

# Radar Visualization Pipeline Report

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

January 16, 2026

## Abstract

Abstract of project

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Planning</b>	<b>2</b>
<b>3</b>	<b>Hardware</b>	<b>2</b>
<b>4</b>	<b>System Overview: From Echo to Visualization</b>	<b>2</b>
<b>5</b>	<b>Visualization</b>	<b>3</b>
5.1	Purpose of the visuals . . . . .	3
5.2	First implementation attempt: C + raylib (prototype) . . . . .	3
5.3	Switch to Processing (final implementation path) . . . . .	4
5.4	How the current Processing radar UI works . . . . .	5
<b>6</b>	<b>Conclusion</b>	<b>6</b>
<b>7</b>	<b>Declaration of Authorship</b>	<b>6</b>

# 1 Introduction

We built a compact ultrasonic “radar-style” scanner that turns echo timing into a live 2D visualization. An ultrasonic pulse is emitted and the return echo duration encodes time-of-flight, which the microcontroller converts to distance. By sweeping the sensor with a servo, each measurement becomes an  $(angle, distance)$  pair that can be rendered in polar form. The project is primarily an educational end-to-end system that makes the hardware–software stack observable: sensing, embedded processing, USB serial transport, OS I/O abstraction, and real-time visualization.

## 2 Planning

## 3 Hardware

## 4 System Overview: From Echo to Visualization

Figure 1 summarizes the end-to-end pipeline. The ultrasonic sensor returns an *echo pulse width*, i.e., a time duration proportional to the time-of-flight. The Arduino controls the servo sweep via PWM, triggers the sensor, measures the echo duration (e.g., with `pulseIn`), converts it to a distance in centimeters, and transmits one newline-terminated ASCII record per measurement (`angle,cm`). Over USB the payload is a byte stream which the operating system exposes as a serial device (e.g., a COM port on Windows). The Processing program opens this port, frames the incoming bytes into lines, parses angle and distance, and updates the visualization state used for rendering.



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

## 5 Visualization

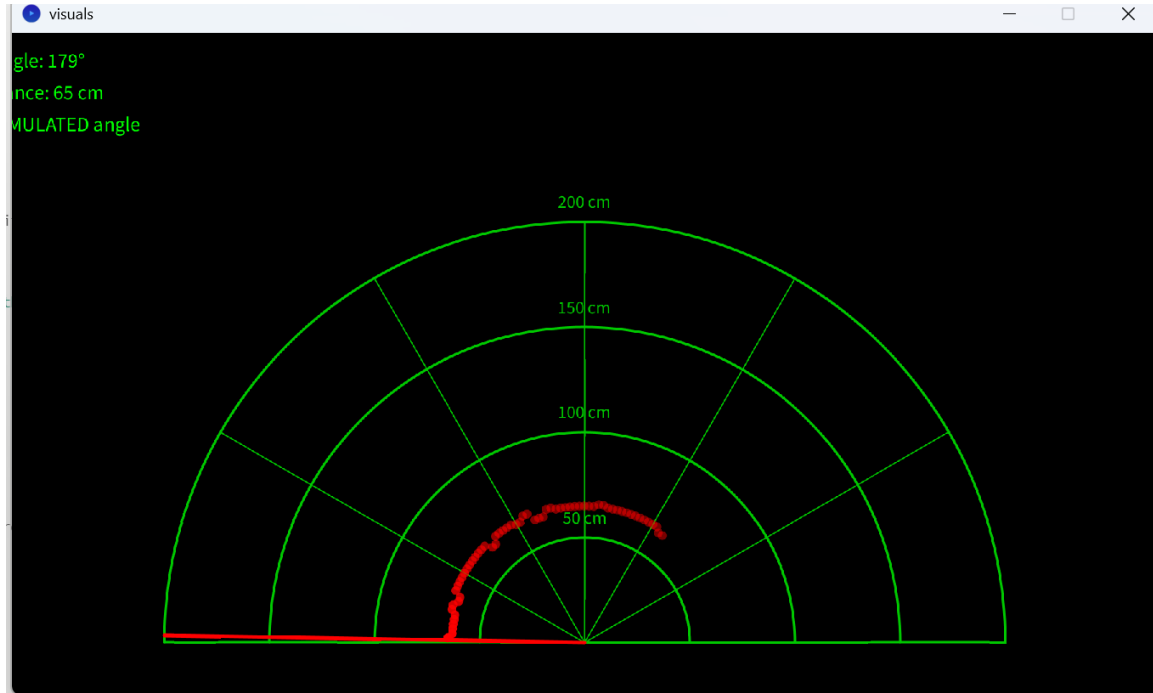


Figure 2: Final Processing-based radar visualization (grid, sweep line, and detections).

