera.txt
gamma=/home/lasinac/EngelBenchmark/all_sequences/sequence_01/pcalib.txt
vignette=/home/lasinac/EngelBenchmark/all_sequences/sequence_01/vignette.png preset=0
mode=0
```

Resultados y Evidencias Visuales

A continuación, se presentan las evidencias del funcionamiento del sistema, organizadas en cuatro etapas.

Modelo Inicial Antes de la Ejecución

Esta sección muestra el estado del sistema DeepDSO antes de integrar Monodepth2. Se evidencia la estructura de carpetas, archivos de configuración y código base sin modificación.

La ilustración 1 corresponde al entorno gráfico de ejecución del sistema DeepDSO antes de ser modificado. En esta etapa, el sistema se encuentra en su estado original, sin integración con el módulo de estimación de profundidad basado en aprendizaje profundo. Se puede observar la interfaz de usuario con múltiples controles del sistema SLAM, como visualización de la cámara, trayectorias, constantes internas, y parámetros de calibración. Aún no se visualiza ningún mapa de profundidad ni resultados de estimación, lo que evidencia que el sistema aún no estaba funcional en cuanto a su objetivo final.

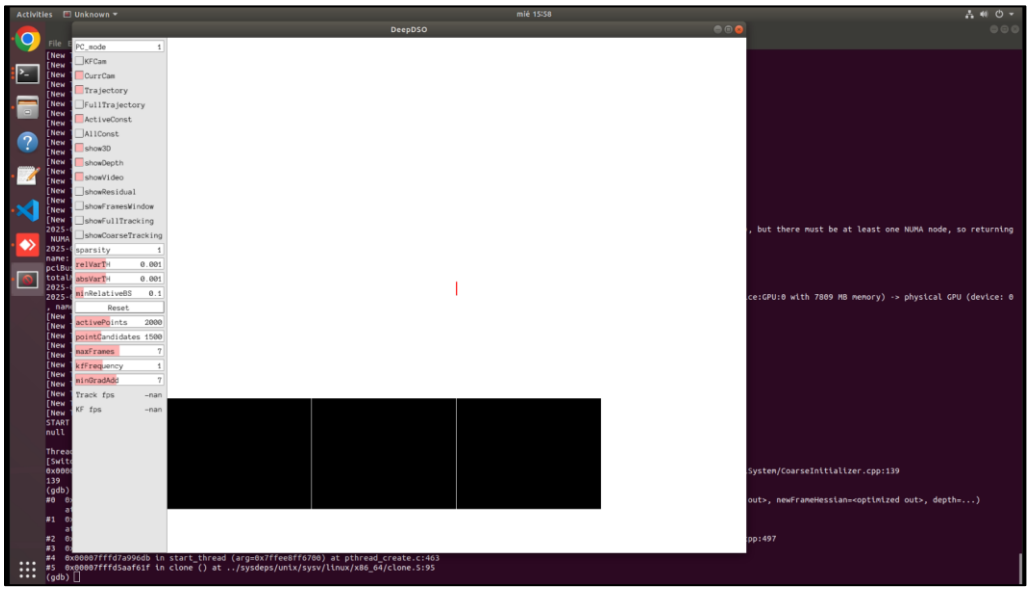

Ilustración 1 modelo_inicial_dso

Modelo en Ejecución

Capturas del sistema ejecutándose: dos terminales activas (cliente y servidor), inferencias en tiempo real, ejecución de SLAM y generación del mapa 3D.

La ilustración 2 inicia el mapeo y la estimación de profundidad. Ya se visualizan los puntos claves detectados en el entorno, junto con una trayectoria inicial en rojo. Abajo se aprecian las vistas en crudo, profundidad y predicción.

