Github Code

Brett Rabbiner and Elizabeth Saunders
Professor Ryo Suzucki
ATLS 4519: How to Hack (Almost) Everything
December 2, 2025

Project 2 Document

Introduction:

Motivation: Overview

This project originated from two major inspirations introduced in class: Mixed Reality development using the Meta Quest 3 and robotics experimentation with Sony Toio robots. In Week 10, we explored How to Hack: Mixed Reality, experimenting with emerging AR/MR technologies such as the Meta Quest 3, Apple Vision Pro, WebXR, Hand Tracking, A-Frame, AR x Tangible interfaces, and Unity. The class demonstrated how immersive interfaces can transform user environments and create novel digital-physical interactions.

In Week 11, during How to Hack: Robots, we shifted to robotics and swarm interaction, working with the Sony Toio platform, reactive robotics, and RVO (Reciprocal Velocity Obstacles) algorithms. This highlighted how physical systems can become playful, interactive, and expressive.

Combining these inspirations, we were motivated to build something more creative, fun, and game-like, contrasting our previous highly practical project, Project 1: The ClueLess Closet. Inspired by telepresence systems and the entertainment value of Mario Kart, we set out to build a First-Person View (FPV) remote vehicle controlled through AR goggles—creating a hybrid system that merges robotics, mixed reality, and gamification.

Summary From Presentation:

a. Class Motivation #1: using the Meta Quest 3 for our final project
    i. How to Hack: Mixed Reality
    ii. Week 10 | October 28, 2025
    iii. Meta Quest 3, Apple Vision Pro, Hand Tracking, A-Frame, WebXR, AR x Tangible, Unity
b. Class Motivation #2: Toio Robots (playing with robots for our final project)
    i. How to Hack: Robots
    ii. Week 11 | November 4, 2025
    iii. Sony Toio, Swarm Robots, RVO Algorithm, Robotic Interfaces
c. Outside Motivation: Mario Kart
    i. Gamification!
    ii. Last project (Project 1, the ClueLess Closet), was much more practical so we hope to build something more fun and creative

Motivation: create a Telepresence or FPV (first-person view) game

Our motivation was to create an FPV telepresence racing experience where the user drives a modified RC car through a custom environment while wearing AR goggles—turning real-world navigation into an immersive game. By customizing the physical environment with obstacles and racetrack elements, and by enabling AR visualization screens through the Meta Quest 3, we aimed to explore how AR can enhance physical play, rather than replace it.

Describing What We Built

We built a telepresence FPV racing platform using a remote-controlled car with a smartphone camera mounted to the front, streaming real-time video into the Meta Quest 3 AR headset using Immersed screencasting. The headset enables the user to view the world from the vehicle's perspective, effectively becoming "miniaturized" into the environment. This transforms a physical RC car into an immersive AR racing machine. Unlike traditional RC driving—where the driver stands outside the scene—we created a driver-in-the-seat perspective, allowing the player to navigate a real-world custom racetrack as if they were sitting inside a tiny car.

Related Work

Related Products/Examples

1. Drones
    a. "'First-person view' drone pilot films Cleveland from a whole new perspective"
    b. Goal from this inspiration: Use an RC car instead of a drone to achieve that same effect
        i. "First Person View Driving Rig for an R/C Car"
            1. No gamification or environmental customization
2. Entertainment
    a. "New Apple Vision Pro video gives us a taste of escaping to its virtual worlds"
3. Video Games
    a. "True First Person Camera in Unreal Engine 4"
    b. "Nintendo brings Mario Kart into the real world with AR RC cars"
        i. However, no AR headset applications (requires Nintendo Switch)
4. Customizing Environment
    a. "The Vision Pro has landed."

Is our project novel and original?

Yes—our project is original and novel. While FPV RC cars and Mario Kart-like gamified racing exist separately, we found no examples that combine:

- FPV RC driving
- AR headset visualization
- Customizable physical racing environments
- A DIY hacked physical platform

The real innovation lies in integrating AR goggles with a physically modified

FPV RC car to create a deeply immersive and playful telepresence experience.

Implementation

Design process and ideation

We began by brainstorming how to merge AR interaction with robotics in a meaningful but manageable format. We ruled out drones for safety and complexity and instead selected an RC car as a stable base platform for prototyping FPV movement.

