Radar Visualization Pipeline Report

Aiysha Mei Frutiger, Sandro Barbazza, Senanur Ates,
University of Basel
Computer Architecture

January 15, 2026

Abstract

Abstract of project

Contents

1 Introduction **2**

2 End-to-End Pipeline **2**

2.1 Hardware layer: actuation and sensing . . . 2

2.2 USB and operating system abstraction . . . 3

2.3 Host application interface (Processing) . . . 4

3 Visualization **4**

3.1 Purpose of the visuals . . . 4

3.2 First implementation attempt: C + raylib (prototype) . . . 4

3.3 Switch to Processing (final implementation path) . . . 5

3.4 How the current Processing radar UI works . . . 6

4 Conclusion **7**

5 Declaration of Authorship **7**

1 Introduction

A “radar screen” is a simple but powerful way to make an invisible physical process visible: a moving sweep line and a few dots are enough to turn raw distance measurements into an intuitive, real-time picture. This project implements a compact ultrasonic “radar” as an educational system that is feasible to build, yet still representative of the layered hardware–software interaction discussed in computer architecture. The central motivation is therefore not to replicate military radar, but to create a small end-to-end system in which sensing, data transfer, and real-time feedback can be observed and reasoned about directly.

At its core, radar-like sensing is based on time-of-flight. A wave is emitted, reflections are received, and the elapsed time is mapped to distance. To place a distance measurement in space, it must also be associated with a direction, i.e., an angle. In this project, direction is obtained by scanning across a range of angles, producing a polar dataset of the form (*angle, distance*) that can be rendered as the familiar 2D radar display. Although ultrasound is used here (physically closer to sonar), the underlying data model and visualization follow the radar principle.

Radar-style ranging and visualization are widely used because they provide robust spatial awareness with relatively simple sensing: examples include obstacle detection and parking assistance, robotics navigation, industrial level and proximity monitoring, and security or presence detection. By implementing a radar-like interface on top of an ultrasonic sensor, the project connects these real-world applications to fundamental architectural concepts such as signal acquisition, data representation, I/O abstractions, and continuous real-time visualization.

2 End-to-End Pipeline

The system forms a complete measurement and visualization pipeline: Sensor → Arduino → USB → Operating System → Processing → Visualization.

Figure 1: End-to-end pipeline from sensing on the Arduino to visualization on the host.

An Arduino-to-PC visualization setup was selected because it exposes the full communication stack discussed in computer architecture in a compact, observable system: a physical device produces signals, a microcontroller converts those signals into data, a transport layer (USB) exposes the device to the host, an operating system abstracts the hardware as files/handles/ports, and an application interprets a continuous byte stream into meaningful state and graphics. This structure makes the transformation of information across layers explicit and traceable.

2.1 Hardware layer: actuation and sensing

On the hardware side, the Arduino functions as a real-time controller responsible for both actuation and measurement. For actuation, a servo motor sweeps the radar head across a defined angular range. The Arduino generates a PWM control signal on the servo pin and implements the sweep using two stepwise loops: a forward sweep from the minimum angle to the maximum angle and a backward sweep from the maximum back to the minimum. After each angle update (`radarServo.write(angle)`), a short delay is introduced so the motor can physically settle at the requested position. This reflects an important architectural constraint: software commands are instantaneous, whereas mechanical systems require settling time before measurements become reliable.

For distance measurement, the Arduino triggers the ultrasonic sensor by emitting a short pulse on the trigger line and measures the resulting echo pulse width on the echo line using `pulseIn(...)`. Conceptually, this converts a timing signal into data: the sensor produces a pulse duration t (in microseconds), and the sketch converts that duration into a distance using the URM37-specific rule

$$\mathrm{cm} = \frac{t}{50}.$$

If no pulse arrives within the timeout or if the measured values are implausible, the sketch outputs `-1` to indicate an invalid or out-of-range reading.

After obtaining an angle and a distance, the Arduino transmits the results to the host system via serial communication using `Serial.println(...)` at 9600 baud. The transmitted format is line-based, human-readable ASCII (e.g., `angle,cm`), which simplifies debugging and supports clear message framing through newline termination.

2.2 USB and operating system abstraction

Microcontroller-to-host communication is transported via USB, where the Arduino is presented as a USB serial device (virtual serial port). Upon connection, the operating system performs USB enumeration: device descriptors are read (VID/PID, class information, endpoints), a suitable driver is loaded (CDC ACM or vendor-specific), and a system-visible device interface is created that applications can open as a byte stream.