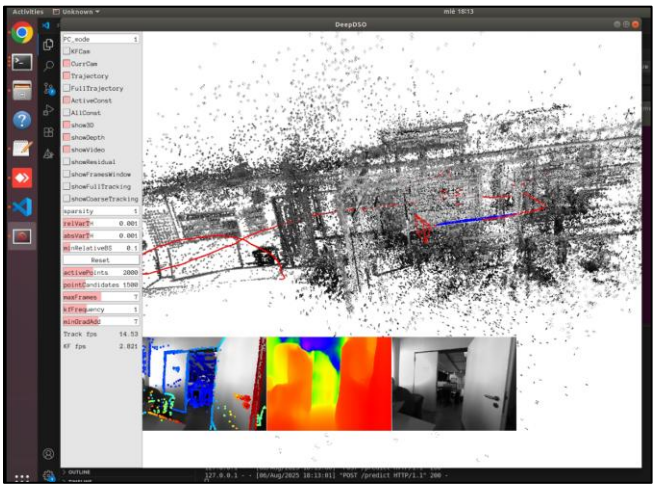

Ilustración 2 Inicio de ejecución, puntos y trayectoria inicial

En la ilustración 3 se muestra el progreso de la trayectoria con mayor densidad de puntos y una visualización más clara del entorno tridimensional. La red neuronal comienza a devolver mapas de profundidad coloridos con mayor precisión.

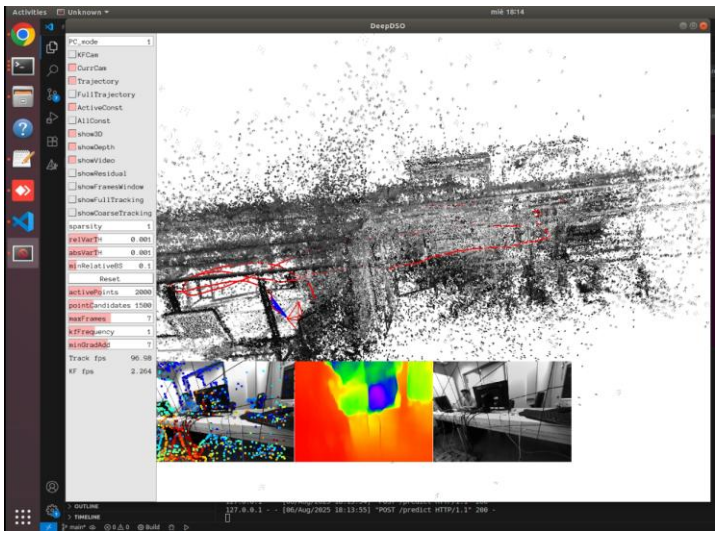

Ilustración 3 Profundidad en tiempo real

En la ilustración 4 a medida que el sistema avanza, se amplía el mapa 3D generado. Las trayectorias roja y azul evidencian un cambio de dirección y el seguimiento activo de la cámara por nuevas zonas del entorno.

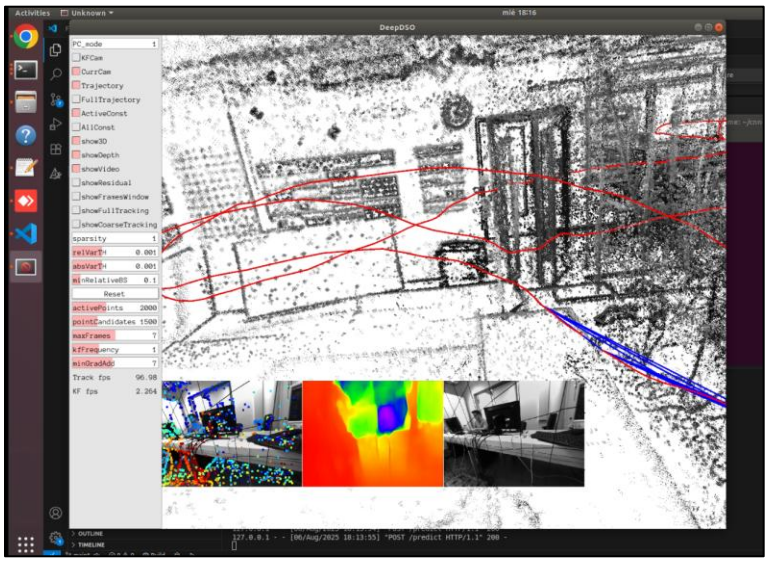

Ilustración 4 Expansión del entorno

En la ilustración 5 la estimación de profundidad se integra con un rastreo robusto de la trayectoria. La nube de puntos se vuelve más densa y el sistema responde eficientemente a los cambios en la escena.