Design Process

Software:

- VSCode - IDE
- XCode - Deploying onto iPhone
- Claude IDE -Generative Environment
- Immersed - screencasting from the iPhone to the Meta Guest 3 AR goggles

Hardware:

- Modify / Build
  - Glued a phone mount to the RC car body (initially backwards — corrected next day)
  - Identified weight imbalance with camera installed
  - Tested temporary counterweights with tape
  - Permanently mounted balancing weights
  - Ran drive tests and calibrated video streaming alignment
  - Created a custom racetrack environment for AR-enhanced gameplay
- Testing
  - Tested control usability when vision is restricted to FPV only
  - Evaluated latency from streaming app to headset
  - Ran multiple trial races to adjust steering stability and mount durability

Division of Work

Brett Rabbiner

Responsible for software development, UX/UI, and project's AR capabilities Motivation: Use and test the AR that we learned in class for this project

Elizabeth Saunders

Responsible for sourcing and creating the physical model + documentation Motivation: Use and test the materials that we learned in class for this project

Describe how you build

In-Depth Explanation of Features and Methods of Technical Architecture

Architectural Pattern: MVVM

KartAR follows the Model-View-ViewModel (MVVM) architectural pattern, which separates concerns and promotes maintainable, testable code. This architecture is particularly effective with SwiftUI's reactive programming model.

Core Components

Component 1: KartARApp.swift (41 lines)

Purpose: Application entry point and orientation management orchestrator.

Key Features:

- OrientationManager class: A custom observable object that manages device orientation state across the entire app
- Dynamic orientation locking: Forces immediate rotation without animation delays, crucial for seamless transitions between portrait and landscape modes
- Published properties: Enables reactive UI updates when orientation changes occur

Technical Implementation: The OrientationManager uses SwiftUI's @Published property wrapper to broadcast orientation changes throughout the app. When a race starts, the system automatically switches to landscape mode by calling lockOrientation(.landscape), providing an immersive cockpit view. Upon race completion, it returns to portrait mode for detailed statistics display.

Component 2: ContentView.swift (567 lines)

Purpose: SwiftUI-based presentation layer that adapts to different game states and device orientations.

UI States and Screens:

- Home Screen: Initial landing with app branding and start button
- Race Type Selection: Choice between Grand Prix (multi-lap), Time Trial (best lap), and Practice (free-form)
- Track Selection: Choose from Monaco GP, Low Poly Circuit, or Mania (procedural) modes

- Scanning Room (Mania only): Visualizes mesh anchor data as the environment is scanned
- Track Placement: AR view with detected surfaces highlighted for track positioning
- Racing Interface: Full HUD with live timing, checkpoint progress, and race controls
- Race Complete: Results screen with final statistics and lap times

Component 3: RacingViewModel.swift (1,416 lines)

Purpose: The brain of the application - manages all game logic, AR session handling, and state coordination.
2.3 Major Systems and Methods

System 1: Game State Machine

Implementation: Uses a Swift enum to define discrete game states, ensuring type-safe state transitions and preventing invalid states.
States: home → raceTypeSelection → scanningRoom (conditional) → trackPlacement → racing → raceComplete

Methods:
- transitionTo(state:): Validates and executes state changes with appropriate cleanup and initialization
- resetGameState(): Returns to home state and clears all runtime data

System 2: Track Loading and Management

Purpose: Handles asynchronous loading, scaling, and positioning of 3D USDZ track models.

Key Methods:
- loadTrackModel(trackType:): Asynchronously loads USDZ files using Entity(contentsOf:) with proper error handling
- scaleTrackModel(entity:scale:): Applies uniform scaling to fit tracks in AR space
- placeTrack(at:anchor:): Positions loaded model at detected AR anchor points
- removeCurrentTrack(): Cleanup method that removes track entity from scene and releases memory

Models are loaded from the app bundle using Bundle.main.url(forResource:withExtension:). The async/await pattern prevents UI freezing during large model loading. Once loaded, models are added to the ARView scene graph and anchored to detected surfaces for stability.

System 3: Checkpoint Detection and Validation

Purpose: Ensures accurate lap completion through sequential checkpoint validation using 3D spatial calculations.