### 5.1 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 ( $0^{\circ}$ – $180^{\circ}$ ) 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.

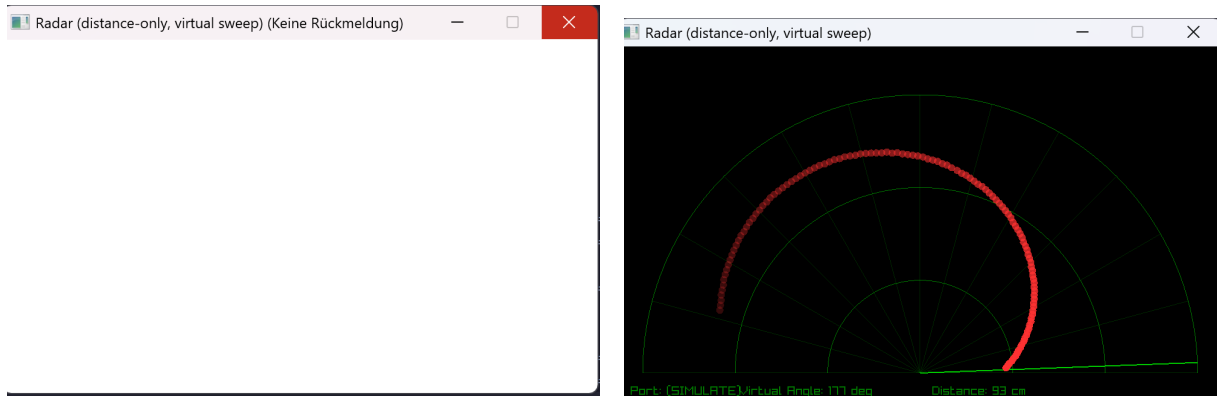
### 5.2 First implementation attempt: C + raylib (prototype)

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.

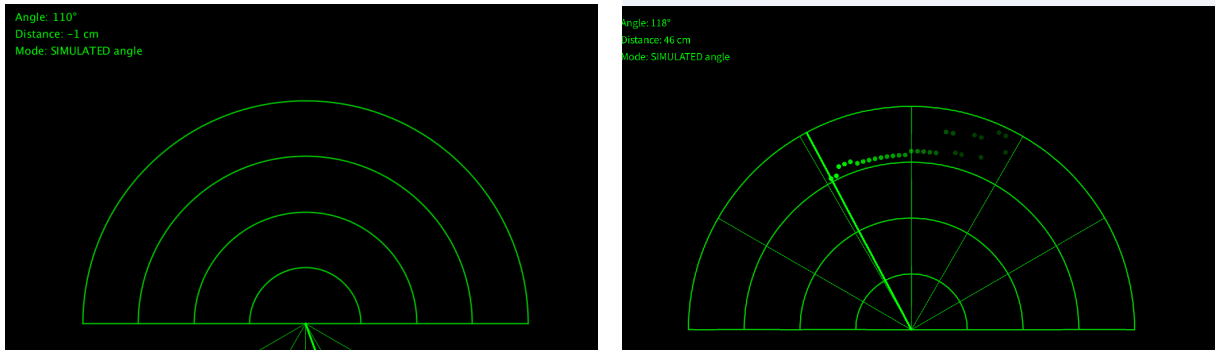
The rendering portion was successful, but low-level serial I/O on Windows became disproportionately time-consuming compared to the core project goal of iterating on the visualization and pipeline behaviour.

### 5.3 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.

## 5.4 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  $0 \rightarrow 180$  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.

## 6 Conclusion

## 7 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.