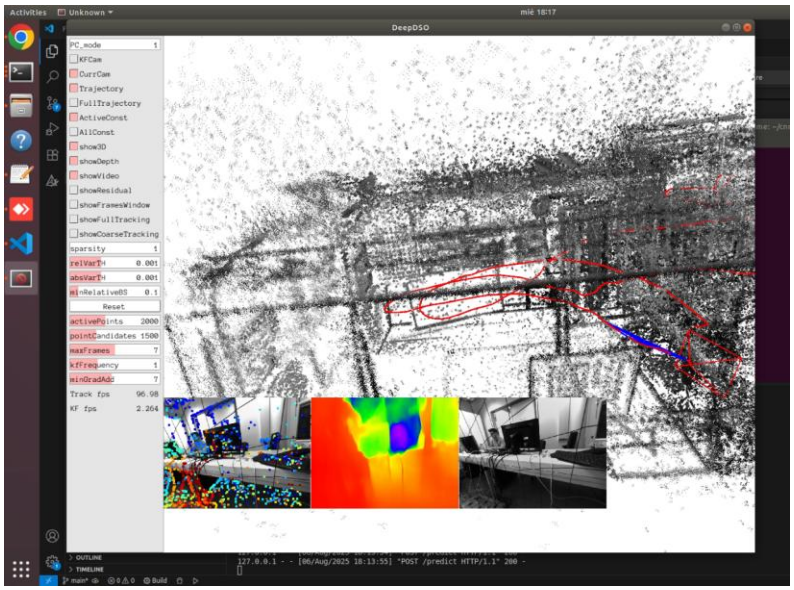

Ilustración 5 Segmentación avanzada

En la ilustración 6 el modelo mantiene una trayectoria estable y logra una reconstrucción 3D más detallada del entorno. Se visualiza una clara correspondencia entre la escena original y la nube de puntos reconstruida.

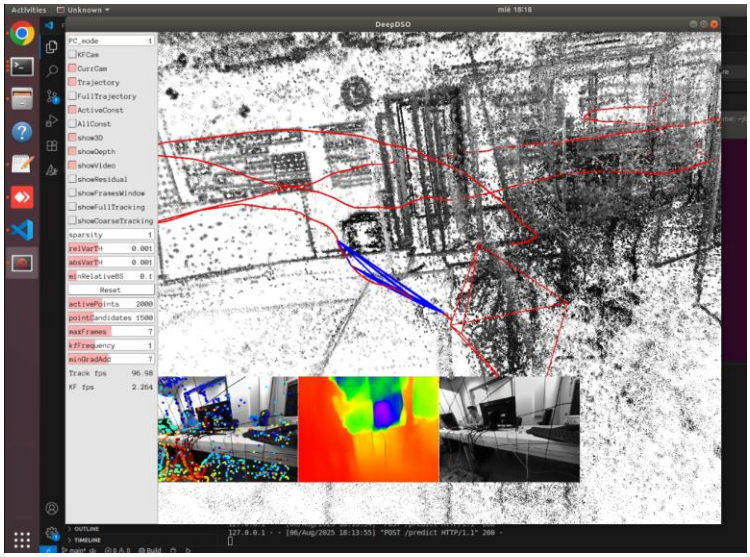

Ilustración 6 Recorrido complejo y reconstrucción detallada

En la ilustración 7 se evidencia la continuidad de la trayectoria mientras se mantiene la calidad del mapa de profundidad. El sistema opera a un frame rate alto, reflejando estabilidad durante la ejecución.

