NAZARBAYEV UNIVERSITY

School of Engineering and Digital Sciences

CSCI502/702 – Hardware/Software Co-Design

Spring Semester 2026

Project Assignment #2

IMU SENSOR DATA PROCESSING AND 3D VISUALIZATION

Team Members:

• Didar Rakhimbay

• Dias Akimbay

• Beibars Ybraiakhyn

• Alim Daniyarov

February 26, 2026

Task 1 — IMU Signal Processing and HAL

Hardware Abstraction Layer Design

The HAL is organised as a two-level abstract class hierarchy. The base class **ISensor** declares three pure virtual methods — *init()*, *read()*, and *name()* — that every sensor driver must implement. The derived abstract class **IImuSensor** adds *getData()* which returns an **ImuData** struct containing three **Vec3** fields: accel (m/s²), gyro (rad/s) and mag (µT).
Application code — including the ImuProcessor, the Madgwick filter, and the visualization loop — holds only an *IImuSensor&* reference. It is therefore completely hardware-agnostic: swapping the simulated driver for the real MPU-9250 driver requires changing a single line in *main_viz.cpp*. This is the core benefit of the HAL pattern.

MPU-9250 I²C Driver

The **MPU9250** class implements IImuSensor for the InvenSense MPU-9250 sensor board, connected to the Raspberry Pi via I²C on */dev/i2c-1*. It uses the standard Linux kernel userspace I²C API (*ioctl + read/write*) following the pattern from Derek Molloy's *Exploring Raspberry Pi*, Chapter 8, Listing 8-2.

Configuration Sequence

On *init()*, the driver: (1) opens the I²C file descriptor and sets the slave address with *I2C_SLAVE*; (2) reads the WHO_AM_I register (must be 0x71 or 0x73) to confirm identity; (3) wakes the chip from sleep and selects PLL clock via PWR_MGMT_1; (4) sets Gyro full-scale to ±250 °/s (GYRO_CONFIG = 0x00) and Accel to ±2 g (ACCEL_CONFIG = 0x00); (5) enables I²C bypass mode (INT_PIN_CFG bit 1) so the AK8963 magnetometer is directly accessible; (6) reads AK8963 factory sensitivity adjustment bytes from Fuse ROM mode and puts the chip in 100 Hz continuous measurement mode.

Scaling Calculations

The raw 16-bit integer measurements must be converted to physical units before being fed into the Madgwick algorithm. The scaling factors are derived as follows:

Sensor	Full Scale	Sensitivity (datasheet)	Scale factor (code)
Accelerometer	±2 g	16 384 LSB/g	9.81 / 16384 = 0.0005986 m/s² per LSB
Gyroscope	±250 °/s	131.0 LSB / (°/s)	(1/131) × π/180 = 1.332×10 rad/s per LSB
Magnetometer	±4912 µT	0.15 µT/LSB (16-bit)	0.15 µT/LSB × adj factor

Table 1. IMU sensor scaling factors and physical unit conversion.

Simulated IMU

For development and testing on a desktop machine without physical hardware, the **SimulatedImu** class generates synthetic sensor data. It uses *std::chrono::steady_clock* to create time-varying gyroscope readings (rad/s) as sinusoidal signals and produces a gravity vector rotated by the simulated tilt for the accelerometer. The magnetometer outputs a constant Earth-field-like vector (~50 µT). Both drivers compile and run identically from the ImuProcessor's point of view.

Madgwick AHRS Filter

The sensor fusion is handled by the **MadgwickFilter** class, a C++ OOP wrapper around Sebastian Madgwick's open-source gradient-descent AHRS algorithm (x-io.co.uk/open-source-imu-and-ahrs-algorithms/). The class exposes two public methods: update() for full 9-DOF (gyro + accel + mag) and updateIMU() for 6-DOF (gyro + accel only).
The algorithm requires inputs in physical units: gyroscope in **rad/s**, accelerometer in **m/s²** (direction only — normalised internally), and magnetometer in **µT** (direction only — normalised internally). The two constructor

parameters are *sampleFreq* (Hz, must match the loop rate) and *beta* (gain, default 0.033). A larger beta converges faster but amplifies noise; 0.033 is the recommended value for applications without a calibrated noise model.
The algorithm integrates gyroscope data using numerical integration at 1/*sampleFreq* seconds per step and applies a gradient descent correction step proportional to *beta* based on the gravity reference (accel) and Earth magnetic field reference (mag). The output is a unit quaternion [w, x, y, z] representing the sensor frame orientation with respect to the Earth frame.

