Bouba District

Entrega 3

Desarrollo de aplicaciones para dispositivos móviles

4º Diseño y desarrollo de videojuegos - Quintana

Ana Ordóñez Gragera

Laura García Martín

Marta Raboso Gómez

Lucía Sánchez Abril

Francisco Rodríguez Martínez

Índice

1. Introducción

Bouba District es un juego competitivo musical que combina mecánicas que mezclan **acción y ritmo**. El jugador deberá enfrentarse a hordas de criaturas adorables pero peligrosas, que se acercan en masa para atacarlo. La **clave** para defenderse será sDesaeguir el ritmo de la **música** y ejecutar las secuencias de **movimientos** correctos dentro del tiempo indicado.

2. Bouba District - Concepto

a. Historia

Un experimento fallido ha provocado que unas criaturas mucosas, los “Boubas” se hayan escapado y quieran conquistar todo aquello con lo que se cruzan. Estas criaturas, atraídas por el sonido, han comenzado a invadir la ciudad, sembrando el caos y el desorden. Utilizando sus dotes innatos para seguir el ritmo, los personajes deberán enfrentarse a estos mutantes e intentar restaurar el orden de la ciudad.

b. Mecánicas de juego

Bouba District es un juego **casual competitivo** donde el jugador se enfrenta a oleadas de enemigos conocidos como “Boubas”. Estos intentarán invadir el territorio del jugador. El objetivo principal es defenderse siguiendo las secuencias

de movimientos que presentan los Boubas, ajustándose al ritmo de la música. Las mecánicas principales de Bouba District se centran en la combinación de acción y ritmo.

- **Ritmo y secuencia de movimientos:** el jugador deberá hacer los swipes correctos en sintonía con la música.
- **Defensa:** el ejecutar los movimientos correctos provocará que los enemigos le den más vida al jugador. El jugador tendrá que aguantar el mayor tiempo posible, consiguiendo la mayor puntuación.
- **Puntuación:** la puntuación incrementa cuando se matan Boubas. Cuantos más movimientos correctos consecutivos se consigan hacer, más puntos se sumarán. También se tiene en cuenta el tiempo total que se ha sobrevivido. Esta puntuación se lleva a un ranking global para fomentar el aspecto competitivo y la rejugabilidad.