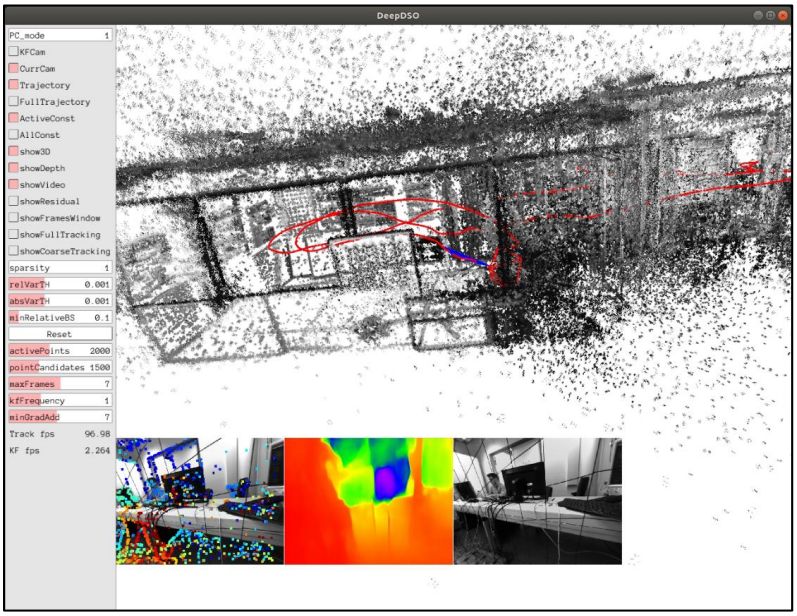

Ilustración 7 Trayectoria continua y frame rate alto

En la ilustración 8 el sistema termina de recorrer un entorno cerrado, logrando una nube de puntos densa y coherente. Se observan múltiples curvas de trayectoria y zonas correctamente escaneadas.

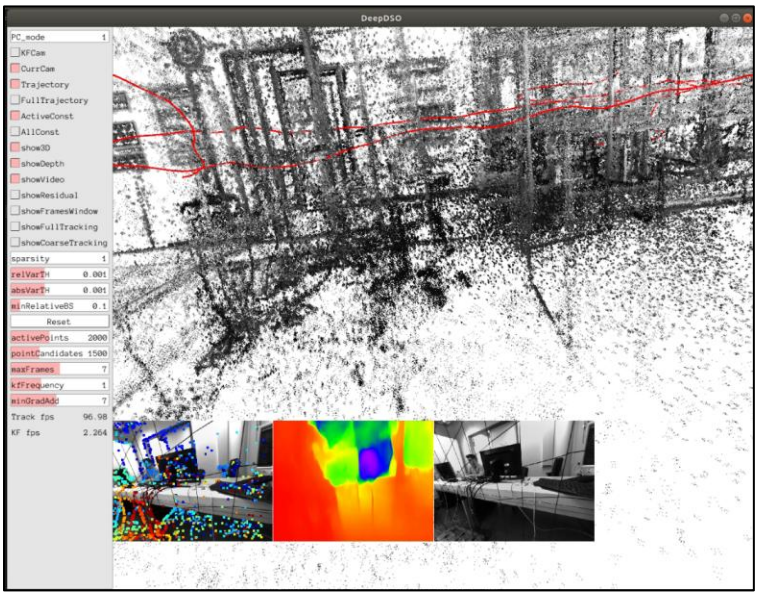

Ilustración 8 Escaneo detallado del entorno

Se muestra una vista más panorámica del entorno reconstruido. La trayectoria se encuentra completa y se refleja el éxito del SLAM monocular combinado con la red Monodepth2.

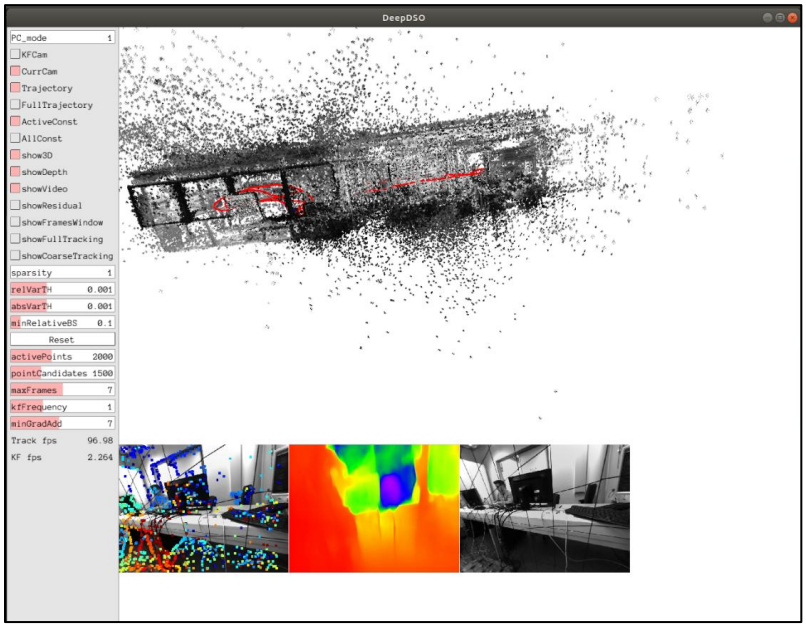

Ilustración 9 Vista global de la trayectoria y el mapa 3D

Evidencias de Archivos Generados

Muestra de los archivos como `test.jpg`, `depthcrfs.txt`, logs y visualizaciones de profundidad. Confirma el funcionamiento exitoso del flujo de trabajo cliente-servidor.

