«Talento Tech»

Desarrollo de Videojuegos

Unity 3D

Clase 12

Buenos Aires Ciudad

Clase N° 12 | Estructuras de datos II

Temario:

- Stacks
- Queue

Objetivos de la clase

En esta clase, los estudiantes aprenderán el funcionamiento de las estructuras de datos **Stacks** y **Queues** en C#, comprendiendo sus principios de **LIFO (Last In, First Out)** y **FIFO (First In, First Out)**. Se explorarán sus aplicaciones en programación y desarrollo de videojuegos, implementando **Stack<T>** y **Queue<T>** en Unity a través de ejemplos prácticos. Además, se analizarán casos de uso como la gestión de historiales de posiciones, brindando herramientas clave para optimizar la lógica y el rendimiento en el desarrollo de software.

📌En el primer curso de Videojuegos en 2D trabajamos con algunas estructuras de datos. Hoy veremos algunas más para ampliar nuestro repertorio.

Stacks

Un **Stack** (pila) es una estructura de datos **LIFO** (*Last In, First Out*), lo que significa que el último elemento agregado es el primero en salir. Funciona como una pila de platos: solo se puede agregar o retirar elementos desde la parte superior.

Métodos principales de Stack<T>

- `Push(T item)`: Agrega un elemento a la cima del stack.
- `Pop()`: Remueve y devuelve el elemento superior del stack.
- `Peek()`: Devuelve el elemento superior sin removerlo.
- `Count`: Devuelve la cantidad de elementos en el stack.

Ejemplo 1:

Añadimos 3 objetos a nuestro Stack, borramos el último guardándolo en una variable y mostramos los valores sobrantes.

```
void Start()
{
  Stack<int> numeros = new Stack<int>();

  numeros.Push(10);
  numeros.Push(20);
  numeros.Push(30);

  int ultimo = numeros.Pop(); // 30 (se elimina)
  int tope = numeros.Peek(); // 20 (sigue en el stack)
  Debug.Log("cantidad:" + numeros.Count);//muestra cantidad de objetos
 }
```

Se crea una pila de enteros

```
Stack<int> numeros = new Stack<int>();
```

- `Stack<int>` representa una estructura de datos tipo pila (LIFO: Last In, First Out).
- Se declara una variable `numeros` que es una pila de enteros.

Se agregan elementos a la pila

```
numeros.Push(10);
numeros.Push(20);
numeros.Push(30);
```

- `Push(10)`: Apila el número `10`.
- `Push(20)`: Apila el número `20` encima de `10`.
- `Push(30)`: Apila el número `30` encima de `20`.

El stack quedaría de esta manera:

TOPE → 30
20
10 (fondo)

Se extrae el último elemento con `Pop()`

```
int ultimo = numeros.Pop(); // 30 (se elimina)
```

- Pop() devuelve el elemento superior (30) y lo elimina de la pila.
- ultimo ahora vale 30.

El stack quedaría de esta manera:

TOPE → 20
10 (fondo)

Se obtiene el elemento superior sin eliminarlo con `Peek()` y se muestra la cantidad de elementos sobrantes con `Count`

- `Peek()` obtiene el elemento superior (`20`) pero **sin eliminarlo**.
- `tope` ahora vale `20`.
- `numeros.Count` devuelve la cantidad de elementos en la pila (2 en este caso).
- Se imprime en la consola: “Cantidad: 2”

Ejemplo de Stack aplicado a juegos.

Usando Stacks crearemos una habilidad que guarde nuestra posición y nos permite volver atrás hasta 10 movimientos.

```
using System.Collections.Generic;
using UnityEngine;

public class BackInTime : MonoBehaviour
{
    Stack<Vector3> positionHistory = new Stack<Vector3>();
    Transform player;
    int maxPositions = 10;
    void Start()
    {
        player = GameObject.Find("Player").transform;
    }

    void Update()
    {
        SavePos();
        LoadPos();

    }
    private void SavePos()
    {
        if (Input.GetKeyDown(KeyCode.X))
        {
            // Si ya hay 10 posiciones almacenadas, eliminamos la más
antigua
            if (positionHistory.Count >= maxPositions)
            {
                // Para mantener solo las últimas 10 posiciones, usamos
una estructura temporal
                Stack<Vector3> tempStack = new Stack<Vector3>();

                // Transferimos los elementos a la estructura temporal,
eliminando el más antiguo
                while (positionHistory.Count > 1)
                {
                    tempStack.Push(positionHistory.Pop());
                }
                positionHistory.Pop(); // Elimina la más antigua
```

```
                while (tempStack.Count > 0)
                {
                    positionHistory.Push(tempStack.Pop());
                }
            }
            // Guarda la posición actual antes de mover al jugador
            positionHistory.Push(player.position);
        }
    }
    private void LoadPos()
    {
        if (Input.GetKeyDown(KeyCode.Z) && positionHistory.Count > 0)
        {
            player.position = positionHistory.Pop(); // Revierte a la
última posición guardada
        }
    }
}
```