Platform differences are visible at this abstraction boundary. On Windows, the driver typically creates a COM port (e.g., `COM7`) and applications access it via Win32 serial APIs (internally via calls such as `CreateFile("`
`textbackslash`
`textbackslash.`
`textbackslash COM7", ...)`). On macOS, the device appears as `/dev/tty.*` or `/dev/cu.*` and is accessed using POSIX I/O. On Linux, similar devices commonly appear as `/dev/ttyACM0` or `/dev/ttyUSB0`. Despite differences in naming and APIs, the core abstraction is consistent: the operating system exposes the physical USB device as a stream-oriented interface.

Two practical properties of serial devices influence system behaviour. First, serial ports are typically exclusive resources: if one process holds the port open, other processes cannot access it simultaneously. Second, many Arduino boards reset upon port opening because control lines such as DTR/RTS are toggled, causing the microcontroller to restart and execute `setup()` again. As a result, initial bytes received by the host may contain boot-time output; the host-side software must therefore tolerate initial noise and synchronize on valid measurement records.

2.3 Host application interface (Processing)

On the host side, Processing accesses the virtual serial port as a byte stream. The protocol uses newline-terminated records so that the stream can be framed into complete messages for parsing. The concrete parsing logic and the radar UI update behaviour are described in detail in Section 3.

3 Visualization

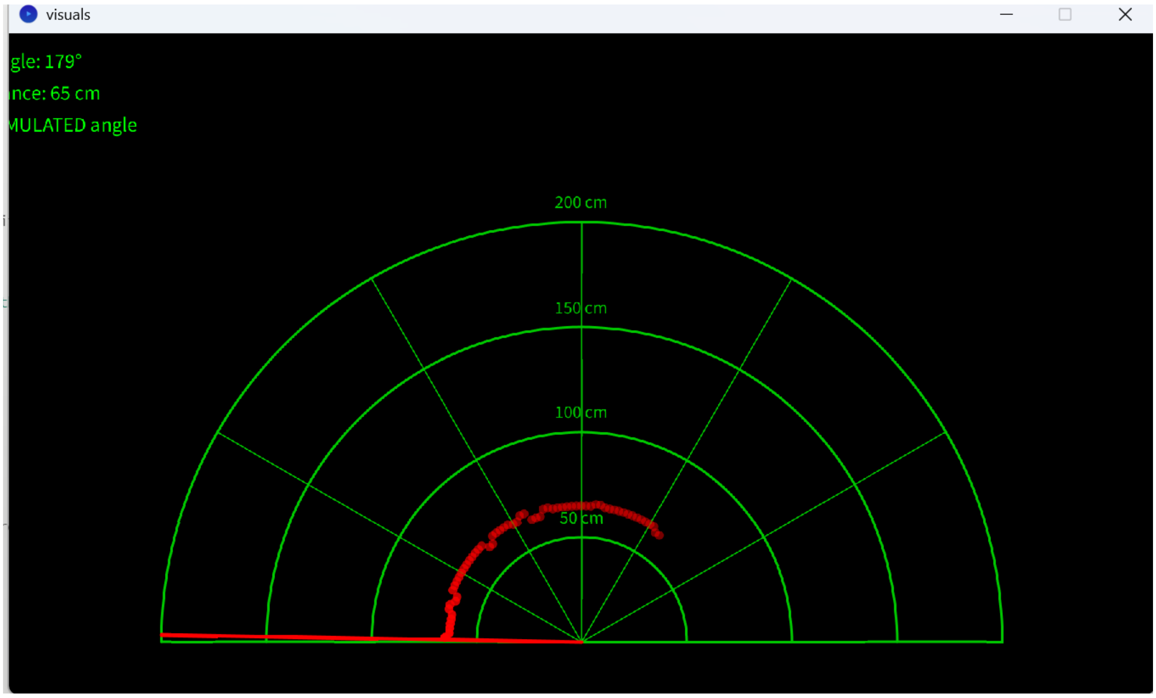


Figure 2: Final Processing-based radar visualization (grid, sweep line, and detections).

3.1 Purpose of the visuals

The visualization provides a real-time radar user interface that makes the sensor data immediately interpretable. The display is structured as a half-circle scan area (0°–180°) with a moving sweep line ("beam") that represents the current scan direction. Detected objects are rendered as distance points ("blips" / trail dots) positioned according to the measured distance at the corresponding angle. To support orientation and scale, the UI includes a range grid (concentric arcs) and distance labels in centimeters. The visualization is explicitly designed to integrate with Arduino sensor data transmitted via USB serial, and, once the servo is present, to use a servo-driven real angle rather than a purely simulated sweep.