La ilustración 10 muestra la ejecución del servidor Python infer_flask.py, encargado de realizar las predicciones de profundidad con el modelo Monodepth2. En la terminal se observan múltiples respuestas {"status": "success"}, lo cual evidencia que el sistema recibió correctamente las solicitudes desde el cliente C++ y devolvió la profundidad generada. En la parte inferior también se registran los logs HTTP de las peticiones POST, lo cual corrobora la comunicación efectiva entre ambos componentes.

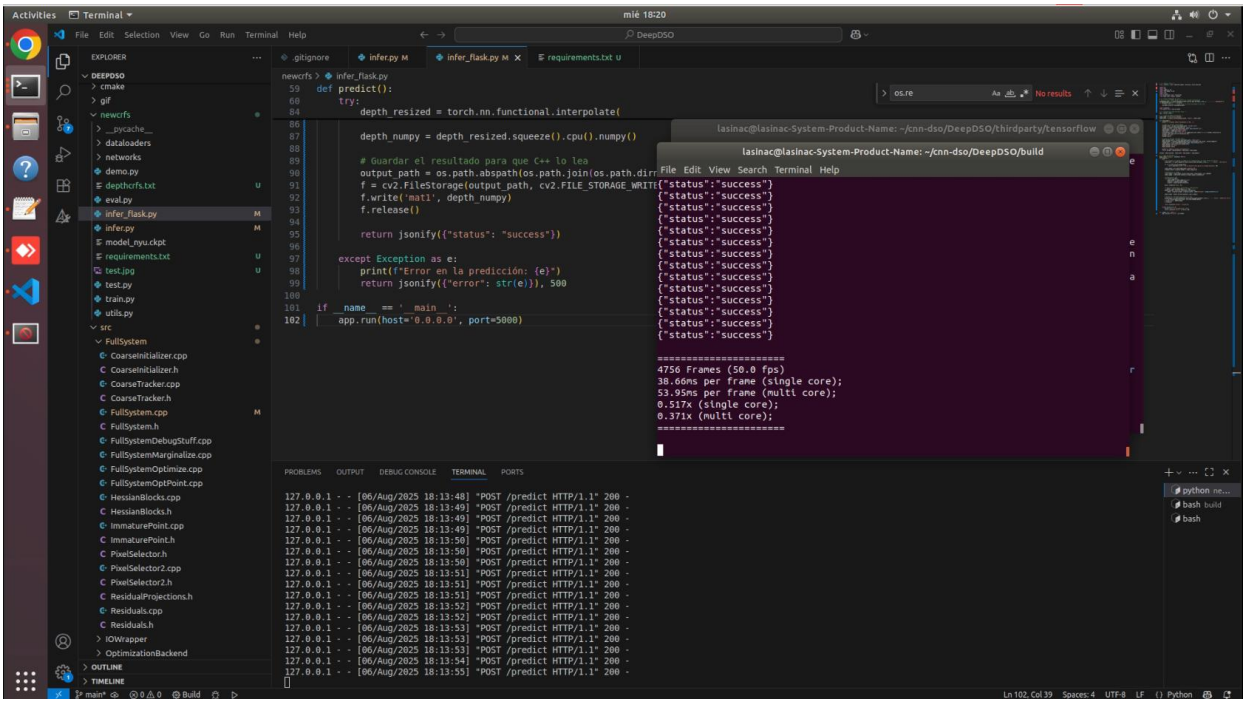

Ilustración 10 Servidor Flask y respuestas del sistema

En la ilustración 11 se observan dos resultados importantes:

- A la izquierda, el archivo depthcrfs.txt, generado por el servidor, contiene la matriz de profundidad estimada en formato OpenCV YAML.