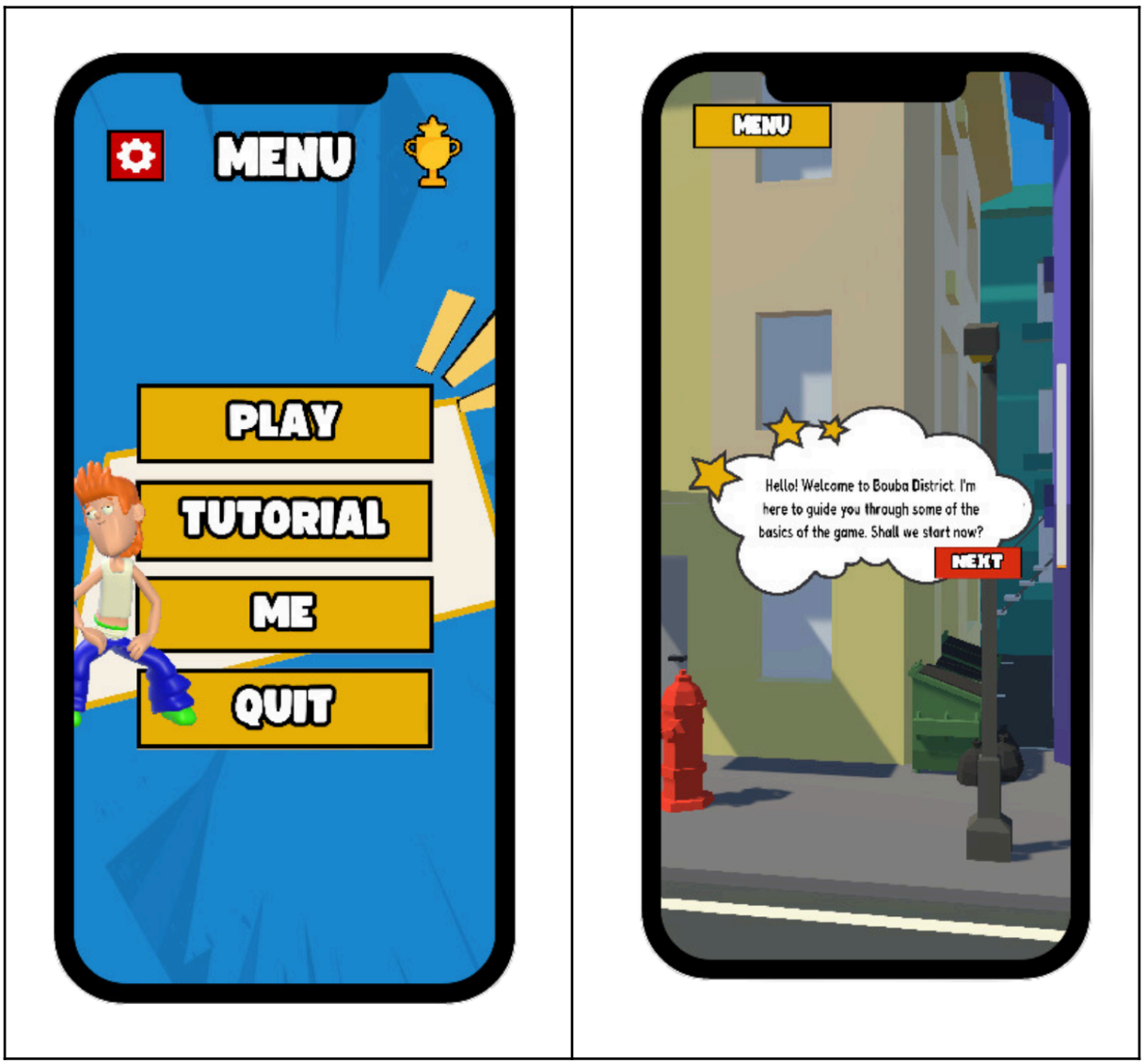
c. Controles

Los controles de Bouba District están diseñados para ser accesibles, simples y precisos.

- **Swipes:** el jugador ejecutará las secuencias arrastrando el dedo en cuatro direcciones: arriba, abajo, derecha e izquierda.
- **Shake:** se irán acumulando puntos según se vayan eliminando enemigos. Cuando la habilidad esté cargada, se agitará el móvil para eliminar a todos los Boubas en pantalla.

3. Tutorial

Bouba District tiene disponible un tutorial que explicará paso a paso las funcionalidades básicas del juego, para aquellos usuarios nuevos o inexpertos. Este tutorial se caracteriza por una progresión paso a paso, interactiva, y donde el jugador puede explorar las funcionalidades del juego a su ritmo, y durante todo el tiempo que considere necesario.



Accesible a través de la pantalla de inicio, comienza con una bienvenida al jugador, donde podrá informarse sobre los objetivos principales del nivel y cuáles son las mecánicas.

Buscamos una representación activa de la progresión en el tutorial, cambiando de posición las cajas de texto cada vez que se interactúa con ellas, además de elementos del fondo en continuo movimiento. Más adelante, se hará una presentación de los Boubas, las criaturas que queremos eliminar, y el funcionamiento clave de los **`swipes`**.