Core Algorithm:

- generateCheckpoints(for:): Creates checkpoint gate entities at predefined 3D coordinates based on track type
- Each checkpoint has a position (SIMD3<Float>), type (start/finish or intermediate), and detection radius
- updateCameraPosition(): Continuously tracks camera position from ARSession frame updates
- checkCheckpointCollision(): Calculates 3D Euclidean distance between camera and each checkpoint
- validateCheckpointSequence(): Ensures checkpoints are crossed in correct order (prevents skipping)

Distance Calculation Formula: $\text{distance} = \text{sqrt}((x2-x1)^2 + (y2-y1)^2 + (z2-z1)^2)$

When distance < threshold (typically 2-3 meters), a checkpoint passage is registered. The system then checks if it's the next expected checkpoint in sequence. If valid, it triggers visual feedback and updates lap progress.

System 4: High-Precision Timing System

Purpose: Provides millisecond-accurate lap timing with real-time updates and historical tracking.

Implementation Details:

- Timer frequency: 16ms (approximately 60 updates per second, matching display refresh rate)
- Uses Timer.publish() with RunLoop.main for accurate time intervals
- Stores lap start time using Date() for high-precision time measurement
- Calculates elapsed time using TimeInterval (Double) for millisecond precision

Key Methods:

- startLapTimer(): Initializes timer and records start time

- updateCurrentTime(): Continuously calculates elapsed time since lap start
- completeLap(): Records lap time, compares with best lap, stores in lap history array
- formatTime(interval:): Converts TimeInterval to formatted string (MM:SS.mmm)
- updateBestLap(): Compares current lap with stored best lap and updates if faster

System 5: AR Session Management

Purpose: Configures and maintains the ARKit session, handling world tracking, plane detection, and scene reconstruction.

Configuration:

- ARWorldTrackingConfiguration: Enables 6DOF (degrees of freedom) camera tracking
- Plane detection: .horizontal and .vertical for comprehensive surface detection
- Scene reconstruction: .mesh for detailed environment geometry (Mania mode)
- Camera position tracking: Real-time updates from ARFrame

Delegate Methods:

- session(_:didUpdate:): Processes frame updates for camera position tracking
- session(_:didAdd:): Handles new mesh anchors during room scanning
- session(_:didUpdate:): Updates existing anchors as environment understanding improves
- session(_:didFailWithError:): Error handling for AR session interruptions

Data Models and Enumerations

RaceType Enumeration

Defines three distinct racing modes with different objectives:

- grandPrix: Multi-lap competitive racing with lap count targets
- timeTrial: Focus on achieving the fastest single lap time
- practice: Free-form practice mode with no lap limits

TrackType Enumeration

Manages different track configurations with associated properties:

- monaco: High-detail Monaco GP circuit (932 KB USDZ, 1 checkpoint)
- lowPoly: Performance-optimized circuit (155 KB USDZ, 6 checkpoints)
- mania: Procedurally generated track (no pre-built model, 8 dynamic checkpoints)

Each track type includes computed properties for USDZ filename and checkpoint count, enabling dynamic track loading based on user selection.

Checkpoint Data Structure

Represents individual checkpoint gates with the following properties:

- position: SIMD3<Float> - 3D coordinates in AR world space
- checkpointType: Enum (startFinish or intermediate) - Determines gate color and behavior
- detectionRadius: Float - Threshold distance for collision detection (typically 2.0-3.0 meters)
- visualEntity: RealityKit Entity - The 3D representation of the gate in AR space

Features and Technical Methods

Race Mode Implementation

Grand Prix Mode

Technical Implementation: Multi-lap race with predefined lap count (typically 3-5 laps). The system tracks total laps completed and automatically ends the race upon completion.

Key Methods:

- startGrandPrix(lapCount:): Initializes race with specified lap target
- checkLapCompletion(): Validates if current lap equals target lap count
- displayRaceProgress(): Shows current lap / total laps in HUD

Time Trial Mode

Technical Implementation: Focuses exclusively on achieving the fastest single lap. The system continues running until manually stopped, constantly comparing new laps against the best recorded time.

Key Methods:

- startTimeTrial(): Initializes infinite lap mode
- compareLapTimes(current:best:): Compares lap times and displays notifications
- showNewBestLapNotification(): Triggers visual and audio feedback for new records

Practice Mode

Technical Implementation: Free-form racing without lap limits or competition. Ideal for learning track layouts and practicing racing lines.

Key Methods:

- startPracticeMode(): Initializes unrestricted racing session
- trackLapStatistics(): Records lap times for analysis without competition logic

Orientation Management System

Dynamic Orientation Switching

The app dynamically switches between portrait and landscape orientations based on racing state, providing optimal viewing experiences for different contexts.

