PIXEL FROG

Nombre del alumno: Sergio Sánchez Fernández

Curso académico: 2º DAM

Tutor del proyecto: Victor Colomo Gómez

ÍNDICE

1. RESUMEN

Pixel Frog es un juego de plataformas en dos dimensiones con estilo pixelart realizado con el software de Unity. En este juego tienes que recoger unas frutas repartidas por el nivel para completarlo mientras evitas trampas y enemigos mortales.

Los niveles están diseñados de manera que el grado de dificultad sea elevado, teniendo en estos una sola vida por intento pero haciendo que el completarlos requiera cierto grado de destreza y que la satisfacción que aporta al jugador sea bastante grande.

El diseño gráfico de estos se compone de un rectángulo que abarca el nivel entero, siendo en este siempre la cámara fija. Cada nivel está diseñado de una manera específica con la cual hacer una sinergia perfecta con los enemigos y trampas que componen los mismos.

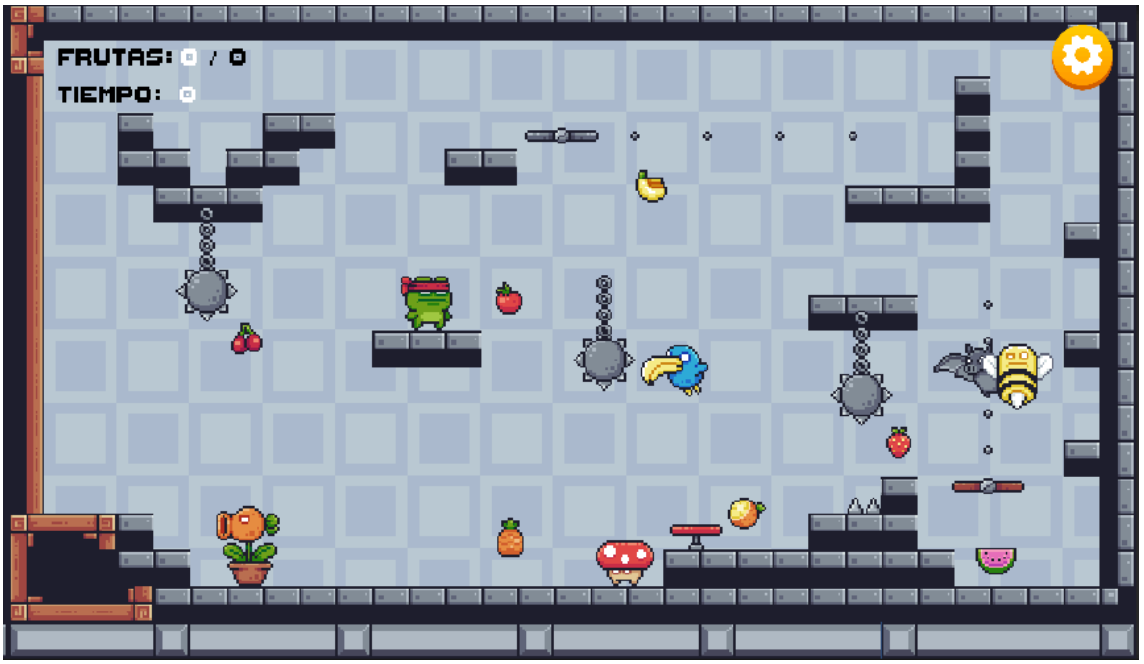


Imagen 1: Ejemplo de diseño de nivel de prueba

La movilidad que tiene el personaje junto con la rapidez con la que los niveles se reinician si el jugador pierde, hace que cada intento carezca de aburrimiento, sea adictivo y frenético.

La puntuación de estos niveles está compuesta por el tiempo que tardas en completarlos, lo que hace a los jugadores rejugar los niveles una y otra vez para mejorar sus anteriores marcas, aprendiendo nuevos caminos y formas de completar el mismo más rápidamente y ganando así más destreza y soltura.

Además, cada vez que superes tu marca personal en cualquier nivel esta se guardará localmente y se subirá a una base de datos que podrás consultar desde el menú principal del juego y así poder competir con los diferentes jugadores que también realizan sus propias marcas.

2. JUSTIFICACIÓN DEL PROYECTO

Pixel Frog nace con la idea de aportar un reto a todos aquellos jugadores que disfrutan al máximo superando sus propios marcajes, aprendiendo de los anteriores y notando una mejoría personal por cada intento realizado.

La idea de verte mejorar cada vez más y ver que eres capaz de superarte a ti mismo una y otra vez trae consigo una increíble satisfacción y sensación de mejoría que no muchas progresiones en los juegos pueden transmitir.

Aunque se trata de juego de corta duración, logra a la perfección esta idea.

Una de las otras razones fundamentales por las que este proyecto se desarrolla es por el interés que yo mismo desarrolle durante la impartición de la tecnología de Unity en las clases del grado.

Me llamo mucha la atención todas las facilidades que aporta Unity en el diseño y desarrollo de videojuegos, sobre todo en el ámbito de los juegos en 2D, siendo así mucho más fácil llevar a cabo un proyecto como este.

Después de mucha investigación previa y de darme cuenta de que un proyecto así no sería tan complicado de realizar con las tecnologías que poseía a mi alcance, me decidí por este proyecto.

Tenía muchas ideas iniciales que durante el desarrollo fue descartando y también muchas otras que fue añadiendo sobre la marcha, pero aun asi estoy muy orgulloso y contento con el resultado obtenido.

3. OBJETIVOS

A. OBJETIVO GENERAL

- Realizar un juego con una jugabilidad rápida, que presente un grado de dificultad elevado y transmita un alto nivel de satisfacción y mejoría personal al completar cada uno de los niveles.

B. OBJETIVOS ESPECÍFICOS

- Crear un menú principal que sea jugable. (Para acceder a los niveles debes usar el personaje para moverte).
- Crear una base de datos con MongoDB para guardar los resultados y poder consultarlos en un ranking creado en el menú principal.
- Crear un total de 4 niveles diferentes.
- Poder darle al jugador a elegir entre 4 aspectos de personaje.
- Crear enemigos y trampas de varios tipos.

4. DESARROLLO

Antes de empezar con las explicaciones del desarrollo de los diferentes aspectos del juego se ha de saber que todo lo relacionado con el apartado gráfico del juego (frames para las animaciones, personajes, enemigos, trampas, mapeado…) pertenece a un Asset gratuito llamado '*Pixel Adventure*' de la Asset Store que proporciona Unity (Se comentará más adelante en el apartado de assets utilizados).

TECNOLOGÍAS UTILIZADAS

Para este proyecto se han utilizado dos tecnologías principales: **Unity** (Versión: 2021.3.17f1) como motor de desarrollo y diseño y **MongoDB Atlas** como servicio de base de datos en la nube.

Unity es un motor de desarrollo de videojuegos multiplataforma ampliamente utilizado en la industria de los videojuegos y aplicaciones interactivas. Así es como funciona:

- **Definición**: Unity es un motor de desarrollo de juegos y una plataforma de creación de contenido 3D en tiempo real. Proporciona un entorno de desarrollo integrado (IDE) que permite a los desarrolladores crear juegos, aplicaciones interactivas, experiencias de realidad virtual y aumentada, entre otros proyectos.
- **Componentes básicos**: En Unity, los elementos básicos para crear un juego son los objetos de juego (game objects) que contienen componentes. Los objetos de juego representan entidades en el juego, como personajes, objetos, luces, cámaras, etc. Los componentes definen el comportamiento y las características de los objetos de juego.
- **Escenas**: Las escenas en Unity representan niveles, pantallas o secciones del juego. Puedes crear múltiples escenas y combinarlas para crear la estructura y flujo del juego. Cada escena puede contener objetos de juego y se pueden cargar y descargar según sea necesario.
- **Scripting**: Unity utiliza el lenguaje de programación C# para escribir scripts y agregar funcionalidades a los objetos de juego. Los scripts permiten controlar el

comportamiento de los objetos, realizar interacciones, implementar la lógica del juego y más.

- **Física y gráficos**: Unity cuenta con un motor de física integrado que permite simular el movimiento y la interacción realista entre objetos. También ofrece capacidades gráficas avanzadas, incluyendo renderizado en tiempo real, efectos visuales, iluminación global, sombreado y más.
- **Plataformas compatibles**: Unity es un motor multiplataforma, lo que significa que puedes desarrollar juegos y aplicaciones para diversas plataformas, como PC, Mac, consolas de videojuegos, dispositivos móviles (iOS y Android), realidad virtual (VR) y realidad aumentada (AR).
- **Ecosistema y comunidad**: Unity cuenta con una amplia comunidad de desarrolladores y una gran cantidad de recursos disponibles, como documentación, tutoriales, foros y activos (modelos 3D, texturas, efectos) en la Asset Store de Unity. Además, ofrece herramientas adicionales para monetización, analíticas y colaboración en equipo.

En **resumen**, Unity es un poderoso motor de desarrollo de videojuegos y aplicaciones interactivas que proporciona herramientas y funcionalidades para crear juegos de alta calidad y experiencias interactivas en diferentes plataformas. Su enfoque en la facilidad de uso, la versatilidad y el soporte multiplataforma lo convierten en una opción popular entre los desarrolladores.

MongoDB Atlas es un servicio de base de datos en la nube ofrecido por MongoDB. Su funcionamiento es el siguiente:

- **Definición**: MongoDB Atlas es una plataforma de base de datos administrada en la nube que permite a los desarrolladores almacenar, administrar y escalar bases de datos MongoDB sin tener que preocuparse por la configuración, administración y mantenimiento de la infraestructura subyacente.

- **Base de datos en la nube**: MongoDB Atlas permite a los desarrolladores aprovechar la flexibilidad y escalabilidad de la nube para sus bases de datos MongoDB. Proporciona un entorno de base de datos completamente administrado en la nube, lo que significa que MongoDB se encarga de las tareas de implementación, configuración, respaldo, seguridad y monitoreo de la infraestructura de base de datos.
- **Escalabilidad y disponibilidad**: MongoDB Atlas permite escalar horizontalmente tu base de datos de forma sencilla y transparente a medida que tu aplicación crece. Puedes aumentar o disminuir la capacidad de almacenamiento y la potencia de procesamiento de tu base de datos de manera rápida y eficiente. Además, Atlas ofrece características de alta disponibilidad y replicación automática para garantizar la continuidad del servicio.
- **Seguridad**: MongoDB Atlas cuenta con características de seguridad avanzadas para proteger tus datos. Ofrece opciones de autenticación, cifrado en tránsito y en reposo, así como controles de acceso basados en roles para garantizar que solo las personas autorizadas puedan acceder y modificar los datos.
- **Integración con servicios en la nube**: Atlas se integra con otros servicios en la nube, como AWS (Amazon Web Services), Azure (Microsoft Azure) y Google Cloud Platform, lo que te permite aprovechar las funcionalidades adicionales de estas plataformas en conjunto con tu base de datos MongoDB.
- **Flexibilidad geográfica**: MongoDB Atlas te permite elegir la ubicación geográfica de tus clústeres de bases de datos para cumplir con los requisitos de cumplimiento normativo y minimizar la latencia para tus usuarios.

