INFORME TÉCNICO ARCANA RING

January 1, 2026

CONTROLADOR ALTERNATIVO MAGNÉTICO PARA TOWER DEFENSE (ANDROID)

DISPOSITIVOS Y FORMATOS

Contents

Resumen ejecutivo

El proyecto investigó el uso de un anillo imán como controlador alternativo para un juego Tower Defense. Inicialmente se diseñó una arquitectura distribuida (aplicación Android -> servidor Node.js -> juego en PC/Unity), pero la transmisión en tiempo real mostró problemas de latencia, estabilidad y despliegue. Por este motivo la implementación final se consolidó en Android: se desarrolló una lectura nativa del magnetómetro, un procesamiento robusto de la señal, detección de gestos fiables para control del juego y la integración de la mecánica del Tower Defense. El prototipo incluye arte original (sprites y animaciones) y un tilemap obtenido de recursos públicos (MitchiriNeko March, omitido en versión final).

A continuación explico con detalle el proceso seguido, las fuentes de los paquetes utilizados, las causas técnicas de los fallos que aparecieron, y el estado actual del prototipo entregado.

1. Proceso seguido

1. Idea y diseño inicial.

Diseñé el sistema para separarlo en dos escenas: una build Android que actúa como sensor, y otra build en PC (Unity) que contiene el gameplay. El móvil debía leer el magnetómetro y enviar los datos al PC por WebSocket; el PC recibiría, deserializaría y usaría esos datos en el juego.

2. Tecnologías y paquetes empleados.

- websocket-sharp (cliente WebSocket para C#): usé el fork con soporte para Unity (https://github.com/Rokobokode/websocket-sharp-unity/tree/unity-support).
- Node.js (servidor): servidor relay implementado con 'ws' en Node – 'server.js' se ejecuta con Node (descargado desde nodejs.org).
- Newtonsoft JSON / JsonSavingSystem: para serializar/deserializar JSON en Unity y guardar datos de diagnóstico; utilicé el paquete de Newtonsoft adaptado a Unity (https://gitlab.com/tkraindesigns/JsonSavingSystem/-/wikis/home).

3. Prototipado y pruebas multiplataforma.

Implementé ‘MagnetometerSender’ (app Android) que lee el magnetómetro, aplica smoothing/calibración y envía JSON por WebSocket. En PC implementé ‘MagnetometerReceiver’ (Unity) que recibe, encola y procesa mensajes con ‘ConcurrentQueue’ y ‘Newtonsoft.Json’. También desarrollé ‘server.js’ como relay para reenviar mensajes entre cliente Android y receptor PC.

4. Problemas prácticos y detección de fallos.

En pruebas reales surgen:

1) Problemas para que el Android enviara datos útiles del magnetómetro o los datos llegaban con latencia inaceptable.

2) Errores y bloqueos en Unity relacionados con permisos/archivos y leaks de memoria, resultado de manipular paquetes y plugins manualmente.

3) Fallos al construir para Android, desarrollando builds con poco progreso. Por todo esto, opté por el desarrollo Android-only y reescribí el sensor para leerlo nativamente con un ‘SensorEventListener’ (JNI ->‘AndroidJavaProxy’) para conseguir lecturas fiables.

5. Desarrollo Android-only y juego.

Con la lectura nativa funcionando se priorizó: lectura robusta del magnetómetro, detección de gestos simples (p. ej. approach), mapeo a controles del juego (navegar grid, seleccionar hechizo, cargar y lanzar), y la lógica de la parte Tower Defense (spawning, pools, proyectiles). El tilemap del escenario fue integrado y las torres/enemigos usan mis sprites personales.

2. WebSocket / NodeServer / Newtonsoft - qué empleé y por qué no funcionó

2.1 Componentes y origen

websocket-sharp (Unity client): utilicé una versión fork con soporte Unity para que el cliente WebSocket corriera dentro de la app Android y en Unity Editor.

```
using WebSocketSharp;
WebSocket ws;
```

