«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

Desarrollar plataformas de caída.

- Implementar plataformas que desaparezcan o caigan tras ser pisadas por el personaje.

Crear plataformas movibles.

- Diseñar plataformas que se desplacen a lo largo de rutas definidas usando animaciones o scripts.
- Sincronizar el movimiento de la plataforma con el personaje para evitar deslizamientos o problemas de física.

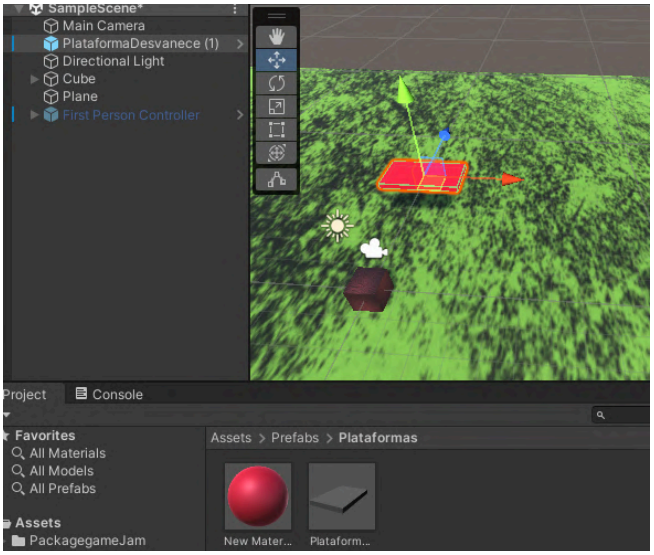
Configurar plataformas rotatorias.

- Implementar plataformas que giren constantemente o en respuesta a eventos.
- Ajustar las propiedades de rotación para mantener un desafío adecuado y realista para el jugador.

Desvanecer o caída.

La primera que haremos 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);
    }
}
```

Crearemos la variable “timeToDestroy” para gestionar el tiempo que tarda en destruirse y posiblemente, podremos usarla para otras cosas más adelante.

Caída:

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

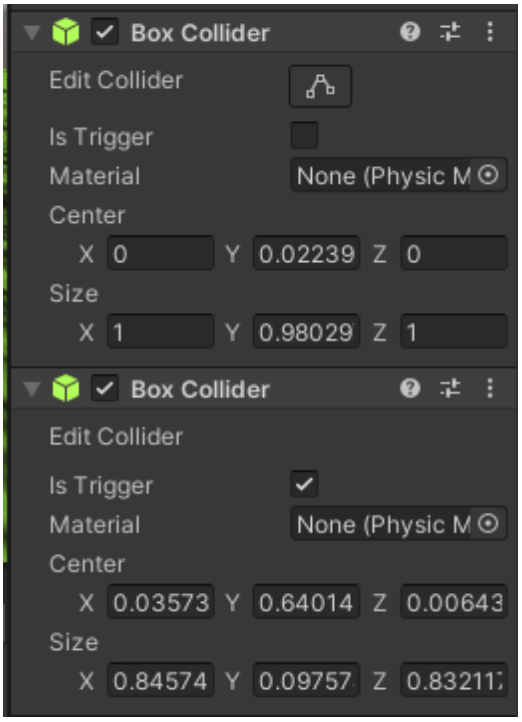
```
        [SerializeField] private float timeToDestroy = 2f;
    private Rigidbody rb;
    private void Start()
    {
        rb = GetComponent<Rigidbody>();
        rb.isKinematic = true;
        rb.constraints = RigidbodyConstraints.FreezePositionZ |
RigidbodyConstraints.FreezePositionX;
    }
 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

Lo que hacemos en este caso es jugar un poco con las propiedades del RigidBody. Convertimos al RigidBody en "**Kinematic**" lo que le sacara las físicas y **fijamos la posición en Z y X**. Entonces, para hacer que "caiga" o le afecte la gravedad, haremos que al tocar al player, la propiedad "isKinematic" pase a ser **false** nuevamente.