Explicando el código:

Variables de la clase:

```
    Stack<Vector3> positionHistory = new Stack<Vector3>();
    Transform player;
    int maxPositions = 10;
```

- `positionHistory`: Una **pila (Stack)** que guarda posiciones del jugador (hasta 10).
- `player`: Referencia al `Transform` del GameObject **"Player"**.
- `maxPositions`: Límite de posiciones almacenadas en la pila.

Método Start():

```
    void Start()
    {
        player = GameObject.Find("Player").transform;
    }
```

- Encuentra el objeto con el nombre **"Player"** y guarda su `Transform` en la variable `player`.
- Esto permite acceder a su posición en `Update()`.

Método Update():

```
    void Update()
    {
        SavePos();
        LoadPos();
    }
```

Llamamos a los métodos para guardar la Posición y cargarla.

Método SavePos():

Verificar si la pila tiene demasiadas posiciones guardadas

```
    private void SavePos()
    {
       if (Input.GetKeyDown(KeyCode.X))
       {
    // Si ya hay 10 posiciones almacenadas, eliminamos la más
antigua
       if (positionHistory.Count >= maxPositions)
        {
```

- `positionHistory` es un `Stack<Vector3>`, que almacena posiciones en orden **LIFO (Last In, First Out)**.
- `maxPositions` es el límite de posiciones que queremos guardar (10 en este caso).
- Si `positionHistory` ya tiene **10 o más elementos**, entonces es necesario eliminar la más antigua para mantener el límite.

Crear una pila temporal:

```
  Stack<Vector3> tempStack = new Stack<Vector3>();
```

- Como `Stack<T>` no permite eliminar directamente el primer elemento (el más antiguo), se usa una **pila temporal (`tempStack`)** para reorganizar los datos.

Transferir los elementos a la pila temporal (excepto el más antiguo)

```
               while (positionHistory.Count > 1)
                {
                    tempStack.Push(positionHistory.Pop());
                }
```

- Se usa un `while` para **extraer** elementos de `positionHistory` **y moverlos a `tempStack`**, excepto el primero (el más antiguo).

- `positionHistory.Pop()` **saca** elementos de la pila y `tempStack.Push()` los **guarda** en el orden inverso.
- Se detiene cuando queda **1 solo elemento** en `positionHistory`, que será el más antiguo.

Eliminar la posición más antigua y Volver a transferir las posiciones a `positionHistory`

```
        positionHistory.Pop(); // Elimina la más antigua
        while (tempStack.Count > 0)
        {
                positionHistory.Push(tempStack.Pop());
        }
```

- Como ya trasladamos casi todas las posiciones a `tempStack`, ahora solo queda **el más antiguo** en `positionHistory`, y lo eliminamos con `Pop()`.

- Ahora tomamos los elementos desde `tempStack` y los devolvemos a `positionHistory`, restaurando el orden original (sin la posición más antigua).

Guardar la nueva posición del jugador

```
            // Guarda la posición actual antes de mover al
jugador
            positionHistory.Push(player.position);
```

- Una vez que hemos asegurado que solo hay **9 elementos** en `positionHistory`, agregamos la nueva posición del jugador.
- `player.position` representa la posición actual del personaje en el juego

Metodo LoadPos()

Cambio de posición

```
    private void LoadPos()
    {
        if (Input.GetKeyDown(KeyCode.Z) && positionHistory.Count
> 0)
        {
            player.position = positionHistory.Pop(); // Revierte
a la última posición guardada
        }
    }
```

- Si apretamos la **Z** y tenemos posiciones guardadas, cambiaremos la posición a la última posición guardada y a la vez, la elimina con **.Pop()**.

Con esto obtendremos una habilidad de guardado y carga de posiciones, algo así como “volver en el tiempo”.

Queues

Una **Queue** (cola) sigue el principio **FIFO** (*First In, First Out*), es decir, el primer elemento agregado es el primero en salir.