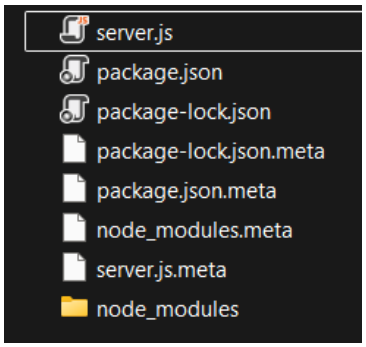
```
Debug.Log("[AutoSender] TryOpen " + url);
try
{
    ws = new WebSocket(url); // WebSocketSharp

    ws.OnOpen += (s, e) =>
    {
        wsConnected = true;
        Debug.Log("[AutoSender] WS Open");
        UpdateStatus("WS Connected");

        // iniciar loop de envío
        if (sendCoroutine == null) sendCoroutine = StartCoroutine(SendLoop());

        // iniciar latido
        if (enableHeartbeat && heartbeatRoutine == null) heartbeatRoutine = StartCoroutine(HeartbeatLoop());
    };
```

Node.js + ‘ws’: el server relay (‘server.js’) corre sobre Node y usa la librería ‘ws’ para aceptar conexiones WebSocket, loguear mensajes y reemitirlos a todos los clientes.



server.js

Archivo Editar Ver

```
// dentro de wss.on('connection', (ws, req) => { ... })
ws.on('message', (msg, isBinary) => {
  if (isBinary) {
    // msg es Buffer
    const buf = msg;
    try {
      // little-endian: timestamp (Int64 LE) + 3 floats LE
      const tBig = buf.readBigInt64LE(0); // BigInt
      const t = Number(tBig); // en ms
      const mx = buf.readFloatLE(8);
      const my = buf.readFloatLE(12);
      const mz = buf.readFloatLE(16);

      const obj = { t, mx, my, mz };
      console.log('[BIN] ' + JSON.stringify(obj));

      // reenvía el BUFFER tal cual a otros clientes (broadcast)
      for (const c of wss.clients) {
        if (c.readyState === WebSocket.OPEN) {
          c.send(buf);
        }
      }
    } catch (err) {
      console.error('Error parsing binary msg:', err);
    }
  } else {
    const txt = msg.toString();
    console.log('[TXT] ' + txt);
    for (const c of wss.clients) {
      if (c.readyState === WebSocket.OPEN) c.send(txt);
    }
  }
});
```

Newtonsoft.Json / JsonSavingSystem: el receptor en Unity deserializaba JSON con Newtonsoft; además se empleó un sistema de guardado JSON para persistir datos de pruebas.

Estas dependencias se obtuvieron de: repositorios públicos (GitHub), la web oficial de Node.js y el Package Manager de Unity.

2.2 Fallos observados en la fase distribuida

Conexión, datos no fiables o con mucha latencia: Aunque el Android conseguía conectar con el Node server, no se lograron enviar lecturas útiles del magnetómetro, o las que llegaron lo hicieron con latencia elevada, que hacía que la interacción no fuera usable para control en tiempo real. Las causas probables: la pila WebSocket en Android/Unity (websocket-sharp) sumó latencia; la red Wi-Fi/routers introdujeron jitter; y el código de envío por frame generaba allocations que afectaban al rendimiento y a la cadencia de envío.

Problemas de permisos y archivos en Unity: Mensajes como “Couldn't create 'Assets/~UnityDirMonSyncFile~...'” y errores relacionados con archivos .meta y Assets/Resources indicaron problemas de permisos o sincronización de carpetas. Esto afectó el editor y provocó que la importación/refresh de assets se bloqueara. Muchos de estos problemas surgieron al intentar instalar paquetes de forma manual (Newtonsoft Json).

Fugas temporales (TLS allocator etc.): El uso intensivo de ‘Debug.Log’, la creación constante de strings JSON y el envío frecuente generaron GC y, en algún momento, warnings sobre allocations temporales no liberadas (“TLS Allocator … has unfreed allocations”), lo que empeoró la estabilidad y las builds.

Builds Android fallidas: La combinación de cambios en plugins, manifiesto Android y DLLs (websocket-sharp, Newtonsoft) provocó builds inconsistentes y errores de integración.

2.3 Conclusión sobre la etapa WebSocket/Node

La cadena completa (sensor móvil -> websocket-sharp -> node -> Unity PC) funcionó a nivel conceptual, pero fue frágil en la práctica: la combinación de latencia de red, overhead de serialización y problemas de estabilidad del entorno me llevó a desestimarla para la entrega final. Por eso el proyecto final se consolidó en Android-only para centrar el esfuerzo en la interacción magnética, que era el objetivo principal del trabajo.