En **resumen**, MongoDB Atlas es un servicio de base de datos en la nube que simplifica la administración y escalabilidad de bases de datos MongoDB. Proporciona un entorno de base de datos completamente administrado, escalable y seguro en la nube, permitiendo a los desarrolladores enfocarse en su aplicación en lugar de en la infraestructura subyacente.

SCRIPTS GENERALES

En este apartado trataremos los scripts generales más importantes del proyecto y que se utilizan muchas veces en diferentes tipos de objetos:

- **Damage Script**: Este script es muy simple pero fundamental para el funcionamiento del juego. Se trata del script que llevan todos los objetos que son capaces de dañar al jugador (Enemigos y trampas).

  En este se accede al box collider que posee el objeto que lleva el script y se ordena que si este colisiona con un objeto con el tag '*Player*' se lleven a cabo las acciones necesarias (Reiniciar el nivel, ejecutar animación de daño.. etc).

```
Mensaje de Unity | 0 referencias
private void OnCollisionEnter2D(Collision2D collision)
{
    /*Condicional que se encarga de cuando el jugador entra en contacto con el
    gameObject que lo lleva, ejecute la funcion PlayerDamaged() ubicada en el
    script de PlayerRespawn*/
    if (collision.transform.CompareTag("Player")) {
        collision.transform.GetComponent<PlayerRespawn>().PLayerDamaged();
    }
}
```

Imagen 2: El contendo del script Damage Script

- **Jump Damage:** Este script lo poseen todos los objetos los cuales son enemigos.

  Lo que hace es, accediendo a un collider que tienen los enemigos en la cabeza, realizar ciertas acciones cuando éste colisiona con el jugador.

  Estas acciones entre otras son: activar la animación de daño del enemigo, restarle una vida y destruir dicho objeto si este se queda sin vidas.

```
//Funcion que se activa al entrar en colision con el collider que lleva el enemigo en la cabeza
Mensaje de Unity | 0 referencias
private void OnCollisionEnter2D(Collision2D collision)
{
    //Al cumplirse la condicion se aplica la fuerza hacia arriba al personaje, y se llama a las
    //funciones LoseLifeAndHit() y CheckLife()
    if (collision.transform.CompareTag("Player"))
    {
        collision.gameObject.GetComponent<Rigidbody2D>().velocity = (Vector2.up * jumpForce);
        LoseLifeAndHit();
        CheckLife();
    }
}

//Funcion que se encarga de restarle una vida al enemigo y ejecutar la animacion de Hit del mismo
1 referencia
public void LoseLifeAndHit() {
    lifes--;
    anim.Play("Hit");
}

//Funcion que comprueba las vidas restantes del enemigo
1 referencia
public void CheckLife() {
    //Si las vidas llegan a 0 la particula de eliminacion se activa, el spriteRenderer se desabilita
    //y se llama a la funcion EnemyDie()
    if (lifes == 0)
    {
        destroyParticle.SetActive(true);
        spriteRenderer.enabled = false;
        Invoke("EnemyDie", 0.2f);
    }
}

//Funcion que destruye el gameobject perteneciente al enemigo
0 referencias
public void EnemyDie() {
    Destroy(gameObject);
}
```

Imagen 3: Contenido del script Jump Damage

- **AIBasic**: Este script es bastante importante ya que define un comportamiento de movimiento para un objeto hacia los puntos que se le indiquen. Este script se utiliza en casi todos los enemigos, en una trampa en particular y en las plataformas móviles (profundizaremos más adelante).

ANIMACIONES

Todas las animaciones que presenta este proyecto (jugador, enemigos… etc) se han realizado de una manera bastante similar utilizando los árboles de animaciones que incluye Unity y el manejo de los mismos. A continuación usaré el ejemplo del árbol de

animaciones del jugador, siendo este el más completo que hay en el juego comparado con todos los demás.

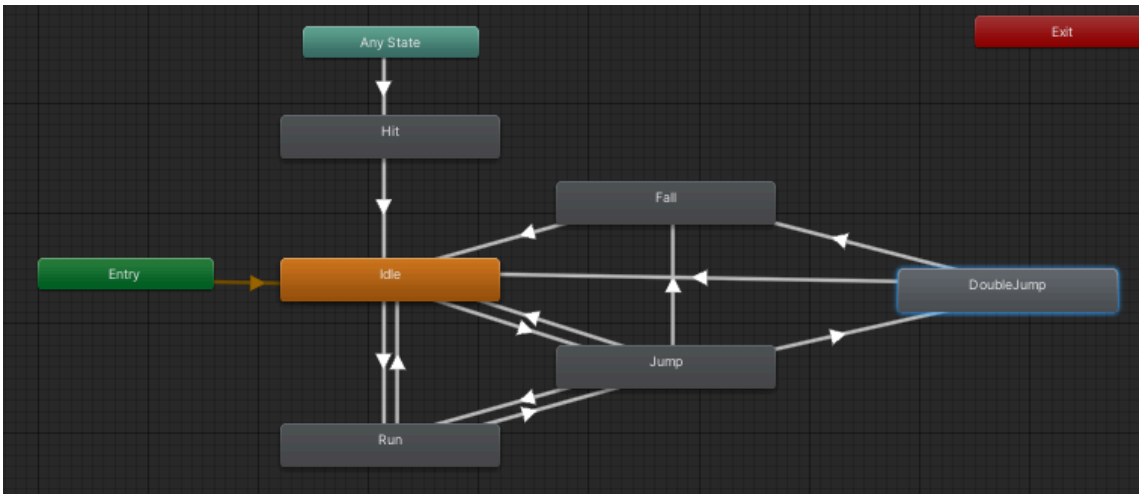


Imagen 4: Árbol de animaciones del jugador

La metodología para crear árboles de animaciones simples es bastante sencilla:

- Primero, claro está, se han de crear las diferentes animaciones a partir de los frames de las mismas. Esto se hace arrastrando dichos frames a la escena y poniendo el nombre deseado a la animación.
- Luego deberemos crear el árbol de animaciones, asignando las diferentes transiciones lógicas entre animaciones (Ej: desde la animación de reposo puedo pasar a la de correr y la de saltar)
- Por último estableceremos mediante unas variables booleanas que condiciones han de darse para pasar de una animación a otra (El valor de estas variables se modificara mediante el código).

Imágenes 5: Set de animaciones del personaje '*Ninja Frog*'

JUGADOR

El pilar fundamental del juego. Este contiene varios componentes:

- **Sprite Renderer**: Se encarga de controlar el sprite perteneciente al objeto del jugador.
- **Animator**: Se encarga de gestionar el árbol de animaciones del jugador.
- **Box Collider 2D:** El jugador posee dos Box Collider: Uno se utiliza para recibir el daño, el cual abarca todo el cuerpo; el otro está ubicado en los pies, este se encarga de ajustar ciertas variables booleanas en los scripts para saber cuando el jugador puede o no saltar y ejecutar las animaciones correspondientes.

Imagen 6: Box Colliders pertenecientes al jugador

- **Rigidbody 2D**: Se encarga de proporcionarle físicas al personaje.
- **Audio Source**: Controla los clips de audio asociados al objeto, en este caso almacena el sonido de '*Hit*' del jugador.

- **PlayerMovement**: Este script es el que se encarga de controlar todo lo relacionado con el movimiento del personaje: recoge los input de las teclas para mover al personaje, modifica las variables booleanas del árbol de animaciones para ajustar la animación del personaje en función de lo que está haciendo, comprueba cuando el personaje puede saltar o no… etc.

```
private void Update()
{
    //Comprobación que me permite saber cuando puedo saltar
    if (Input.GetKey("space"))
    {
        if (CheckGround.isGrounded)
        {
            canDoubleJump = true;
            rb2.velocity = new Vector2(rb2.velocity.x, jumpSpeed);
        }
        else
        {
            if (Input.GetKeyDown("space"))
            {
                if (canDoubleJump)
                {
                    anim.SetBool("doubleJump", true);
                    rb2.velocity = new Vector2(rb2.velocity.x, DoublejumpSpeed);
                    canDoubleJump= false;
                }
            }
        }

    }
    //Comprobación para ver si estoy saltando y ajustar las animaciones ya implementadas
    if (CheckGround.isGrounded == false)
    {
        anim.SetBool("isJumping", true);
        anim.SetBool("isRunning", false);
    }
    else
    {
        anim.SetBool("isJumping", false);
        anim.SetBool("doubleJump", false);
        anim.SetBool("isFalling", false);
    }

    if (rb2.velocity.y < 0)
    {
        anim.SetBool("isFalling", true);
    }
    else if (rb2.velocity.y > 0)
    {
        anim.SetBool("isFalling", false);
    }
}
```

```
Mensaje de Unity | 0 referencias
void FixedUpdate()
{
    //Movimiento del personaje
    //Si me muevo a la derecha
    if (Input.GetKey("d") || (Input.GetKey("right")))
    {
        rb2.velocity = new Vector2(runSpeed, rb2.velocity.y);
        sp.flipX = false;
        anim.SetBool("isRunning", true);
    }
    else if (Input.GetKey("a") || (Input.GetKey("left")))
    {
        rb2.velocity = new Vector2(-runSpeed, rb2.velocity.y);
        sp.flipX = true;
        anim.SetBool("isRunning", true);
    }
    else {
        rb2.velocity = new Vector2(0,rb2.velocity.y);
        anim.SetBool("isRunning", false);
    }

    //Implementación de salto mejorado
    if (betterJump) {
        if (rb2.velocity.y < 0) {
            rb2.velocity += Vector2.up * Physics2D.gravity.y * (fallMultiplier) * Time.deltaTime;
        }
        if (rb2.velocity.y > 0  && !Input.GetKey("space"))
        {
            rb2.velocity += Vector2.up * Physics2D.gravity.y * (lowJumpMultiplier) * Time.deltaTime;
        }
    }
}
```