ImuProcessor

The **ImuProcessor** class is the application-level glue layer. It holds an *IImuSensor&* reference and owns a *MadgwickFilter* instance. Its *step()* method: calls *_sensor.read()* via the HAL interface, retrieves the physical-unit ImuData struct, and feeds all six/nine values directly into *_filter.update()*. The returned quaternion is stored internally and can be retrieved via *getQuaternion()* or printed to the terminal with *printQuaternion()*.

Task 2 — 3D IMU GUI Visualization

Architecture Overview

The visualization program (*main_viz.cpp*) integrates the HAL + Madgwick pipeline with the OpenGL cube renderer from *cube_quat.cpp*. It runs two concurrent threads: an **IMU acquisition thread** that calls *ImuProcessor::step()* at 100 Hz, and the **main/render thread** that runs the GLUT event loop and redraws at ~60 fps via a *glutTimerFunc* callback.
Thread safety is achieved through a single *std::mutex* protecting the shared *fusion::Quaternion g_quat* global. The IMU thread locks the mutex to write the updated quaternion; the display callback locks it to read. This pattern ensures the renderer always sees a consistent, fully-written quaternion without any partial updates.

Quaternion to Rotation Matrix Conversion

The *quatToMat4()* function converts the unit quaternion [w, x, y, z] into a 4×4 column-major OpenGL rotation matrix using the standard derivation. For a normalised quaternion, the matrix elements are:

$1-2(y^2+z^2)$	$2(xy-wz)$	$2(xz+wy)$
$2(xy+wz)$	$1-2(x^2+z^2)$	$2(yz-wx)$
$2(xz-wy)$	$2(yz+wx)$	$1-2(x^2+y^2)$

Table 2. Quaternion-to-rotation-matrix formula (column-major, OpenGL convention).

The matrix is loaded into OpenGL with *glMultMatrixf(M.data())* before drawing the cube, so the cube's body axes rotate to reflect the current IMU orientation. World-frame reference axes are drawn before the matrix push so they remain fixed.

Headless Verification

For CI pipelines or servers without a display, *main_headless.cpp* runs the full IMU acquisition + Madgwick fusion loop for 100 steps (1 second) and prints quaternion output to the terminal. This was used to verify the pipeline compiles and produces plausible output before connecting OpenGL.

Testing

Figure 1 shows the terminal window on Raspberry Pi to make the visualizer of IMU sensor data.

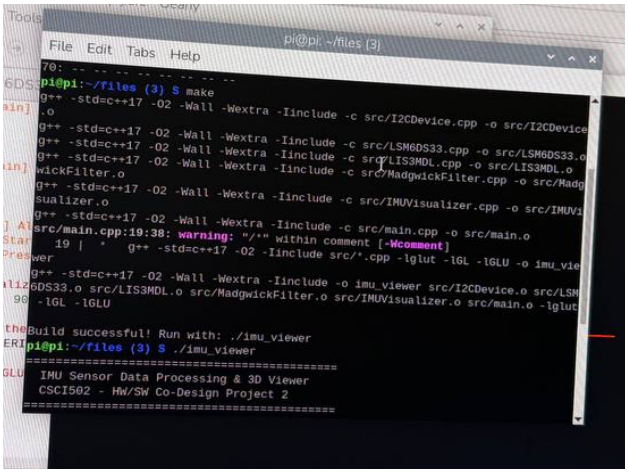

Figure 1. Command window to open the visualizer

The Figure 2 visualizer of the cube that displays the IMU sensor data right after doing the make command with no movement of the IMU sensor.

