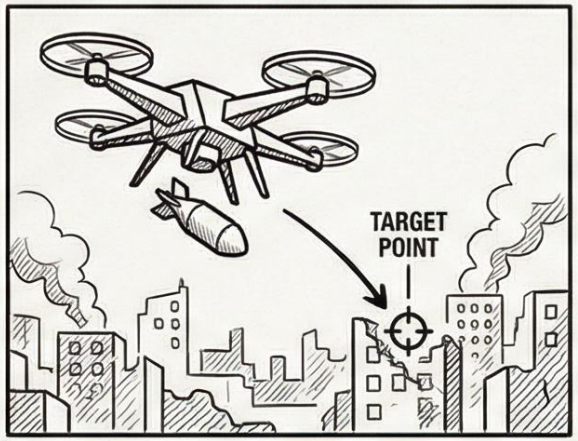
God's Eyes' View

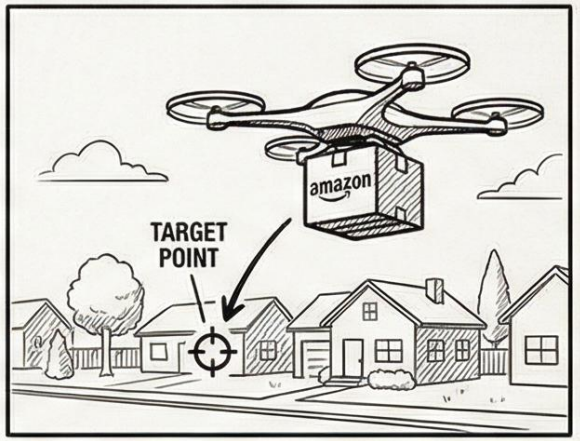
GAME CONTROL: SYNCHRONIZED DRONE OPERATIONS

3-PLAYER COOPERATIVE SYSTEM

LEFT SCREEN (WAR ZONE)

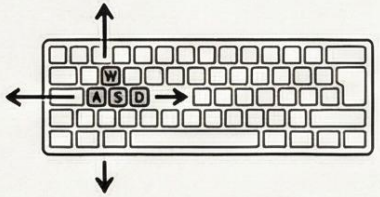


RIGHT SCREEN (RESIDENTIAL AREA)



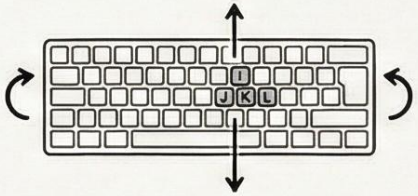
PLAYER 1 (LEFT) CONTROLS: W, A, S, D

- HORIZONTAL MOVEMENT (NO ROTATION)

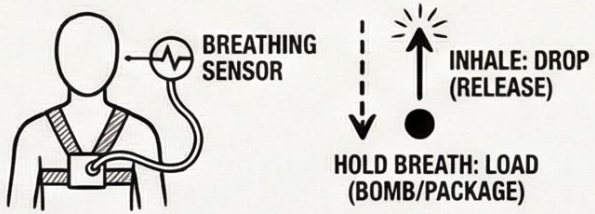


PLAYER 2 (RIGHT) CONTROLS: I, J, K, L

- VERTICAL MOVEMENT & ROTATION



PLAYER 3 (BREATHING) TRIGGER MECHANISM



NOTE: ALL THREE PLAYERS MUST WORK TOGETHER TO ALIGN AND DROP SIMULTANEOUSLY ON TARGETS. CYCLE REPEATS.

Domestic Tension

Wafaa Bilal "Domestic Tension," 2007

The Asymmetry of Violence

The decision makers

– Corporate Executives, Engineers, Military strategists, Drone operators

–Treating war and logistics solely as ‘abstract data’