Imágenes 7: Contenido del script de PlayerMovement

- **CheckGround**: Este script está asociado al box collider que lleva el personaje en los pies. Es el que se encarga de controlar si el personaje puede saltar o no en función de una variable booleana llamada isGrounded.

```
public class CheckGround : MonoBehaviour
{

    //Variable que determina si estamos en el suelo.
    //La declaramos static para poder acceder a ella desde otros scripts.
    public static bool isGrounded;

    //Función que pone isGrounded a true si tocamos el suelo
    Mensaje de Unity | 0 referencias
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Ground"))
        {
            isGrounded = true;
        }
    }

    //Función que pone isGrounded a false si no tocamos el suelo
    Mensaje de Unity | 0 referencias
    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.CompareTag("Ground"))
        {
            isGrounded = false;
        }
    }
}
```

Imagen 8: Script de CheckGround

- **PlayerRespawn**: Este es el script que se encarga de realizar las acciones correspondientes cuando se llama a la función PlayerDamaged() desde el script anteriormente comentando Damage Script.

```
//Se activa cuando el jugador recibe daño, ejecuta la animacion
//de recibir daño, reinicia el nivel desde el principio y ejecuta la transicion
2 referencias
public void PLayerDamaged() {
    gameObject.GetComponent<Collider2D>().enabled = false;
    anim.Play("Hit");
    hitSound.Play();
    interfaz.gameObject.SetActive(false);
    transition.SetActive(true);
    Invoke("ChangeToActualScene", 1);
}

//Funcion que reinicia la escena actual
0 referencias
void ChangeToActualScene() {
    SceneManager.LoadScene(SceneManager.GetActiveScene().name);
}
```

Imagen 9: Contenido del script de PlayerRespawn

- **PlayerSelect**: Este script se encarga de asignar los sprites y animaciones correspondientes al objeto del jugador en función del aspecto que elija este en el menú principal. Se asigna mediante un Player Prefab de Unity (variable local).

```
//Funcion que comprueba el valor de nuestro PlayerPrefab llamado PlayerSelected
//y asigna los sprites y animaciones que estan asiganda a dicho valor
2 referencias
public void ChangePlayerInMenu() {
    switch (PlayerPrefs.GetString("PlayerSelected"))
    {
        case "Frog":
            spriteRenderer.sprite = playersRenderer[0];
            anim.runtimeAnimatorController = playersController[0];
            break;
        case "PinkMan":
            spriteRenderer.sprite = playersRenderer[1];
            anim.runtimeAnimatorController = playersController[1];
            break;
        case "VirtualGuy":
            spriteRenderer.sprite = playersRenderer[2];
            anim.runtimeAnimatorController = playersController[2];
            break;
        case "MaskDude":
            spriteRenderer.sprite = playersRenderer[3];
            anim.runtimeAnimatorController = playersController[3];
            break;
        default:
            break;
    }
}
```

Imagen 10: Script de PlayerSelect

TRAMPAS

Todas las trampas diseñadas en este juego se han diseñado de la misma manera:

- Se han creado las animaciones siguiendo los pasos indicados en el apartado *'ANIMACIONES'* explicado anteriormente.
- Se les ha asignado un box collider en función a sus proporciones en tamaño para dañar al jugador.

Las trampas son:

- **Sierra**: Aparte de tener el diseño mencionado anteriormente, también lleva incorporado el script de **AIBasic** para poder implementar una sierra móvil.

- **Pinchos**: Objeto pequeño en forma de pinchos que sigue el diseño indicado al principio.
- **Bola de pinchos**: Este objeto a parte de seguir el diseño general, tiene dos variantes con animaciones diferentes en función de lo que se quiera usar. Una hace que la bola gire 360º y la otra que gire 180º, eso aporta más versatilidad al diseño de niveles.

Imagen 11: Sierra y su box collider

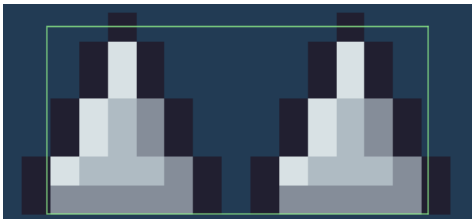


Imagen 12: Pinchos y su box collider

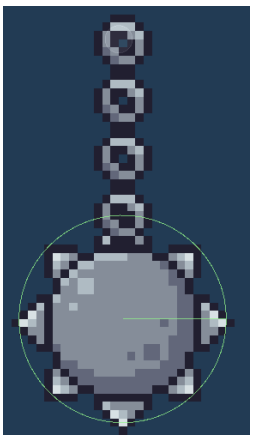


Imagen 13: Bola de pinchos y su box collider

ENEMIGOS

Hay varios enemigos desarrollados en este proyecto, pero primero vamos a ver el contenido del script **AIBasic**; este script hace que el objeto que lo lleva se mueva por los puntos pasados como parámetro en el editor con un tiempo de espera entre punto y punto controlado por una variable. Aparte de controlar esto, también lleva definida una corrutina que detecta la dirección a la que está dirigido nuestro enemigo y así poder darle la vuelta a su sprite renderer para que esté bien orientado.

También como se comentó al principio, todos los enemigos poseen un box collider en la cabeza que permite al jugador acabar con ellos al saltar sobre los mismos.

```
void Update()
{
    //Empezamos la corrutina definida mas abajo
    StartCoroutine(CheckEnemyMoving());
    //Vamos actualizando la posicion de nuestra IA en funcion de su posicion actual y el siguiente punto
    //al que tiene que dirigirse
    transform.position = Vector2.MoveTowards(transform.position, moveSpots[i].transform.position,
        speed * Time.deltaTime);

    //Condicianal que se activa si nuestra IA ha llegado al siguiente punto que se le indique
    if (Vector2.Distance(transform.position, moveSpots[i].transform.position) < 0.1f) {

        //Condicional que comprueba si nuestro tiempo de espera es menor o igual que cero para
        //que nuestra IA avance al siguiente punto o siga en el tiempo de espera
        if (waitTime <= 0) {
            //Condicional que comprueba si el siguiente punto es diferente al ultimo en su comportamiento,
            //avanzando asi al siguiente o volviendo a la posicion inicial del array
            if (moveSpots[i] != moveSpots[moveSpots.Length-1])
            {
                i++;
            }
            else
            {
                i = 0;
            }

            waitTime = startWaitTime;
        }
        else
        {
            waitTime -= Time.deltaTime;
        }

    }
}
```

```
//Corrutina que sirve para detectar si nuestra IA de esta
//moviendo a la izquieda o a la derecha y poder asi encararlo a la direccion adecuada
1 referencia
IEnumerator CheckEnemyMoving()
{
    actualPos = transform.position;
    yield return new WaitForSeconds(0.5f);

    if (transform.position.x > actualPos.x)
    {
        spriteRenderer.flipX = true;
        anim.SetBool("Idle", false);
    }
    else if (transform.position.x < actualPos.x)
    {
        spriteRenderer.flipX = false;
        anim.SetBool("Idle", false);
    }
    else if (transform.position.x == actualPos.x)
    {
        anim.SetBool("Idle", true);
    }

}
```

Imagenes 14: Contenido del script de AIBasic

Los diferentes enemigos son los siguientes:

- **Seta**: Enemigo terrestre que cuenta con su box collider en el cuerpo para dañar al jugador y en la cabeza para poder ser eliminado. Tiene incorporado el script AIBasic para su movimiento.

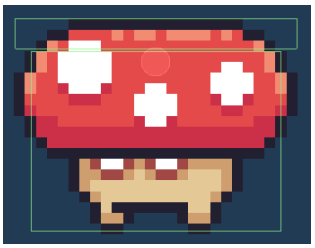


Imagen 15: Enemigo seta con sus box colliders

- **Murciélago y pájaro**: Poseen las mismas configuraciones que el enemigo *'seta'* pero estos son enemigos aéreos.

Imagen 16: Enemigo murciélago con sus box colliders

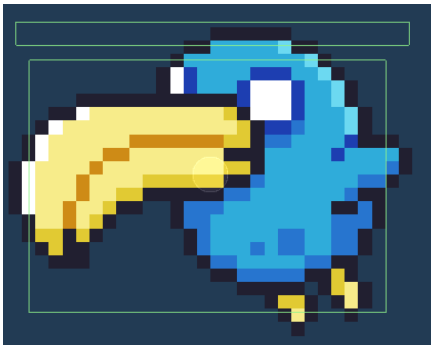


Imagen 17: Enemigo pájaro con sus box colliders

- **Planta**: Este enemigo es algo diferente ya que no posee el script AIBasic. Aparte la funcionalidad diferente que posee, la cual se almacena en el script llamado PlantEnemy, es que dispara unos proyectiles hacia delante para eliminar al jugador. Estos proyectiles solo se pueden esquivar y desaparecen después de un tiempo indicado en una variable del código.

  Los proyectiles que dispara tienen su propio box collider y son instanciados mediante código cada vez que pasa un tiempo determinado definido en una variable. El código que define el comportamiento de los proyectiles se encuentra en el script Bullet Plant.

```
Mensaje de Unity | 0 referencias
private void Start()
{
    waitedTime = waitTimeToAttack;
}

//Condicional que controla el ataque del enemigo planta según el valor de las variables
//waitedTime y waitTimeToAttack
Mensaje de Unity | 0 referencias
private void Update()
{
    //Si el tiempo de espera a terminado, lo vuelve a reiniciar y hace que la planta
    //lance la bullet con la animación asignada
    if (waitedTime <= 0)
    {
        waitedTime = waitTimeToAttack;
        anim.Play("Attack");
        Invoke("LaunchBullet", 0.5f);
    }
    //Al contrario, va restando el tiempo real a la variable hasta que acabe
    else
    {
        waitedTime -= Time.deltaTime;
    }
}

//Funcion utilizada para invocar el proyectil
0 referencias
public void LaunchBullet()
{
    //Creo e instancio el objeto bullet en la posicion pasada como parametro
    GameObject newBullet;
    newBullet = Instantiate(bulletPrefab, launchSpawnPoint.position, launchSpawnPoint.rotation);
}
```

Imagen 18: Contenido del script PlantEnemy

Imagen 19: Enemigo planta con sus box colliders

```
private void Start()
{
    //Destruye el objeto bullet al pasar el tiempo indicado en
    //la variable lifeTime despues de generarse
    Destroy(gameObject, lifeTime);
}

Mensaje de Unity | 0 referencias
private void OnCollisionEnter2D(Collision2D collision)
{
    Destroy(gameObject);
}

//En funcion del valor de la variable left, este condicional hace que
//las bullet se generen hacia la izquieda o hacia la derecha
Mensaje de Unity | 0 referencias
private void Update()
{
    if (left)
    {
        transform.Translate(Vector2.left * speed * Time.deltaTime);
    }
    else
    {
        transform.Translate(Vector2.right * speed * Time.deltaTime);
    }
}
```

Imagen 20: Contenido del script Bullet Plant

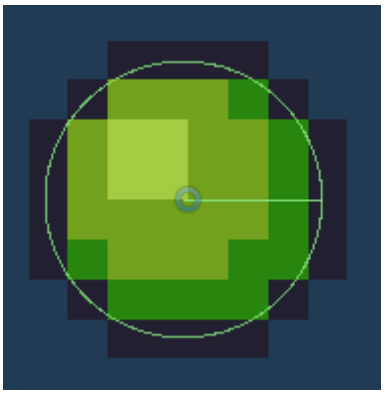


Imagen 21: Proyectil del enemigo planta

- **Abeja**: Este enemigo es volador y también lleva incorporado el script de AIBasic para moverse entre los puntos que se le indique. Lo especial que tiene este enemigo es que posee un ataque de proyectil basado en raycast, esto significa que proyecta un rayo invisible debajo suya y cuando el jugador lo atraviesa el enemigo le dispara un proyectil.

