DOCUMENTACIÓN

"Los sueños no tienen fecha de caducidad,

rendirse ni siquiera es la última opción"

PRELIMINARES

Este documento sirve como una guía exhaustiva y un registro detallado del desarrollo del juego de cartas, inspirado en el rico y complejo universo de **"Game Of Throne"**. A lo largo de estas páginas, se desglosará el proceso de creación del código (noches de desvelos), las decisiones de diseño y las principales implementaciones de mecánicas que capturan la esencia de intriga, estrategia y poder que caracteriza a la famosa serie. Con el objetivo de trasladar esa misma intensidad (P/V) y profundidad que caracteriza la obra de George R.R Martin al formato interactivo de un juego de cartas, se ha desarrollado un sistema de juego que permite a los jugadores sumergirse en los conflictos dinásticos, las alianzas políticas y las batallas épicas que son sinónimos de **"Game Of Throne"**.

La documentación está estructurada para facilitar la comprensión de cada etapa del desarrollo, desde la concepción inicial hasta la implementación final. Se exhorta al lector a explorar los entresijos del código, las estructuras de datos, las interfaces medievales y los algoritmos, algunos de ellos inefables, que hacen de este juego sea una experiencia única y envolvente… Sin más que hablar, abróchese los cinturones y agárrese de su asiento porque en las siguientes páginas estaremos hablando de la razón fundamental de las noches de desvelos del autor.

DESARROLLO

class Card

Me parece primordial comenzar por traducir los cimientos en donde nace todo, la clase base, diría yo, la clase madre. Sin más preámbulos les presento a continuación la clase Card:

```
public class Card : ScriptableObject
{
    // Propiedades (Campo)
    public new string name;                                                    // Nombre de la carta
    public string faction;                                                     // Faccion de la carta
    public int power;                                                          // Poder(unidad), daño(clima) o incremento(aumento)
    public string description;
    public Sprite artWork;                                                     // Imagen principal
    public Sprite portrait;                                                    // Imagen del marco
    81 referencias
    public enum card_position { M, R, S, MR, MS, RS, MRS, I, C, L};            // Posiciones en que se puede ubicar
    98 referencias
    public enum kind_card { golden, silver, climate, clear, bait, increase, leader };  // Tipos de carta
    public kind_card typeCard;                                                 // Tipo de carta
    public card_position cardPosition;                                         // Tipo de posición
    public bool isUnity;                                                       // Es carta unidad?
    public bool isHeroe;                                                       // Es carta héroe?
    public delegate void EffectDelegate(params object[] item);
    public EffectDelegate effect;                                              // Delegado que almacena el efecto(Método)
    private AudioClip clip;                                                    // Audio de las cartas al colocarse
    public int affectedRow;                                                    // Fila que afectan las cartas climas
```

La clase Card es una representación de una carta, definida como un ScriptableObject en Unity. Un ScriptableObject es un tipo especial de objeto en Unity que permite almacenar datos y comportamientos independientes de la escena o de los objetos de juego. La clase Card contiene varias propiedades que describen características y comportamientos específicos de una carta, como su nombre(name), facción(faction), poder(power), descripción(description), imagen(artWork), si es unidad(isUnity) o un héroe(isHeroe), un delegado para almacenar el effecto en forma de método (effect), y un audio clip(clip) para cuando que se activa cuando la carta es colocada. Además, define dos enumeraciones, card_position(cardPosition) y kind_card(typeCard), para especificar las posiciones en las que se puede ubicar la carta y los tipos de carta que puede ser, respectivamente. Estas propiedades permiten una gran flexibilidad en la definición de las cartas, permitiendo que cada una tenga características y comportamientos únicos según el diseño del juego.