– Never exposed to physical danger

– Monopolizing profit and safety

The decision followers

– Warehouse workers, Delivery workers, Soldiers, Civilians in war

– People who must bear the system with their bodies

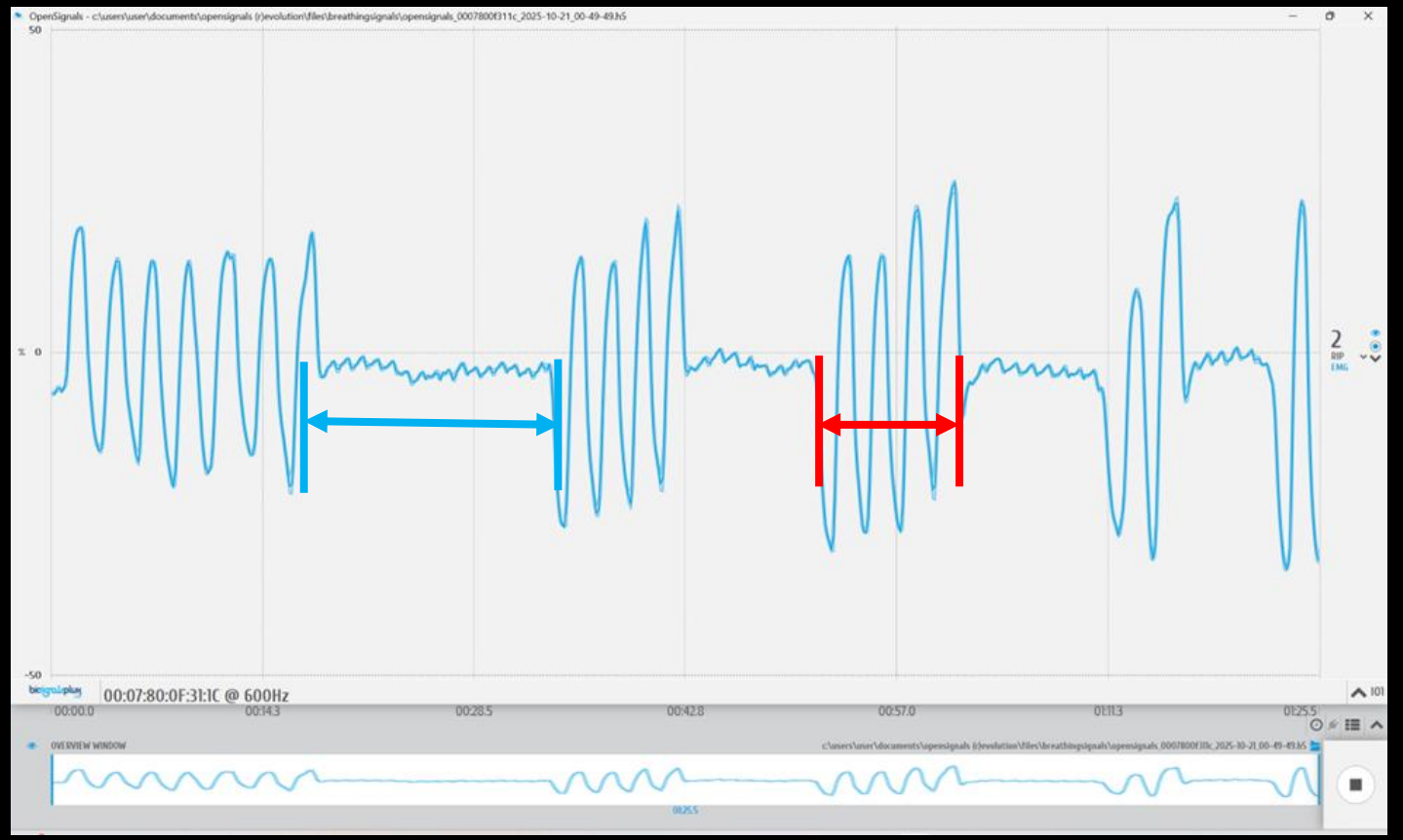
– Existing at the point where drone simulations and algorithms become reality and fall