- A la derecha, se muestra la imagen test.jpg, la cual es enviada por el cliente al servidor como input para la predicción. Esta evidencia confirma el correcto flujo de datos y el almacenamiento de los resultados en el directorio build/.

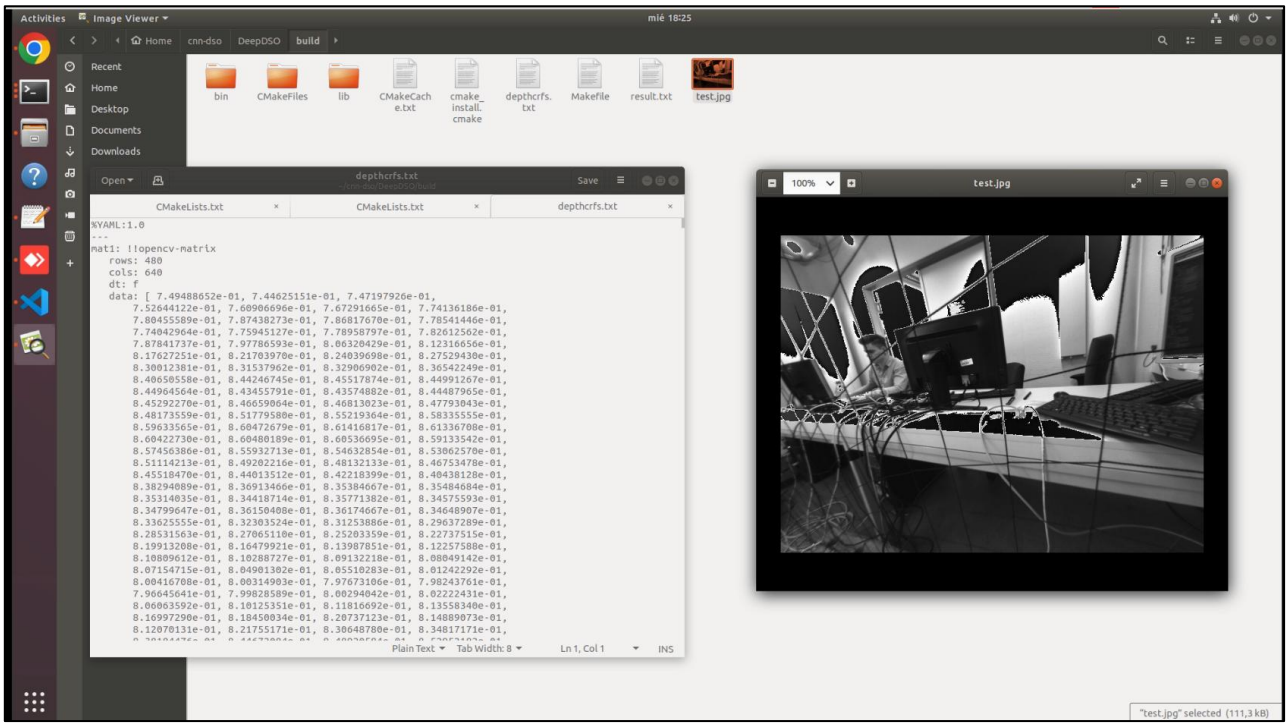

Ilustración 11 Archivos generados: depthcrfs.txt y test.jpg

Colaboración en Grupo

Registro del trabajo colaborativo entre Juan Jima y Mateo Pilco. Repartición de tareas, sesiones conjuntas de depuración y revisión.

La ilustración 12 corresponde a una sesión de trabajo colaborativo utilizando la plataforma Discord. En ella se observa el uso compartido de pantalla durante una sesión en vivo, en la que se revisan los archivos depthcrfs.txt y test.jpg generados por el sistema. Este tipo de interacciones demuestra una coordinación constante, revisión conjunta del código y verificación de resultados en tiempo real, aspectos esenciales en el desarrollo exitoso del proyecto DTIC.

