System Design: iPhone Prototype

Simulated Projectile Hit Detection

1 Overview

This document describes the system architecture for prototyping the hit-detection system on an iPhone. The iPhone provides all required hardware in a single device: a camera capable of 60 fps capture with exposure control, a 6-axis inertial measurement unit (gyroscope + accelerometer) sampled at up to 100 Hz, a Neural Engine for real-time machine-learning inference, and ARKit—Apple's visual-inertial odometry framework that fuses camera and IMU data into a world-space tracking solution.

The phone is mounted on the gun with its rear camera aligned along the barrel axis. A Bluetooth Low Energy (BLE) button wired into the trigger mechanism provides the fire event.

2 Physical Configuration

The iPhone is rigidly attached to the gun such that the rear camera's optical axis is parallel to and aligned with the barrel. The camera's forward direction (the $-z$ axis of the phone's camera coordinate system) corresponds to the barrel direction $\hat{\boldsymbol{b}}$.

The BLE trigger button is a simple momentary switch paired to the phone via CoreBluetooth. When pressed, it sends an event that the app timestamps and uses to initiate the ballistic calculation.

3 Software Architecture

The application is structured as five layers, each mapping to one or more Apple frameworks.

3.1 Layer 1: ARKit Session (Sensor Fusion)

Framework: ARKit (`ARSession`, `ARWorldTrackingConfiguration`)

An `ARSession` configured for world tracking is the foundation of the system. At each frame (60 Hz), ARKit delivers an `ARFrame` object containing:

- The captured camera image as a `CVPixelBuffer`.
- The device's 6DoF pose as a 4×4 homogeneous transform matrix `ARFrame.camera.transform`, expressed in a stable world coordinate frame. This is the matrix $M(t_i)$ from the mathematical model.
- The camera intrinsics as a 3×3 matrix via `ARFrame.camera.intrinsics`, whose diagonal entries are the focal lengths f_x, f_y in pixels and whose off-diagonal entries give the principal point (u_0, v_0). No manual calibration is needed.

ARKit internally performs visual-inertial SLAM: it fuses IMU readings (gyroscope and accelerometer) with visual feature tracking across frames to maintain a drift-corrected world-frame pose. This eliminates the need to implement gyroscope bias correction, gravity subtraction, quaternion integration, or drift compensation.

Extracting barrel state. From the 4×4 transform $T(t_i)$:

- The barrel direction $\hat{\boldsymbol{b}}$ is the third column of the 3×3 rotation submatrix (the camera's forward axis, negated per Apple's convention that $-z$ is forward).
- The barrel angular velocity $\boldsymbol{\omega}$ is obtained by numerically differentiating the rotation between consecutive frames, or by supplementing with direct `CMMotionManager` gyroscope readings at a higher rate if needed.
- The muzzle position $\boldsymbol{r}_m$ is the translation column of $T(t_i)$ plus a fixed offset (calibrated once) along $\hat{\boldsymbol{b}}$ from the phone's position to the muzzle tip.

3.2 Layer 2: Vision / CoreML (Target Detection)

Frameworks: Vision (`VNCoreMLRequest`), CoreML

Each `ARFrame`'s pixel buffer is passed to a Vision request that locates the target in the image. Two approaches are available, depending on the target type:

Colour segmentation (simple targets). For high-contrast targets against sky (e.g. a bright orange disc), a `CIFilter`-based colour threshold followed by `VNDetectContoursRequest` extracts the target contour. A bounding circle fit yields the centroid (u_i, v_i) and apparent diameter s_i.

Neural detector (complex scenes). For robustness to varying lighting, orientation, and background, a lightweight object detector (MobileNet-SSD or YOLOv8-nano) is trained on a small dataset of the target (20–50 images) and exported to CoreML format. A `VNCoreMLRequest` runs the model on the Neural Engine, returning a bounding box from which (u_i, v_i, s_i) are computed. Inference time is typically 3–8 ms on the Neural Engine.

Output. Regardless of method, the detection layer produces, per frame:

- Target centroid (u_i, v_i) in pixel coordinates.
- Apparent diameter s_i in pixels.
- A detection confidence score (used by the Kalman filter to weight the observation).

