«Talento Tech»

Desarrollo de Videojuegos

Unity 3D

Clase 03

Buenos Aires Ciudad

Clase N° 3 | Mecánicas de Plataforma

Temario:

- Implementación de plataformas.
- Plataformas de caída.
- Plataformas movibles
- Plataformas rotatorias.

Objetivos de la clase

En esta clase vas a aprender a crear distintos tipos de plataformas para construir escenarios más dinámicos y desafiantes.

Ya implementaste movimiento, salto, doble salto y dash. Ahora vas a diseñar plataformas que **respondan al jugador** y **ponen a prueba esas habilidades**.

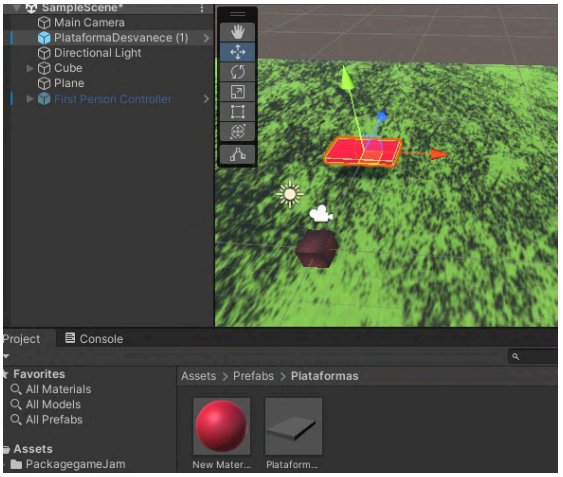
Vas a trabajar con:

- **Plataformas de caída**, que se destruyen o desploman al ser pisadas.
- **Plataformas movibles**, que se desplazan entre puntos.
- **Plataformas rotatorias**, que giran constantemente.

Estas mecánicas te permitirán aplicar colisiones, triggers, físicas y lógica de movimiento. Al final, vas a tener las bases para construir un **nivel vertical jugable**, donde el entorno y el personaje interactúan de forma fluida.

Plataformas de caída.

A esta altura ya dominás los fundamentos del movimiento y la física en Unity. Ahora vamos a dar el siguiente paso en el diseño de entornos dinámicos: **crear plataformas que reaccionan al jugador**.



En este caso, vas a trabajar con plataformas que **se destruyen o colapsan al ser pisadas**, generando tensión y obligando al jugador a actuar rápido. Son ideales para niveles verticales, secuencias de escape o tramos que exigen precisión y ritmo.

Entonces crearemos plataformas que:

- Se destruyan tras ser pisadas.
- O bien, **caigan primero** y luego se destruyan.
- Respondan solo si el jugador está parado **encima**, y no si las toca lateralmente.

Variante 1: Plataforma que desaparece después de ser tocada

La primera que trabajaremos será la típica plataforma que al tocarla desaparece o empieza a caer. Para esto empezaremos colocando nuestras propias plataformas que serán algunos cubos con la escala modificada. También podríamos convertirlos en prefab y ponerles algún material para identificarlos. Y ahora pasaremos a crear un simple código que si la plataforma colisiona con un objeto con el **tag** de “Player”, ésta se destruirá en 2 sec.

```
    [SerializeField]
    private float timeToDestroy = 2f;
    private void OnCollisionEnter(Collision collision){
        if (collision.gameObject.CompareTag("Player"))
        {
            Destroy(gameObject, timeToDestroy);
        }
    }
```

- `CompareTag("Player")`: asegura que solo reacciona al personaje.
- `Destroy(gameObject, timeToDestroy)`: borra la plataforma luego del tiempo indicado.

¿Qué hace este código?

- Detecta la colisión con el jugador (por tag).
- Inicia una cuenta regresiva (timeToDestroy) y luego destruye la plataforma.
- Ideal para efectos de desvanecimiento o trampas simples.