  La metodología de la programación es muy parecida al del enemigo planta, instanciando un objeto de tipo beeBullet hacia abajo para dañar al jugador.

```
private void FixedUpdate()
{
    //Se instancia el raycast hacia abajo de donde se encuentra el enemigo con la distancia definida
    //en la variable distanceRayCast
    RaycastHit2D hit2D = Physics2D.Raycast(transform.position, Vector2.down, distanceRayCast);


    if (hit2D.collider != null)
    {
        //Si el rayo colisiona con el Player y el tiempo de enfriamiento de ataque es menor que 0; se
        //ejecuta la animacion de ataque, se llama a la funcion LaunchBullet() y se reinicia el tiempo
        //de espera del ataque
        if (hit2D.collider.CompareTag("Player") && actualCoolDownAttack < 0)
        {
            Invoke("LaunchBullet",0.5f);
            anim.Play("Attack");
            actualCoolDownAttack = coolDownAttack;
        }
    }
}

//Esta funcion instancia un objeto de tipo beeBullet hacia abajo para atacar al jugador
0 referencias
void LaunchBullet()
{
    GameObject newBullet;
    newBullet = Instantiate(beeBullet, transform.position, transform.rotation);
}
```

Imagen 22: Contenido del script BeeAttack

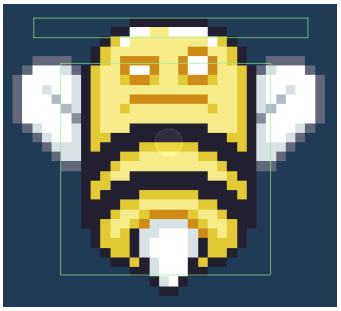


Imagen 23: Enemigo abeja con sus box colliders.

```
public class BeeBullet : MonoBehaviour
{
    //Variables que controlan la velocidad con la que
    //se lanzan y el tiempo que tardan en desaparecer las balas
    public float speed = 2.0f;
    public float lifeTime = 2.0f;

    Mensaje de Unity | 0 referencias
    private void Start()
    {
        //Destruye el objeto bullet al pasar el tiempo
        //indicado en la variable lifeTime despues de generarse
        Destroy(gameObject,lifeTime);
    }

    //Funcion que hace que los proyectiles se generen hacia abajo con
    //la velocidad definida en la variable speed
    Mensaje de Unity | 0 referencias
    private void Update()
    {
        transform.Translate(Vector2.down * speed * Time.deltaTime);
    }
}
```

Imagen 24: Contenido del script BeeBullet

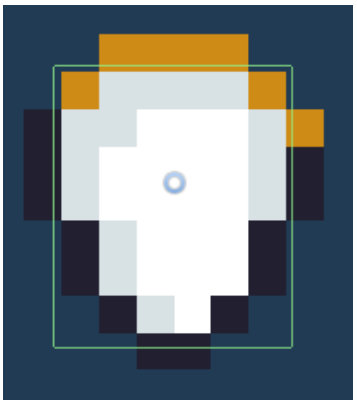


Imagen 25: Proyectil del enemigo abeja

OBJETOS DE MOVILIDAD

Se han implementado varios objetos que aportan movilidad al personaje. Estos objetos ayudan también a expandir las posibilidades del diseño de los niveles aumentando así el número de opciones disponibles y la versatilidad a la hora de diseñar.

Los objetos implementados son los siguientes:

- **Plataformas**: Han sido diseñadas para que sean doble sentido, es decir, el jugador puede subirse a ellas desde arriba y desde abajo. En lo que respecta a código tienen bastante poco ya que Unity incluye un componente llamado '*Platform Effector 2D*' que ya implementa este efecto de doble sentido.

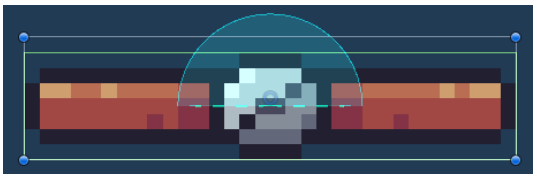


Imagen 26: Plataforma doble sentido

- **Plataformas móviles**: Estas son las plataformas normales combinadas con el script **AIBasic** para hacer que recorran distancias con patrones horizontales y verticales con la longitud que se desee.

Imagen 27: Plataforma doble sentido con desplazamiento horizontal

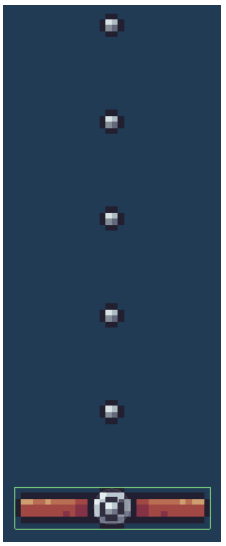


Imagen 28: Plataforma doble sentido con desplazamiento vertical

- **Trampolin**: Este objeto posee sus respectivas animaciones y un pequeño script llamado '*Trampoline*' que se encarga de impulsar hacia arriba al jugador en función del valor de la variable '*jumpforce*' cuando este entra en contacto con él.

```
//Variables para hacer referencia a los componentes en el editor
public Animator anim;
public float jumpForce;

Mensaje de Unity | 0 referencias
private void OnCollisionEnter2D(Collision2D collision)
{
    //Condicional que hace que si el jugador toca el trampolín este salte por los aires aplicandole
    //verticalmente la fuerza jumpForce y ejecuta la animación del trampolin mediante el animador
    if (collision.transform.CompareTag("Player"))
    {
        collision.gameObject.GetComponent<Rigidbody2D>().velocity = (Vector2.up * jumpForce);
        anim.Play("JumpTrampoline");
    }
}
```

Imagen 29: Contenido del script Trampoline

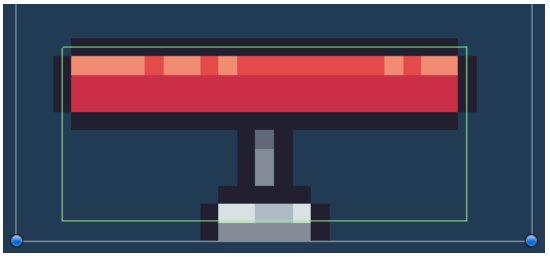


Imagen 30: Trampolín y su box collider

FRUTAS

Este objeto es el recolectable de los niveles que condiciona la victoria del jugador. Hay 8 frutas diferentes en total pero todas tienen la misma función, ser recolectadas por el jugador para determinar el fin del nivel. Cada una posee una animación y box collider diferentes.

También poseen un script llamado *'FruitCollected'* que sirve para que cuando el jugador colisiona con estas, ejecuten una animación y sonido de recogida, desaparezcan después de unos instantes y comprueben cuántas frutas quedan para determinar el fin del nivel.

```
private void OnTriggerEnter2D(Collider2D collision)
{
    /*Condicional que al chocar el jugador con una fruta hace que esta se
     * desactive y aparezca la animación y sonido de recogida. Pasados unos
     * instantes destruye dicha fruta. También detecta si se han acabado
     * las frutas del nivel*/
    if (collision.CompareTag("Player")) {
        GetComponent<SpriteRenderer>().enabled= false;
        gameObject.transform.GetChild(0).gameObject.SetActive(true);
        FindObjectOfType<FruitManager>().AllFruitCollected();
        Destroy(gameObject, 0.2f);
        clip.Play();
    }
}
```

Imagen 31: Contenido del script FruitCollected

Imágenes 32: Las diferentes frutas y sus box colliders

CONDICIÓN DE VICTORIA

La condición de victoria de los niveles de este juego consiste en obtener una serie de objetos recolectables (frutas) sin morir. Todos estos objetos recogibles son hijos de un objeto padre el cual posee un script que comprueba el número de hijos que quedan para que cuando este sea igual a 0 el nivel se complete.

Este script se llama '*FruitManager*' y ya que es el que determina el fin de nivel, también tiene implementado el guardado de datos local y en la nube usando la base de datos de MongoDB (se explicará más adelante).

```
private void Update()
{
    //Compruebo constantemente cuantas frutas quedan en el nivel para actualizar el
    //texto que lleva la cuenta de las mismas y para acabar el nivel cuando se recojan todas
    AllFruitCollected();
    totalFruits.text = totalFruitsInLevel.ToString();
    fruitsColleted.text = (totalFruitsInLevel - transform.childCount).ToString();
}

//Función que detecta si todas las frutas del nivel han sido recogidas para activar la transicion del nivel,
//guardar los records localmente y en la nube (si se han hecho) y cambiar la escena al Main Menu
2 referencias
public void AllFruitCollected() {
    if (transform.childCount == 0) {
        //Una vez las frutas han sido recogidas se desactiva el collider del jugador
        player.GetComponent<Collider2D>().enabled = false;
        transition.SetActive(true);
        canva.gameObject.SetActive(false);
        safeScoreLocallyAndOnline();
        Invoke("ChangeSceneToMainMenu", 1f);
    }
}

//Funcion que cambia la escena al Menu Principal
0 referencias
void ChangeSceneToMainMenu()
{
    SceneManager.LoadScene("MainMenu");
}
```

Imagen 33: Contenido del script FruitManager que se encarga de comprobar la condición de victoria del nivel.