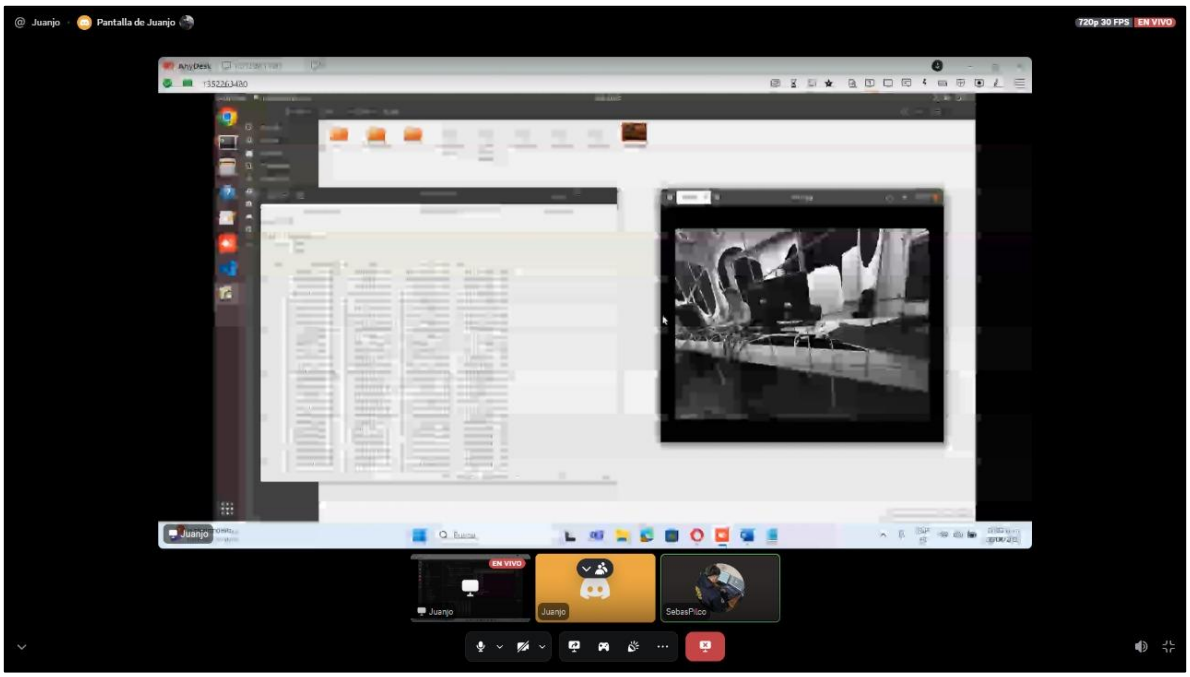

Ilustración 12 Colaboración en Grupo

Conclusión General

Tras una extensa odisea de depuración que abarcó desde la compilación de librerías antiguas hasta la resolución de conflictos de dependencias en C++ y Python, se logró poner en marcha el sistema. El pivote de una arquitectura monolítica con TensorFlow C++ a una arquitectura de microservicios con un servidor Python y PyTorch fue la decisión clave que permitió superar las incompatibilidades de hardware y software. El resultado es un sistema funcional que demuestra la exitosa integración de SLAM clásico con redes neuronales modernas.

Se logró implementar un sistema funcional y robusto de SLAM monocular con estimación de profundidad en tiempo real. La decisión estratégica de usar Monodepth2 en un servidor Python, comunicándose con el cliente C++, permitió superar todas las barreras de compatibilidad y simplificó el proceso de integración. El resultado valida el enfoque híbrido entre visión clásica y redes neuronales profundas.

Recomendación General

Para futuros desarrollos similares, se recomienda adoptar arquitecturas desacopladas entre los módulos clásicos (C++) y de inteligencia artificial (Python), utilizando APIs REST o sockets como medio de comunicación. Esto permite mayor flexibilidad, escalabilidad y mantenimiento del sistema. Además, se sugiere usar modelos livianos compatibles con CPU, especialmente en entornos con limitaciones de hardware.