**Nombre:** Alejandro García Ruiz

**Grupo:** C-112

Facultad de Matemática y Computación

Universidad de La Habana

|  |  |
| --- | --- |
| Informe Proyecto de programación  Gwent-Serk |  |

El proyecto es un juego inspirado en Gwent de CD Projekt. Usa el motor de Unity y tiene temática de Berserk.

Cuenta con 10 escenas en total, siendo la primera el menú principal, una escena para crédito, 5 escenas para ofrecer un tutorial básico, una para cada posible resultado de la partida (los dos ganadores y un empate), y la última con el juego en sí.

Cuando clickeamos el botón de *Iniciar Partida,* el juego corre la escena del juego y comienza la Primera Ronda. Al principio, el primer jugador robará 10 cartas, de estas puede elegir hasta 2 cartas para regresar al mazo y se le repartirá la misma cantidad automáticamente, una vez conforme se debe oprimir el botón correspondiente en la zona derecha de la mano. Luego de esto, el segundo jugador debe hacer lo mismo. Una vez que ambos estén conformes con sus cartas, comienza el juego.

Las **Cartas** se dividen en distintos tipos: Clima, Aumento, Despeje, Carta de unidad y Señuelo.

**Clima:** Modifica la franja a la que está asignada y la contraria, reduce a 1 de poder todas las cartas.

**Aumento:** Modifica la franja a la que está asignada, suma 1 de poder a todas las cartas.

**Despeje:** Elimina los efectos de todas las cartas de clima jugadas en el campo

**Señuelo:** Regresa a la mano la carta con más poder de la franja en la que se juega y, si surge efecto, te permite jugar de nuevo.

**Cartas de Unidad:** Estas son las cartas principales y pueden o no contar con un efecto. Se dividen en cartas de Plata, que son la mayoría, y cartas de Oro, a las cuales no les afecta ni las cartas de Clima ni las de Aumento. Ambas cuentan con distintas propiedades: Nombre, Poder, Descripción de la Carta, Franja en la que juegan, Facción y Tipo.

**Cartas de Líder**: Son cartas únicas y específicas de cada bando, que siempre están en el campo y tienen una habilidad más fuerte que el resto, la cual se usa una sola vez en cada partida. Solo cuentan con: Nombre, Descripción y Facción.

**Mazos:** Cada facción cuenta con el suyo propio, en el que están solo las cartas que comparten facción con su respectivo líder, además de cartas de Clima, Aumento, Despeje y Señuelo.

El **Tablero** está dividido en 6 franjas (3 para cada jugador), estas se dividen en: Cuerpo a Cuerpo, Distancia y Asedio. En estas se juegan las cartas de unidad y cada carta se juega en una zona específica. En la parte izquierda de cada franja se puede observar un espacio reservado para jugar las cartas de Clima (Eclipse) o de Aumento (Furia). Cada jugador cuenta un espacio para tener su mazo, su cementerio y la carta de su líder. Además, en la zona izquierda del tablero se encuentra reservado un espacio para mostrar la descripción completa de cada carta a tiempo real, este se activa simplemente pasando el *mouse* por encima de la carta deseada (a la izquierda de la descripción se le hace zoom a la carta para poder apreciarla mejor). Cada jugador cuenta con distintos contadores, uno encima de su líder que llevará la cantidad de victorias, uno para cada franja que contará la cantidad de poder en ellas y el último que llevará la suma de cada franja, o sea, el total.

**La Partida** puede llegar hasta 3 Rondas (De 3 a Ganar 2). Una vez que los jugadores tienen sus cartas, es cuestión de jugarlas. El objetivo para ganar una ronda es tener mayor cantidad de puntos que el oponente y esta termina una vez que ambos jugadores se quedan sin cartas o se rinden. Es un juego por turnos, estos pasan solo si se utiliza la habilidad del Líder de la Facción o si juega una carta (a excepción de la carta de señuelo que, si te devuelve una, te permite jugar de nuevo). Una vez que el jugador decide que no va jugar más, se puede rendir usando el botón que se encuentra en la parte izquierda de la pantalla (La Foto de Griffith). El ganador de la Ronda se decide pulsando el botón de la derecha (La Foto de Gutts).

**Scripts:**

**Cartas de Líder**: Script asignado a las cartas de Líder para identificarlas; contiene todas sus propiedades.

**Clase Carta**: Script asignado a todas las cartas excepto las de líder y es utilizado para identificarlas; contiene todas sus propiedades.

**Clase Clima**: Script asignado a las franjas donde se juegan los climas y aumentos; utilizada para guardar la lista de las cartas en la franja.