– Shouldering the burden of danger, fatigue, injury, dismissal, and burnout

‘Order from Amazon’: How tech giants are storing mass data for Israel’s war

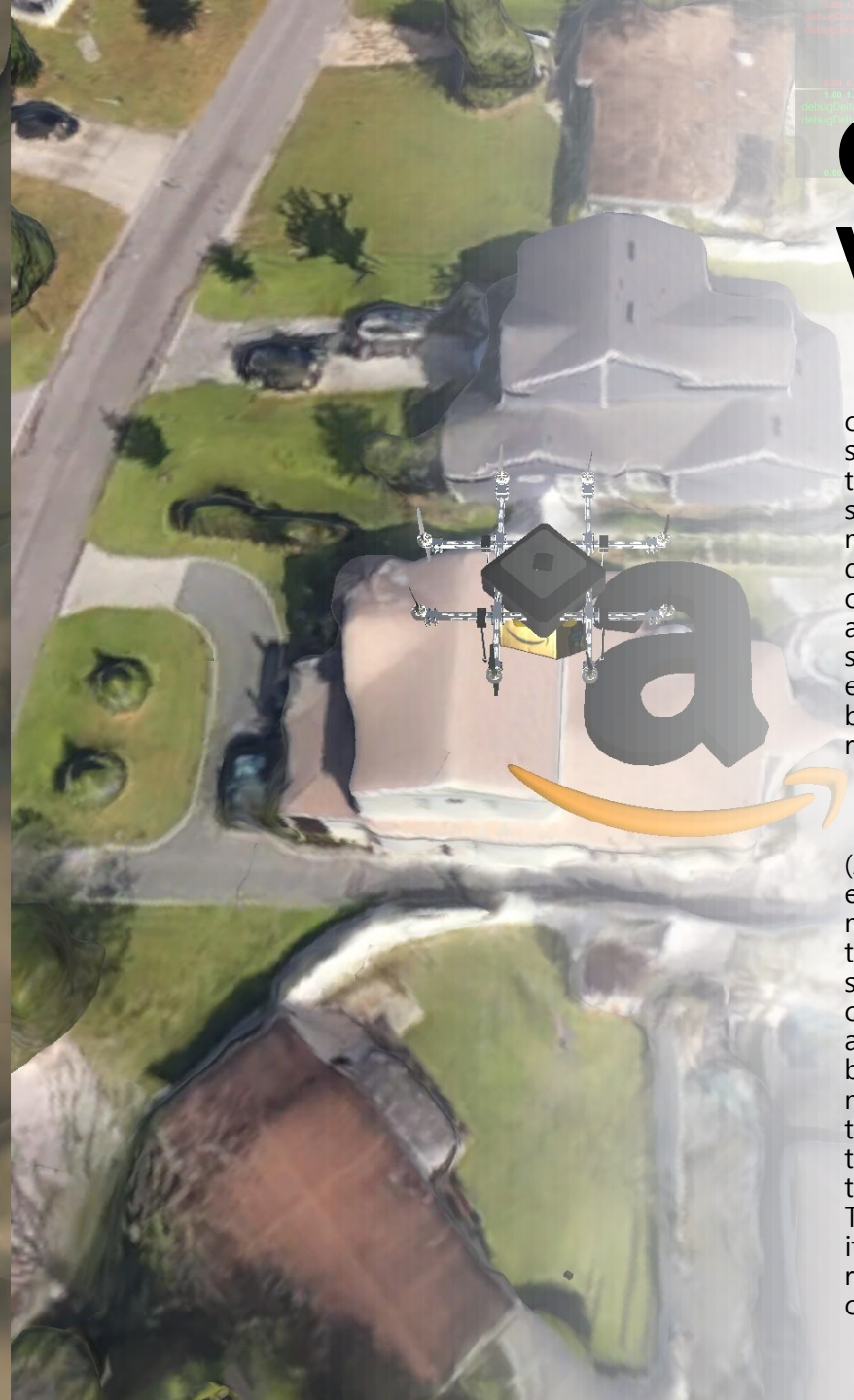
The Israeli army is using Amazon’s cloud service to store surveillance information on Gaza’s population, while procuring further AI tools from Google and Microsoft for military purposes, an investigation reveals.