Variante 2: Plataforma que cae y luego desaparece

Ahora haremos que primero empiece a caerse y termine por desaparecer. Esto será cuestión de agregarle un RigidBody y manipularlo.

```
   [SerializeField] private float timeToDestroy = 2f;
    private Rigidbody rb;
    private void Start(){
        rb = GetComponent<Rigidbody>();
        rb.isKinematic = true;
              rb.constraints  =  RigidbodyConstraints.FreezePositionZ  |
RigidbodyConstraints.FreezePositionX;
    }
    private void OnCollisionEnter(Collision collision){
        if (collision.gameObject.CompareTag("Player"))
        {rb.isKinematic = false;
         Destroy(gameObject, timeToDestroy);
        }}
```

Aquí se aprovecha el `Rigidbody`:

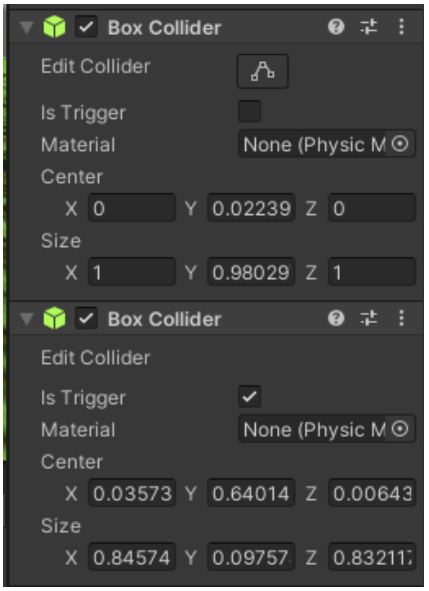
- Está en modo **Kinematic** hasta que el jugador lo pisa.
- Se le permite caer **solo en eje Y**.
- Se destruye automáticamente luego de un tiempo.

Lo que hacemos en este caso es **modificar las propiedades del componente Rigidbody** para controlar cuándo se activa la física. Al inicio, configuramos el Rigidbody como **"Kinematic"**, lo que desactiva la influencia de la gravedad y evita que la plataforma se mueva. Además, **congelamos los ejes X y Z** para que, cuando finalmente caiga, lo haga solo en el eje Y (vertical), manteniendo un comportamiento controlado.

Cuando el jugador entra en contacto con la plataforma, cambiamos la propiedad `isKinematic` a **`false`**, lo que permite que la gravedad la afecte y comience a caer. Sin embargo, esto trae un problema: **la plataforma se activa incluso si el jugador la toca desde abajo o desde los costados**, por ejemplo, al saltar y chocar con su base. Para

solucionar esto y hacer la mecánica más precisa y versátil, vamos a implementar un sistema que **detecte si el jugador está realmente parado encima de la plataforma**, evitando falsos positivos y mejorando la experiencia de juego.

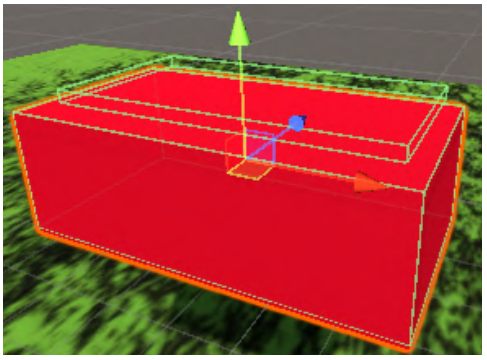
✅Mejora: Detectar si el jugador está parado sobre la plataforma



Para mejorar la precisión del comportamiento de la plataforma, vamos a implementar un sistema que **verifique si el jugador realmente está parado sobre ella**, y no simplemente tocándola desde los costados o desde abajo.

Paso 1: Agregar un *trigger* superior

Primero, añadimos un **segundo Box Collider** a la plataforma. Este nuevo collider debe ubicarse **encima de la superficie**, un poco más pequeño que el original, y marcarse como **"Is Trigger"**. Su función será detectar si el jugador entra en contacto desde arriba.



Paso 2: Reemplazar `OnCollisionEnter` por `OnTriggerStay`

Luego, modificamos el código para que use `OnTriggerStay`, el cual se ejecuta mientras el jugador permanece dentro del área del trigger. Además, hacemos una verificación adicional para comprobar que el jugador está **por encima** de la plataforma:

```
        private void OnTriggerStay(Collider other) {
        if (other.gameObject.CompareTag("Player")) {
            // Obtener la posición del jugador y la plataforma
            Vector3 playerPosition = other.transform.position;
            Vector3 platformPosition = transform.position;
            // Verificar si el jugador está por encima de la plataforma
            if (playerPosition.y > platformPosition.y) {
                rb.isKinematic = false;
                Destroy(gameObject, timeToDestroy);
                Debug.Log("El jugador está encima de la plataforma.");
            }
        }  }
```

¿Cómo funciona?