CREACIÓN DE NIVELES

La creación de niveles de este proyecto ha resultado increíblemente fáciles debido a las facilidades que aporta Unity y al haber escogido un Asset tan completo.

La metodología ha sido siguiente:

- Primero se ha creado un Grid el cual es una cuadrícula que define el tamaño de los bloques que componen los niveles, es este caso después de varias pruebas decidí que el tamaño perfecto eran celdas era de 0.16 x 0.16.
- Seguidamente se crea un objeto de tipo TileMap para este objeto Grid, este TileMap es como una especie de 'paleta de bloques' desde el que podemos elegir qué bloque queremos 'pintar' sobre nuestro grid. Se configuró el TileMap previo a esto para que los bloques tuvieran un collider en función de su tamaño

y que se crearán con un tipo de material el cual no diese problemas a la hora de colisionar con el jugador.

- Una vez hecho esto lo único que restaba era echarle imaginación y diseñar los niveles a mi gusto en función de los recursos que tenía (enemigos, trampas… etc).

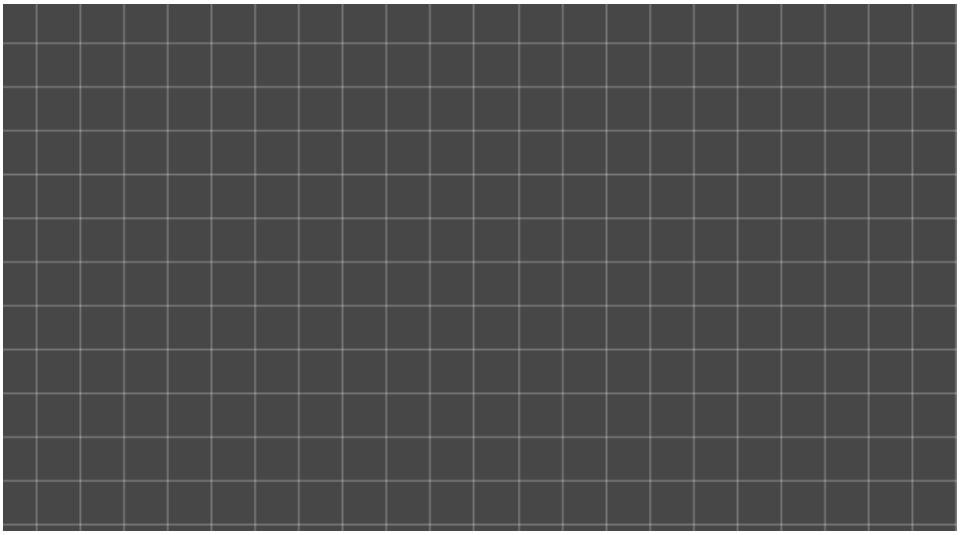


Imagen 34: Grid con tamaño 0.16 x 0.16 por bloque

Imagen 35: TileMap (Paleta) desde donde se seleccionan los bloques para ponerlos en el grid.

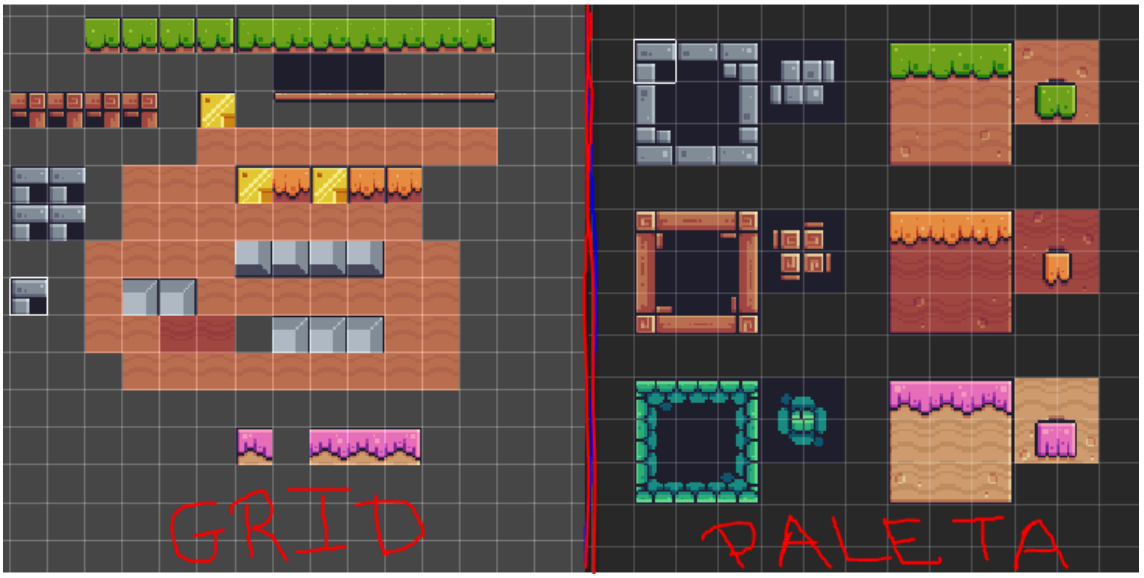


Imagen 36: Ejemplo 'pintando' bloques sobre el grid

- Una vez diseñados los niveles solo quedaba arrastrar mis enemigos y trampas previamente diseñados para tener un nivel completo y colocar las frutas para su recolección. A continuación los 4 diferentes niveles que he diseñado:

Imagen 37: Diseño nivel 1

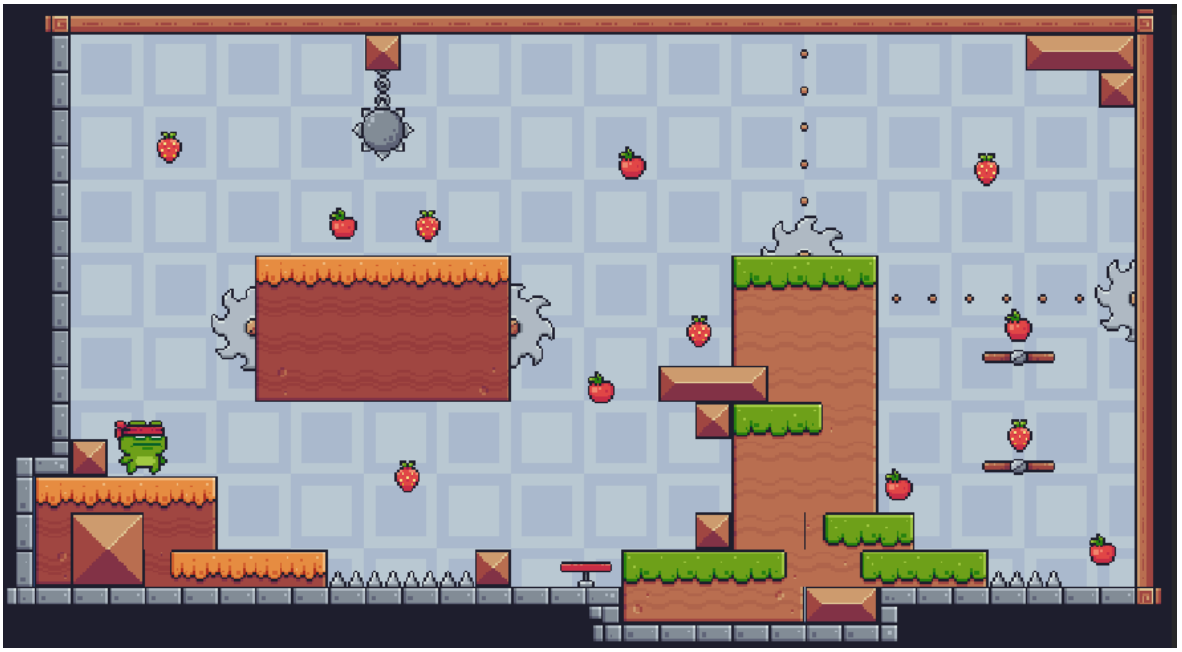


Imagen 38: Diseño nivel 2

Imagen 39: Diseño nivel 3

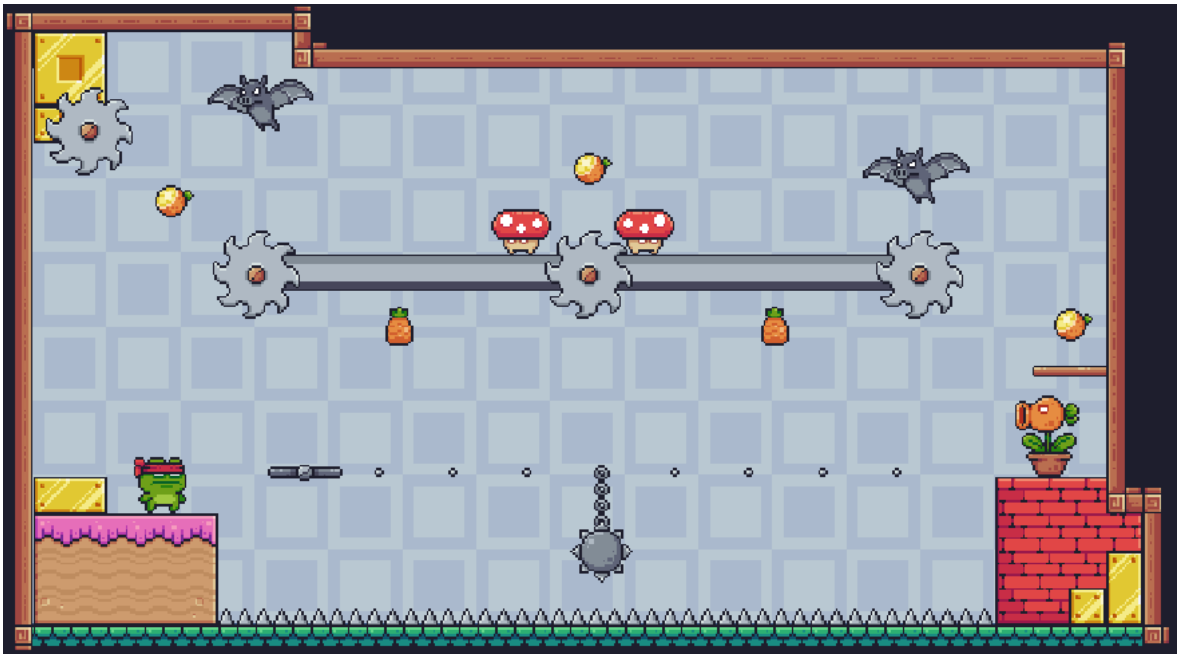


Imagen 40: Diseño nivel 4

Imagen 41: Diseño del menú principal

INTERFAZ

Las interfaces en este proyecto están compuestas por los menús de pausa ubicados en el menú principal y en los niveles y las tablas que muestran el registro de los récords de los jugadores.