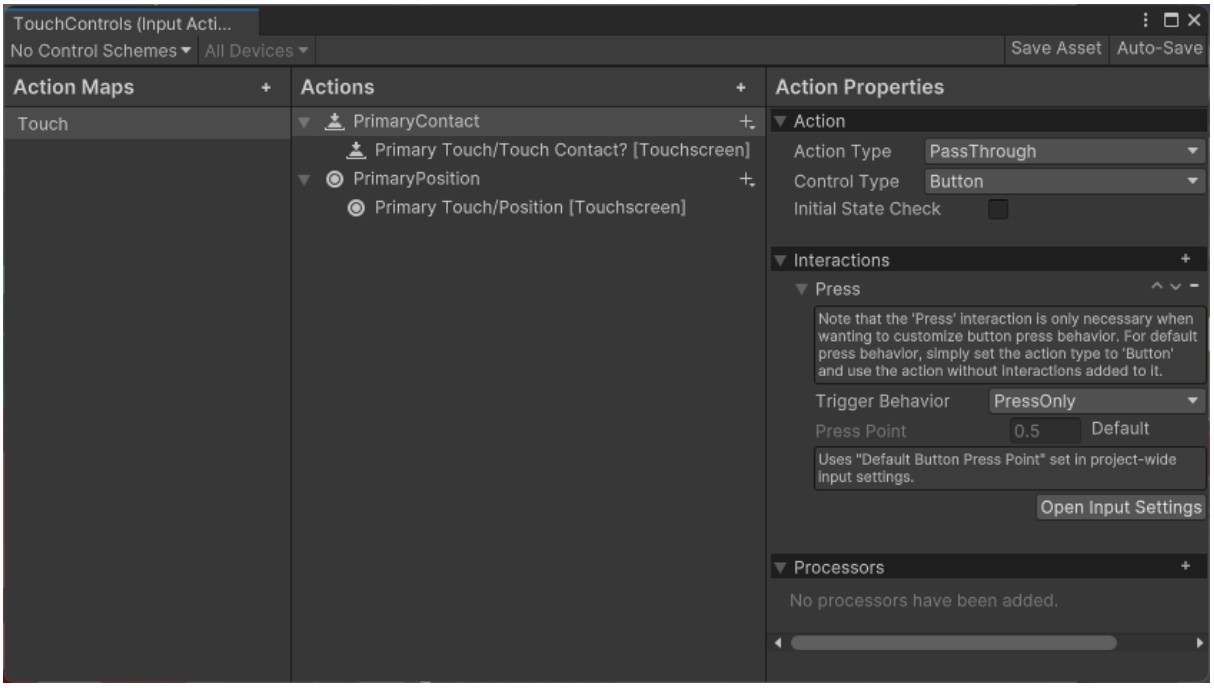
Finalmente, el tutorial acabará cuando el jugador se sienta listo, tras haber matado a varios Boubas, para volver a la pantalla de inicio y comenzar el juego.

4. Swipes

a. Implementación

Para la implementación del contacto táctil de la pantalla se ha hecho uso del ***New Input System*** de Unity.

Se definió un nuevo conjunto de ***Input Actions*** **(.inputactions)** que se utilizarán para leer los datos de la interacción del usuario con la pantalla.



Primary Contact controla si ha habido un contacto o no con la pantalla mientras que *Primary Position* almacena la posición de ese contacto. Este esquema está limitado a un solo contacto (dedo) por pantalla.

Utilizando las herramientas de Unity se generó un archivo C# automáticamente (**`TouchControls.cs`**).

Ahora se implementa un nuevo *script* **`InputManager`** encargado de encapsular las funcionalidades específicas para procesar los *swipes* –o deslizamientos. Esta nueva clase ***singleton*** compone la anterior. Primero se inicializa correctamente.

```
private:
    TouchControls touchControls;
    Camera mainCamera;

private:
    void Awake() {
        touchControls = new TouchControls();
        mainCamera = Camera::main;
    }

    void OnEnable() {
        touchControls.Enable();
    }

    void OnDisable() {
        touchControls.Disable();
    }
```