**Clase Franja**: Script asignado a todas las franjas, en el que se guardan la lista de las cartas en ella. Calcula la cantidad de puntos en la franja y en donde se encuentran la mayoría de los métodos que controlan los efectos de las cartas. Guarda los booleanos que se encargan de activar el efecto de clima y de aumento.

**Clase Mano**: Script asignado a las manos de los jugadores y guarda la lista de las cartas que tienen en la mano.

**ClaseCementerio**: Asignado a ambos cementerios, controla la lista de las cartas eliminadas

**GameManager**: Este script es el que lleva el conteo de las rondas y el que calcula el ganador (está asignado al botón de la derecha). Cambia a la escena de quien gane, guarda el ganador de la primera y segunda ronda y activa la animación de quien comienza.

**Iniciar Partida**: Este script es el encargado de cambiar entre las escenas del menú, créditos, tutoriales y la partida.

**Puntos**: Script asignado al contador de puntos de Gutts y suma los contadores de cada franja de Gutts.

**PuntosGriffith**: Script asignado al contador de puntos de Griffith y suma los contadores de cada franja de Griffith.

**SoundController**: Script asignado a un objeto del menú y se encarga de que la música se reproduzca de forma continua sin interrumpirse entre escenas.

**EndSong**: Para la música.

**StartSong**: Da *play* a la música.

**CardInfo**: Script encargado de mostrar el cuadro de texto con todas las propiedades de la carta en tiempo real. Está asignado a todas las cartas excepto el Líder.

**CardInfoL**: Script encargado de mostrar el cuadro de texto con todas las propiedades de la carta en tiempo real. Está asignado solo a las cartas de Líder.

**DrawCards**: Script asignado al mazo de Gutts y que guarda la lista de las cartas del mazo. Se encarga de repartir las cartas en las 3 rondas.

**eDrawCards**: Script asignado al mazo de Griffith y que guarda la lista de las cartas del mazo. Se encarga de repartir las cartas en las 3 rondas.

**Rendirse**: Script asignado al botón de rendirse y es el que controla esta función, cuando se activa cambia solo al turno del oponente.

**GriffithCards**: Asignado al botón para mantener las cartas de Griffith una vez que el jugador está conforme.

**GuttsCards**: Asignado al botón para mantener las cartas de Gutts una vez que el jugador está conforme.

**JDevolver**: Asignado a todas las cartas y es el que controla que, cuando se hace click derecho sobre las cartas, se devuelvan al mazo, pero solo hasta 2 veces.

**JugaCarta**: Asignado a todas las cartas del mazo. Controla que cuando se haga click sobre la carta, esta se juegue en la zona que le corresponde.

**Amanecer**: Asignado a todas las cartas de despeje y controla su efecto, cuando se juega esta carta, desactiva la función de todas las franjas para que realice el efecto de clima. Además, mueve todas las cartas de clima al cementerio al igual que a ella misma.

**Eclipse**: Asignado a todas las cartas de eclipse y es quien se encarga de elegir a qué franja se le vuelve *verdadero* el booleano que controla el efecto de clima.

**Furia**: Asignado a todas las cartas de furia y es quien se encarga de elegir a qué franja se le vuelve *verdadero* el booleano que controla el efecto de aumento.

**Puck**: Asignado a las cartas de señuelo y se encarga de correr el método *Puck()* en la **ClaseFranja**. Este calcula el mayor de los puntos y regresa a la mano la carta que coincida con esa cantidad de puntos

**Danan**: Asignado a la carta *Danan* y controla su efecto. Llama al método *Danan()* de **ClaseFranja** para conocer la suma de todas las franjas, calcula el promedio de las cartas, y le pasa ese parámetro al método *AplicarDanan()* en **ClaseFranja** para igualar todas las cartas al promedio entre ellas.

**Ejércitos**: Asignado a las cartas Soldados de la Banda del Halcón y Ejército Demoníaco y llama al método *AplicarEjércitos()* en la **ClaseFranja** para multiplicar su cantidad de puntos por la cantidad de cartas iguales en la franja.

**Farnesse**: Asignado a las cartas *Farnesse*. Se encarga de instanciar un eclipse en el clima de Distancia de Gutts y activa el efecto de Clima en ambas franjas de distancia.

**FloraHabilidad**: Asignado a las cartas de *Flora*, llama al método *Flora()* en la **ClaseFranja**. Este método controla que, si en la franja está una carta *Shierke* y una Flora, le suma 2 a *Shierke* y 1a *Flora.*

**GriffithHabilidad**: Asignado al Líder de Falconia. Llama al método RgresarMuerto() en la ClaseCementerio. Este método activa el método de PlayCard() de una carta random en la lista.