Primero los menus, se ha de saber que para acceder a estos desde los niveles y el menu principal se hace pulsando un boton circular de fondo amarillo con un engranaje blanco que se encuentra siempre en la esquina superior derecha de la pantalla.

Al pulsarlo, nuestro juego se pondrá en pausa y nos mostrará el menú correspondiente en función de en qué escena nos encontremos.

Imagen 42: Botón para acceder al menú de pausa ubicado en la esquina superior derecha.

Existen dos menús de pausa implementados en este proyecto, ambos se han diseñado de la misma manera. Lo único diferente entre ambos son las acciones que podemos realizar con ellos:

- **Menú de pausa 1 (Main Menu)**: Este es el menú de pausa que pertenece a la escena principal del juego, el Main menu. Al abrirlo tendremos tres botones con los que podremos interactuar:
    - **Reanudar**: Nos hará volver a la escena actual desactivando el menú de pausa y reanudando el tiempo.
    - **Cambiar nombre**: Nos mostrará un mensaje y un campo donde podemos introducir el nombre de usuario al que queramos cambiar (este nombre es el que se usa en la base de datos). También puedes salir con otro botón si cambias de idea.
    - **Salir del juego**: Cierra el juego.

Imagen 42: Menú de pausa de la escena Main Menu

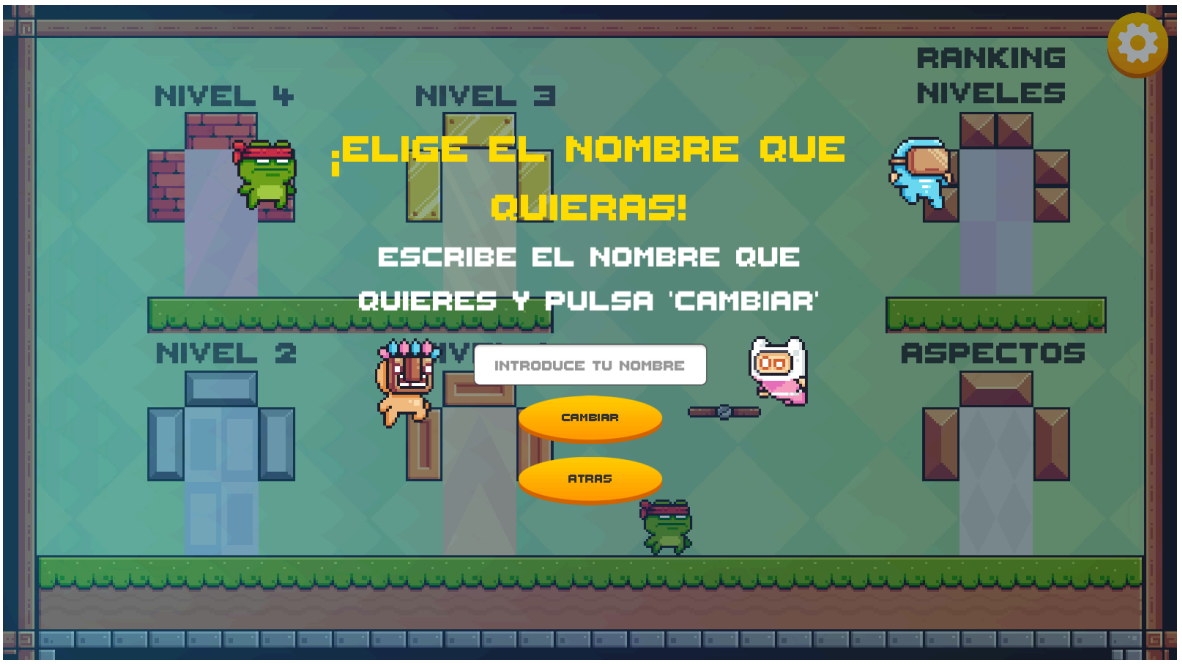


Imagen 43: Ventana mostrada al pulsar el botón '*Cambiar nombre*'

- **Menú de pausa 2 (Niveles)**: Este es el menú que se nos mostrará al abrir el menú de pausa en cualquiera de los niveles jugables. Estas son las opciones disponibles al abrirlo:
    - **Reanudar**: Reanuda la escena y desactiva el menú de pausa.

- **Reiniciar**: Reinicia el nivel desde el principio.
- **Menú principal**: Nos lleva a la escena del Main Menu.
- **Salir del juego**: Cierra el juego

Imagen 44: Menú de pausa de los niveles

También en las esquina superior izquierda en todos los niveles tenemos dos contadores diferentes para informar al jugador que se actualizan en tiempo real:

- Uno posee el número de frutas que ya hemos recogido frente a las totales en el nivel.
- El otro posee el tiempo que ha transcurrido hasta el momento

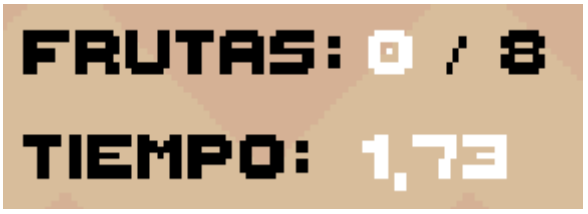


Imagen 45: Contador de frutas y de tiempo

Por otro lado tenemos las tablas que se mostrarán al jugador al acercarse a la puerta de 'Ranking niveles' ubicada en el menú principal. Se tratan de unas sencillas tablas donde podemos diferenciar el ranking de los 10 mejores tiempos en cada nivel. En la

parte de abajo de la tabla se encuentran unos botones para poder cambiar entre las tablas de los diferentes niveles y poder ver sus rankings.

Imagen 46: Tabla de ranking de puntuaciones de niveles

Finalmente tenemos una pantalla emergente que solamente se mostrará si detecta que es la primera vez que entramos al juego. Poseyendo una breve descripción sobre la funcionalidad del juego y controles seguido de un campo input para introducir el nombre del jugador:

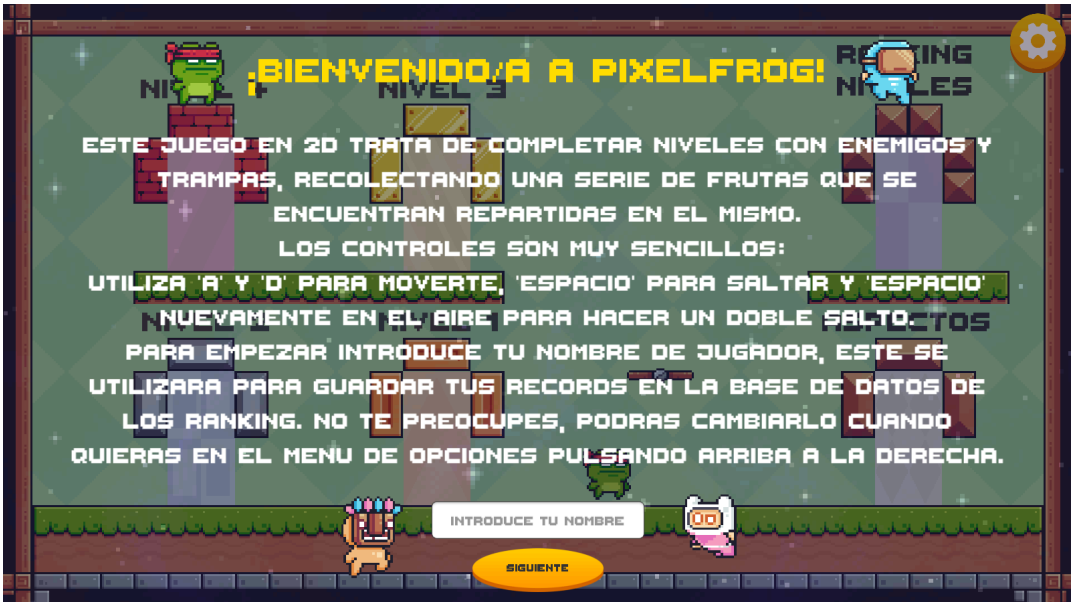


Imagen 47: Ventana emergente al entrar al juego por primera vez

GUARDAR Y RECUPERAR DATOS EN MONGODB

Lo primero para poder realizar cualquier acción sobre una base de datos de MongoDB desde Unity es importar los archivos dll. de MongoDB a nuestro proyecto, estos son los que nos proporcionan las clases y los recursos para realizar la conexión y sus posteriores acciones sobre la base de datos.

Simplemente hay que descargarlos y arrastrarlos a la carpeta del proyecto.

Respecto al **guardado de datos local** es bastante simple, en mi script que controla la condición de victoria de los niveles (FruitManager) compruebo si el record hecho en ese momento es el primero o mejor que el anterior y de ser asi lo guardo en una variable local de Unity (PlayerPrefs).