A continuación, se implementan los eventos **OnStartTouch** y **OnEndTouch** que encapsulan las funcionalidades de las ***input actions*** y que se utilizarán propiamente para para procesar los ***inputs*** del usuario. Estos eventos se disparan cuando el usuario hace contacto con la pantalla y deja de hacerlo respectivamente. Ambos trasladan la información de la posición del dedo en espacio de la pantalla además del tiempo. Se hace uso de una clase auxiliar **TouchUtils**.

```
public delegate void StartTouch(Vector2 position, float time);
public event StartTouch OnStartTouch;
public delegate void EndTouch(Vector2 position, float time);
public event EndTouch OnEndTouch;

public Vector2 PrimaryPosition()
{
 return TouchUtils.ScreenToWorld(mainCamera,
touchControls.Touch.PrimaryPosition.ReadValue<Vector2>());
}

private void Start()
{
    touchControls.Touch.PrimaryContact.started += ctx =>
StartTouchPrimary(ctx);
```

```
    touchControls.Touch.PrimaryContact.canceled += ctx => EndTouchPrimary(ctx);

}

private void StartTouchPrimary(InputAction.CallbackContext context)
{
    if(OnStartTouch != null) OnStartTouch(TouchUtils.ScreenToWorld(mainCamera,
touchControls.Touch.PrimaryPosition.ReadValue<Vector2>()),
                        (float) context.startTime);
}

private void EndTouchPrimary(InputAction.CallbackContext context)
{
    if (OnEndTouch != null) OnEndTouch(TouchUtils.ScreenToWorld(mainCamera,
touchControls.Touch.PrimaryPosition.ReadValue<Vector2>()),
                       (float)context.time);
}
```

Finalmente, se genera una tercera y última clase **SwipeDetection** encargada de calcular los ***swipes*** y su orientación. Esta clase conecta el sistema de ***input*** con el resto de sistemas de juego. Utilizando la referencia a la clase **InputManager** suscribimos métodos propios a los anteriormente creados. Estos nuevos métodos **SwipeStart** y **SwipeEnd** almacenan los datos de posición y tiempo en variables locales, en este último método se procesa el ***swipe*** –es decir el ***swipe*** se completa cuando el usuario deja de hacer contacto con la pantalla.

```
private Vector2 startPosition;
private float startTime;
private Vector2 endPosition;
private float endTime;

private void Awake()
{
    inputManager = InputManager.Instance;
}

private void OnEnable()
{
    inputManager.OnStartTouch += SwipeStart;
    inputManager.OnEndTouch += SwipeEnd;
}
```

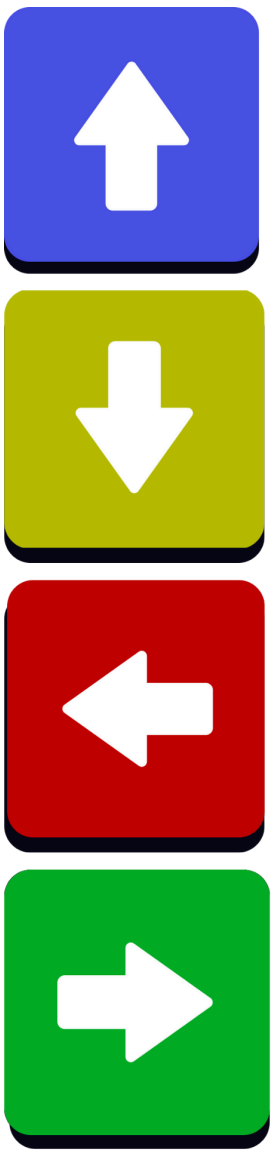
```
private void OnDisable()
{
    inputManager.OnStartTouch -= SwipeStart;
    inputManager.OnEndTouch -= SwipeEnd;
}

private void SwipeStart(Vector2 position, float time)
{
    startPosition = position;
    startTime = time;
}
private void SwipeEnd(Vector2 position, float time)
{
    endPosition = position;
    endTime = time;
    DetectSwipe();
}
```