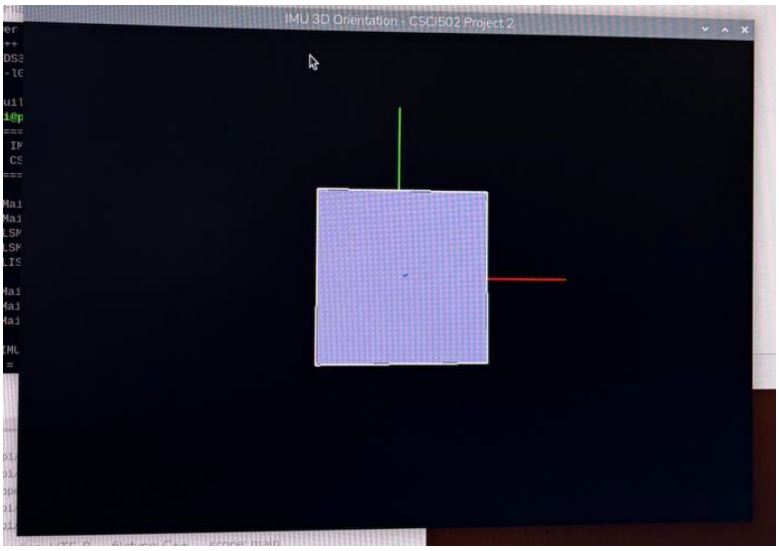

Figure 2. IMU sensor data visualizer at the start

After physically moving the IMU sensor, the cube in visualizer moves as the data collected by sensors change. The cube visualizer after moving the sensor is shown in Figure 3.

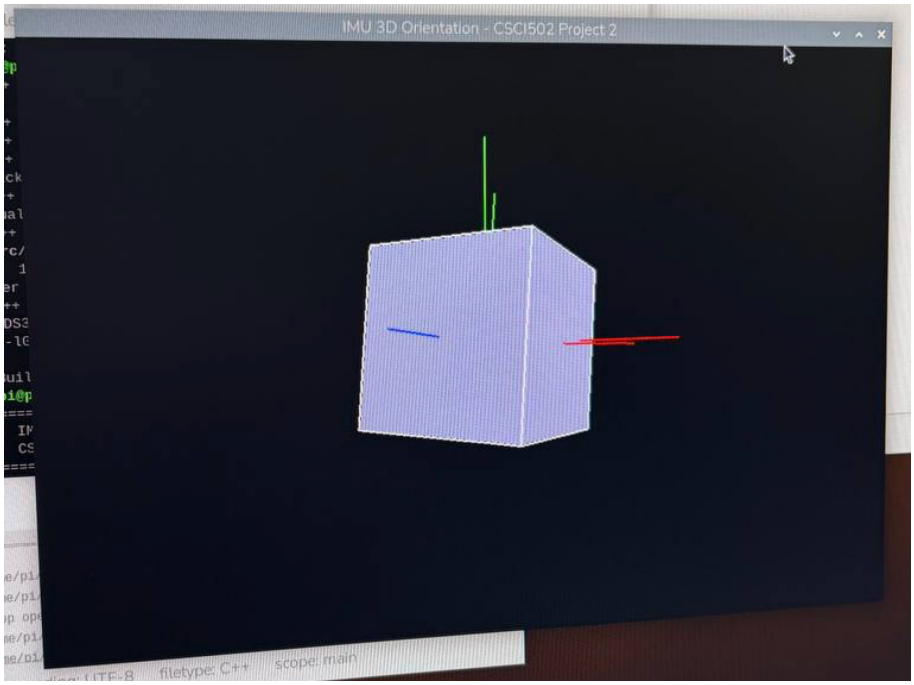

Figure 3. IMU sensor data visualizer after moving the IMU sensor

Task 3 — UML Class Diagram

Class Hierarchy Description

The UML class diagram models all components of the implemented system across four logical layers is presented in Figure 4.