3.3 Layer 3: Kalman Filter (State Estimation)

Framework: Custom implementation using `simd` types (`simd_float3`, `simd_float3x3`)

The Kalman filter maintains the target's estimated state $(\boldsymbol{r}, \boldsymbol{v}, \boldsymbol{a})$ in the inertial (world) frame. Its operation per frame is:

1. **Observation construction.** From the detection output (u_i, v_i, s_i) and the camera intrinsics, compute the distance $d_i = Sf/s_i$ and the unit ray direction $\hat{\boldsymbol{e}}_i$. Form the barrel-frame position $\boldsymbol{r}_i^b = d_i\, \hat{\boldsymbol{e}}_i$.
2. **Inertial transform.** Rotate into the world frame using ARKit's transform: $\boldsymbol{r}_i = M(t_i)\, \boldsymbol{r}_i^b$.
3. **Predict step.** Propagate the state estimate forward by Δt using the constant-acceleration process model (Eq. 6 of the mathematical model).
4. **Update step.** Incorporate the observation $\boldsymbol{r}_i$, weighted by the detection confidence and the measurement noise covariance, to refine the state estimate.

The filter's 9×9 state covariance matrix and the predict/update steps map naturally to `simd_float3x3` block operations. The entire filter update is approximately 50 lines of Swift.

3.4 Layer 4: Ballistics Engine (Hit Calculation)

Framework: Custom implementation using `simd` types

This layer executes once, at trigger time t_0. It implements Sections 4–7 of the mathematical model:

1. **Read state.** Extract the Kalman filter's current estimate $(\hat{\boldsymbol{r}}_0, \hat{\boldsymbol{v}}, \hat{\boldsymbol{a}})$.
2. **Projectile initial conditions.** Compute $\boldsymbol{p}_0 = \boldsymbol{r}_m$ and $\boldsymbol{V} = v_p\,\hat{\boldsymbol{b}} + \boldsymbol{\omega} \times \boldsymbol{r}_m$ from the ARKit pose and the barrel's angular velocity at t_0.
3. **Relative state.** Compute $\boldsymbol{\delta}_0$, $\boldsymbol{\delta}_v$, $\boldsymbol{\delta}_a$.
4. **Intercept time.** Compute the scalar coefficients α, β, γ and solve the quadratic for τ^*.
5. **Miss distance.** Evaluate $\rho = \|\boldsymbol{\Delta}(\tau^*)\|$ and the spread radius $R(v_p\,\tau^*)$.
6. **Hit test.** Return HIT if $\rho \le R(v_p\,\tau^*)$ and $v_p\,\tau^*$ is within the maximum effective range; MISS otherwise.

The entire computation is a handful of dot products, a square root, and a comparison. It completes in microseconds.

3.5 Layer 5: Trigger Input and Output

Frameworks: CoreBluetooth (`CBCentralManager`, `CBPeripheralDelegate`), CoreHaptics, AVFoundation

Trigger input. A BLE momentary button paired to the phone sends a characteristic-change notification when pressed. The app's `CBPeripheralDelegate` receives this event, timestamps it against the most recent `ARFrame`, and invokes the ballistics engine.

Feedback output. The hit/miss result is communicated to the shooter via:

- **Haptic:** A sharp tap (hit) or a long buzz (miss) via CoreHaptics.
- **Audio:** A distinct tone for hit vs. miss via AVFoundation.
- **Visual (optional):** An AR overlay on the camera feed showing the predicted impact point and spread circle, rendered via ARKit's `SCNNode` or RealityKit entity.

4 Framework Summary

Pipeline Step	Framework	Key API
Camera frames	ARKit	`ARFrame.capturedImage`
Device pose / $M(t)$	ARKit	`ARFrame.camera.transform`
Intrinsics (f, u_0, v_0)	ARKit	`ARFrame.camera.intrinsics`
Target detection	Vision + CoreML	`VNCoreMLRequest`
Matrix / vector math	simd / Accelerate	`simd_float3`, `simd_float3x3`
Kalman filter	Custom (simd)	$\sim$50 lines of Swift
Trigger event	CoreBluetooth	`CBPeripheralDelegate`
Hit/miss feedback	CoreHaptics + AV	Haptic pattern + audio cue

5 Data Flow Timing

The per-frame pipeline operates within the 16.7 ms budget of 60 fps:
The trigger-time ballistic computation (Layer 4) adds negligible latency (< 0.01 ms).

Step	Time
ARKit frame delivery	< 1 ms
Vision/CoreML detection	3–8 ms
Geometry + inertial transform	< 0.1 ms
Kalman filter update	< 0.1 ms
Total per frame	**4–10 ms**

6 Development Sequence

The prototype should be built incrementally, with each stage independently testable:

1. **ARKit tracking verification.** Run an ARKit session and display the device's orientation and position overlaid on the camera feed. Verify that tracking remains stable during fast swings in an outdoor environment.