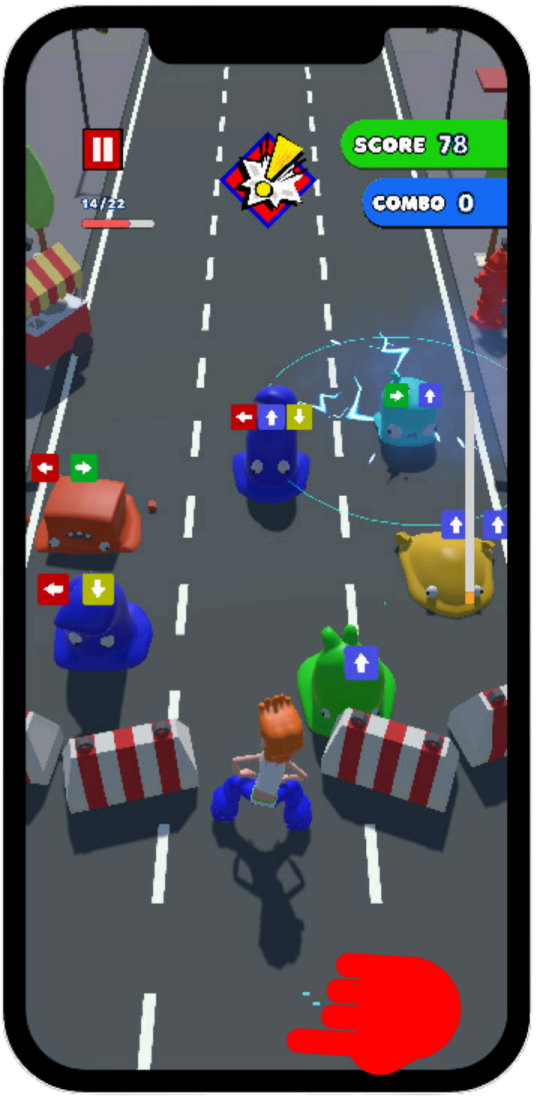
El método **DetectSwipe** asume varias funciones. Primero debe asegurarse de que el ***swipe*** es válido, tanto la distancia como el tiempo deben ajustarse a unas constantes de distancia mínima (**minimunDistance**) y tiempo máximo (**maximunTime**). Una vez que se comprueba que el ***swipe*** es válido, se almacena si se ejecutó a ritmo utilizando la clase beatManager. A continuación se comprueba si la entrada está en ***cooldown*** –el usuario solo puede introducir un número de notas cada cierto tiempo. Si está en *cooldown* finaliza la ejecución, en caso contrario se procesa el *swipe* para obtener su dirección utilizando la función **GetSwipeDirection**. Esta función utiliza operaciones de vectores básicas junto con la constante **directionThreshold**. La dirección del *swipe* determina qué nota se loguea en cada momento de tal manera que se introduce una diferente en cada caso al **noteManager**, encargado de procesarlas. Finalmente, solo si la nota se introdujo a ritmo, se inicia el contador del *cooldown*.

```
private void DetectSwipe()
{
    if(Vector3.Distance(startPosition, endPosition) >= minimunDistance  &&
      (endTime - startTime) <= maximunTime)
    {
        var onBeat = beatManager.IsOnBeat();
```

```
        if (cooldown.IsCoolingDown) return;

        var swipeDir = GetSwipeDirection(startPosition, endPosition);
        switch (swipeDir)
        {
            case SwipeDirection.Up:
                noteManager.InputNote(notes[0], onBeat);
                break;
            case SwipeDirection.Down:
                noteManager.InputNote(notes[1], onBeat);
                break;
            case SwipeDirection.Right:
                noteManager.InputNote(notes[2], onBeat);
                break;
            case SwipeDirection.Left:
                noteManager.InputNote(notes[3], onBeat);
                break;
        }

        if(onBeat) cooldown.StartCooldown();
    }
}

private SwipeDirection GetSwipeDirection(Vector2 startPosition, Vector2
endPosition)
{
    Vector3 direction = endPosition - startPosition;
    Vector2 direction2D = new Vector2(direction.x, direction.y).normalized;

    if (Vector2.Dot(Vector2.up, direction2D) > directionThreshold)
    {
        return SwipeDirection.Up;
    }
    else if (Vector2.Dot(Vector2.down, direction2D) > directionThreshold)
    {
        return SwipeDirection.Down;
    }
    else if (Vector2.Dot(Vector2.right, direction2D) > directionThreshold)
    {
        return SwipeDirection.Right;
    }
    else if (Vector2.Dot(Vector2.left, direction2D) > directionThreshold)
    {
        return SwipeDirection.Left;
    }
    return SwipeDirection.Null;
```