Portrait Mode (Default and Post-Race):

- Comprehensive HUD with detailed race statistics
- Current lap and total lap display
- Real-time timer with millisecond precision
- Checkpoint progress indicator
- Best lap time tracking and comparison
- Full race control buttons (Reset, Stop, Start)

Landscape Mode (During Racing):

- Immersive cockpit-style racing view
- Minimal HUD for maximum visibility
- Top-left: Compact lap counter and current time
- Top-right: Best lap time and track information
- Center: Checkpoint/lap completion messages
- Bottom-right: Quick access race controls

AR Room Scanning (Mania Mode)

Purpose: Mania mode creates procedurally generated race tracks by scanning the physical environment and using real-world geometry as the track foundation.

Technical Process:
- ARKit's scene reconstruction generates a 3D mesh of the environment using the LiDAR sensor
- Mesh anchors are continuously updated and processed through the ARSession delegate
- The system analyzes mesh geometry to identify suitable paths for track placement
- Checkpoints are dynamically generated and positioned along the procedural track path
- The system ensures proper spacing and sequence for checkpoint validation

Key Methods:
- startRoomScanning(): Initializes ARSession with scene reconstruction enabled
- processMeshAnchors(_:): Analyzes mesh geometry for track suitability
- generateProceduralTrack(): Creates track path based on mesh data
- placeCheckpointsOnPath(path:): Distributes 8 checkpoints along generated path

Performance Optimization Techniques

Memory Management:
- Entities are properly removed from parent scene graph on cleanup, preventing memory leaks
- AR anchors are explicitly removed and dereferenced when no longer needed
- Timers are invalidated and deallocated when races complete to prevent background processing
- USDZ models are loaded asynchronously to prevent blocking the main thread

Rendering Optimization:
- Efficient checkpoint collision detection using distance calculations instead of complex physics
- Minimal UI updates during racing to maintain high frame rates
- Optimized 3D model poly counts (Low Poly track: only 155KB)
- Material sharing for checkpoint gates reduces draw calls

- Procedural generation in Mania mode is optimized to complete within 2-3 seconds

How to Replicate Our Project

Prerequisites and Environment Setup

Hardware Requirements:
- Mac computer running macOS 13.0 (Ventura) or later
- iOS device with A12 Bionic chip or later (iPhone XS/XR or newer)
- LiDAR sensor recommended for enhanced AR (iPhone 12 Pro or newer)
- USB-C or Lightning cable for device connection
- Minimum 4GB RAM on Mac (8GB recommended)

Software Requirements:
- Xcode 15.0 or later (free download from Mac App Store)
- iOS 18.0 SDK (included with Xcode 15.0+)
- Swift 5.0 or later (included with Xcode)
- Apple Developer account (free for personal development, $99/year for App Store distribution)
- Git version control system (pre-installed on macOS)

Step-by-Step Installation Guide

Step 1: Install Xcode
- Open Mac App Store
- Search for 'Xcode'
- Click 'Get' and then 'Install' (requires ~15GB disk space)
- Wait for installation to complete (may take 30-60 minutes)
- Open Xcode and agree to license agreement
- Xcode will install additional components automatically

Step 2: Clone the Repository
- Open Terminal application (found in /Applications/Utilities/)
- Navigate to desired directory: cd ~/Documents
- Clone the repository: git clone https://github.com/SbbarB/SbbarB.github.io.git
- Navigate to the KartAR directory: cd SbbarB.github.io/KartAR

Step 3: Open the Project

- From Terminal, open the Xcode project: open KartAR.xcodeproj
- Alternatively, double-click KartAR.xcodeproj in Finder
- Wait for Xcode to load and index the project (may take 1-2 minutes)

Step 4: Configure Signing and Capabilities

- In Xcode, click on 'KartAR' project in the left sidebar
- Select 'KartAR' target under TARGETS
- Click 'Signing & Capabilities' tab
- Check 'Automatically manage signing'
- Select your Apple Developer account from 'Team' dropdown
- If no team appears, click 'Add Account' and sign in with Apple ID
- Change Bundle Identifier to make it unique (e.g., com.yourname.KartAR)
- Verify required capabilities are enabled: Camera access and ARKit support

Step 5: Connect and Trust iOS Device

- Connect your iPhone/iPad to Mac using cable
- Unlock your iOS device
- If prompted on device, tap 'Trust This Computer'
- Enter device passcode if requested
- In Xcode toolbar at top, click device dropdown (near Run button)
- Select your connected iOS device from list