2. **Target detection.** Add the Vision/CoreML detection pipeline and draw a bounding box around the detected target on the camera feed. Verify detection at various ranges and lighting conditions.

3. **Kalman filter integration.** Wire in the Kalman filter and display the estimated target state $(\boldsymbol{r}, \boldsymbol{v}, \boldsymbol{a})$ in real time. Verify that the velocity and acceleration estimates are physically reasonable.

4. **Trigger and ballistics.** Pair the BLE button, implement the ballistics engine, and display the hit/miss result on screen. Test against a target with known trajectory (e.g. a ball thrown on a predictable arc).

5. **Feedback and polish.** Add haptic and audio feedback. Optionally add an AR overlay showing the predicted impact point and spread radius.

7 Calibration Requirements

The following quantities must be measured or configured before use:

- **Target physical diameter** S: measured once per target type.
- **Phone-to-muzzle offset**: the displacement vector from the phone's position (as reported by ARKit) to the muzzle tip, expressed in the phone's local frame. Measured once when the phone is mounted.
- **Projectile parameters**: muzzle speed v_p and the spread function $R(\ell)$, specified per ammunition type.
- **Camera intrinsics**: provided automatically by ARKit—no manual calibration needed.

8 Limitations and Risks

- **ARKit tracking under fast motion.** ARKit's visual-inertial tracking can degrade during very rapid swings if the image is too motion-blurred for feature extraction. Mitigation: force a short camera exposure time via `AVCaptureDevice` (this is compatible with ARKit). Outdoor scenes with strong visual features (horizon, trees, structures) also help.
- **Target detection at long range.** At distances beyond $\sim$40 m, the target may occupy very few pixels, making detection and size estimation less reliable. Mitigation: use a narrower-FOV lens attachment (clip-on telephoto), or restrict the prototype to shorter ranges initially.
- **BLE latency.** Bluetooth Low Energy has a connection interval of 7.5–30 ms, introducing a small delay between trigger pull and event receipt. For a prototype this is acceptable; a wired trigger (via the Lightning/USB-C port) would eliminate it.
- **Constant-acceleration assumption.** The ballistic model assumes constant target acceleration over the projectile's flight time. This is well-justified for clay pigeons and most birds over the 0.05–0.2 s flight window, but may break down for targets executing sharp manoeuvres.

A Alternate Architecture: Raw CoreMotion + AVFoundation

For applications requiring lower latency or finer control over sensor timing, the ARKit layer can be replaced with direct sensor access. This section outlines the changes.

A.1 Camera: AVFoundation

Replace the ARKit session with an `AVCaptureSession` configured for 60 fps video capture. Use `AVCaptureVideoDataOutput` to receive each frame as a `CVPixelBuffer` with a precise timestamp.

Camera intrinsics are not provided automatically in this configuration. They must be obtained either by a one-time calibration procedure (e.g. OpenCV's checkerboard calibration, results stored in the app) or by querying `AVCaptureDevice`'s lens intrinsics if available on the device.

A.2 IMU: CoreMotion

Use `CMMotionManager` to receive gyroscope and accelerometer data at up to 100 Hz. The gyroscope provides the angular velocity $\boldsymbol{\omega}(t)$ directly.

The rotation matrix $M(t)$ must be maintained manually by integrating $\boldsymbol{\omega}$. Apple's `CMDeviceMotion.attitude` (available via the sensor fusion mode of CoreMotion) provides a quaternion-based orientation estimate that handles gyroscope bias and gravity, but it does not include visual correction and will drift over time.

A.3 Sensor Synchronization

In the ARKit architecture, camera frames and IMU data are synchronized internally. With raw sensors, the developer must align timestamps between `AVCaptureOutput` callbacks and `CMDeviceMotion` updates. Both use `CMClockGetHostTimeClock` as their time base, so direct comparison is valid, but interpolation of IMU readings to camera frame times is the developer's responsibility.

A.4 Tradeoffs

	ARKit	**Raw Sensors**
Sensor fusion	Automatic	Manual
Drift correction	Visual-inertial	Gyro-only (drifts)
Camera intrinsics	Automatic	Manual calibration
Frame-IMU sync	Automatic	Manual
Latency	$\sim$1 frame	Sub-frame possible
Development effort	Low	High

The raw-sensor approach is recommended only if profiling reveals that ARKit's processing pipeline introduces unacceptable latency, or if the application must run on a device that does not support ARKit world tracking.