b. Funcionamiento



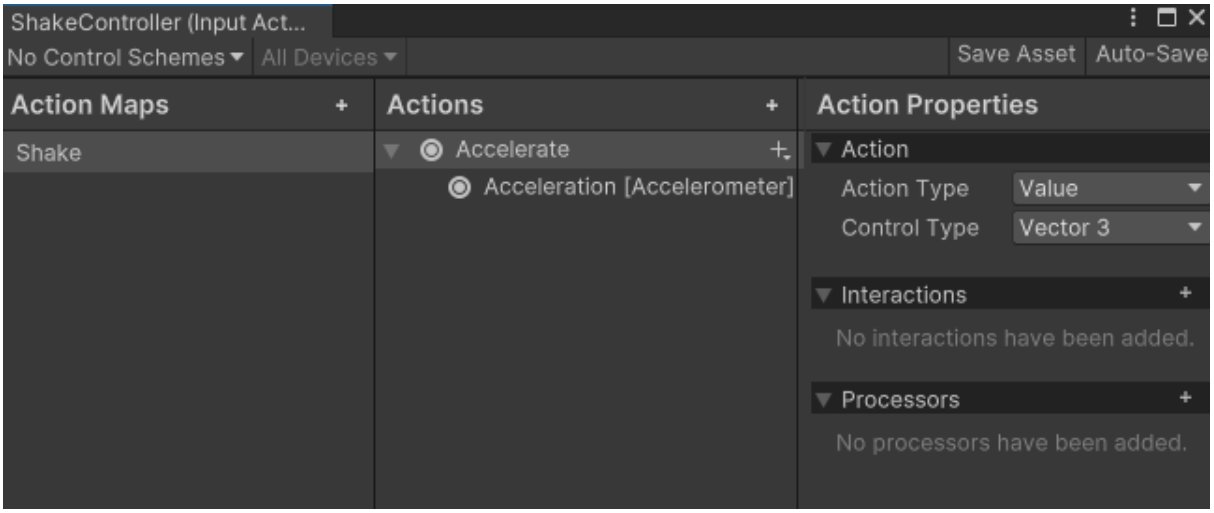


Observamos cómo aparecen las combinaciones de los swipes encima de las cabezas de los Boubas. Estas nos indican cuáles son las combinaciones necesarias para acabar con ellos antes de que nos alcancen. Representada cada flecha mediantes colores vivos como rojo, amarillo, verde, morado. Al realizar los **`swipes`**, si se ha conseguido la combinación correcta, se irán descartando las flechas de cada Bouba, cambiando al color gris, para después continuar con una animación de muerte al terminar la secuencia.

Debajo del jugador, si no se ha realizado la secuencia correctamente, aparecerá un indicador de **`‘’Missed’’`**.

5. Acelerómetro

La implementación del uso del **acelerómetro**, al igual que los "**swipes**", fue implementada gracias al uso del **New Input System** de Unity.



Se creó un nuevo archivo .**inputactions** llamado "**ShakeController**", que gracias a su "**action**" "**Accelerate**", guardará el valor de un Vector 3, resultado de los datos registrados en el acelerómetro del dispositivo móvil. Gracias al generador automático de archivos C#, tras crear este Input Actions, se creó el archivo **"ShakeController.cs".**

Para poder utilizar este sensor y asignarle una funcionalidad concreta en el videojuego, se creó la clase **"SpecialAbilitiesManager.cs".** Primeramente, se declaran las variables a utilizar. Cabe destacar el evento "**OnShockWave**" el cual será llamado tras detectar una agitación en el teléfono, cuya intensidad esta definida en la variable "***shakeThreshold***". Posteriormente, se encuentra un slider el cual mostrará cuán de cargada está la habilidad para poder usarse y finalmente encontramos el propio Inputaction "**shakeController**" y "**_lastAcceleration**" que se encargará de guardar la aceleración del dispositivo en el instante anterior.