Illusion of Choice



Holding breath → The will to resist

The moment of breathing→ A return to the system



God’s Eyes’ View

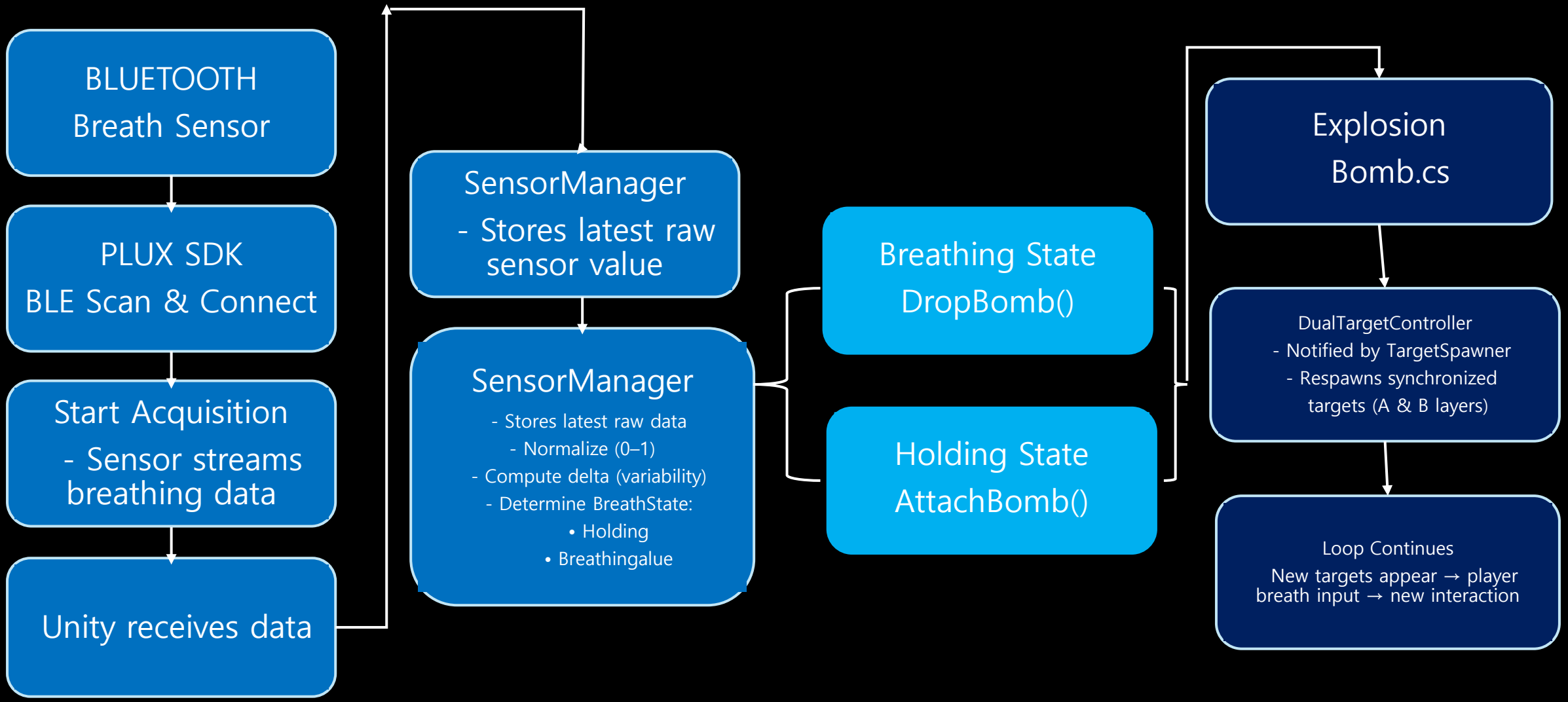
This work does not simply equate war with consumption. While delivery drones and bombing drones serve different purposes, both operate within the same technological and economic framework. At the core of this structure lies the asymmetry of violence. Those in decision-making positions handle logistics and warfare as abstract data, remaining completely insulated from real risks. In contrast, delivery workers, warehouse employees, soldiers, and local residents exist at the points where drone simulations and algorithms manifest into reality, constantly exposed to risks of death, fatigue, injury, termination, and burnout. This unequal system of violence abstracts and renders invisible those who become victims.

The fact that Amazon's cloud infrastructure (AWS) supports both military technology and consumer ecosystems reveals that, although war and consumption may appear to be distinct realms, they actually operate on the same technological infrastructure and automation structure. Moreover, we may feel that we are making choices, yet we can only select from the narrow range allowed by the system. The players in this work also believe they are making decisions, but ultimately, they are merely executing the decision-making loops designed by the platform. The act of holding one’s breath symbolizes the will to resist, while the moment of inhalation signifies the inevitability of participation in the automated system. This reflects the conditions of modern individuals who find it difficult to reject the convenience of Amazon, and the reality that even in opposing military technology, one cannot easily escape from violent infrastructures.