3. Lectura del magnetómetro y procesamiento (implementación actual)

3.1 Lectura nativa del magnetómetro en Android

Tras los problemas detectados con el uso de Input.compass y con la transmisión de datos mediante WebSockets, se optó por una lectura directa y nativa del sensor de campo magnético del dispositivo Android. Para ello, se accede a la API de Android (android.hardware.SensorManager) desde Unity utilizando el mecanismo de interoperabilidad AndroidJavaObject y AndroidJavaProxy.

```
try
{
    // Obtener la actividad de Unity actual
    AndroidJavaClass unityPlayer = new AndroidJavaClass("com.unity3d.player.UnityPlayer");
    unityActivity = unityPlayer.GetStatic<AndroidJavaObject>("currentActivity");

    AndroidJavaClass contextClass = new AndroidJavaClass("android.content.Context");
    string SENSOR_SERVICE = contextClass.GetStatic<string>("SENSOR_SERVICE");

    sensorManager = unityActivity.Call<AndroidJavaObject>("getSystemService", SENSOR_SERVICE);

    AndroidJavaClass sensorClass = new AndroidJavaClass("android.hardware.Sensor");
    int TYPE_MAGNETIC_FIELD = sensorClass.GetStatic<int>("TYPE_MAGNETIC_FIELD");

    sensor = sensorManager.Call<AndroidJavaObject>("getDefaultSensor", TYPE_MAGNETIC_FIELD);

    if (sensor == null)
    {
        Debug.LogWarning("[AndroidMagnetometer] No se encontró sensor MAGNETIC_FIELD en el dispositivo.");
        return;
    }

    // Crear y registrar el listener
    var sensorManagerClass = new AndroidJavaClass("android.hardware.SensorManager");
    int SENSOR_DELAY_GAME = sensorManagerClass.GetStatic<int>("SENSOR_DELAY_GAME");

    listenerProxy = new MagListenerProxy(OnSensorChangedCallback, OnAccuracyChangedCallback);

    bool registered = sensorManager.Call<bool>("registerListener", listenerProxy, sensor, SENSOR_DELAY_GAME);
    Debug.Log("[AndroidMagnetometer] Listener registrado: " + registered);

    running = registered;
}
catch (Exception ex)
{
    Debug.LogError("[AndroidMagnetometer] Error de inicialización: " + ex);
}
```

En concreto, se registra un SensorEventListener sobre el sensor TYPE_MAGNETIC_FIELD, lo que permite recibir eventos con lecturas del campo magnético en los tres ejes espaciales:

- values[0]: componente X (eje horizontal del dispositivo)
- values[1]: componente Y (eje vertical del dispositivo)
- values[2]: componente Z (eje perpendicular a la pantalla)

```
// Si no es Android (Editor), mostrar valores de Input.compass
if (Application.platform != RuntimePlatform.Android && useUnityFallbackInEditor)
{
    Vector3 v = Input.compass.rawVector;
    Volatile.Write(ref latestX, v.x);
    Volatile.Write(ref latestY, v.y);
    Volatile.Write(ref latestZ, v.z);
    Interlocked.Exchange(ref latestTs, DateTimeOffset.UtcNow.ToUnixTimeMilliseconds());
}
```

Estas lecturas están expresadas en microteslas (μT), que es la unidad estándar del Sistema Internacional para medir la densidad de flujo magnético.

La frecuencia de actualización se configura mediante el parámetro SENSOR_DELAY_GAME, lo que proporciona un compromiso adecuado entre:

- Latencia baja, necesaria para interacción en tiempo real.
- Consumo moderado de recursos, evitando saturar el hilo principal o la batería del dispositivo.

Esta aproximación ofrece un mayor control sobre la frecuencia de muestreo y una reducción significativa de la variabilidad observada al usar las capas de abstracción estándar de Unity, además de evitar los problemas de sincronización y latencia inherentes a la transmisión de datos por red.

3.2 Preprocesado de la señal magnética

Las lecturas del magnetómetro son altamente sensibles al entorno y al ruido, por lo que no pueden utilizarse directamente para detección de gestos. Por ello, se implementa una capa de preprocesado que transforma la señal bruta en una señal más estable y utilizable.