Step 6: Build and Run

- Click the Run button (play icon) in Xcode toolbar, or press ⌘R
- Xcode will build the project (first build takes 2-5 minutes)
- Watch build progress in top center of Xcode window
- If first-time developer on device, go to Settings > General > Device Management on iOS device
- Tap your Apple ID under 'Developer App' and tap 'Trust [Your Apple ID]'
- Return to Xcode and run again
- App will launch on your device automatically
- Grant camera permissions when prompted

Troubleshooting Common Issues

Issue: 'No devices connected' or device not appearing
Solutions:

- Unplug and reconnect the device
- Restart Xcode
- Restart your iOS device
- Try a different USB cable or port
- Check cable supports data transfer (not just charging)

Issue: Build errors related to signing
Solutions:

- Ensure Bundle Identifier is unique (change from default)
- Verify Team is selected in Signing & Capabilities
- Try unchecking and rechecking 'Automatically manage signing'
- Sign out and back into Apple account in Xcode preferences

Issue: 'ARKit not available' runtime error
Solutions:

- Verify device has A12 Bionic chip or later
- Ensure iOS version is 18.0 or later (Settings > General > About)
- ARKit cannot run in simulator - must use physical device

Issue: Track won't place in AR view
Solutions:

- Ensure adequate lighting in your environment
- Move device slowly to help ARKit detect surfaces
- Point camera at textured surfaces (avoid blank walls)
- Avoid reflective surfaces like mirrors or glass
- Wait for yellow plane detection indicators to appear

Testing and Deployment

Testing the App:

- Test all three race modes (Grand Prix, Time Trial, Practice)
- Verify each track loads correctly (Monaco, Low Poly, Mania)
- Check orientation switching between portrait and landscape
- Validate checkpoint detection by driving through gates
- Test lap timing accuracy with a stopwatch
- Verify best lap tracking across multiple laps
- Test in different lighting conditions and environments
- Try Mania mode room scanning in various room sizes

Deployment Options:

Option 1: Personal Use (Free)

- Run directly from Xcode to your device (no special requirements)

- App remains installed for 7 days before requiring rebuild
- Suitable for personal development and testing

Option 2: TestFlight Beta Testing

- Requires $99/year Apple Developer Program membership
- Upload builds to App Store Connect
- Invite up to 10,000 external testers
- 90-day build expiration period

Option 3: App Store Release

- Requires $99/year Apple Developer Program membership
- Submit app for App Store review
- Must comply with App Store Review Guidelines
- Provide app description, screenshots, and privacy policy
- Review typically takes 24-48 hours

How to Replicate Our Project

1. How We Built the Project

Development Approach: KartAR was developed using an iterative, feature-driven approach with a focus on creating an immersive augmented reality racing experience. The project leverages Apple's cutting-edge ARKit and RealityKit frameworks to bridge the physical and digital worlds.

Technology Stack Selection

Primary Technologies:

- Swift 5.0+ - Apple's modern, type-safe programming language chosen for iOS development
- SwiftUI - Declarative UI framework for building responsive, adaptive interfaces
- ARKit - Apple's augmented reality framework for world tracking, plane detection, and spatial awareness
- RealityKit - High-performance 3D rendering engine optimized for AR experiences
- Metal - Low-level graphics API for efficient GPU rendering

Rationale: These technologies were selected because they provide native integration with iOS hardware capabilities, including the LiDAR sensor and advanced camera systems. ARKit's world tracking and scene reconstruction features are essential for creating realistic AR experiences, while RealityKit offers optimized 3D rendering performance.

Development Phases

Phase 1: Foundation and Core AR Implementation
We began by establishing the foundational AR infrastructure:

- Setting up the ARKit session with world tracking configuration
- Implementing plane detection for both horizontal and vertical surfaces
- Creating the camera position tracking system for real-time spatial awareness
- Developing the mesh anchor processing system for room scanning (Mania mode)

Phase 2: 3D Asset Integration
We integrated high-quality 3D racing track models in USDZ format:

- Monaco GP Track (932 KB) - A detailed replica of the iconic circuit
- Low Poly Circuit (155 KB) - A performance-optimized stylized track
- Implemented asynchronous model loading to prevent UI blocking
- Created anchor-based placement system for positioning tracks on detected surfaces