**GuttsHabilidad**: Asignado al Líder de Sacrificios. Activa el método *Gutts()* de la **ClaseFranja** de Cuerpo a Cuerpo de Sacrificios. Este método busca en la lista de cartas y, si se encuentra la carta *Casca*, instancia la carta *Gutts Jugable* en la franja.

**IrvineHabilidad**: Asignado a las cartas *Irvine*. Llama al método *Irvine()* en la **ClaseFranja** de Distancia de Sacrificios. Este método calcula el menor y, la carta que coincida con esos puntos, la manda al cementerio.

**LocusHabilidad**: Instancia una carta *Furia* en la zona de Clima de Cuerpo a Cuerpo de Falconia y activa el método *Furia()* en la **ClaseFranja** de Cuerpo a Cuerpo de Falconia.

**SkelletonKnightHabilidad**: Asignado a la carta *SkelletonKnight*. Crea un numero random entre 1 y 3, según el que sea llama al método SkelletonKnight() en la ClaseFranja correspondiente. Este método crea un numero random y elimina la carta de la lista que ocupe esa posición

**ZoddHabilidad**: Asignado a la carta *Zodd el Inmortal*. Busca la cantidad de cartas jugadas en cada fila del enemigo y calcula la menor distinta de cero y, si es menor que 4, se la pasa como parámetro al método *Zodd()* en la **ClaseFranja** correspondiente.

**Casca**: Asignado a las cartas de Casca. Recibe el menor de cada franja enemiga, los compara y el menor de ellos lo pasa como parámetro al método AplicarCasca() en la ClaseaFranja. Este método manda la carta cuyo poder coincida con el parámetro al cementerio.

**Judeau**: Recibe el mayor de cada franja, los compara y el mayor de ellos lo pasa como parámetro al método EliminarMayor() en la ClaseaFranja. Este método manda la carta cuyo poder coincida con el parámetro al cementerio.

**Rosine**: Recibe el mayor de cada franja, los compara y el mayor de ellos lo pasa como parámetro al método EliminarMayor() en la ClaseaFranja. Este método manda la carta cuyo poder coincida con el parámetro al cementerio.

**SEGUNDO PROYECTO**

Este segundo proyecto de programación trata esencialmente de crear tus propias cartas para jugar GwentSerk. En la pantalla de inicio se encuentra un nuevo apartado, ¨Personalizar Cartas¨, precisamente el lugar para diseñarlas. El botón de la derecha dirige a la sección en donde se elige la foto de tu nueva carta, a la derecha, donde se escribe su descripción pues esta no forma parte del código de la carta.

Cuando se introduce el código de la carta, pasa por un proceso para devolver el objeto ModelCard que se usa en Unity, a rasgos sencillos, es pasarlo por un Lexer que devuelve una lista de Tokens, y esta, se pasa como parámetro al Parser que se encarga de devolver este ModelCard

La clase **Token** cuenta con un string que representa su valor, la Localización con la línea y la columna donde se encuentra y un TokenType, su tipo, un Enum con los tipos de tokens posibles; texto, número, identificador, etc. Además de contar con un Diccionario de TokenType y string que representa los tipos de tokens con su correspondiente valor (algo que se usa en el Lexer).

El **Lexxer** (llamado Lexxer por cuestiones estéticas) es el que se encarga de convertir y separar la cadena entrante en tokens. Este trabaja con expresiones regulares que reconocen símbolos que se utilizan en el lenguaje, números y texto. El método que realiza el proceso es Tokenize(), este primero separa el código en líneas con el método Split() de string y luego se va iterando por cada una de ellas, primero se ve si hay algún “ indicando que es un string hasta que se encuentre otro, sino, se compara el valor con el diccionario de los TokenType con sus valores en busca de símbolos útiles o palabras reservadas, y si nada coincide se asume que es un identificador. Cuando se termina el proceso se devuelve una lista de Tokens.

Antes de pasar al Parser se deben abordar ciertos puntos, por ejemplo:

La clase abstracta **DSL** que cuenta con una lista de string que son los errores que puede tener, la Localización, un método Validation() que controla la semántica, e Implement() que devuelve un object que representa el valor que toma esta clase.

La clase abstracta **Statement** que posee lo mismo que DSL pero en vez de un Implement que devuelve un objeto, este, no devuelve nada.

La clase abstracta **ExpressionDSL** que hereda de DSL y que solo añade un IDType que es un Enum con los tipos de expresiones posibles y CheckType(IDType) que recibe un IDType y devuelve verdadero o falso dependiendo de si coincide con el ID de la expresión.