3.2.1 Calibración inicial (baseline)

Durante una fase de calibración inicial, el sistema toma un conjunto de N muestras con el imán en reposo o en una posición neutra. A partir de estas muestras se calcula un vector medio:

$$\text{baseline} = \frac{1}{N}\sum_{i=1}^{N}\vec{B}_i$$

Este vector representa el campo magnético ambiental en ese momento y ubicación. En las lecturas posteriores, dicho baseline se resta del valor actual:

$$\vec{B}_{corr} = \vec{B}_{raw} - \text{baseline}$$

De este modo, el sistema deja de depender de valores absolutos y pasa a centrarse en variaciones inducidas por el imán, lo cual es esencial debido a la gran variabilidad del campo magnético terrestre y de las interferencias del entorno.

3.2.2 Suavizado (Low-pass filter / EMA)

Incluso tras la calibración, la señal presenta oscilaciones rápidas causadas por ruido eléctrico, vibraciones o micro-movimientos del dispositivo. Para mitigar este efecto se aplica un filtro paso bajo de tipo Exponential Moving Average (EMA):

$$\vec{B}_{smooth} = \alpha \cdot \vec{B}_{corr} + (1 - \alpha) \cdot \vec{B}_{prev}$$

smooth=α·Bcorr+(1−α)·B

donde lowPassAlpha controla el equilibrio entre:

- Reactividad (valores altos de α),
- Estabilidad (valores bajos de α).

Este filtro permite eliminar ruido de alta frecuencia sin introducir un retardo excesivo, algo crítico para la interacción en tiempo real.

3.2.3 Ventana deslizante (Windowing)

Para detectar cambios temporales, no basta con analizar una muestra aislada. Por ello, el sistema mantiene una ventana deslizante de las últimas muestras procesadas (posición magnética y timestamp).

Esto permite:

- Comparar el estado actual con uno anterior.
- Calcular diferencias temporales (delta) de posición o magnitud.
- Detectar cambios rápidos frente a variaciones lentas del entorno.

Por ejemplo, se calcula la diferencia de magnitud entre la muestra más reciente y la más antigua de la ventana:

$$\Delta \mid B \mid = \mid B_{actual} \mid - \mid B_{anterior} \mid$$

3.3 Detección de gestos magnéticos básicos

Sobre la señal preprocesada se implementan detectores de gestos simples, priorizando la robustez sobre la complejidad. Estos gestos constituyen la base de la interacción con el juego.

3.3.1 Gesto de acercamiento (Approach)

El gesto de *acercamiento* se detecta cuando la magnitud total del campo magnético aumenta de forma significativa en un intervalo corto de tiempo. Matemáticamente:

$$\Delta \mid B \mid = \mid B_{actual} \mid - \mid B_{pasado} \mid$$

Si este valor supera un umbral configurado (approachThreshold), el sistema interpreta que el imán se ha acercado al dispositivo.

Este gesto se utiliza como acción principal de activación, por ejemplo:

- Iniciar la carga de un hechizo.
- Confirmar una acción en el juego.

El enfoque basado en magnitud es relativamente robusto, ya que no depende de la orientación exacta del dispositivo, sino de la intensidad del campo.

```
else if (deltaMag >= approachThreshold)
    detected = CalibPosition.Acercar;
```

3.3.2 Gestos direccionales (heurísticos)

Además del acercamiento, se implementa una detección heurística de direcciones básicas (izquierda, derecha, arriba, abajo). Esta detección se basa en observar qué componente del vector magnético presenta un cambio dominante:

- Cambios predominantes en X → izquierda / derecha
- Cambios predominantes en Y → arriba / abajo

Para evitar falsas detecciones, se emplean:

- Un umbral mínimo de cambio (dirThreshold)
- Una tolerancia cruzada (crossTolerance) que exige que los otros ejes permanezcan relativamente estables

Sin embargo, este tipo de detección es inherentemente frágil, ya que:

- El magnetómetro no distingue entre rotación del dispositivo y movimiento del imán.
- El campo ambiental fluctúa constantemente.
- La orientación del móvil altera el significado de los ejes X e Y.

Por estas razones, los gestos direccionales resultan inconsistentes y afectan negativamente a la fiabilidad del control, especialmente en situaciones de juego dinámicas.

4. Fuentes del grid y arte visual

Grid / Tilemap: el escenario se montó usando tilemaps descargados de recursos públicos de internet (tilepacks gratuitos disponibles en plataformas como itch.io). El tilemap fue adaptado y configurado en Unity con Tilemap Renderer y colliders para el pathfinding.