Phase 3: Game Logic and Checkpoint System
The racing mechanics were built around a sophisticated checkpoint validation system:

- Designed checkpoint gates with 3D collision detection using distance calculations
- Implemented sequential validation to ensure proper lap completion
- Created visual feedback system with color-coded gates (green for start/finish, orange for intermediate)
- Developed the lap completion detection algorithm

Phase 4: Timing System and Race Modes
We implemented a high-precision timing system with millisecond accuracy:

- Created a 16ms update interval timer for real-time lap tracking
- Developed lap comparison algorithms for best lap tracking
- Implemented three distinct race modes: Grand Prix, Time Trial, and Practice
- Created formatted time display system (MM:SS.mmm)

Phase 5: User Interface and Orientation Management
The final phase focused on creating an adaptive, immersive user experience:

- Developed portrait mode with comprehensive HUD showing all race statistics
- Created landscape cockpit view for immersive racing experience
- Implemented dynamic orientation locking system that responds to race state
- Built responsive UI components that adapt to different screen orientations

Future Work

Where Our Design Could Lead (Possible Use Cases)

Our system opens pathways to multiple applications beyond entertainment. By combining telepresence, robotics, and AR visualization, the platform could evolve into solutions for:

- Education & STEM Learning — Students could remotely explore miniature environments to learn about physics, navigation, robotics, or spatial reasoning through hands-on play.
- Telepresence Exploration — Users could drive into spaces that are difficult or unsafe to physically enter, such as disaster sites, industrial zones, or research environments.
- Remote Social Interaction — People could visit others' homes, museums, or events through physical robots controlled via AR, experiencing a more embodied presence than video calls.
- Rehabilitation & Accessibility — Individuals with limited mobility could experience movement and physical navigation through proxy devices to enhance agency and joy.
- Theme Park & Museum Experiences — Interactive miniature exhibits where visitors become "tiny drivers" navigating fictional worlds.
- Mixed Reality Esports / Competitions — Large-scale multiplayer FPV racing leagues with custom environments and augmented overlays.

Future Expansion (If We Had More Time)

If this project continued, we would focus on expanding the system across both hardware and software dimensions. Planned improvements include:

- Real-Time Control through AR Gestures Replace physical remote control with hand-tracking steering and acceleration inside the Meta Quest 3 interface.
- Custom AR Overlays & Game Elements Add digital checkpoints, hazards, power-ups, or time trials visible only through the AR headset.
- Multi-Player Racing Support multiple RC vehicles racing on the same track with live AR scoring.
- Higher-Quality Camera System Use a lightweight FPV camera and transmitter for lower latency and better stability than phone streaming. Integrated Headset App Build a dedicated application for Meta Quest rather than relying on external screen-casting.
- Improved Vehicle Design CAD-designed mount, 3D-printed chassis, better balancing, and swappable parts.

- Environment-Responsive Gameplay Sensor-based interactions (e.g., IR checkpoints, RFID tags, or computer vision-based obstacles).
- Safety & Durability Testing Evaluate collision resistance, battery life, frame rate consistency, and network performance under extended use.

Conclusion

This project demonstrates the creative potential of combining mixed reality, robotics, and telepresence to transform a simple RC car into an immersive first-person racing experience. By merging physical play with AR visualization through the Meta Quest 3, we built a system that challenges traditional boundaries between virtual and real environments. Our FPV racing platform showcases how playful hardware hacking and rapid prototyping can lead to novel interaction concepts that feel surprising, engaging, and future-oriented.

While our current prototype focuses on a single-player experience and foundational AR integration, the design opens a pathway toward richer applications—from education to accessibility, entertainment, and remote exploration. With additional development, multiplayer racing, real-time AR overlays, gesture-based control, and improved hardware could expand this platform into a fully interactive telepresence gaming ecosystem.

Ultimately, our work highlights that innovation does not always require complex technology—sometimes, combining simple tools with curiosity and creativity can result in something unexpectedly exciting.

Summary of Important Links

- Github Code
- Presentation
- Demo Video

AI Usage Disclosure Statement

This project utilized artificial intelligence (AI) tools to support its development. Specifically, AI was used for tasks including (but not limited to) content generation, code assistance, data analysis, written summaries, taking notes, and/or design optimization. All AI-generated outputs were reviewed and refined by human contributors to ensure accuracy, quality, and alignment with the project's goals. We are committed to transparency and responsible use of AI technologies.