**Trabajo Práctico - Programación**

Leer la función ***PerformMovement*** del script /Assets/Scripts/Mechanics/**KinematicObject.cs** e identificar si se utilizan **colliders o raycasts** para detectar las superficies que el objeto tiene por delante.

Tener en cuenta la variable “***hitBuffer***” y la llamada a la función ***Cast*** en la línea **135.**

En el código proporcionado, se utilizan raycasts para detectar las superficies que el objeto tiene por delante. La función body.Cast(move, contactFilter, hitBuffer, distance + shellRadius) en la línea 135 es una llamada a un raycast 2D, que se utiliza para verificar si hay colisiones en la dirección de movimiento del objeto.

Aquí hay una breve explicación a entendimiento de cómo funciona esta detección de colisiones utilizando raycasts segun lo visto en clase e investigacion aparte:

**Body.Cast(move, contactFilter, hitBuffer, distance + shellRadius)**: Esta línea de código realiza un raycast desde la posición actual del objeto en la dirección **move** (que representa la dirección de movimiento del objeto) con una distancia de **distance + shellRadius.contactFilter** es un filtro que define qué capas de colisión se deben considerar durante el raycast. El resultado del raycast se almacena en el arreglo **hitBuffer**.

**var count = body.Cast(move, contactFilter, hitBuffer, distance + shellRadius)**: La función **Cast** devuelve el número de colisiones detectadas. El resultado se almacena en la variable count, que indica cuántas colisiones se encontraron en la dirección del movimiento del objeto.

**Recorriendo el hitBuffer:** Después de realizar el raycast, el código recorre el arreglo hitBuffer para verificar cada colisión detectada. Si encuentra una superficie que el objeto puede ocupar (es decir, una superficie cuya normal tenga un ángulo adecuado para que el objeto se siente en ella), actualiza la variable **IsGrounded** para indicar que el objeto está en el suelo. También ajusta la velocidad del objeto según la normal de la superficie para simular la física del salto y la caída adecuadamente.

En resumen, el código utiliza raycasts para detectar las superficies que el objeto tiene por delante y ajusta el comportamiento del objeto en consecuencia. El uso de raycasts junto con el arreglo **hitBuffer** permite detectar las colisiones de manera eficiente y realizar cálculos precisos para el movimiento del objeto.

La técnica utilizada aquí se llama "raycasting". Imagina que el objeto lanza líneas imaginarias, llamadas rayos, hacia adelante para ver si toca algo. En este caso, se están lanzando rayos desde la posición actual del objeto en la dirección en la que se está moviendo.

Cuando uno de estos rayos encuentra una superficie, el código registra esa información y ajusta el comportamiento del objeto en consecuencia. Por ejemplo, si el objeto está en el aire, se detiene para simular la gravedad que lo atrae hacia abajo. Si toca el suelo, se marca como "IsGrounded" para indicar que está en el suelo y puede saltar o caminar.

¿Qué tipo de estructura de datos se utiliza para almacenar los tokens (objetos coleccionables) para que puedan ser animados todos al mismo tiempo en el archivo /Assets/Scripts/Mechanics/**TokenController.cs**

Tener en cuenta la variable “***tokens***” y que se puede acceder a ***cualquiera de sus elementos*** desde la función ***Update***()

En el código proporcionado, los tokens (objetos coleccionables) se almacenan en un arreglo (array) de tipo **TokenInstance[]**. La variable **tokens** es un arreglo de objetos de la clase **TokenInstance**. Un arreglo es una estructura de datos que puede contener múltiples elementos del mismo tipo. En este caso, el arreglo **tokens** almacena todas las instancias de **TokenInstance** que se animarán.

Cada elemento en el arreglo **tokens** es una instancia de la clase **TokenInstance**. Los elementos del arreglo se pueden acceder individualmente usando índices. Por ejemplo, **tokens[0]** se refiere al primer elemento del arreglo, **tokens[1]** al segundo elemento y así sucesivamente.