Este sistema se basa en un **doble chequeo**:

1. El jugador debe estar **dentro del trigger superior** (indica contacto con la parte de arriba).
2. Su posición vertical (**Y**) debe ser **mayor** que la de la plataforma (indica que está efectivamente arriba, no al costado ni debajo).

Solo si se cumplen ambas condiciones, la plataforma activa su caída y comienza la cuenta regresiva para ser destruida.

Plataformas movibles.

En esta etapa vas a diseñar plataformas que **se desplazan a lo largo de rutas definidas**, permitiendo que el jugador acceda a nuevas zonas o supere obstáculos. Este tipo de mecánica es muy común en juegos de plataformas, acción o aventuras, ya que **agrega movimiento al entorno** y exige al jugador atención, sincronización y planificación.

Además de construir el movimiento, vas a resolver un problema clave: **cómo hacer que el jugador se mueva junto con la plataforma sin resbalarse ni quedar desfasado**, lo que implica integrar lógica de colisiones y posicionamiento.

Para hacer esto, podemos utilizar el concepto que vimos en el **Nivel1** de los **"CheckPoints"** que marcaban 2 posiciones para que la plataforma rebote entre ellas.

Así que lo que haremos será crear 2 **emptyObjects** dentro de la plataforma con un **BoxCollider** cada uno:

Luego procederemos a realizar el código de la plataforma, dentro del Parent contenedor de los 3 GameObjects:

Movimiento entre dos puntos:

Primero, estructuramos la plataforma con:

- Un objeto principal (**Parent**).
- Tres hijos (**Childs**):
    - El objeto físico de la plataforma (el que se mueve).
    - Punto A (posición inicial).
    - Punto B (posición final).

```
 private Transform posA;
 private Transform posB;
 private Transform platMove;
 private Transform currentTarget;
[SerializeField] private float speed = 10f;
    void Start(){
        platMove = transform.GetChild(0);
        posA = transform.GetChild(1);
        posB = transform.GetChild(2);
        currentTarget = posA;
      }
     void Update(){
        Movimiento();
    }
    private void Movimiento(){
        // Calcula la distancia hacia el objetivo actual
         float distance = Vector3.Distance(platMove.transform.position,
currentTarget.position);
        Debug.Log(distance);
        // Si está cerca del objetivo, cambia al otro punto
        if (distance < 0.1f) {
            if (currentTarget == posA){
                currentTarget = posB;
            }
            else{
                currentTarget = posA;
            }
        }
        // Mueve la plataforma hacia el objetivo actual
                    Vector3 direction = (currentTarget.position -
platMove.transform.position).normalized;
                  platMove.transform.Translate(direction * speed *
Time.deltaTime);
    }
```

- **`Vector3.Distance()`** calcula qué tan cerca está la plataforma del objetivo actual.
- Si está muy cerca (`< 0.1f`), cambia el objetivo (`posA` ↔ `posB`).
- **`Translate()`** mueve el objeto suavemente hacia el objetivo.
- Se usa `.normalized` para mantener una dirección constante sin importar la distancia.

Tengan en cuenta que para hallar los Empty que representarán las posiciones hay distintas maneras:

- Podemos usar el inspector para dejar un prefab ya seteado.
- Podemos usar “GetChild” que nos buscara los childs del objeto mediante un “index”: https://docs.unity3d.com/6000.0/Documentation/ScriptReference/Transform.GetChild.html
- O buscar cualquier otra forma que nos parezca.

🗨️**Comentario:** Ahora, este código ya fue semi-explicado en el Nivel1, pero vamos a repasarlo un poco.

Repaso rápido.

Variables y Propiedades.

```
 private Transform posA;
 private Transform posB;
 private Transform platMove;
 private Transform currentTarget;
[SerializeField] private float speed = 10f;
```

- **`private Transform posA;` y `private Transform posB;`**: Representan los dos puntos entre los cuales se moverá el objeto.
- **`private Transform platMove;`**: Es la referencia al objeto que se moverá entre los puntos `posA` y `posB`.
- **`private Transform currentTarget;`**: Mantiene el objetivo actual hacia el cual el objeto `platMove` se está moviendo.
- **`[SerializeField] private float speed = 10f;`**: Controla la velocidad del movimiento de la plataforma. El atributo `[SerializeField]` permite editar este valor desde el Inspector de Unity.

