Technical Documentation: Hogan's Alley VR

Enrique Bellido & Iñaky Ordiales

January 2026

1 Introduction

Hogan's Alley is a classic arcade shooter released by Nintendo in 1984. It was a pioneer in using the "NES Zapper" light gun, a peripheral that allowed players to physically aim at the television screen. The core gameplay loop challenged players to react instantly to pop-up targets, requiring them to distinguish between armed criminals and innocent bystanders in a simulated police training environment.

This mechanic makes *Hogan's Alley* an ideal candidate for an Extended Reality (XR) adaptation. The original game's design—standing in a fixed position and rotating to aim—maps perfectly to 3-DOF (Three Degrees of Freedom) VR. By utilizing a smartphone's gyroscope, we can replicate the physical sensation of the light gun without requiring complex locomotion systems. This approach eliminates the risk of motion sickness common in VR movement while preserving the immersion of physical aiming, effectively turning the mobile device into a "Magic Window" into the virtual firing range.

2 1. Design Rationale

2.1 VR vs. AR Implementation

The decision to implement this project as a **Virtual Reality (VR)** experience (specifically 3-DOF "Magic Window") rather than Augmented Reality (AR) was driven by the need for **environmental immersion**.

- **Environmental Control:** "Hogan's Alley" relies on a specific atmosphere—a police training range with specific cover points (walls, windows). AR would project enemies into the user's uncontrolled real-world room, breaking the "Alley" theme.
- **Field of View:** A VR approach allows us to control the lighting and background completely, ensuring the targets are always visible against a contrasted background, which is critical for a reflex shooter.

2.2 Scene Architecture Strategy

We utilized a multi-scene approach (`Menu`, `Game`, `GameOver`) rather than a single scene with UI overlays.

- **State Hygiene:** Reloading the `GameScene` guarantees a complete reset of all variables, object pools, and timers without complex "Reset()" functions.
- **Memory Management:** Assets required only for the menu are unloaded during gameplay, optimizing performance on mobile devices.

3 2. Project Structure & Assets

3.1 Game Objects and Components

- **Main Camera (The Player):** Configured for 3-DOF tracking. It acts as the "head" of the player.
- **Target Prefabs (Billboarding):**
  - We used 2D Sprites in a 3D world to mimic the original 1984 aesthetic.
  - **Orientation:** Scripts force sprites to always face the camera (‘transform.LookAt‘), ensuring they don't look like flat paper when the player rotates.
- **Spawn Points:** Invisible spatial anchors that dictate where targets can appear, allowing us to easily redesign the level layout without changing code.

4 3. VE Mechanics & Interaction

4.1 Locomotion (3-DOF Magic Window)

Locomotion is restricted to **Rotational Movement** only. This mimics the experience of standing in a fixed booth at a shooting range.

- **Mobile Implementation:** We access the `AttitudeSensor` (Gyroscope). By remapping the sensor's quaternion to Unity's coordinate system, the device screen becomes a direct window into the virtual world.
- **PC Implementation:** Standard Mouse-Look logic (‘Input.GetAxis‘) provides a fallback for testing and desktop play.

4.2 Shooting Mechanics: Raycast vs. Projectiles

We chose **Raycasting (Hitscan)** over physical projectiles for three reasons:

1. **Responsiveness:** Arcade shooters require instant feedback. Raycasts resolve immediately in the same frame, whereas rigidbodies introduce travel time and physics calculation latency.
2. **Precision:** Raycasts travel infinitely straight from the center pixel, perfectly simulating a "laser sight" mechanic essential for reflex games.
3. **Performance:** Avoiding physics calculations for bullets reduces CPU load on mobile devices.

4.3 Collision Detection Logic

The interaction relies on **BoxColliders** attached to the target sprites.

- **Hierarchy Handling:** Since the graphical model might be a child of an empty parent (the "root" of the prefab), the shooting script checks both the hit object and its parent for the ‘npcKilled‘ component. This ensures that shooting the "arm" or "head" of a sprite still registers as a kill for the main entity.

5 4. Developed Scripts

5.1 CameraController.cs

Manages the player's perspective. It detects the platform (Mobile vs PC) and switches between Gyroscope reading and Mouse Delta reading. This abstraction allows the same build to work on both simulators and physical devices.

```
void Update() {
    #if UNITY_STANDALONE || UNITY_EDITOR
        // PC: Read Mouse Delta
        float mouseX = Mouse.current.delta.x.ReadValue() * sensitivity;
        yRotation += mouseX;
        transform.localRotation = Quaternion.Euler(xRotation, yRotation, 0f);

    #elif UNITY_IOS || UNITY_ANDROID
        // Mobile: Read Gyroscope
        if (AttitudeSensor.current != null) {
            Quaternion q = AttitudeSensor.current.attitude.ReadValue();
            // Remap coordinate system for Unity
            transform.localRotation = Quaternion.Euler(90, 0, 0) * new Quaternion(q.x, q.y, -q.z, -q.w);
        }
    #endif
}
```

5.2 PlayerShooter.cs

Handles the firing input. It abstracts the trigger (Mouse Click vs Touch Tap) and performs the Raycast. It includes logic to debug the ray trajectory and validate the hit target hierarchy.

```
void Shoot() {
    // Cast ray from viewport center (0.5, 0.5)
    Ray ray = Camera.main.ViewportPointToRay(new Vector3(0.5f, 0.5f, 0));
    RaycastHit hit;

    if (Physics.Raycast(ray, out hit)) {
        // Check for component on hit object OR parent
        npcKilled target = hit.transform.GetComponent<npcKilled>();
        if (target == null && hit.transform.parent != null)
             target = hit.transform.parent.GetComponent<npcKilled>();

        if (target != null) target.killed();
    }
}
```

5.3 npcManager.cs

The central orchestrator. It manages the Coroutine loop for spawning (‘SpawnGroupRoutine‘), handles the "Race Condition" so rounds don't overlap, and stores the final score in ‘PlayerPrefs‘ before switching scenes.

```
IEnumerator SpawnGroupRoutine() {
    // 1. Select random enemies and locations
    List<GameObject> group = SelectRandomGroup();

    // 2. Wait for player input OR timeout (3 seconds)
    float elapsed = 0f;
    while (elapsed < 3f && !npcClickedInThisTurn) {
        elapsed += Time.deltaTime;
        yield return null;
    }

    // 3. Race Condition Fix: Wait if a hit is currently processing
    if (npcClickedInThisTurn) {
        yield return new WaitUntil(() => isProcessingHit == false);
    } else {
        foreach (GameObject npc in group) npc.SetActive(false);
    }
}
```

5.4 npcKilled.cs

A bridge component. It resides on the target prefabs and forwards the "I have been shot" event back to the main Manager, passing its own GameObject reference so the Manager knows if it was an Enemy or Innocent.

```
public class npcKilled : MonoBehaviour {
    public npcManager Manager;

    public void killed() {
        // Notify manager to update score and hide this specific NPC
        Manager.OnNpcClicked(transform.gameObject);
    }
}
```

6 5. Play Instructions

1. **Device:** Launch on iPhone (Landscape) or PC.
2. **Aiming:** Rotate the physical device (Mobile) or move the mouse (PC).
3. **Rules**
   - Shoot **Armed Bad Guys** (+1 Point).
   - Do NOT shoot **Civilians/Cops** (-1 Point).
4. **Loop:** Targets appear for 3 seconds. The round ends when time runs out (40s).