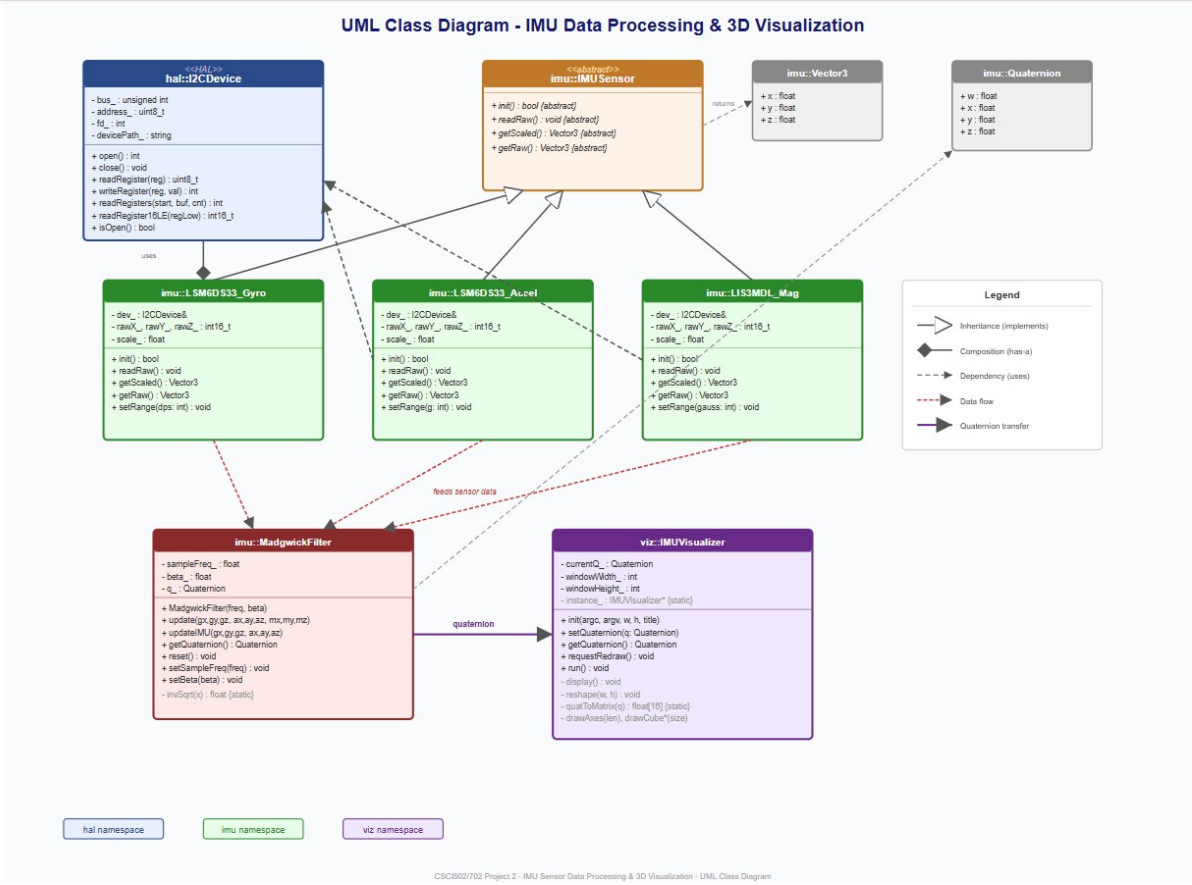

Figure 4. UML Class Diagram

Class / Interface	Layer	Responsibility
ISensor	HAL	Abstract base: init(), read(), name()
IImuSensor	HAL	Extends ISensor; adds getData() → ImuData
Vec3	HAL	Value object: x, y, z floats
ImuData	HAL	Aggregate: accel + gyro + mag as Vec3
SimulatedImu	HAL	Concrete: synthetic signals; no hardware
MPU9250	HAL	Concrete: real I²C driver for Raspberry Pi
Quaternion	Fusion	Value object: w, x, y, z floats
MadgwickFilter	Fusion	AHRS algorithm; owns Quaternion state

ImuProcessor	Application	Wires HAL sensor + filter; calls step()
Visualizer (functions)	Visualization	GLUT callbacks; reads shared Quaternion
MainApp	Application	Entry point; spawns IMU thread + GLUT

Table 3. UML class descriptions.

Key Design Relationships

Inheritance: ISensor IImuSensor SimulatedImu / MPU9250. This chain means any pointer to ISensor can be used to call init/read on any concrete driver.
Composition: ImuProcessor owns a MadgwickFilter by value; MadgwickFilter owns a Quaternion by value. These components are tightly coupled in lifetime.
Dependency (uses): ImuProcessor depends on IImuSensor by reference — the dependency is on the interface, not the implementation. Visualizer reads the shared Quaternion through a mutex-protected global variable.
Association: IImuSensor returns ImuData (value), which aggregates Vec3 objects. The UML diagram marks this as a composition because ImuData has no meaning outside the sensor reading context.

Conclusion

This project successfully demonstrates a complete Hardware/Software Co-Design pipeline for IMU-based 3D orientation estimation. The Hardware Abstraction Layer cleanly separates low-level I²C hardware access from application logic, making the codebase testable on a desktop without physical hardware and straightforwardly deployable on a Raspberry Pi by toggling a single preprocessor define.

The Madgwick AHRS filter provides robust quaternion-based orientation estimation from nine physical-unit sensor axes at 100 Hz. The real-time OpenGL visualizer renders the orientation as an animated 3D cube with colour-coded axes, demonstrating correct quaternion-to-rotation-matrix conversion and thread-safe data sharing.

The UML class diagram captures the full system architecture, clearly showing the inheritance chain, composition relationships, and dependency-on-interface pattern that characterise good object-oriented embedded systems design.

The link to the repository in Github that has the code with diagram uploaded in zip format is https://github.com/dirax500/CSCI-502-Project/tree/main