Assets: todos los sprites y animaciones de torres y enemigos fueron creados en Krita por mí (pixel art), lo cual garantiza que el estilo visual de la demo sea original. Los sonidos fueron realizados y editados empleando MuseScore, bibliotecas públicas y Audacity.

5. Fallos e inconsistencias del prototipo entregado

5.1 Fluctuación constante del campo magnético

En las pruebas se observó que el campo magnético registrado por el sensor del teléfono presenta fluctuaciones continuas que dificultan la detección consistente de gestos direccionales. Aunque la calibración en reposo (cálculo de una baseline promedia) mejora inicialmente la relación señal/ruido, dicha referencia deja de ser fiable tras un periodo corto (minutos en algunos casos) cuando cambian las condiciones del entorno.

Causas físicas y del sensor:

- Campo geomagnético (Tierra): el magnetómetro mide la suma del campo terrestre y los campos locales. El campo terrestre es estable pero la señal absoluta depende de la orientación del dispositivo.
- Interferencias locales: estructuras metálicas, mesas, equipos eléctricos, altavoces, cargadores, imanes cercanos (incluso imanes en altavoces) y cables pueden introducir variaciones significativas.
- Sensor noise y resolución: los magnetómetros en móviles tienen ruido y resolución limitada (μT); pequeñas variaciones pueden ser del orden de la señal útil y generar falsos positivos.
- Drift y temperatura: el sensor puede mostrar drift temporal y depender de la temperatura del dispositivo.

- Orientación del teléfono: cambiar la rotación del dispositivo cambia la proyección de los componentes del campo; sin compensación por actitud, la misma acción física del imán produce vectores distintos.
- Movimiento del usuario: movimientos no intencionales de la mano o la vibración son interpretados por el sensor y generan ruido.

Impacto en la detección:

- Los detectores direccionales basados en diferencias de ejes (X / Y) fallan porque las pequeñas fluctuaciones o interferencias pueden tener magnitud similar al cambio real que genera el imán.
- La calibración que haces en reposo deja de ser válida pasado un tiempo porque el entorno o la orientación cambian (o el propio sensor tiene drift). Es decir, calibrar ayuda, pero no es una solución definitiva si las condiciones varían durante la sesión.

Conclusión práctica:

Por esto los gestos direccionales resultan muy inconsistentes y el gameplay (que depende de que esos gestos sean fiables y repetibles) se ve afectado. Para experimentos fiables hay que: 1) reducir el ruido con filtros más avanzados y compensación por orientación; 2) usar gestos que sean robustos frente al ruido (p. ej. acercar y rotar el imán en lugar de pequeños desplazamientos laterales); 3) registrar y analizar CSV para ajustar umbrales por dispositivo y lugar.

5.2 Calibración se vuelve irrelevante con el tiempo

Tal como se ha observado, calibrar en reposo es útil pero la baseline puede "romperse" en minutos si cambian las condiciones (movimiento del usuario, un equipo cercano, etc.). Por eso la práctica experimental recomendada es:

- Recalibrar con frecuencia (o pedir al usuario que realice un gesto de recalibración),
- o implementar calibración adaptativa que detecte drift y actualice la baseline lentamente cuando el usuario no está emitiendo gestos.

5.3 Hechizos que no se lanzan correctamente

En la versión entregada se han detectado fallos intermitentes en el lanzamiento de hechizos. Los motivos identificados en el análisis del código y de las pruebas son los siguientes:

1. Ausencia de objetivo (target == null): algunas llamadas a CastSpell o a SpellProjectile.Setup se producen sin que la torre tenga un objetivo válido (enemigo en rango), con lo que el proyectil no recibe la información necesaria para dirigirse y no se ejecuta la acción esperada.
2. Inicialización incorrecta del pool: si BulletPool.Instance no está disponible en el momento en que una torre solicita un proyectil (problema de orden de inicialización), el flujo recurre a un fallback que no siempre instancia o configura correctamente el proyectil.
3. Lógica de carga basada en eventos pulsados: la detección de acercamiento puede emitir pulsos cortos en lugar de estados start/stop; con ello la mecánica de "carga" (mantener acercamiento para aumentar potencia) no logra iniciarse o completarse, de modo que el hechizo no se lanza con la potencia esperada o no se lanza en absoluto.
4. Paquete de manejo de estados incompleto: el panel de calibración no pausa la máquina de estados del juego, por lo que pueden ejecutarse spawners, corutinas o detecciones en segundo plano que interfieren con el flujo de lanzamiento. Además no existe un manejador centralizado de GameOver que detenga oleadas y permita un reinicio limpio.
5. Problemas de sincronización en Setup del proyectil: en algunos casos los parámetros (potencia, velocidad, objetivo) no se transfieren antes de activar el objeto, provocando comportamiento indefinido o la devolución inmediata al pool.