Para el **guardado en la nube** (ubicado en el mismo script que el guardado local)se realizan las mismas comprobaciones que para el guardado local, posterior a eso establezco la conexión a la base de datos con una función llamada '*establecerConexion()*' que contiene todo lo necesario para ello. Una vez establecido la conexión inserto un documento de tipo BSON que contiene tres campos (nombre de jugador, tiempo empleado y nivel actual) mediante un método asíncrono perteneciente a la librería de MongoDB.Driver importada mediante los archivos dll. mencionados anteriormente.

```
void establecerConexion() {
    try
    {
        string connectionString = "mongodb+srv://adminUser:[redacted]@cluster0.uigfjpa.mongodb.net/";

        // Creamos el cliente de MongoDB
        client = new MongoClient(connectionString);
        // Obtenemos la referencia a la base de datos
        database = client.GetDatabase("HighScoresDB");
        // Obtenemos la referencia a la colección
        collection = database.GetCollection<BsonDocument>("HighScores");

        Debug.Log("Conexion exitosa");
    }
    catch (DnsClient.DnsResponseException)
    {
        Debug.Log("No se ha podido establecer la conexion con el servidor");
        throw;
    }
    catch (System.Exception)
    {
        Debug.Log("Ocurrio un error inesperado");
        throw;
    }
}
```

Imagen 48: Función que establece la conexión con la BBDD de MongoDB

```
//Funcion que se encarga de guardar localmente y en la nube el tiempo empleado
//al acabar el nivel si este es mayor que la marca anterior o si no se habia hecho ninguno antes
1 referencia
void safeScoreLocallyAndOnline() {
    //Variables que almacenan el nombre del PlayerPref que contiene el record del nivel,
    //el nombre del nivel actual y el tiempo que has necesitado para acabarlo
    string nombrePlayerPref = "";
    string escenaActual = SceneManager.GetActiveScene().name;
    float tiempo = timer.timer;
    string playerName = PlayerPrefs.GetString("PlayerName");
    //Mediante la variable escenaActual asigno un valor al nombre
    //del PlayerPref que quiero analizar a continuacion
    switch (escenaActual)
    {
        case "Nivel1":
            nombrePlayerPref = "levelOneScore";
            break;
        case "Nivel2":
            nombrePlayerPref = "levelTwoScore";
            break;
        case "Nivel3":
            nombrePlayerPref = "levelThreeScore";
            break;
        case "Nivel4":
            nombrePlayerPref = "levelFourScore";
            break;

    }
    //Compruebo si se habia hecho algun record antes
    if (PlayerPrefs.GetFloat(nombrePlayerPref) == 0)
    {
        //Se guarda el record localmente en el PlayerPrefab
        PlayerPrefs.SetFloat(nombrePlayerPref, tiempo);
        //Se establece la conexion con la BBDD de MongoDB y se inserta el record
        if (Application.internetReachability != NetworkReachability.NotReachable)
        {
            SaveScoreToDataBase(playerName, tiempo, escenaActual);
        }
    }
    else
    {
        //Comparamos si el tiempo actual es mejor que el record anteriormente
        //guardado, si lo es, se actualiza el record
        if (PlayerPrefs.GetFloat(nombrePlayerPref) > tiempo)
        {
            //Se guarda el record localmente en el PlayerPrefab
            PlayerPrefs.SetFloat(nombrePlayerPref, tiempo);
            //Se establece la conexion con la BBDD de MongoDB y se inserta el record
            if (Application.internetReachability != NetworkReachability.NotReachable)
            {
                SaveScoreToDataBase(playerName, tiempo, escenaActual);
            }
        }
    }
}
```

Imagen 49: Función que se encarga de guardar los datos localmente y en la nube

```
//Esta funcion realizara una insercion con los datos pasados como parametros a la BBDD
2 referencias
public async void SaveScoreToDataBase(string user, float score, string level)
{
    establecerConexion();
    // Creamos un documento BSON con los datos pasados como parametros
    var document = new BsonDocument
    {
        { "Usuario", user },
        { "Puntuacion", score },
        { "Nivel", level }
    };

    // Insertamos el documento en la colección
    await collection.InsertOneAsync(document);

    Debug.Log("Documento insertado correctamente.");
}
```

Imagen 50: Función que realiza la inserción en la MongoDB mediante un documento de tipo BSON.

Ahora en lo que respecta a la **recuperación de los datos**, estos se realizan en un script llamado '*DatabaseController*' ubicado en los gameobject que contiene las tablas de los ranking de las puntuaciones de los niveles. La conexión se realiza al principio de la misma manera que la función 'establecerConexion()' explicada anteriormente.

La función asíncrona principal para realizar esto es la llamada QueryCollection(), en esta realizo los siguientes pasos:

- Primero creo los parámetros de la consulta quiero realizar; en este caso filtro los documentos para que sean iguales al nombre de un nivel pasado por parámetro, los ordeno ascendentemente por el campo puntuación y limito los resultados a solamente 10 registros.
- Lo que me devuelve dicha consulta lo recorro mientras existan documentos y voy almacenando los datos en los atributos de la clase HighScore (creada por mi) para posteriormente guardarlo en una lista de este tipo y devolverla.
- En otra función llamada MostrarScoresNivel() obtengo la lista de objetos de tipo HighScore devuelta por la función anterior y la voy recorriendo para asignar sus valores a los campos de texto de la tabla en donde se muestran los ranking de puntuaciones de los jugadores.

```
//Esta funcion asincrona de tipo Task devuelve una lista de la clase HighScore la cual
//contiene los registros de los niveles
1 referencia
public async Task<List<HighScore>> QueryCollection(string level, GameObject gameObject)
{
    // Creamos el filtro que tendra la consulta, en este caso obtendra solo los documentos
    // donde el campo Nivel sea igual al level pasado como parametro
    var filter = Builders<BsonDocument>.Filter.Eq("Nivel",level);
    // Creamos un filtro de ordenacion, en este caso ordenara los resultados ascendentemente
    // por el campo puntuacion
    var sort = Builders<BsonDocument>.Sort.Ascending("Puntuacion");
    // Creamos un objeto de tipo BsonDocument que contendra las opciones de nuestra busuqeda
    var options = new FindOptions<BsonDocument>
    {
        //El filtro de ordenacion que hemos creado
        Sort = sort,
        //Esto hace que solo obtenga los primeros cinco documentos
        Limit = 10
    };
    // Realizamos la consulta pasandole el filtro y las opciones de busqueda.
    // Esto devuelve un puntero de tipo IAsyncCursor<BsonDocument> que recorreremos ahora
    var result = await collection.FindAsync(filter,options);

    //Lista en la que guardare los registros para luego devolverla
    List<HighScore> resultList = new List<HighScore>();

    // Recorremos el puntero devuelto en la consulta
    while (await result.MoveNextAsync())
    {
        //Variable que me lleva la cuenta de los registros que entran para saber el orden de los mismos
        int contador = 0;
        //Por cada documento devuelto en la consulta sacamos los campos que necesitamos
        var batch = result.Current;
        foreach (var document in batch)
        {
            //Variables para almacenar los datos de los documentos
            var user = "";
            var score = " ";
            var nivel = "";

            // Accede a los campos del documento
            user = document["Usuario"].AsString;
            score = document["Puntuacion"].AsDouble.ToString("f2") + " s";
            nivel = document["Nivel"].AsString;

            //Sumo uno al contador e inicializo un objeto HighScore para poder guardar
            //los datos de la consultas en sus atributos
            contador++;
            var highscore = new HighScore();
            highscore.Username = user;
            highscore.rank = contador;
            highscore.score = score;
            highscore.level = level;
            //Añado el objeto highscore a la lista definida anteriormente
            resultList.Add(highscore);
        }
    }
    //Devuelvo la lista
    return resultList;
}
```

Imagen 51: Función 'QueryCollection' encargada de realizar la consulta a la BBDD de MongoDB.

```
//Funcion asincrona que sirve para escribir los registros obtenidos con la
//funcion QueryCollection() en las tablas de las puntuaciones
1 referencia
public async void MostrarScoresNivel(string level, GameObject gameObject) {
    //Resultado de llamar a la funcion QueryCollection (Lista de HighScores)
    var result = await QueryCollection(level, gameObject);

    //Utilizo un bucle foreach para iterar la lista de los highScores
    foreach (var item in result)
    {
        //Al estar el script introducido en el objeto padre que contiene los textos,
        //accedo a ellos con las siguientes funciones y cambio su valor a los registros de la BBDD
        var row = gameObject.transform.Find("Row" + item.rank);
        row.Find("RankText").GetComponent<Text>().text = item.rank.ToString();
        row.Find("UserText").GetComponent<Text>().text = item.Username;
        row.Find("ScoreText").GetComponent<Text>().text = item.score;
    }
}
```

Imagen 52: Funcion 'MostrarScoresNivel' encargada de introducir en las tablas los datos obtenidos de la función 'QueryCollection'

```
//Objeto que voy a utilizar para guardar los datos de las consultas
4 referencias
public class HighScore{
    2 referencias
    public string Username { get; set; }
    3 referencias
    public int rank { get; set; }
    2 referencias
    public string score { get; set; }
    1 referencia
    public string level { get; set; }
}
```

Imagen 53: Clase HighScore creada para almacenar los datos de la consulta

CASOS DE USO

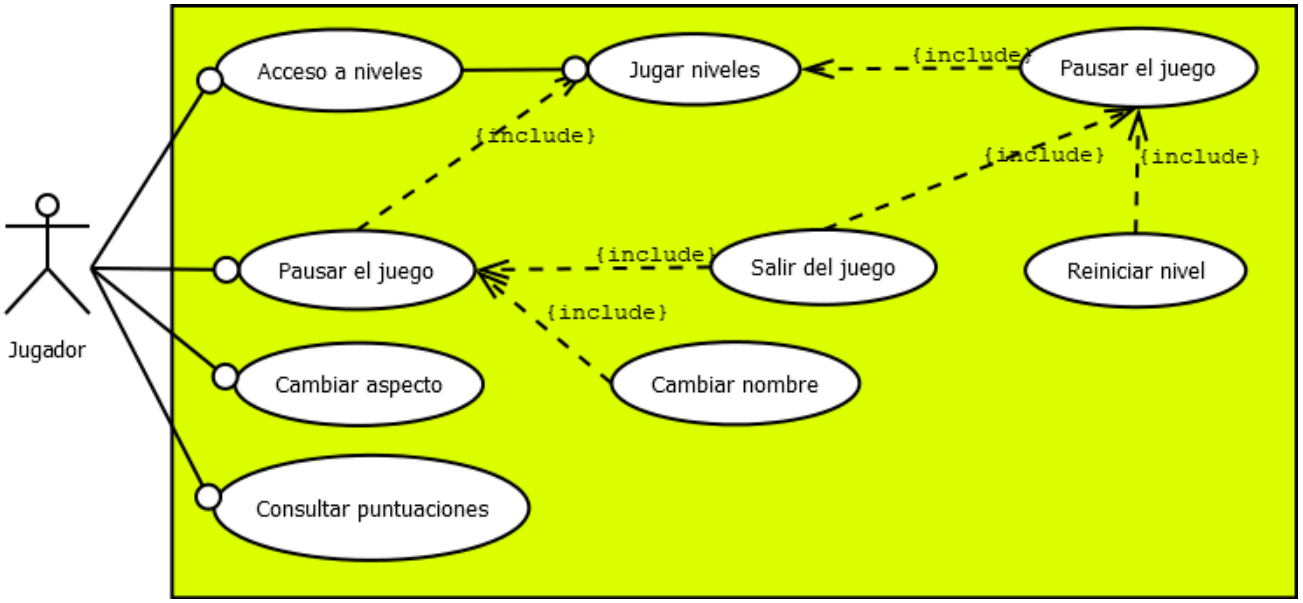


Imagen 54: Diagrama de casos de uso de Pixel Frog.