Equipment and Environment

Unity 3D+ C#
BioSignal Plux Piezo-Electric Respiration (PZT) Sensor
PLUX Unity API [Windows]
https://github.com/pluxbiosignals/unity-sample

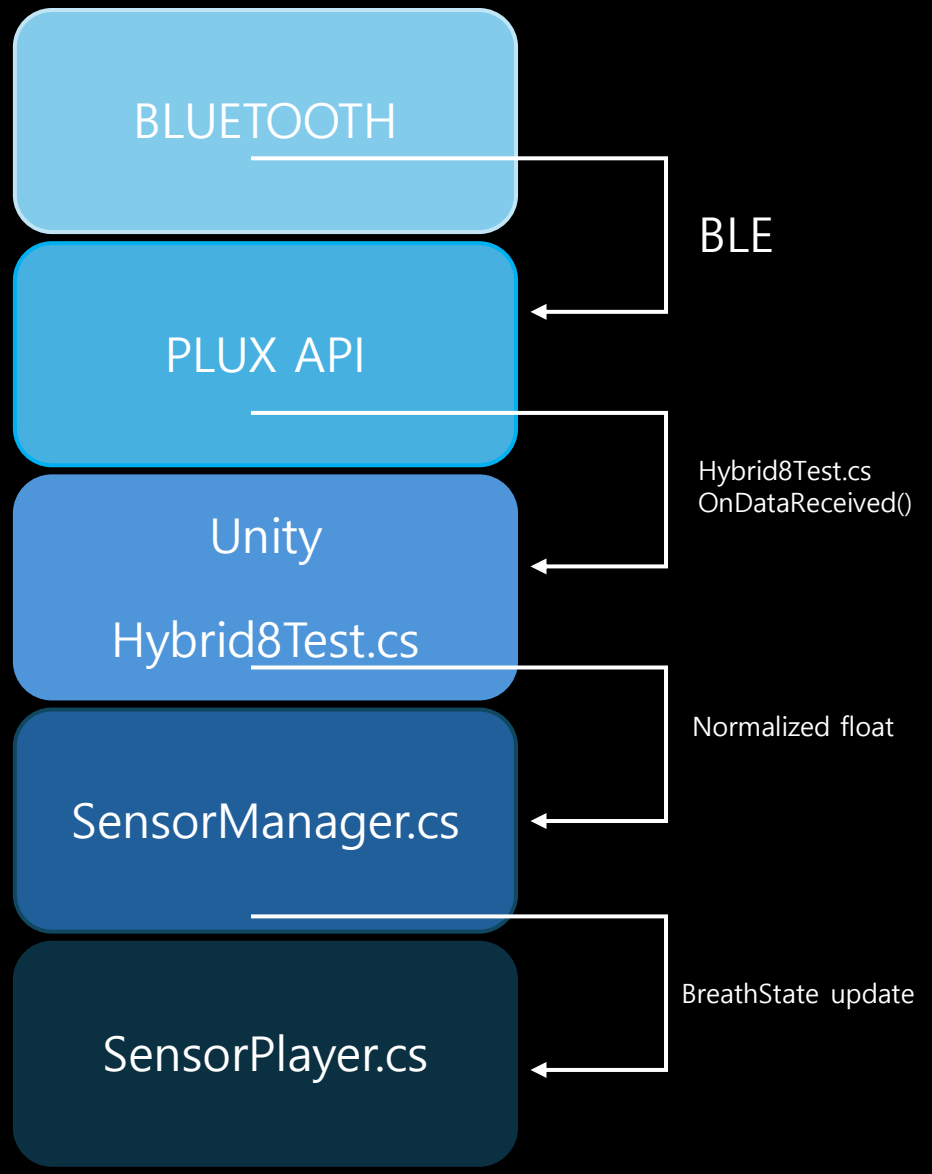
Flow Chart


BLUETOOTH
Breath Sensor
PLUX SDK
BLE Scan & Connect
Start Acquisition
- Sensor streams breathing data
Unity receives data
SensorManager
- Stores latest raw sensor value
SensorManager
- Stores latest raw data
- Normalize (0–1)
- Compute delta (variability)
- Determine BreathState:
• Holding
• Breathingalue
Breathing State
DropBomb()
Holding State
AttachBomb()
Explosion
Bomb.cs
DualTargetController
- Notified by TargetSpawner
- Respawns synchronized targets (A & B layers)
Loop Continues
New targets appear → player breath input → new interaction