5.4 Otros fallos de UX/visualización

Si se quisiera pulir el proyecto en mayor medida, cabría tener en cuenta que:

1) No se muestra la vida de los enemigos: falta un health bar en el prefab de enemigos y falta código que sincronice la barra con el valor de salud.
2) La Gestión de modos (GridPlacement / SpellCasting) no es fiable: la confusión entre modos (y la falta de indicador visual claro) provoca que gestos tengan efectos distintos según el estado y el usuario no sepa en qué modo está.

6. Correcciones y recomendaciones prácticas

6.1 Problemas de detección magnética

Algunas posibles soluciones a los problemas encontrados podrían ser:

1. Compensar orientación del dispositivo**:** antes del filtro, transforma el vector magnético usando la actitud del giroscopio para trabajar en un referencial

estable (evita que girar el teléfono cambie la interpretación de gestos). En la práctica: obtener Input.gyro.attitude (o la matriz de rotación nativa) y aplicar la rotación inversa al vector crudo.

2. Mejorar filtros**:** además del EMA, aplicar filtros de ventana y/o filtros basados en mediana para eliminar picos espurios; usar thresholds adaptativos (p. ej. comparar con desviación típica local).
3. Priorizar gestos robustos**:** usar *approach* y *spike* (aumento puntual) como gestos primarios; los gestos direccionales solo si su detección es estable con pruebas en el mismo dispositivo y entorno.
4. Calibración adaptativa**:** permitir recalibración rápida y/o actualizar baseline lentamente cuando no se detectan gestos.
5. Registrar CSV para tuning**:** cada sesión debe poder registrar ts,mx,my,mz,|B|,gesture para calcular TPR/FPR y ajustar umbrales por dispositivo.

6.2 Hechizos, pooling y orden de inicialización

1. Validar target en CastSpell: si target == null deberías mostrar una notificación y evitar instanciar; o cambiar la lógica para que un hechizo se dirija a la posición seleccionada en el grid en lugar de depender de target.
2. Implementar estado de carga robusto: el detector debe emitir un evento de *start* (cuando Acercar entra) y un evento de *stop* (cuando Acercar sale), no solo pulsos. La RingSpellController usará esos eventos para iniciar/parar cronómetro de carga.
3. Revisar SpellProjectile.Setup y su llamada: asegurar que los parámetros (power, speed, target) se transfieren correctamente y que el projectile se activa y se devuelve al pool al terminar.

6.3 Pausa, calibración y GameOver

1. Panel de calibración debe pausar el juego: al abrir el panel se debe cambiar el estado del juego a Paused: detener spawners, detener detección (o ignorar eventos), pausar corutinas relevantes y mostrar menú. Al cerrar, volver al estado anterior.
2. Implementar Game Over y reinicio: añadir gestión centralizada de estado (GameManager) con OnGameOver() que detiene oleadas, muestra UI y ofrece Restart (reload scene) o MainMenu.

3. Mostrar vida de enemigos: añadir un World-space canvas o un sprite/slider child en el prefab de enemigos que se actualice desde su script Enemy en cada TakeDamage.

6. Estado final entregado y limitaciones

El prototipo Android-only funciona: lectura nativa de magnetómetro, detección de approach y direcciones, mapeo de eventos a acciones de juego.

El tilemap y los assets gráficos son parte del proyecto; los sprites de las torres y enemigos son creación propia.

Las limitaciones empíricas (variabilidad del campo, drift, interferencias y ergonomía) condicionan la fiabilidad de gestos direccionales: esto se documenta como un resultado experimental, no como un error de implementación.

Los problemas restantes (gestos pobremente detectados, hechizos no lanzados consistentemente, ausencia de reinicio por Game Over, falta de indicadores de vida) están descritos arriba con causas técnicas y las correcciones necesarias.