Métodos principales

- `Enqueue(T item)`: Agrega un elemento al final de la cola.
- `Dequeue()`: Remueve y devuelve el primer elemento.
- `Peek()`: Obtiene el primer elemento sin eliminarlo.
- `Count`: Retorna la cantidad de elementos en la cola.

Utilizaremos ejemplos muy similares para que puedan comprar las diferencias de uso y profundizar en la lógica.

Ejemplo Básico:

```
class EjemploQueue
{
    static void Main()
    {
        // Crear una cola de tipo string
        Queue<string> cola = new Queue<string>();

        // Enqueue: Agregamos elementos a la cola
        cola.Enqueue("Jugador1");
        cola.Enqueue("Jugador2");
        cola.Enqueue("Jugador3");

        Console.WriteLine($"Elementos en la cola: {cola.Count}");

        // Peek: Obtenemos el primer elemento sin eliminarlo
        Console.WriteLine($"Primer elemento en la cola (sin eliminar):
{cola.Peek()}");

        // Dequeue: Removemos y obtenemos el primer elemento
        string atendido = cola.Dequeue();
```

```
        Console.WriteLine($"Atendiendo a: {atendido}");

        // Verificar el nuevo primer elemento
        Console.WriteLine($"Nuevo primer elemento: {cola.Peek()}");

        // Verificar la cantidad de elementos después del Dequeue
        Console.WriteLine($"Elementos restantes en la cola:
{cola.Count}");
    }
}
```

El resultado en consola seria algo asi:

Elementos en la cola: 3
Primer elemento en la cola (sin eliminar): Jugador1
Atendiendo a: Jugador1
Nuevo primer elemento: Jugador2
Elementos restantes en la cola: 2

En este ejemplo, simulamos una cola de jugadores esperando turno en un juego. Primero agregamos tres jugadores con `Enqueue()`, luego usamos `Peek()` para ver quién es el primero sin sacarlo, después con `Dequeue()` lo eliminamos y lo atendemos, y finalmente mostramos cómo cambia la cola.

Ejemplo de Queues en juegos

Adaptamos la idea que hicimos con Stacks pero usando Queues

```
    Queue<Vector3> positionHistory = new Queue<Vector3>();
    Transform player;
    int maxPositions = 10;

    void Start()
    {
        player = GameObject.Find("Player").transform;
    }

    void Update()
    {
        SavePos();
        LoadPos();
    }
```

```
    private void SavePos()
    {
        if (Input.GetKeyDown(KeyCode.X))
        {
            // Guarda la posición actual del jugador
            positionHistory.Enqueue(player.position);

            // Si ya hay más de 10 posiciones almacenadas, eliminamos
la más antigua
            if (positionHistory.Count > maxPositions)
            {
                positionHistory.Dequeue(); // Elimina la más antigua
            }
        }
    }

    private void LoadPos()
    {
      if (Input.GetKeyDown(KeyCode.Z) && positionHistory.Count > 0)
      {
        List<Vector3> tempList = new List<Vector3>(positionHistory);
        player.position = tempList[tempList.Count - 1]; // Recupera la
última posición guardada

            // Elimina la última posición
         tempList.RemoveAt(tempList.Count - 1);
         positionHistory = new Queue<Vector3>(tempList);
       }
    }
```

Explicando el código:

SavePos()

```
private void SavePos()
    {
        if (Input.GetKeyDown(KeyCode.X))
        {
            // Guarda la posición actual del jugador
            positionHistory.Enqueue(player.position);

            // Si ya hay más de 10 posiciones almacenadas,
eliminamos la más antigua
            if (positionHistory.Count > maxPositions)
            {
                positionHistory.Dequeue(); // Elimina la más
antigua
            }
        }
    }
```

Guarda la posición actual

- Se usa `Enqueue(player.position)` para agregar la posición actual del jugador a la cola.

Mantiene un máximo de 10 posiciones

- Si la cola supera las 10 posiciones (`positionHistory.Count > maxPositions`), elimina la más antigua con `Dequeue()`, asegurando que solo se almacenen las últimas 10 posiciones.

LoadPos()

Verifica si se presiona la tecla "Z", si hay posiciones almacenadas y Convierte la cola en una lista para facilitar el acceso

```
private void LoadPos()
    {
        if (Input.GetKeyDown(KeyCode.Z) && positionHistory.Count
> 0)
        {
           List<Vector3> tempList = new
List<Vector3>(positionHistory);
```

- Solo ejecuta la carga de posiciones si el jugador presiona la tecla **Z** y hay posiciones guardadas en la cola.
- Se crea una nueva lista `tempList` a partir de `positionHistory`.
- La cola (`Queue<Vector3>`) no permite acceso directo a sus elementos por índice, por lo que se convierte en una `List<Vector3>`.
- Esto permite acceder fácilmente al último elemento.