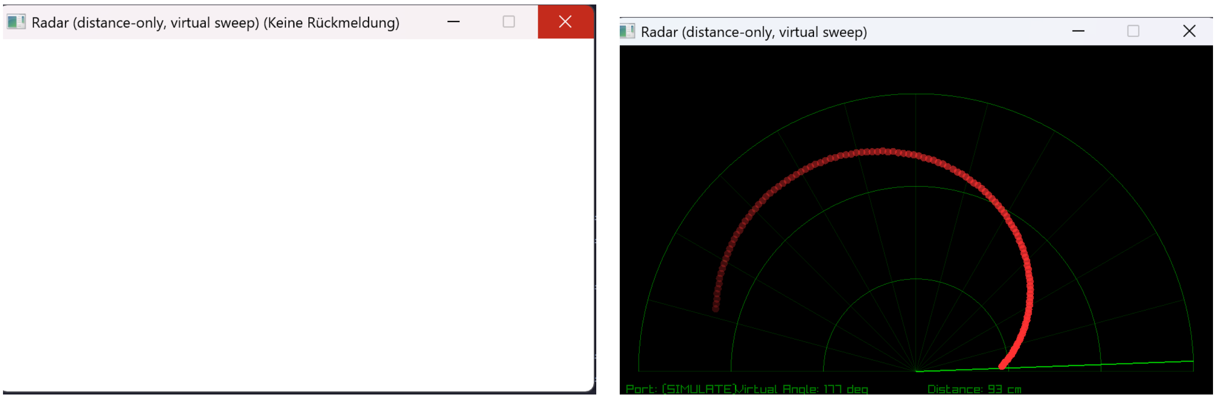
3.2 First implementation attempt: C + raylib (prototype)

The initial prototype was implemented as a Windows program in C using `raylib`. The rendering pipeline drew the radar background (arcs and angle divider lines), a sweeping beam, and blips with a fading lifetime, creating the intended "radar refresh" effect.

A major part of this prototype was the serial integration, implemented directly against the Windows API. Communication was handled via Windows COM ports using `CreateFileA`, with the connection configured through `DCB` settings (baud rate, 8N1). To avoid blocking and to keep the UI responsive, the design relied on non-blocking reads using `COMMTIMEOUTS` and queue inspection via `ClearCommError` / `COMSTAT`. A robust "read line" mechanism was implemented by accumulating bytes into a buffer, detecting newline termination (`\n`), trimming `\r\n`, parsing numeric values, and ignoring unrelated text.

In addition, an auto-detection / handshake approach was introduced to support a plug-and-play experience. The program scanned `COM1..COM64` automatically, toggled DTR to trigger Arduino reset, and waited for a signature string such as `RADAR_READY` to confirm the correct port. The intended workflow was: connect USB → start program → automatic detection → live radar output.

In practice, this setup proved unreliable for consistent operation (Figure 3). Windows serial behaviour is sensitive to port contention (e.g., an open Serial Monitor), timing immediately after reset, and partial or malformed lines during Arduino boot; additionally, COM port numbering differs across machines. The resulting failure modes ranged from missing or malformed measurements to crashes caused by unexpected serial buffer contents (Figure 3a). During debugging, the visualization often ran only in a simulated/virtual-sweep configuration, indicating that the prototype did not yet provide a robust sensor-driven end-to-end path (Figure 3b).



(a) Prototype failure case: the application becomes unresponsive / renders a blank window after receiving unexpected or incomplete serial data.

(b) Prototype running with virtual sweep / simulated values (`SIMULATE` mode), illustrating that the displayed "distance" did not reliably represent real sensor measurements.

Figure 3: Screenshots from the C+`raylib` prototype highlighting practical issues: unreliable serial I/O and non-robust handling of malformed input, motivating the switch to Processing.

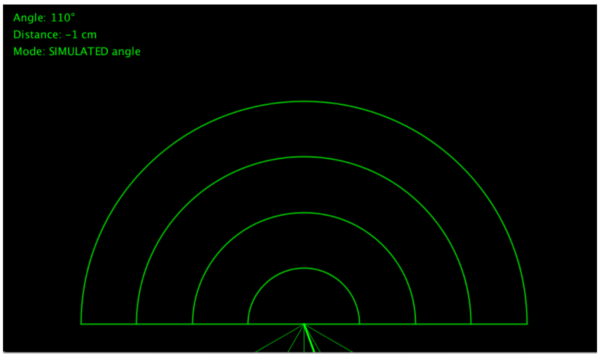
The rendering portion was successful, but low-level serial I/O on Windows became disproportionately time-consuming compared to the core project goal of iterating on the visualization and pipeline behaviour.