Data Input

In my project, the breath sensor (a PLUX Hybrid-8 device) communicates with Unity through the PLUX SDK, which handles Bluetooth Low Energy (BLE) scanning, device connection, and real-time data streaming.

The SDK provides all the low-level Bluetooth communication and data handling, allowing Unity to receive real-time breathing data. Unity then processes this data and integrates it into the game logic.



Data Input Acquisition

Step 1 — Bluetooth Device Scan
Unity calls the PLUX SDK to scan for nearby BLE devices.
pluxDevManager.GetDetectableDevicesUnity(...);
→ Sensor appears in the device list.

Step 2 — Bluetooth Connection
Unity establishes a Bluetooth connection through the SDK.
pluxDevManager.PluxDev(selectedDeviceName);
→ Sensor is now paired and ready to stream.

Step 3 — Start Real-Time Data Acquisition
Unity tells the sensor to begin streaming raw biosignal data.
pluxDevManager.StartAcquisitionBySourcesUnity(...);
→ Sensor begins sending analog breathing data at a fixed sampling rate (e.g., 100 Hz).

Step 4 — Receive Raw Data in Unity (Callback)
The SDK pushes each incoming BLE packet into Unity through:
void OnDataReceived(int nSeq, int[] data)
→ data[] contains raw breathing samples (integer values).

Step 5 — Pass Raw Data to SensorManager
Unity stores the most recent breathing value in SensorManager.SensorData

Data Processing

Step 6 — Normalize the Data (SensorPlayer.cs)
float newData = Mathf.Clamp(sensorManager.SensorData / 65000f, 0f, 1f);
→ Converts raw sensor signal into a clean 0–1 value.
→ This value becomes BreathData.Value

Step 7 — Analyze Breathing Pattern (Delta Calculation)
SensorPlayer computes how fast the signal is changing:
BreathData.Delta = |newData - previousData| / deltaTime;
→ This is used to detect whether the player is breathing or holding breath.

Step 8 — Determine Breath State (Holding vs. Breathing)
A queue-based smoothing algorithm checks whether delta is below a threshold:
Low variability → **Holding breath**
High variability → **Breathing**
BreathState = currentState ? Holding : Breathing;

Step 9 — Trigger Gameplay Actions
When breath state changes, Unity triggers game logic:

