

«Talento Tech»

Desarrollo de Videojuegos

# Unity 3D

Clase 12



# 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: Uso de una pila (Stack) en Unity

En este ejemplo, vamos a:

- Crear una pila de enteros
- Apilar 3 valores
- Quitar el último (con Pop())
- Ver el valor en la cima (con Peek())
- Mostrar cuántos elementos quedan

```
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 que almacena enteros.
- Las pilas siguen el comportamiento **LIFO** (*Last In, First Out*)

## 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() saca el **elemento que está en la cima** (en este caso, 30).
- Ese valor se guarda en la variable ultimo.

El stack quedaría de esta manera:

TOPE → 20
10 (fondo)

## Se mira el nuevo tope con **Peek()**

```
int tope = numeros.Peek();
```

- Peek() **devuelve el elemento de la cima, pero sin eliminarlo**.
- En este caso, tope vale **20**.

## Se consulta cuántos elementos quedan con Count

```
Debug.Log("cantidad:" + numeros.Count);
```

- Count devuelve cuántos elementos tiene la pila en ese momento.
- Luego del Pop(), quedan 2 elementos → se imprime.

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 (máximo 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();  
}
```

Llama constantemente a los métodos para **detectar las teclas** y ejecutar guardar/cargar posición.

### Método SavePos():

#### Detectar entrada del usuario y verificar límite

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

- Solo se ejecuta cuando el jugador presiona la tecla X.
- **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**, dejando solo el más antiguo.
- positionHistory.Pop() **saca** elementos desde el tope de la pila y tempStack.Push() los **guarda** en orden inverso.
- Se detiene cuando queda **1 solo elemento** en positionHistory, que será el **más antiguo** (el del fondo).

### Eliminar la posición más antigua y restaurar las demás

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

- Como ya trasladamos todas las posiciones **excepto la más antigua** a tempStack, ahora solo queda **el elemento más antiguo** en positionHistory, y lo eliminamos con Pop().
- Después 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



## Método LoadPos():

### Restaurar posición anterior

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

- **Solo se ejecuta** cuando el jugador presiona la tecla **Z** y hay posiciones guardadas en la pila.
- Cambia la posición del jugador a la **última posición guardada** (la del tope de la pila).
- Al usar **.Pop()**, no solo obtiene la posición sino que también **la elimina** de la pila.
- Esto permite "retroceder" paso a paso a través del historial de posiciones.

Con esto obtenemos una habilidad de guardado y carga de posiciones que simula "volver en el tiempo", donde:

- **Tecla X:** Guarda la posición actual (máximo 10 posiciones)
- **Tecla Z:** Retrocede a la última posición guardada

## Queues

Una **Queue (cola)** sigue el principio **FIFO (First In, First Out)**, es decir, el primer elemento agregado es el primero en salir. Es como una fila de personas: quien llega primero, es atendido primero.

### 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.

### 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**. Como es FIFO, **Jugador1** (el primero en llegar) es el primero en ser atendido.

## Ejemplo de Queues en juegos

Adaptamos la idea que hicimos con Stacks pero usando Queues

```
public class PositionHistory : MonoBehaviour{
    Queue positionHistory = new Queue();
    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)
        {
            LoadOldestPosition();
        }
    }
}
```

```

    }
}

private void LoadOldestPosition()
{
    if (positionHistory.Count > 0)
    {
        oldestPosition = positionHistory.Dequeue(); // Saca la MÁS
ANTIGUA
        player.position = oldestPosition;
        Debug.Log($"Restaurada posición más antigua. Restantes:
{positionHistory.Count}");
    }
}

```

### Explicando el código:

```

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

```

- **positionHistory:** Una cola (Queue) que guarda posiciones del jugador (máximo 10).
- **player:** Referencia al Transform del GameObject "Player".
- **maxPositions:** Límite de posiciones almacenadas en la cola.

### 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();
}
```

Llama constantemente a los métodos para detectar las teclas y ejecutar guardar/cargar posición.

## Método 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
        }
    }
}
```

- Se usa **Enqueue(player.position)** para agregar la posición actual del jugador **al final** de la cola.
- Como es FIFO, las posiciones se almacenan en **orden cronológico**: la primera guardada queda al frente.
- Si la cola supera las **10 posiciones** (`positionHistory.Count > maxPositions`), se elimina automáticamente **la más antigua**.
- **Dequeue()** siempre elimina el **primer elemento** de la cola (el más antiguo), siguiendo el principio FIFO.
- Esto mantiene automáticamente el límite sin necesidad de reorganizar datos.

## LoadPos()

Detectar entrada y llamar a carga

```
private void LoadPos()
{
    if (Input.GetKeyDown(KeyCode.Z) && positionHistory.Count > 0)
    {
        LoadOldestPosition();
    }
}
```

- Solo se ejecuta cuando el jugador presiona la tecla Z Y hay posiciones almacenadas en la cola.
- Llama al método LoadOldestPosition() para restaurar la posición más antigua.

## Método LoadOldestPosition():

Recuperar y usar la posición más antigua

```
private void LoadOldestPosition()
{
    if (positionHistory.Count > 0)
    {
        Vector3 oldestPosition = positionHistory.Dequeue(); // Saca
la MÁS ANTIGUA
        player.position = oldestPosition;
        Debug.Log($"Restaurada posición más antigua. Restantes:
{positionHistory.Count}");
    }
}
```

- **Verifica** que hay posiciones disponibles en la cola antes de proceder.
- **Dequeue()** extrae **la posición más antigua** (la primera que se guardó) de la cola.
- **Asigna** esa posición a **player.position**, **moviendo al jugador** a esa ubicación.
- **Elimina** automáticamente la posición de la cola, evitando que se pueda usar más de una vez.
- Como es **FIFO**, el jugador retrocede en el **mismo orden** que guardó las posiciones.

# Comportamiento del sistema:

## Al guardar posiciones (Tecla X):

- Las posiciones se almacenan en **orden cronológico**
- La cola mantiene automáticamente un máximo de 10 posiciones
- Ejemplo: Guardas  $A \rightarrow B \rightarrow C \rightarrow D$  (A queda al frente, D al final)

## Al cargar posiciones (Tecla Z):

- Recuperas las posiciones en el **mismo orden** que las guardaste
- Ejemplo: Cargas  $A \rightarrow B \rightarrow C \rightarrow D$  (en ese orden secuencial)

## Diferencia clave con Stack:

- **Stack (LIFO)**: Guardas  $A \rightarrow B \rightarrow C \rightarrow D$ , cargas  $D \rightarrow C \rightarrow B \rightarrow A$  (orden inverso)
- **Queue (FIFO)**: Guardas  $A \rightarrow B \rightarrow C \rightarrow D$ , cargas  $A \rightarrow B \rightarrow C \rightarrow D$  (mismo orden)

Con esto obtenemos una mecánica de "retroceso cronológico" donde el jugador puede **revisar su camino** en el mismo orden que lo recorrió, como si fuera **reproduciéndolo hacia atrás** paso a paso. Esto ofrece una experiencia de juego completamente diferente al Stack y demuestra perfectamente cómo la elección de la estructura de datos afecta el comportamiento del sistema.



## 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

**BA** Buenos  
Aires  
Ciudad