Función Start().

```
    void Start(){
        platMove = transform.GetChild(0);
        posA = transform.GetChild(1);
        posB = transform.GetChild(2);
        currentTarget = posA;
      }
```

Este método se ejecuta una vez al inicio del juego. Aquí se inicializan las referencias y valores:

1. **`platMove = transform.GetChild(0);`**: Toma el primer hijo del GameObject actual como la plataforma que se moverá.
2. **`posA = transform.GetChild(1);`**: Toma el segundo hijo del GameObject actual como el primer punto de destino.
3. **`posB = transform.GetChild(2);`**: Toma el tercer hijo del GameObject actual como el segundo punto de destino.
4. **`currentTarget = posA;`**: Inicializa el objetivo actual en el punto `posA`.

Función Movimiento()

```
 private void Movimiento(){
        // Calcula la distancia hacia el objetivo actual
        float distance = Vector3.Distance(platMove.transform.position,
currentTarget.position);
        Debug.Log(distance);
        // Si está cerca del objetivo, cambia al otro punto
        if (distance < 0.1f) {
            if (currentTarget == posA){
                currentTarget = posB;
            }
            else{
                currentTarget = posA;}
        }  // Mueve la plataforma hacia el objetivo actual
      Vector3      direction      =      (currentTarget.position      -
platMove.transform.position).normalized;
     platMove.transform.Translate(direction * speed * Time.deltaTime) }
```

1. **`float distance = Vector3.Distance(platMove.transform.position, currentTarget.position);`**: Calcula la distancia entre la posición actual de la plataforma (`platMove`) y la posición del objetivo actual (`currentTarget`).

2. **`if (distance < 0.1f)`**: Verifica si la plataforma está suficientemente cerca del objetivo actual.
    - Si es verdad, cambia el objetivo:
        - **`currentTarget = posB;`**: Si el objetivo era `posA`, ahora será `posB`.
        - **`currentTarget = posA;`**: Si el objetivo era `posB`, ahora será `posA`.
3. **`Vector3 direction = (currentTarget.position - platMove.transform.position).normalized;`**: Calcula la dirección hacia el objetivo actual. El método `.normalized` asegura que el vector de dirección tenga una longitud de 1, para usarlo como dirección.
4. **`platMove.transform.Translate(direction * speed * Time.deltaTime);`**: Mueve la plataforma en la dirección calculada.
    - `speed`: Controla qué tan rápido se mueve.
    - `Time.deltaTime`: Hace que el movimiento sea independiente de la velocidad del hardware.

¡Perfecto! Con esto nuestra plataforma se moverá repetidamente de un lado a otro.

PlayerLock.

Uno de los problemas más comunes al trabajar con plataformas móviles es que, **cuando el jugador se sube a la plataforma, esta no lo transporta junto con su movimiento**. El personaje queda físicamente sobre ella, pero no se traslada a la par, lo que genera una sensación poco realista o frustrante.

Este comportamiento se puede resolver de distintas maneras. En esta clase vamos a implementar una **solución sencilla pero efectiva**, que consiste en **sincronizar la posición del jugador con la plataforma** mientras esté en contacto con ella.

Para lograrlo, vamos a colocar el siguiente código **dentro del objeto físico de la plataforma** (es decir, el *Child* que se mueve, no el *Parent* contenedor del sistema):

```
// Lock Player a la plataforma
private bool lockPlayer;
private Transform player;
void Start(){
   lockPlayer = false;
}
 void Update(){
   PlayerLock();
 }
private void PlayerLock(){
  //Si el player esta lockeado y la variable NO es null, lo muevo junto
con la plataforma
 if (lockPlayer && player != null){
   Debug.Log("Moviendo");
   player.transform.position    =    new    Vector3(transform.position.x,
player.transform.position.y, transform.position.z);
    }}
private void OnCollisionEnter(Collision collision){
  //Si toco al jugador y "lockPlayer" es false, obtengo el componente y
pongo "lockPlayer" como true
   if (!lockPlayer && collision.gameObject.CompareTag("Player")){
       Debug.Log("Locked");
       player = collision.gameObject.GetComponent<Transform>();
       lockPlayer = true;
    }}
```

```
private void OnCollisionExit(Collision collision){
       if (lockPlayer && collision.gameObject.CompareTag("Player")){
           Debug.Log("UnLocked");
           player = gameObject.GetComponent<Transform>();
           lockPlayer = false;
       }}
```

¿Qué logramos?