"**TriggerShockWave**" invocará al evento "**_onShockWave**" cuando sea llamado y consumirá la carga de la habilidad especial completamente. "**Awake**" inicializa el "**inputactions**" anteriormente creado y cuando se detecta algún cambio en el acelerómetro, se ejecutará un evento que guardará los valores actuales en un

Vector3. "**Start**", habilita el acelerómetro en el dispositivo, configura el evento "**AddShockWavePoints**" y establece el valor máximo de **_pointsSlider**".

```
      public class SpecialAbilitiesManager : MonoBehaviour
      {
          event Action _onShockWave;
            public event Action OnShockWave { add { _onShockWave += value; }
remove { _onShockWave -= value; } }

          [SerializeField] EnemyManager _enemyManager;
          [SerializeField] Slider _pointsSlider;

          [SerializeField] int _requiredShockWavePoints;
          int _shockWavePoints;
          bool _shockWaveAviable;
           //Variables para Input
           private ShakeController _shakeController;
           private Vector3 _lastAcceleration;
           private float _shakeThreshold = 2.5f;
           public bool isShockWaveAviable { get { return _shockWaveAviable; } }
           public int ShockWavePoints { get { return _shockWavePoints; }
          private set { _shockWavePoints = value; UpdatePointSlider(value); } }
          ...
```

```
       public void TriggerShockWave()
       {
           _onShockWave?.Invoke();
           // Consume points:
           _shockWaveAviable = false;
           ShockWavePoints = 0;
       }

       private void Awake()
       {
           _shakeController = new ShakeController();
            _shakeController.Shake.Accelerate.performed      +=      ctx      =>
DetectShake(ctx.ReadValue<Vector3>());
       }
       private void Start()
       {
           //Busca accelerometer
           if (Accelerometer.current != null)
           {
```

```
            InputSystem.EnableDevice(Accelerometer.current);
            Debug.Log("Accelerometer enabled.");
        }
        else
        {
            Debug.LogError("No accelerometer available on this device.");
        }
        _enemyManager.OnEnemyCured += AddShockWavePonints;
        _pointsSlider.maxValue = _requiredShockWavePoints;
    }
...
```

Se implementa la recarga de la habilidad especial, mediante una función que lee si está disponible, pero que en caso contrario la recarga.

```
    void AddShockWavePonints(int points)
{
    if (_shockWavePoints + points >= _requiredShockWavePoints)
    {
        ShockWavePoints = _requiredShockWavePoints;
        _shockWaveAviable = true;
        Debug.Log("ShockWave aviable.");
    }
    else
    {
        ShockWavePoints += points;
    }
}
void UpdatePointSlider(int value)
{
    _pointsSlider.value = value;
}

...
```

Finalmente, se encuentra el método de detección de la agitación, el cual comprueba la disponibilidad del acelerómetro y calcula la aceleración, que comparada con el "threshold" se decide si se llama al método "**`TriggerShockWave`**" para activar la habilidad, pero en caso contrario, actualiza el valor "**`_lastAcceleration`**".

```
            private void DetectShake(Vector3 acceleration)
           {
               // Comprueba si el acelerómetro está disponible
               if (Accelerometer.current == null)
               {
                   Debug.LogError("Accelerometer not available.");
                   return;
               }

               // Calcula el cambio de aceleración
               Vector3 deltaAcceleration = acceleration - _lastAcceleration;

               // Si el cambio supera el umbral, activa la habilidad
                      if (deltaAcceleration.sqrMagnitude > _shakeThreshold *
_shakeThreshold && isShockWaveAviable)
               {
                   TriggerShockWave();
                   Debug.Log("Shake detected! ShockWave triggered.");
               }

               _lastAcceleration = acceleration;
           }
}
```

b. Funcionamiento

El acelerómetro se representa mediante una **barra de progreso vertical** ubicada en el lado izquierdo de la interfaz. Esta barra se llena progresivamente a medida que el jugador avanza en el nivel, matando a los Boubas. Una vez completada, desbloquea la función del acelerómetro del dispositivo, permitiendo al jugador agitar el móvil para ejecutar una **onda expansiva (shock wave)** que acaba con todos los Boubas que se encuentren en la pantalla en ese momento.