Recupera la última posición guardada y mueve al jugador allí

```
player.position = tempList[tempList.Count - 1]; // Recupera la última posición guardada
```

- Se accede al **último elemento de la lista** con `tempList[tempList.Count - 1]`.
- Se asigna esta posición al `player.position`, moviendo al jugador al último punto guardado.

Elimina la última posición de la lista y Reconstruye la queue sin la última posición

```
// Elimina la última posición
tempList.RemoveAt(tempList.Count - 1);
positionHistory = new Queue<Vector3>(tempList);
```

- Se elimina el último elemento de `tempList` porque ya fue utilizado para mover al jugador.
- Esto evita que el jugador pueda volver a esa posición más de una vez.
- Se crea una nueva cola (`Queue<Vector3>`) con los elementos restantes de `tempList`.
- Se asigna esta nueva queue a `positionHistory`, reemplazando la original.
- Como resultado, la última posición se ha eliminado y la cola conserva el resto de posiciones en el mismo orden.

Con esto obtendremos un resultado similar a lo que hicimos en Stacks.

Como verán las partes más “elaboradas” de los códigos se invierten con cada herramienta por tener lógicas similares pero casi opuestas. Siempre recuerden que hay más formas de hacerlo, especialmente formas que darán mejor rendimiento por su optimización, pero en muchos casos elegimos otros caminos por su simpleza a la hora de explicar las herramientas y su fácil lectura.

Volver en el Tiempo:

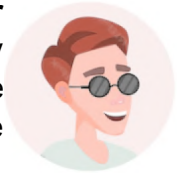
*Tras semanas de arduo trabajo, el equipo de **TalentoLab** ha logrado avances impresionantes en el desarrollo del **proyecto Nexus**. Los sistemas de movimiento, animaciones y mecánicas básicas ya están en funcionamiento, y la narrativa empieza a sentirse más integrada con la jugabilidad.*

*Sin embargo, un nuevo desafío ha llegado a la mesa. **Uno de los clientes de TalentoLab ha solicitado una mecánica especial para su juego:** un **poder temporal que permita al personaje volver a posiciones anteriores**, como si estuviera rebobinando el tiempo. Esta habilidad será clave para superar ciertos obstáculos y darle una capa extra de estrategia al gameplay.*

*El equipo de Desarrolladores ha analizado este pedido y ha decidido que la mejor manera de implementarlo es utilizando **estructuras de datos avanzadas: Stacks y Queues**.*

Ejercicios prácticos:

El equipo de **TalentoLab** ha logrado implementar con éxito la mecánica de **rebobinado**, permitiendo que el personaje vuelva a posiciones anteriores utilizando **Stacks** y **Queues**. Sin embargo, durante las pruebas, los diseñadores han notado un problema: **el jugador puede abusar de la habilidad**, regresando en el tiempo sin ninguna restricción y generando situaciones que rompen el balance del juego. Elizabeth y Giuseppe han pensado que serías el indicado para lograr un tiempo de espera entre guardado y guardado



Objetivo del ejercicio

*Para equilibrar la mecánica, **se deberá implementar un cooldown entre cada posición guardada en la Stack/Queue**. Esto significa que:*

- *Cada vez que el personaje guarde una posición en la memoria, deberá esperar un tiempo antes de poder almacenar la siguiente.*
- *El tiempo de espera puede ser ajustable para que la mecánica se sienta justa y estratégica.*
- *Esto evitará el abuso del sistema y fomentará que el jugador **planifique con cuidado cuándo usar la habilidad**.*

Materiales y recursos adicionales.

Stacks
https://learn.microsoft.com/es-es/dotnet/api/system.collections.generic.stack-1?view=net-8.0

Queue
https://learn.microsoft.com/es-es/dotnet/api/system.collections.generic.queue-1?view=net-8.0

Preguntas para reflexionar.

1. ¿Qué otras cosas podríamos armar con estas herramientas?
2. ¿Son 2 herramientas que actúan de la misma manera?

Próximos pasos.

En la próxima clase veremos una introducción a Pools de Objects que nos permitirá nuevas formas de manejar nuestros objetos en la escena.


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