En la función **Update()**, se recorre este arreglo usando un bucle for para animar todos los tokens al mismo tiempo. El bucle itera a través de cada elemento en el arreglo tokens y actualiza su animación en cada cuadro. Aquí está el fragmento relevante del código:

for (var i = 0; i < tokens.Length; i++)

{

var token = tokens[i];

// Resto del código para animar el token

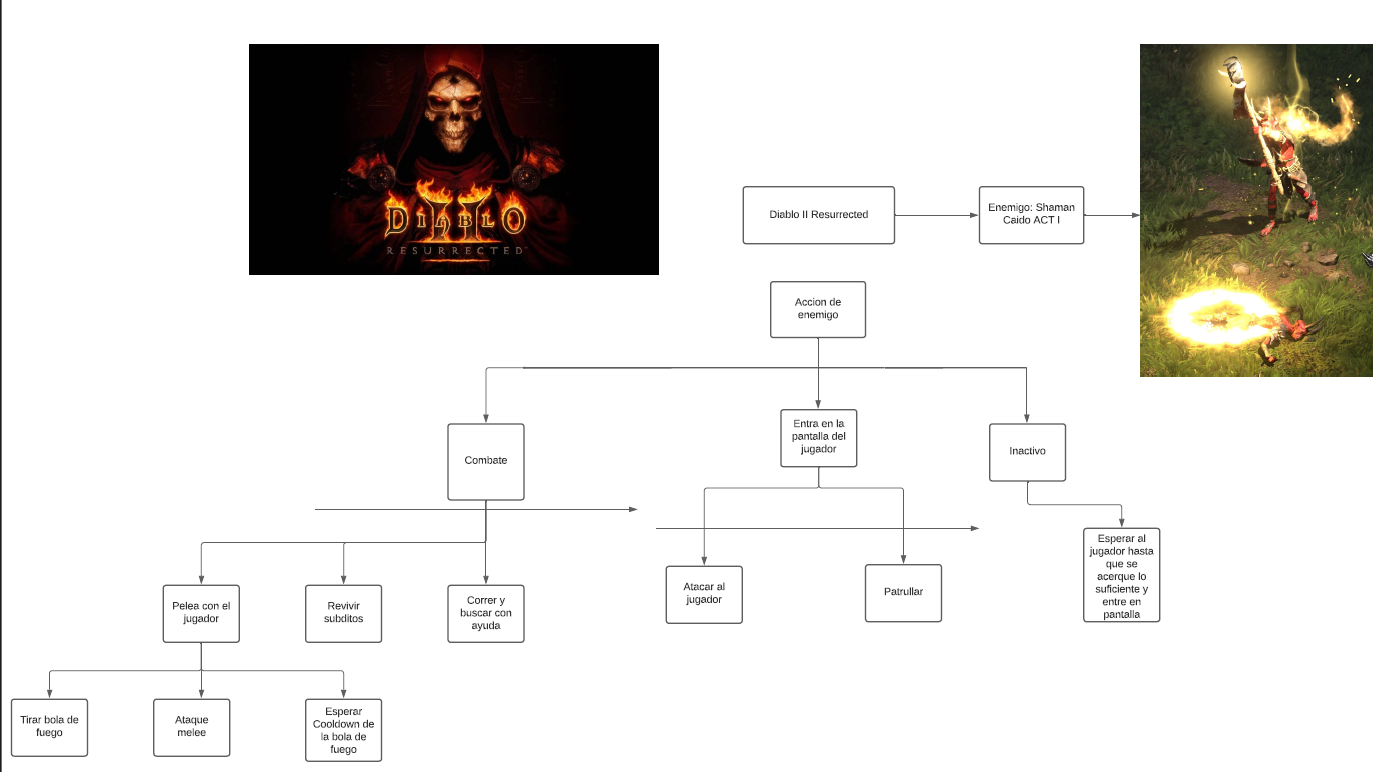
// ...

}

Este enfoque permite manejar y animar múltiples tokens al mismo tiempo utilizando un único script y una estructura de datos basada en un arreglo. Cada elemento del arreglo representa un token individual que se puede acceder y manipular desde la función **Update()**.

**Diagrama de comportamiento**

**Diablo II resurrected**



Jugar el juego corriendo el ejecutable /Build/**Platformer.exe** y redactar los bugs encontrados y sus posibles causas.

Los bugs encontrados fueron de diferentes fuentes, sin embargo sin mucho conocimiento dichos bugs pueden considerarse como “Features” sin mucho conocimiento referente al juego.

Al ser un juego de plataformas es muy comun observar valga la redundancia, plataformas que no posea una colision o no se puedan subir en ellas por parte del game desing, ya sea para molestar al jugador haciendolo creer que dicha plataforma sirve y es utilizable para el transcurso del juego o solo para una trampa y provocar ya sea una muerte intencional o un breve retraso en su juego.

Adicional se pudo apreciar como el jugador no era capaz de tomar una moneda, claramente ya eso si seria un problema siempre y cuando no esté con esa finalidad en el juego, esto puede deberse directamente a un problema en la parte de programacion a la interaccion con el player y este pick up item o hasta con la misma colision del item seria algo que podria verse en el codigo directamente con algo de experiencia o mencionar al equipo de desarollo para primero conocer si estas cosas son intencionales o un problema en si.