El diseño de esta mecánica tiene un doble propósito: por un lado, añade una funcionalidad exclusiva para dispositivos móviles, aprovechando los **sensores de movimiento** para introducir una interacción física que no sería posible en otras plataformas.

Por otro lado, ofrece una **herramienta estratégica** para el jugador, ya que debe decidir cuidadosamente cuándo y dónde utilizar esta habilidad para maximizar su efectividad.

Además, esta implementación busca enriquecer la experiencia del jugador al integrar una interacción directa con el dispositivo, fomentando una conexión más inmersiva con el juego. La mecánica también premia la destreza del jugador, ya que llenar la barra requiere eliminar enemigos activamente, añadiendo una capa extra de **dinamismo** y **emoción** al gameplay.

6. Conclusiones

La implementación de controles táctiles, como los **`swipes`**, y de funciones basadas en el movimiento, como el **shake**, ha permitido ofrecer una experiencia de juego inmersiva y optimizada para dispositivos móviles. Estos controles, diseñados específicamente para aprovechar las capacidades únicas de los sensores de los dispositivos, enriquecen la jugabilidad y la hacen atractiva tanto para jugadores casuales como para aquellos más competitivos.

a. Controles

La mecánica de **`swipes`** sincronizados con el ritmo musical combina acción y música de manera efectiva, creando una experiencia dinámica y entretenida. Por su parte, el uso del **`shake`** como habilidad especial introduce un componente estratégico, añadiendo variedad y profundidad al gameplay. Estas funcionalidades destacan por su capacidad de sacar provecho de las características y sensores de los dispositivos móviles, ofreciendo una jugabilidad adaptada al **formato vertical** y **táctil**.

b. Rejugabilidad

El sistema de puntuación basado en **combos**, **tiempo** de supervivencia y un **ranking** global fomenta la **rejugabilidad** al incentivar a los jugadores a mejorar sus marcas **personales** y **competir** con otros. Este enfoque no solo mantiene a los usuarios comprometidos, sino que también promueve la repetición de partidas para superar desafíos y explorar distintas estrategias.

c. Tutorial

La incorporación de mecánicas como el **`shake`** y las **`secuencias`** de ritmo requirió diseñar un tutorial claro y eficiente. Garantizar que los jugadores comprendieran rápidamente las **dinámicas** de juego fue un paso crucial para **evitar confusiones** y facilitar una experiencia inicial positiva.

d. Desafíos superados

El diseño e implementación de controles táctiles y por movimiento en dispositivos móviles presentó retos significativos, especialmente para garantizar precisión y responsividad en hardware diverso. Sin embargo, el resultado ha sido satisfactorio, logrando una sincronización fluida entre las mecánicas rítmicas y las acciones del jugador, evitando frustraciones típicas en juegos de ritmo.

Además, fue necesario ajustar las interfaces y mecánicas al formato vertical, asegurando que todos los elementos del juego se adaptaran de manera uniforme a las características de los distintos dispositivos móviles.

e. Balance y éxito del proyecto

Bouba District logra un equilibrio destacado entre **simplicidad** y **profundidad**, ofreciendo un juego fácil de aprender pero con mecánicas lo suficientemente innovadoras para mantener el interés del jugador. Su desarrollo ha combinado controles **intuitivos**, diseño **adaptado** al formato móvil y una experiencia **musical** envolvente. Esto ha sido posible gracias a una planificación sólida, la iteración constante basada en el **feedback** recibido y la capacidad de adaptarse a las necesidades y expectativas de los jugadores.