Luego están los constructores que le dan vida a estas cosas llamadas “cartas”, de una forma u otra materializan en un objeto estás propiedades:

Estos constructores permiten crear instancias con diferentes configuraciones. Todos acepta parámetros básicos como nombre, facción, poder, si es una unidad

```
// Constructores (Sobrecargado)
14 referencias
public Card(string name, string faction , int power, bool IsUnity, bool IsHeroe, Sprite artWork, Sprite portrait, kind_card typeCa
57 referencias
public Card(string name, string faction, int power, bool IsUnity, bool IsHeroe, Sprite artWork, Sprite portrait, kind_card typeCar
8 referencias
public Card(string name, string faction, int power, int affectedRow, bool IsUnity, bool IsHeroe, Sprite artWork, Sprite portrait,
```

```
rd, string description, EffectDelegate effect, AudioClip clip = null)...
d, card_position cardPosition, string description, EffectDelegate effect, AudioClip clip = null)...
kind_card typeCard, card_position cardPosition, string description, EffectDelegate effect, AudioClip clip = null)...
```

o héroe, imagen, tipo de carta, descripción, efecto y opcionalmente un clip de audio. El segundo constructor añade una posición específica para la carta(cardPosition). El tercer constructor incluye un parámetro adicional affectedRow, el cual indica la fila afectada por las cartas climas.

… y un método que es el encargado de reproducir el audio:

```
public void ActiveClip();
```

class DataBase

Las 166 líneas de códigos perteneciente a esta destacable clase se caracterizan por contener en sí tres listas de objetos Card que representan mazos los tres mazos de cartas Stark, Targaryen y Dead (Caminantes Blancos). Al instanciar la clase DataBase, se llama al método `public void CreateCard()`, que inicializa las cartas con sus determinadas propiedades.

…y pues, en estas listas se almacenan los caminantes blancos, lo dragones de Daenerys y demás:

```
public List<Card> deckStark = new List<Card>();
public List<Card> deckTargaryen = new List<Card>();
public List<Card> deckDead = new List<Card>();
```

class Panels

A continuación… el Script encargado de acoger las cartas un vez ubicadas en el campo ya sea en un panel de cartas climas, líder, cuerpo a cuerpo asedio, distancia o de aumento:

```
public class Panels : MonoBehaviour
{
    public List<GameObject> cards = new List<GameObject>();                    // Cartas en un panel
    public List<Card.card_position> position = new List<Card.card_position>(); // Posiciones que acepta el panel
    public int maxItems;                                                       // Máxima cantidad de cartas
    public int itemsCounter;                                                   // (0) Cantidad de cartas actualmente

    4 referencias
    public int CounterUnity()                                                  // Cantidad de cartas de tipo Unidad...
    4 referencias
    public void RemoveAll(List<Card> cementery)                                // Remueve todas las cartas...
    1 referencia
    public void Remove()                                                       // (1) Remueve cartas innecesarias del panel...
    2 referencias
    public int PowerRow()                                                      // (2) Poder acumulado del panel (fila)...
    1 referencia
    private void UnDragging()                                                  // (3) Toda carta que se coloque en el panel se
    Mensaje de Unity | 0 referencias
    private void Start()...
    Mensaje de Unity | 0 referencias
    private void Update()                                                      // Actualización de propiedades...
}
```

La clase Panels en Unity se encarga de gestionar un conjunto de objetos GameObject (cartas ya instanciadas) que representan cartas dentro de un panel. Esta clase mantiene una lista de cartas (cards) y una lista de posiciones (position) que acepta el panel, así como contadores para el máximo número de cartas (maxItems) y la cantidad actual de cartas (itemsCounter). La clase proporciona métodos para contar el número de cartas de tipo Unidad (`CounterUnity()`), eliminar todas las cartas del panel y añadirlas a una lista de cementerio (`RemoveAll()`), eliminar cartas innecesarias del panel (`Remove()`), calcular el poder acumulado de las cartas en el panel (`PowerRow()`), y desactivar el script (Drag) el cual desactiva el arrastre de las cartas una vez que se colocan en el panel (`UnDragging()`). Estas operaciones son fundamentales para la lógica de juego, permitiendo la interacción dinámica entre las cartas y el panel, y asegurando que las reglas del juego se apliquen correctamente. Y bueno, no estaría demás agradecer a este Scrpit, pues gracias a él las cartas no se salen de control ni comienzan a flotar ni dar vueltas locas en el campo ni semejantes atrocidades del multiverso.