Drone Movement

```
void MovementUpDown()
{
    if((Mathf.Abs(Input.GetAxis("Vertical")) > 0.2f || Mathf.Abs(Input.GetAxis("Horizontal")) > 0.2f)){
        if(Input.GetKey(KeyCode.I) || Input.GetKey(KeyCode.K))
        {
            ourDrone.velocity = ourDrone.velocity;
        }
        if(!Input.GetKey(KeyCode.I) && !Input.GetKey(KeyCode.K) && !Input.GetKey(KeyCode.J) && !Input.GetKey(KeyCode.L)){
            ourDrone.velocity = new Vector3(ourDrone.velocity.x, Mathf.Lerp(ourDrone.velocity.y, 0, Time.deltaTime * 5), ourDrone.velocity.z);
            upForce = 281;
        }
        if(!Input.GetKey(KeyCode.I) && !Input.GetKey(KeyCode.K) && (Input.GetKey(KeyCode.J) || Input.GetKey(KeyCode.L))){
            ourDrone.velocity = new Vector3(ourDrone.velocity.x, Mathf.Lerp(ourDrone.velocity.y, 0, Time.deltaTime * 5), ourDrone.velocity.z);
            upForce = 110;
        }
        if(Input.GetKey(KeyCode.J) || Input.GetKey(KeyCode.L))
        {
            upForce = 410;
        }
        if(Mathf.Abs(Input.GetAxis("Vertical")) < 0.2f && Mathf.Abs(Input.GetAxis("Horizontal")) > 0.2f)
        {
            upForce = 136;
        }

    }
    if (Input.GetKey(KeyCode.I))
    {
        upForce = 460;
        if(Mathf.Abs(Input.GetAxis("Horizontal")) > 0.2f)
        {
            upForce = 500;
        }
    }
    else if (Input.GetKey(KeyCode.K))
    {
        upForce = -200;
    }
    else if (!Input.GetKey(KeyCode.I) && !Input.GetKey(KeyCode.K)&&(Mathf.Abs(Input.GetAxis("Vertical"))<0.2f && Mathf.Abs(Input.GetAxis("Horizontal"))<0.2f)) {
        upForce = 98.1f;
    }
}

private float movementForwardSpeed = 600.0f;
private float tiltAmountForward = 0;
private float tiltVelocityForward;

참조 1개
void MovementForward()
{
    if(Input.GetAxis("Vertical") != 0)
    {
        ourDrone.AddRelativeForce(Vector3.forward * Input.GetAxis("Vertical") * movementForwardSpeed);
        tiltAmountForward = Mathf.SmoothDamp(tiltAmountForward, 20 * Input.GetAxis("Vertical"), ref tiltVelocityForward, 0.1f);
    }
}

private float wantedYRotation;
public float currentYRotation;
private float rotateAmountByKeys = 2.5f;
private float rotationYVelocity;
```

```
void Rotation()
{
    if (Input.GetKey(KeyCode.J))
    {
        wantedYRotation -= rotateAmountByKeys;
    }
    if (Input.GetKey(KeyCode.L))
    {
        wantedYRotation += rotateAmountByKeys;
    }

    currentYRotation = Mathf.SmoothDamp(currentYRotation, wantedYRotation, ref rotationYVelocity, 0.25f);
}

private Vector3 velocityToSmoothDampToZero;
참조 1개
void ClampingSpeedValues()
{
    if(Mathf.Abs(Input.GetAxis("Vertical")) > 0.2f && Mathf.Abs(Input.GetAxis("Horizontal")) > 0.2f)
    {
        ourDrone.velocity = Vector3.ClampMagnitude(ourDrone.velocity, Mathf.Lerp(ourDrone.velocity.magnitude, 10.0f, Time.deltaTime * 5f));
    }
    if(Mathf.Abs(Input.GetAxis("Vertical")) > 0.2f && Mathf.Abs(Input.GetAxis("Horizontal")) < 0.2f)
    {
        ourDrone.velocity = Vector3.ClampMagnitude(ourDrone.velocity, Mathf.Lerp(ourDrone.velocity.magnitude, 10.0f, Time.deltaTime * 5));
    }
    if(Mathf.Abs(Input.GetAxis("Vertical")) < 0.2f && Mathf.Abs(Input.GetAxis("Horizontal")) > 0.2f)
    {
        ourDrone.velocity = Vector3.ClampMagnitude(ourDrone.velocity, Mathf.Lerp(ourDrone.velocity.magnitude, 5.0f, Time.deltaTime * 5));
    }
    if(Mathf.Abs(Input.GetAxis("Vertical")) < 0.2f && Mathf.Abs(Input.GetAxis("Horizontal")) < 0.2f)
    {
        ourDrone.velocity = Vector3.SmoothDamp(ourDrone.velocity, Vector3.zero, ref velocityToSmoothDampToZero, 0.95f);
    }
}

private float sideMovementAmount = 300.0f;
private float tiltAmountSideways;
private float tiltAmountVelocity;
참조 1개
void Swere()
{
    if(Mathf.Abs(Input.GetAxis("Horizontal")) > 0.2f)
    {
        ourDrone.AddRelativeForce(Vector3.right * Input.GetAxis("Horizontal") * sideMovementAmount);
        tiltAmountSideways = Mathf.SmoothDamp(tiltAmountSideways, -20 * Input.GetAxis("Horizontal"), ref tiltAmountVelocity, 0.1f);
    }
    else
    {
        tiltAmountSideways = Mathf.SmoothDamp(tiltAmountSideways, 0, ref tiltAmountVelocity, 0.1f);
    }
}

public AudioSource droneSound;
참조 0개
void DroneSound()
{
    droneSound.pitch = 1 + (ourDrone.velocity.magnitude / 100);
}
}
```