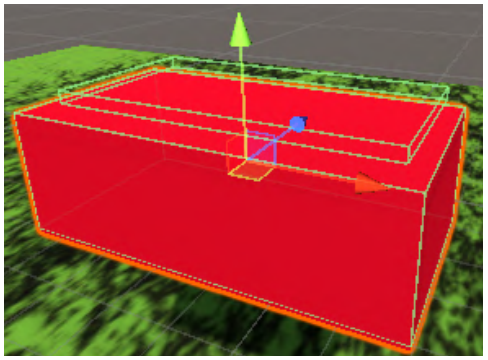
Ahora nace un nuevo problema o cuestión de "balance". Podrán notar que aunque toque a mi plataforma con la cabeza, esta se caerá. Por esto, crearemos un sistema sencillo para modificar esta situación. Lo que agregara más versatilidad a nuestro **diseño** del juego.

¿Está parado encima de mí?

Como dice el título, empezaremos por crear una forma de que detecte los pies o si mi personaje está parado encima.



Primero, añadiremos otro **Box Collider** a nuestra plataforma, lo editaremos para que quede encima de la caja, lo achicamos un poco y le marcaremos como que será "**IsTrigger**"



Y seguiremos cambiando el código reemplazando el "**OnCollisionEnter**" por un "**OnTriggerStay**", a la vez que pedimos los datos del objeto que chocamos para ver si su **Y** es mayor a la nuestra:

```
        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.");
            }
        }  }
```

Esto funcionará a fuerza de doble chequeo. **Si** el personaje está tocando mi collider superior (*el **IsTrigger***) y **Si** su **Y** es mayor a la mia (*si esta mas arriba que la plataforma*), entonces se cae y se destruye.

Si recordamos bien la principal diferencia entre el estado **Enter** y el **Stay** es que el Enter detecta el primer frame de contacto y el Stay detecta todo el tiempo dentro del Collider.

Movible.

Seguiremos con una plataforma movible. 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:

```
 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);
    }
```

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.

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

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.

Seguiremos con algo importante, pero optativo. Podemos ver que al subirnos a la plataforma, esta no nos lleva y tendremos que movernos manualmente. Esto se puede modificar de distintas maneras. Ahora usaremos una de ellas, que igualara la posición del Player a la de la plataforma. Dentro de nuestra plataforma “física”(El Child, no el parent Contenedor), colocaremos el siguiente código:

```
// 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;
        }
    }
```

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.

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;
        }
    }
```

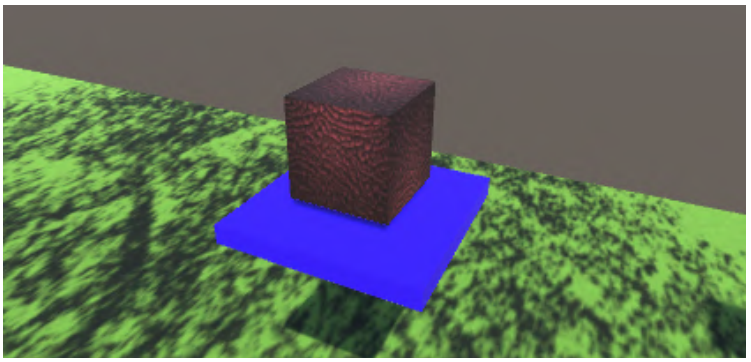
Este método se ejecuta cuando algo deja de colisionar con la plataforma.
Verifica si:

- **`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.

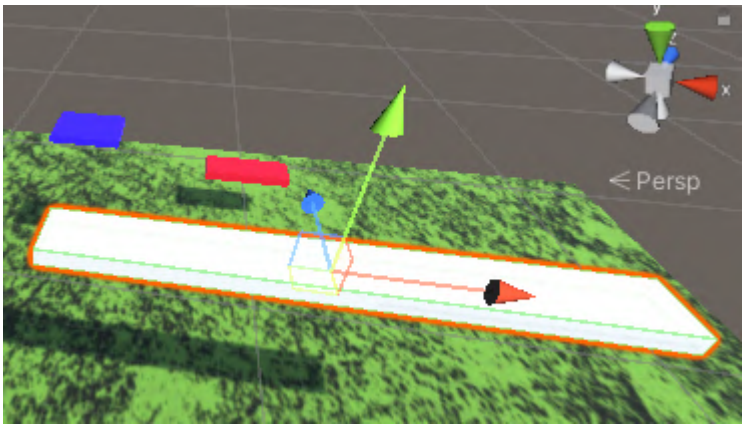
Al terminar esto nuestra plataforma no solo se moverá, sino que mantendrá a nuestro personaje moviéndose con ella
.



Rotatoria.

Por último, haremos una plataforma sencilla con con movimiento “rotativo”

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.

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.

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.


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