El struct **Int** que sirve para modelar los números en el lenguaje, posee un double con su valor y un método por cada operación que se puede hacer con él

Las ExpressionDSL se pueden dividir entre Unarias y Binarias

**Unarias:**

**UnaryExp<T>** es una clase abstracta que hereda de ExpressionDSL y que posee un T con su valor y sobrescribe el método ToString.

**UnaryObj<object>** hereda de UnaryExp y es para los objetos

**UnaryVal** que almacena el valor del token que recibe en el constructor

**UnaryOp** usada para los números negativos o los booleanos !

**Binarias:**

**BinaryExp<T>** clase abstracta que hereda de ExpressionDSL y que tiene una expresión izquierda, una operación representada por un token y una expresión derecha.

**NumericalBinary** hereda de BinaryExp<Int> que simula las operaciones de números con un miembro izquierdo y derecho

**BooleanExpression** también hereda de BinaryExp y es utilizada para simular las expresiones booleanas del tipo ==, ¡=, >, <, <=, >=

**BooleanBinary** es usada para las operaciones entre booleanos con los operadores || y &&

**StringBinary** para las operaciones de concatenación de string usando @ y @@

Luego de explicadas las expresiones, hay que hacerlo con las enunciaciones:

Las **Enunciation** heredan de Statement y tienen una ExpressionDSL que almacena su valor, un Token con su nombre, la localización, Type() que devuelve el IDType de su valor, un Token con la operación que puede o no tener(dígase ++, --, etc.), su respectivo método Implement y Validation y un Scope.

La clase **Scope** contiene un diccionario de string y expresiones que son en esencia las variables que están definidas en el contexto actual. Posee un Scope static llamado Global y es en el cual están todas las variables definidas

**IF** es la clase que hereda de Statement y que simula el comportamiento de un if de C# con su método Implement(), esta contiene el cuerpo del if, la expresión booleana como condición y el cuerpo del else en caso de tenerlo.

**For** la clase que hereda de Statement y que simula el comportamiento de un foreach de C# con su método Implement(), posee el nombre del identificador, la lista por donde se itera y el cuerpo de las instrucciones.

**While** la clase que hereda de Statement y que simula el comportamiento de un while de C# con su método Implement(), posee su condición y el cuerpo con las instrucciones.

**EnunBlock**, en esencia es una Lista de Statement y su método de Implementarse es recorrer cada una implementándose.

El **Parsel** (También llamado así por cuestiones estéticas):

Funciona iterando por todos los tokens que arroja el Lexer, esto lo hace mediante un int que controla la posición en la lista, un token TokenPlus que representa al siguiente y varios métodos para recogerla siendo los más importantes LookAhead y Consume.

**LookAhead(TokenType)** es utilizado para revisar el siguiente token y tiene 2 sobrecargas:

Si recibe un TokenType revisa el token en la posición siguiente, si coincide actualiza TokenPlus, y si no, lanza un error dado que no coinciden. Si el parámetro es null entonces lo actualiza sin preguntar

Si recibe una lista de TokenType revisa si contiene al tipo del siguiente

**Consume(TokenType)** se cerciora de que el próximo token es del mismo tipo y suma uno a la posición, si no, lanza un error (si recibe una lista de tipos lo va consumiendo en orden)

El Parser método **Parse()** que devuelve una lista de DSL que son los efectos y las cartas

Para parsear cada carta y efecto se utilizan pequeños métodos para analizar si la semántica de cada objeto está bien, como por ejemplo ParseName(), ParsePower(), ParseType(), entre bastantes más dando como resultado el objeto CardDSL o EffectDSL, estos tienen como propiedades expresiones y Statement todavía.

En el case de los efectos existe la clase **SavedEffect** en la que se guarda en el efecto ahora sí con campos interactuables como string, estos SavedEffect se guardan en una clase static llamada **EffectSaver**

La clase CardDSL en su método Implement devuelve una **ICard** que guarda todos sus campos en tipos accesibles como string y double y que se guardan en la clase static **CardSaver**

Una vez que se inicia el juego se utiliza un prefab en Unity llamado **CardTransler** que usa el método **CulturalAppropiation()** para transferir los datos de la ICard del CardSaver al prefab (incluyendo la foto y la descripción que se había usado antes), se instancia y se añade al mazo correspondiente para su uso.

Por último, la clase static **ErrorThrower** es la encargada de almacenar los errores de todo el proceso en una lista de string. Tiene un método Reset que vacía la lista y un método ThrowNReset que devuelve un string con todos los errores y luego hace Reset.