- Mientras el jugador está **sobre** la plataforma, su posición en X y Z se actualiza junto con ella.
- Al separarse, se libera la referencia (`lockPlayer = false`).
- Esto crea la sensación de que el jugador **viaja con la plataforma** sin necesidad de subclases físicas o jerarquías.

Este código implementa un sistema que "bloquea" al jugador a la plataforma cuando esta detecta una colisión con el jugador. La plataforma fuerza al jugador a moverse junto con ella mientras está "bloqueado" (lockPlayer = true), y lo "desbloquea" (lockPlayer = false) cuando el jugador deja de estar en contacto con la plataforma.

Explicación detallada

Variables:

```
    private bool lockPlayer;
    private Transform player;
```

`lockPlayer`: Variable booleana que indica si el jugador está "bloqueado" a la plataforma.

- Si es `true`, la plataforma moverá al jugador junto con ella.

`player`: Almacena la referencia al `Transform` del jugador para poder manipular su posición.

Función Start():

```
void Start(){
   lockPlayer = false;
}
```

Inicializa la variable `lockPlayer` como `false` para indicar que, al inicio, el jugador no está "bloqueado" a la plataforma.

Función PlayerLock():

```
private void PlayerLock(){
  //Si el player esta lockeado y la variable NO es null, lo muevo junto
con la plataforma
 if (lockPlayer && player != null){
   Debug.Log("Moviendo");
   player.transform.position    =    new    Vector3(transform.position.x,
player.transform.position.y, transform.position.z);
    }
}
```

Verifica si **`lockPlayer` es `true` y si la variable `player` no es `null`**.
Si ambas condiciones son verdaderas:

- Mueve al jugador para que coincida con la posición **X** y **Z** de la plataforma.
- Mantiene la posición **Y** del jugador intacta para evitar que se desplace verticalmente.

Función OnCollisionEnter():

```
private void OnCollisionEnter(Collision collision){
  //Si toco al jugador y "lockPlayer" es false, obtengo el componente y
pongo "lockPlayer" como true
   if (!lockPlayer && collision.gameObject.CompareTag("Player")){
       Debug.Log("Locked");
       player = collision.gameObject.GetComponent<Transform>();
       lockPlayer = true;
    }
 }
```

Este método se ejecuta cuando algo colisiona con la plataforma.

Verifica si:

- **`lockPlayer` es `false`**, para evitar que vuelva a bloquear al jugador si ya está bloqueado.
- **El objeto colisionado tiene el tag `Player`**, asegurándose de que solo interactúe con el jugador.

Si ambas condiciones son ciertas:

- **`player`**: Guarda la referencia al `Transform` del jugador para manipular su posición.
- **`lockPlayer = true`**: Activa el estado de "bloqueo".
- **`Debug.Log("Locked")`**: Escribe un mensaje en la consola para depuración.

Función OncollisionExit():

```
    private void OnCollisionExit(Collision collision){
        if (lockPlayer && collision.gameObject.CompareTag("Player")){
            Debug.Log("UnLocked");
            player = gameObject.GetComponent<Transform>();
            lockPlayer = false;
        }
    }
```

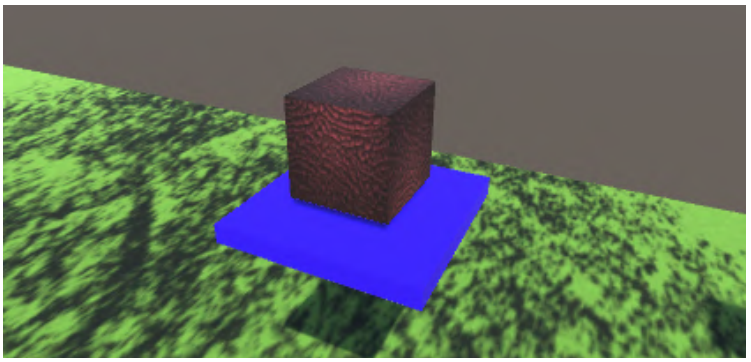
Este método se ejecuta cuando algo deja de colisionar con la plataforma.
Verifica si:

- **`lockPlayer` es `true`**, indicando que el jugador estaba bloqueado.
- **El objeto colisionado tiene el tag `Player`**, asegurándose de que solo interactúe con el jugador.