Drone Bomb

```
public void DropBomb()
{
    if (!HasBomb || currentBomb == null) return;

    HasBomb = false;

    currentBomb.transform.SetParent(null);

    Rigidbody rb = currentBomb.GetComponent<Rigidbody>();
    if (rb != null)
    {
        rb.isKinematic = false;
        rb.useGravity = true;
    }

    // Notify bomb it is dropped (enable collider + ignore drone collision)
    Bomb bomb = currentBomb.GetComponent<Bomb>();
    if (bomb != null)
    {
        bomb.OnDropped();
    }

    Debug.Log("Bomb dropped");

    currentBomb = null;
}
```

Bomb states are managed separately and respond to breath-driven events.

Breath Manager

```
public class BreathManager : MonoBehaviour
{
    public List<DroneBombController> drones = new List<DroneBombController>();

    private bool wasHoldingBreath = false;

    참조 1개
    public void OnBreathHold()
    {
        wasHoldingBreath = true;

        foreach (var drone in drones)
        {
            if (drone != null && !drone.HasBomb)
                drone.AttachBomb();
        }
    }

    참조 1개
    public void OnBreathRelease()
    {
        if (!wasHoldingBreath) return;
        wasHoldingBreath = false;

        foreach (var drone in drones)
        {
            if (drone != null && drone.HasBomb)
                drone.DropBomb();
        }
    }
}
```

Breath controls bomb behavior: Holding attaches a bomb, releasing drops it.

Target Spawner

```
public void SpawnAt(int posIndex, int prefabIndex)
{
    if (spawnPoints == null || spawnPoints.Length == 0)
    {
        Debug.LogError($"[{name}] spawnPoints가 비어 있어서 스폰 못함");
        return;
    }

    if (posIndex < 0 || posIndex >= spawnPoints.Length)
    {
        Debug.LogError($"[{name}] posIndex {posIndex} 범위 밖 (길이 {spawnPoints.Length})");
        return;
    }

    if (targetPrefabs == null || targetPrefabs.Length == 0)
    {
        Debug.LogError($"[{name}] targetPrefabs가 비어 있음");
        return;
    }

    if (prefabIndex < 0 || prefabIndex >= targetPrefabs.Length)
    {
        Debug.LogError($"[{name}] prefabIndex {prefabIndex} 범위 밖 (길이 {targetPrefabs.Length})");
        return;
    }

    if (currentTarget != null)
        Destroy(currentTarget);

    Vector3 pos = spawnPoints[posIndex].position;

    currentTarget = Instantiate(targetPrefabs[prefabIndex], pos, Quaternion.identity);

    var t = currentTarget.GetComponent<TargetObject>();
    if (t != null)
        t.SetSpawner(this);
}

참조 1개
public void NotifyHit()
{
    var controller = FindObjectOfType<DualTargetController>();
    if (controller != null)
        controller.TargetHit();
}
```

The TargetSpawner generates target objects at predefined positions, and when a target is destroyed, it coordinates with the DualTargetController to respawn synchronized targets across both layers.

Future Work

1. Expanding into a Multimodal, Multi-device Asymmetric Cooperative System

The current system operates as a local multiplayer experience in which all players share the same screen and the same information.
In the future, I plan to extend this into a multimodal, multi-device cooperative environment where participants join from VR, PC, or mobile and only see partial information.

2. Introducing Failure Penalties to Encourage Meaningful Decision-making

Because the current version has no penalties or risks, players tend to drop bombs excessively.
To address this, I plan to introduce a failure penalty, such as *game over after five missed targets*.
This form of risk introduces hesitation, strategy, and responsibility, shifting the breath-based interaction from a repetitive action into a decision-driven experience.

3. Moving Beyond a Direct One-to-One Mapping Between Amazon and Warfare

In its current form, the system can appear to directly equate commercial logistics (Amazon) with military violence.
Future iterations will diversify the world by introducing additional agents and objects—for example, entities other than drones in motion, or objects other than pigeons falling to prevent a literal one-to-one interpretation.
This will allow the project to explore more complex forms of systemic and infrastructural violence, rather than relying on a direct metaphor.