class Player

Próximamente estaremos paseándonos por el laberinto de esta emotiva clase la cual abarca toda aquella característica que posee un jugador destacando algunas propiedades como son: el nombre del jugador, su facción, su mazo de cartas, y el poder acumulado por rondas.

```
// Propiedades
#region Property
public string playerName;                          // Nombre del jugador
public string faction;                             // Facción
public List<Card> deck = new List<Card>();         // Mazo del jugador
public int[] powerRound;                           // Puntos acumulados por rondas
public int takeCardStartGame = 0;                  // Cantidad de cartas cambiadas antes de la batalla

public bool myTurn;                                // Dicta el turno del jugador
public bool skipRound;                             // Dicta si el jugador pasa la ronda
public bool oneMove;                               // Dicta si el jugador ya ha jugado una carta
public Text counterDeck;                           // Cantidad de cartas en el mazo
public Text counterCementery;                      // Cantidad de cartas en el cementerio

// Paneles
public List<Card> cementeryCards;                  // Cartas enviadas al cementerio
public GameObject leader;                          // Carta líder
public GameObject hand;                            // Cartas de la mano
public GameObject[] field;                         // Cartas del campo(Melee-Range-Siege)
public GameObject[] increase;                      // Cartas de aumento
public GameObject climate;                         // Cartas clima
public GameObject panelTakeCard;                   // Panel para robar carta antes de la batalla
public GameObject infoTakeCard;                    // Boton-Info que indica poder robar cartas
#endregion
```

Aparte de las propiedades básicas como lo son: playerName, faction, deck. Existen otras primordiales como el conjunto de GameObject el cual hace referencia a los distintos paneles del campo:

```
GameObject leader;        // Carta líder
GameObject hand;          // Cartas de la mano
GameObject[] field;       // Array de los paneles del campo(Mel-Sig-Dst)
GameObject[] increase;    // Array de los paneles del aumento
GameObject climate;       // Panel de las cartas climas
```

Véase los métodos más importantes de esta clase:

```
public void Cementery()
private void GeneralPower(int round)
public void TakeCard(int num = 0)
private void BackImageAndDrag()
```

public void Cementery()

Envía todas las cartas, del campo al cementerio

private void GeneralPower(int round)

Retorna el poder acumulado por cada fila y lo almacena en el array powerRound en la posición equivalente al número de ronda

public void TakeCard(int num = 0)

Instancia las cartas en la mano (sean las 10 iniciales, 2 o 1)

private void BackImageAndDrag()

Actualiza el estado de las cartas. Si el jugador no está en su turno se voltean las cartas y se desactiva el Script Drag.

class GameManager

Bueno, ha llegado el momento tan esperado, el famoso lugar donde ocurre la magia, la clase GameManager, la encargada de gestionar toda la lógica, interacción y demás reglas del juego. Ánimo que sin ella los jugadores ganarían con puntuación 0.

```
// Propiedades (Campo)
public static int round;                                    // (1) Número de Ronda
public bool skipRound;                                      // (2) True(si algún jugador pasa la ronda)
public GameObject[] numberRound;                            // (3) Panel-Display hace referencia al número de ronda
public GameObject panelGameOver;                            // (4) Panel que muestra los resultados a final del juego
public GameObject panelRound;                               // (5) Panel que muestra los resultados a final de ronda
public  Player player1;                                     // (7) Jugador 1
public  Player player2;                                     // (8) Jugador 2
private Player start;                                       // (9) Jugador que comienza cada ronda
private Player playerEnd;                                   // (10) Jugador que termina cada ronda
static public Player currentPlayer;                         // (11) Jugador actual
```

En este fragmento de código, se definen varias variables y objetos que son esenciales para la gestión del flujo de un juego de turnos o rondas. La variable round es un entero estático que almacena el número de ronda actual, lo que indica en qué etapa del juego se encuentra. El booleano skipRound indica si algún jugador ha optado por pasar la ronda, lo que podría modificar la lógica del juego. Tenemos numberRound, un arreglo de objetos GameObject que se utiliza para mostrar el número de ronda actual en la interfaz de usuario, panelGameOver y panelRound son objetos GameObject que representan paneles en la interfaz de usuario, uno para mostrar los resultados finales del juego y otro para mostrar los resultados al final de cada ronda. Luego tenemos player1 y player2 son instancias de la clase Player que representan a los dos jugadores. start y playerEnd son variables privadas de tipo Player que almacenan al jugador que comienza y al jugador que termina cada ronda, respectivamente. Finalmente, currentPlayer es una variable estática pública de tipo Player también que indica al jugador actualmente en turno, lo que es crucial para determinar cuándo es el turno de cada jugador y aplicar las reglas del juego de manera adecuada.