Si ambas condiciones son ciertas:

- **`player`**: Se limpia la referencia al jugador.
- **`lockPlayer = false`**: Cambia el estado de "bloqueo" a `false`, indicando que el jugador ya no está adherido a la plataforma.
- **`Debug.Log("UnLocked")`**: Escribe un mensaje en la consola para depuración.

Al terminar esto nuestra plataforma no solo se moverá, sino que mantendrá a nuestro personaje moviéndose con ella
.

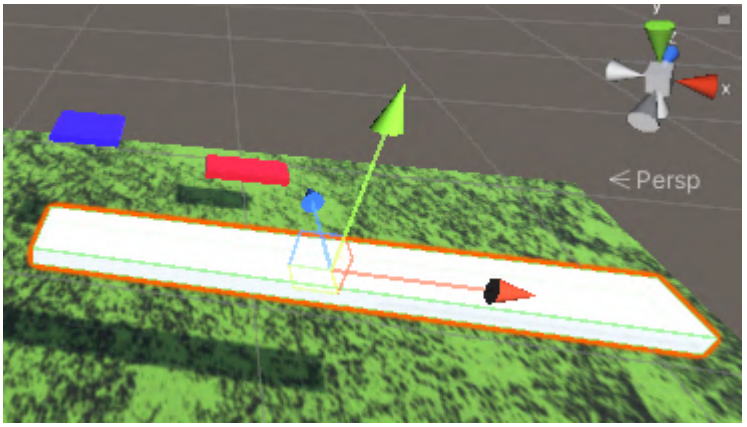


Plataformas rotatorias.

Ya implementaste plataformas que **reaccionan al jugador** (caen o desaparecen), y otras que **se desplazan entre puntos**. El siguiente paso es incorporar plataformas que **giren continuamente**, generando un nuevo tipo de desafío: el equilibrio dinámico.

Este tipo de plataforma no cambia de posición, pero sí de **orientación**, lo que obliga al jugador a adaptarse al movimiento circular y calcular mejor sus saltos o tiempos. Es un recurso frecuente en niveles intermedios o avanzados de juegos de plataformas 3D.

Primero crearemos una plataforma y le modificaremos la escala:



Y le colocaremos un código bastante simple de rotación:

```
[SerializeField]
private float speed = 1f;
void Update()
{
    transform.Rotate(0,speed,0);
}
```

Con esto, nuestra plataforma rotará sobre el eje Y, como si fuera la hélice de un helicóptero. El desafío siempre será intentar maniobrar sobre ella para poder alcanzar otros espacios.

¿Qué hace este código?

- Gira la plataforma sobre su **eje Y**, es decir, de forma horizontal.
- El valor de `speed` determina **cuántos grados rota por frame** (ajustable desde el Inspector).
- `Update()` lo hace funcionar **continuamente** en tiempo real.

Materiales y recursos adicionales.

Colliders:
https://docs.unity3d.com/es/2018.4/Manual/CollidersOverview.html

Translate:
https://docs.unity3d.com/6000.0/Documentation/ScriptReference/Transform.Translate.html

Rotate:
https://docs.unity3d.com/ScriptReference/Transform.Rotate.html

Destroy:
https://docs.unity3d.com/6000.0/Documentation/ScriptReference/Object.Destroy.html

Preguntas para reflexionar.

- ¿Qué importancia tiene aprender a crear variedades de opciones con ideas simples o limitadas?
- ¿Por qué es importante “exprimir” el conocimiento básico?
- ¿Cómo pueden aportar las plataformas de un Videojuego a inmersión y narrativa?

Próximos pasos.

En la próxima clase exploraremos las animaciones en entornos 3D. Si bien ya vimos animaciones en 2D y pudimos conocer tanto el Animation como Animator, ahora veremos cómo crear nuestras propias animaciones simples e importaremos personajes con animaciones más complejas para poder utilizarlas.

Situación en TechLab.

El cliente está encantado con los avances del prototipo. Los controles del personaje y las habilidades como el doble salto y el dash han sido un éxito rotundo. Sin embargo, el equipo de diseño de niveles ha planteado un nuevo desafío: las mecánicas actuales necesitan escenarios más dinámicos y desafiantes para realmente destacar.