A continuación un breve explicación de cada caso de uso:

Caso de Uso: Consultar puntuaciones
Actor: Jugador.
Descripción: Se muestra una tabla con las puntuaciones de los jugadores por nivel.
Precondiciones: Que el jugador se identifique con su nombre de jugador (la primera vez que entra).
Curso Normal
1 - El jugador se acerca a la puerta asignada a las puntuaciones.
2 - El sistema muestra la tabla de las puntuaciones de los jugadores por nivel.
Alternativas
1 - Se muestra un mensaje de error junto a la tabla por no tener conexión a internet o por un error inesperado.

Caso de Uso: Pausar el juego
Actor: Jugador.
Descripción: Se pausa la escena actual y se muestra el menú de pausa.
Precondiciones: Que el jugador se identifique con su nombre de jugador (la primera vez que entra).
Curso Normal
1 - El jugador pulsa el botón de ajustes situado en la parte superior derecha.
2 - El sistema pausa la escena actual y muestra el menú de pausa.

Caso de Uso: Cambiar aspecto
Actor: Jugador.
Descripción: Se cambia el aspecto actual del personaje.
Precondiciones: Que el jugador se identifique con su nombre de jugador (la primera vez que entra).
Curso Normal
1 - El jugador se acerca a la puerta asignada a los aspectos.
2 - El jugador hace clic sobre el aspecto al que desea cambiar.
3 - El sistema cambia el aspecto del personaje al elegido por el jugador.

Caso de Uso: Acceso a niveles
Actor: Jugador.
Descripción: Se cambia de escena a la del nivel seleccionado.
Precondiciones: Que el jugador se identifique con su nombre de jugador (la primera vez que entra).
Curso Normal
1 - El jugador se acerca a la puerta y pulsa la tecla ‘E’.
2 - El sistema cambia la escena a la del nivel seleccionado.

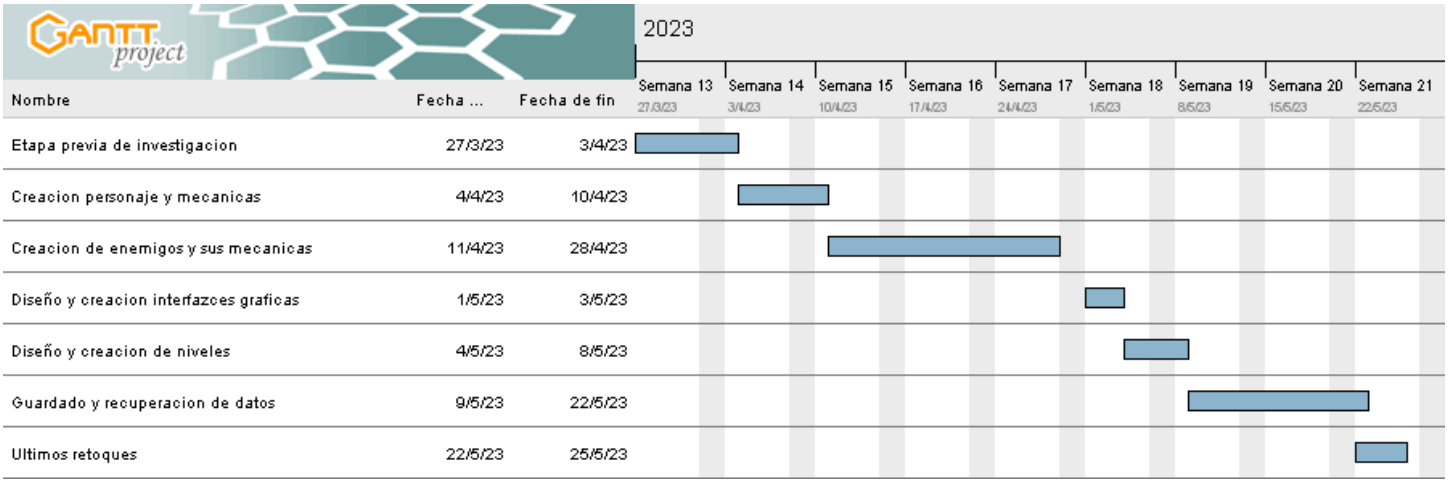
Caso de Uso: Jugar niveles
Actor: Jugador.
Descripción: El jugador completa el nivel en el que se encuentra.
Precondiciones: Que el jugador haya accedido al nivel.
Curso Normal
1 - El jugador recolecta todos los recogibles y completa el nivel.
2 - El sistema cambia la escena a la del menú principal.
Alternativas
1 - El nivel se reinicia porque el jugador muere o elige la opción reiniciar nivel desde el menú de pausa.

Caso de Uso: Cambiar nombre
Actor: Jugador.
Descripción: El jugador cambia su nombre.
Precondiciones: Que el jugador active el menú de pausa en el menú principal.
Curso Normal
1 - El jugador introduce su nuevo nombre y pulsa el botón ‘cambiar’.
2 - El sistema cambia la variable local del nombre del jugador.
Alternativas
1 - El jugador cancela el cambio de nombre pulsando el botón ‘atrás’.

Caso de Uso: Salir del juego
Actor: Jugador.
Descripción: Se cierra el juego.
Precondiciones: Que el jugador active el menú de pausa desde cualquier escena.
Curso Normal
1 - El jugador pulsa el botón ‘Salir del juego’ en el menú de pausa.
2 - El sistema cierra la aplicación.

Caso de Uso: Reiniciar nivel
Actor: Jugador.
Descripción: Se reinicia el nivel actual.
Precondiciones: Que el jugador active el menú de pausa desde cualquier nivel.
Curso Normal
1 - El jugador pulsa el botón ‘Reiniciar nivel’ en el menú de pausa.
2 - El sistema reinicia el nivel actual.

DIAGRAMA DE GANTT



Imágenes 55: Diagrama de Gantt

5. CONCLUSIONES

Se ha obtenido a la perfección el objetivo principal del proyecto: conseguir un juego de una dificultad elevada, con una jugabilidad frenética y adictiva y con una sistema de mejoría satisfactorio para el jugador.

Respecto a los objetivos secundarios: se ha creado un menú principal jugable, se ha aportado al jugador la opción de elegir entre cuatro aspectos, se han diseñado 4 niveles diferentes con dificultad escalable, el guardado y recuperación de datos con MongoDB ha sido un éxito y se han creado enemigos de y trampas de diferentes tipos para aportar versatilidad al diseño de niveles.

Todos los objetivos del proyecto se han cubierto y completado con creces.

6. LÍNEAS DE INVESTIGACIÓN FUTURAS

Posibles desarrollos futuros:

- Nueva mecánica en el personaje que permite deslizarse por las paredes pudiendo saltar desde ellas.
- Crear nuevos niveles, pudiendo ser estos más grandes que los actuales haciendo que la cámara siga al personaje por el mismo.
- Implementación de nuevos enemigos y trampas.
- Añadir nuevas animaciones al aparecer y morir.
- Creación a mano de nuevos aspectos.

7. WEBGRAFÍA

Autor	Fecha	Título	Fuente	
LuisCanary	2020 - 2021	Plataformas 2D Unity	Youtube	https://www.youtube.com/playlist?list=PLNEAWvYbJJ9kZpaIg2RfzAc_KZixBgchT
Mr.Pineapple Studio	2020, 20 de abril	How to: Connect your Unity game to Database, MongoDB Atlas, leaderboard, multiplayer cross platform	Youtube	https://www.youtube.com/watch?v=v-3NNV93jDE&t=1421s
Fundación Wikimedia, Inc.	2023, 23 mayo	Unity (motor de videojuego)	Wikipedia	https://es.wikipedia.org/wiki/Unity_(motor_de_videojuego)
Unity technologies	2023	Unity Documentation	Unity	https://docs.unity.com
Unity Technologies	2023, 2 de junio	Unity Manual	Unity	https://docs.unity3d.com/Manual/index.html
MongoDB, Inc.	2023	MongoDB Documentation	MongoDB	https://www.mongodb.com/docs/
Legis Music	2023	Musica libre de derechos	LegisMusic	https://legismusic.com/es/musica-sin-copyright/
Pixabay GmbH	2023	Efectos de sonidos sin copyright	PixaBay	https://pixabay.com/es/sound-effects/

8. ANEXOS

Assets utilizados:

- **Pixel adventure 1 y Pixel adventure 2**: Los dos assets principales del juego que contienen todo lo relacionado con el apartado gráfico: animaciones, sprites, bloques… etc.
  https://assetstore.unity.com/packages/2d/characters/pixel-adventure-1-155360
  https://assetstore.unity.com/packages/2d/characters/pixel-adventure-2-155418
- **Free Pixel Font - Thaleah**: Fuente de letra pixelada usada en el juego.
  https://assetstore.unity.com/packages/2d/fonts/free-pixel-font-thaleah-140059
- **Simple Button Set 01**: Set de botones utilizados en el apartado de menús del juego.
  https://assetstore.unity.com/packages/2d/gui/icons/simple-button-set-01-153979
- **Pixel Skies Demo Background Pack**: Fondo utilizado en el mensaje de bienvenida a los nuevos jugadores.
  https://assetstore.unity.com/packages/2d/environments/pixel-skies-demo-background-pack-226622

Estructura de carpetas: La estructura de carpetas empleada en el proyecto ha sido la siguiente:

- **Animations**: Para almacenar las animaciones del fondo y de las transiciones de nivel.
- **Assets**: Para almacenar los assets utilizados durante el proyecto
- **Music and sounds**: Para almacenar la música y los efectos de sonido del juego.
- **Prefabs**: Para almacenar los prefabs del proyecto una vez diseñados e implementados (enemigos, trampas…).
- **Scenes**: Para almacenar las diferentes escenas del proyecto.
- **Scripts**: Para almacenar todos los scripts utilizados en el proyecto. Esta carpeta a su vez se subdivide en:
  - **Enemies**: Los scripts relacionados con los enemigos
  - **Player**: Los scripts relacionados con el jugador.
  - **Probs**: Los scripts relacionados con los elementos de movilidad
  - **Scripts generales**: Resto de scripts sin categoría fija.

9. OTROS PUNTOS

Retos personales:

- La verdad que la realización de este proyecto en si ya ha sido un reto para mi ya que es la primera vez que hago algo similar, pero si tuviera que escoger algo del proyecto que mas me costo eso seria la conexión con la base de datos de MongoDB y el guardado y lectura de datos de la misma.

Agradecimientos:

- Quiero agradecer a mi amigo Mario Manuel Padilla por aportar diferentes ideas durante el proyecto y por ser mi tester oficial de niveles y versiones tempranas del juego.
- También a todas las demás personas que se ofrecieron a probar mi juego una vez estaba acabado para darme feedback y poder mejorar.