Si bien algunos de los métodos más importantes de la clase:

```
private void ButtonSkipTurn()

private void ButtonSkipRound()

private Player Winner()

private void PanelGameOver()
```

ButtonSkipTurn()

En estás pocas líneas ocurre algo muy lindo pues, se gestiona la lógica del juego cuando un jugador decide saltar su turno. Primero, establece la propiedad oneMove del jugador actual (currentPlayer) en false, indicando que el jugador no ha realizado ninguna acción en su turno. Luego, verifica si la opción de saltar la ronda (skipRound) no está activada. Si no está activada, el método cambia el turno entre los jugadores: si es el turno del player1, se cambia a player2, y viceversa. Esto se logra invirtiendo el valor de myTurn de cada jugador y actualizando currentPlayer. Si la opción de saltar la ronda está activada, en lugar de cambiar el turno, se llama al método `ButtonSkipRound()`. No te desesperes, ahora viene la explicación de ese método:)

ButtonSkipRound()

Alto responsable de manejar la lógica cuando un jugador decide saltar toda la ronda. Vamos paso a paso, primero, verifica si el jugador que terminó la ronda (playerEnd) es el jugador actual (currentPlayer). Si es así, el método procede a verificar si hay un ganador del juego mediante la llamada al método `Winner()`, calma que te lo explico en el próximo párrafo. Si hay un ganador, se muestra el panel de fin de juego (PanelGameOver) para indicar el resultado final del juego. Si no hay un ganador, el método prepara el juego para la siguiente ronda. Esto incluye restablecer la variable skipRound a false, y asegurarse de que el jugador actual no haya realizado ninguna acción en su turno (oneMove a false). Luego, el método determina quién comienza la siguiente ronda basándose en el poder acumulado de cada jugador en la ronda actual (powerRound[round]). Se actualizan las variables start, playerEnd, currentPlayer, y se incrementa el contador de rondas (round). Finalmente, se llama a los métodos `Cementery()`, y `TakeCard()`, para ambos jugadores. Si el jugador que terminó la ronda no es el jugador actual, el método simplemente llama a `ButtonSkipTurn()` y establece skipRound a true, indicando que el jugador actual ha decidido saltar su turno.

Winner()

Lo prometido es dedua, acá te explico. Determina el ganador del juego hasta la ronda actual, comparando el poder acumulado de cada jugador en cada ronda. Inicialmente, se establecen dos contadores (winner1 y winner2) en cero para llevar un registro del número de rondas ganadas por cada jugador. Luego, se itera a través de todas las rondas hasta la actual (round + 1), comparando el poder acumulado de los jugadores en cada ronda. Si hay un empate, ambos contadores

se incrementan. Después de comparar todas las rondas, si uno de los contadores alcanza el valor de 2, lo que indica que un jugador ha ganado dos rondas consecutivas, se devuelve ese jugador como el ganador. Si no hay un ganador claro después de comparar todas las rondas, el método devuelve null, de lo contrario retorna el jugador.

`PanelGameOver()`

Lamento informarte que aquí no hay mucho de que hablar, esto no significa que no le demos importancia, pues este método permite mostrar gráficamente los resultados alcanzados por cada jugador a lo largo del juego.

Nota: Véase que la inicialización de las propiedades de esta clase ocurre en el método `void Start()`.

Conclusiones

Felicidades si has llagado hasta acá, ha recorrido mucho. Cierto es que el proceso de creación del juego ha sido meticuloso y desafiante, involucrando largas noches de trabajo para desarrollar un código que refleje la complejidad del universo de "Game Of Thrones". Se anima a los lectores a profundizar en el código y las estructuras subyacentes. Venga, ¿A qué esperas?, *¡A jugar!*

```
// Gracias a todos los que me apoyaron a lograr algo que parecía imposible
// Continuará…
```

Jery Rodríguez Fernández