La nueva solicitud del cliente:

El cliente quiere implementar un nivel vertical que aproveche al máximo las habilidades del personaje. Este nivel deberá estar lleno de plataformas interactivas que brinden variedad, dinamismo y retos a los jugadores. Han solicitado específicamente los siguientes tipos de plataformas:

1. **Plataformas Estáticas**: Elementos básicos que sirvan como puntos de apoyo.
2. **Plataformas de Caída**: Superficies que se desintegren o caigan al ser pisadas, añadiendo un componente de tiempo y tensión.
3. **Plataformas Movibles**: Mecanismos que permitan al personaje desplazarse a nuevas áreas, ya sea horizontal o verticalmente.
4. **Plataformas Rotatorias**: Superficies que giren en torno a un eje, desafiando la precisión y el equilibrio del jugador.

El desafío del nivel vertical

El cliente espera que el jugador experimente una progresión emocionante en este nivel, donde el doble salto y el dash sean esenciales para avanzar. Cada tipo de plataforma debe estar cuidadosamente diseñado para aportar algo único al nivel:

- **Las plataformas de caída** pondrán a prueba los reflejos y la rapidez de decisión del jugador.
- **Las plataformas movibles** deberán crear oportunidades estratégicas para explorar.
- **Las plataformas rotatorias** agregarán un nivel de complejidad que requerirá precisión y paciencia.

Requisitos técnicos

La interacción entre el personaje y las plataformas debe sentirse natural y fluida. Además, algunas plataformas podrían necesitar ajustes adicionales en su comportamiento físico, como detección de colisiones y sincronización de movimientos con el tiempo.

El reto adicional

El cliente quiere ver un nivel vertical completamente funcional al final de esta clase. ¿Podemos garantizar que las plataformas sean lo suficientemente diversas y emocionantes para captar la atención de los inversionistas?

¡Manos a la obra!

Comencemos con las bases: diseñar e implementar cada tipo de plataforma. Una vez que tengamos todas las piezas, construiremos el nivel vertical y realizaremos pruebas para asegurarnos de que cumpla con los estándares. ¡Es hora de subir el nivel, literalmente!

Ejercicios prácticos:

Elizabeth, en su rol de Lead Game Designer desea asignarte el diseño de las plataformas del juego. El cliente quiere que llevemos el diseño un paso más allá. El nivel vertical de **Nexus** debe ser una experiencia única, y para lograrlo, necesitamos crear plataformas híbridas que combinen las mecánicas existentes o que introduzcan nuevas ideas.

TalentoLab te desafía a mostrar tu creatividad e ingenio, diseñando plataformas que sorprendan al jugador y añadan profundidad al nivel. Estas combinaciones no solo pondrán a prueba tus habilidades técnicas, sino que también demostrarán tu capacidad para innovar.

1. **Fusionando mecánicas:**
   Diseña una nueva plataforma que combine las características de dos tipos existentes. Algunas ideas incluyen:
   - Una plataforma **rotatoria** que también **se mueve entre puntos definidos**.
   - Una plataforma **movible** que, tras alcanzar ciertos puntos, realiza un **giro completo** en el eje X o Y.
2. **Añadiendo interactividad:**
   Desarrolla plataformas que respondan a la interacción del jugador:
   - Una plataforma **movible** que, al ser tocada, empieza a caer tras unos segundos.
   - Una plataforma que **gire en el eje Y** mientras se mueve verticalmente, desafiando al jugador a mantenerse sobre ella.
3. **Creando patrones avanzados:**
   Implementa plataformas con trayectorias o comportamientos complejos:
   - Una plataforma **movediza** con múltiples puntos de parada, siguiendo un patrón más elaborado.
4. **Construyendo un prototipo híbrido:**
   Usa las plataformas creadas para diseñar una sección de nivel donde estas mecánicas combinadas sean el núcleo del desafío.
   - Diseña el recorrido para que el jugador tenga que adaptarse a diferentes tipos de plataformas en rápida sucesión.
   - Prueba y ajusta la dificultad, asegurándote de que las plataformas híbridas sean funcionales y divertidas.


Buenos Aires
aprende
Agencia de Habilidades para el Futuro
Buenos
Aires
Ciudad