3.3 Switch to Processing (final implementation path)

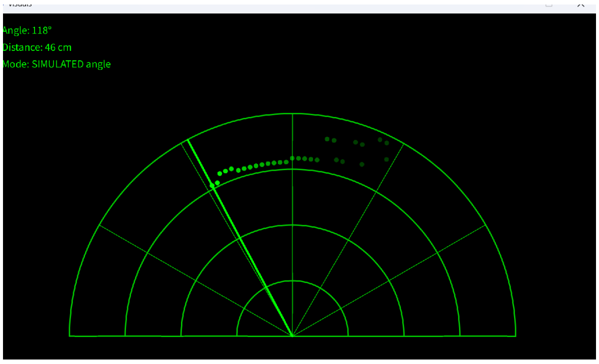
The final implementation moved to Processing to reduce OS-specific complexity and accelerate iteration. Processing's serial library provides a high-level interface that matches the project’s protocol design directly: selecting the correct port and baud rate is straightforward, `bufferUntil('\n')` supports line-based framing, and `serialEvent()` receives complete records. This removed a large class of platform-specific pitfalls and allowed focus to shift back to the visualization logic and the end-to-end data pipeline.

3.4 How the current Processing radar UI works

Figure 4 shows two intermediate visualization iterations developed while validating the scan logic and UI behaviour.



(a) Early Processing prototype: the sweep and grid were rendered with an inverted orientation. The simulated angle mode enabled rapid debugging of the coordinate mapping before servo feedback was available.



(b) Intermediate iteration: a fading trail was implemented by storing recent measurements per angle bucket and decreasing their alpha over time. This effect was later adjusted/removed to match the desired radar behaviour.

Figure 4: Evolution of the Processing-based radar visualization during development.

The Processing-based system can be described as a clear data pipeline from connection to rendered output:

1. **Arduino plugged in.** After connection, the Arduino begins transmitting newline-terminated serial records. Two formats are supported to allow smooth development progression:
   - *distance-only* (early stage without servo feedback)
   - *angle,distance* (final stage with servo-based real angle)
2. **Processing reads serial.** Processing opens the selected COM port at the configured baud rate and buffers input until newline termination (`\n`). On each complete line, the handler:
   - trims whitespace and line endings,
   - ignores non-data noise (e.g., boot messages or handshake strings),
   - parses either a single distance value or a pair *angle* + *distance*.

   If only distance is present, the program updates the current distance and continues using the simulated sweep angle. If angle and distance are provided, the real servo angle becomes the authoritative scan direction.
3. **Angle handling: simulated vs. real.** During early development, the servo motor was not yet integrated, but the visualization needed a scan direction to behave like radar. A simulated sweep angle was therefore implemented: it increments from 0 → 180 and then

back 180 $\rightarrow$ 0, matching the intended servo motion. This allowed testing and refining the full UI independent of motor hardware. Once the servo is available, the real angle sent by the Arduino overrides the simulated angle, keeping the display physically consistent with the scanning head.

4. **Visual update logic.** Each frame renders the radar in layered steps:
   - draw the grid (arcs and divider lines),
   - draw the current sweep line,
   - update the "measurement bucket" corresponding to the current sweep angle.

   If a distance value is valid (e.g., within `MAX_CM`), a detection point is stored for that angle bucket. If the value is invalid or out of range, the bucket is cleared—replicating the radar behaviour where the sweep effectively "erases" old detections when it rescans empty space.
5. **Trail persistence and fading.** To create persistence without accumulating unlimited points, the system stores one point per small angle segment (defined by `ANGLE_BUCKET_DEG`). Each stored point fades after creation: its alpha decreases over a configured time window (from full visibility down to a minimum alpha). When the sweep returns to the same bucket, that entry is overwritten—either by a fresh detection, or by a clear if nothing is detected. This produces the intended radar effect: detections persist briefly, then decay, and disappear naturally on refresh.

4 Conclusion

5 Declaration of Authorship

ChatGPT was used solely to improve the clarity and coherence of the report's language. All ideas, analyses, and interpretations presented reflect the group's own work and research.