Informe y Manual de Usuario del proyecto de Programación Battle Cards, 2022

Rodrigo Mederos González - Daniel Machado Pérez

Estudiantes de Ciencia de la Computación, Facultad de Matemática y Computación, Universidad de La Habana

rodrigo.mederos@estudiantes.matcom.uh.cu

daniel.machado@estudiantes.matcom.uh.cu

Resumen

En el año 2022, como parte de la asignatura de Programación de Primer Año de la carrera Ciencia de la Computación de la Universidad de la Habana, fue orientado un proyecto a realizar por los estudiantes como parte de su evaluación del Segundo Semestre. La tarea consistía en diseñar un juego de tipo **Trading Card Game** llamado **Battle Cards**, con la particularidad de permitir la extensibilidad de crear nuevas cartas mediante un mini lenguaje de programación elaborado por nosotros. Para este caso en particular la aplicación visual fue desarrollada en **Godot**, utilizando **C#**.

Palabras Clave: Lenguaje, AST, Parser, Lexer, ChackSemantic, Evaluate, Expression, Card, Effect,Jugador Virtual, Board, Game, Godot.

1. Sobre el uso de Godot

Figura 1: Godot

La interfaz visual del juego se genera a través de un motor llamado **Godot**, que trabaja con una jerarquía bien definida, a través de árboles de escenas y nodos. Para este proyecto se han creado 4 escenas: **Main**, **Menu**, **Game**, **CardSupport**. El árbol está estructurado con **Main** como la raíz, cuyo nodo principal es de tipo **Node**, y **Menu** y **Game** como subnodos de **Main**, ambos con un nodo principal de tipo **Node2D**. La escena **CardSupport** no pertenece inicialmente al árbol de nodos, y su nodo principal es un **Node2D**. **Menu**, **Game** y **CardSupport** tienen vinculado un script, cuya clase principal lleva el mismo nombre de la respectiva escena, y es una subclase de la clase del tipo del nodo principal de la escena, en este caso **Node2D**. A través de estos scripts y otros que fueron añadidos no vinculados a **Godot**, es que se desarrolla la jugabilidad de **Cold War**. Las escenas **Menu**, **Game** y **CardSupport** constituyen en sí, cada una, un árbol de nodos, con gran cantidad de subnodos que facilita **Godot** para manejar las interacciones del juego y cambiar los estados del mismo. El juego comienza por la escena **Menu**, que se utiliza para configurar toda la partida, mediante un sistema de botones que permiten, entre otras cosas, seleccionar el **Deck**, las **Cartas**, el **Modo de Juego** y acceder al **Código Fuente** para crear nuevas **Cartas** y **Decks**. Luego de configurada la partida, el usuario se traslada a la escena **Game**, que es donde se desarrolla la dinámica del juego. Allí se manejará la interacción entre las **Cartas** y entre los jugadores para, mediante un **Flujo de Estados**, obtener un ganador.

2. Sobre las Clases y las Abstracciones

En la carpeta raíz del juego existen 6 scripts: **Board**, **CardSupport**, **CardTemplate**, **Game**, **Menu**, **PlayerTemplate**. De ellos existen 3 vinculados a nodos de **Godot**.

2.1 CardTemplate

Esta clase se encarga del trabajo con las **Cartas** "lógicas", que son guardadas en forma de archivos .json. La función principal de esta clase es deserializar los archivos y guardar la información, para luego ser transmitida a **CardSupport**. Además es la superclase de **Unit** y **Politic**:

2.1.1 Unit

Es la clase que engloba las cartas lógicas de tipo **Unidad**.

2.1.2 Politic

Es la clase que engloba las cartas de tipo **Política**. En este tipo de cartas existe una particularidad y es que no poseen **Health** ni **Attack**, solamente **Effect**, y son destruidas automáticamente luego de ejecutar el efecto.

2.2 CardSupport

Esta clase se encarga de manejar la interfaz visual de las cartas, como nodos de **Godot**, que serán añadidos al árbol de nodos del juego para poder interactuar con ellas. Contiene una propiedad de tipo **CardTemplate** (que será **Unit** o **Politic** según corresponda), para manejar los parámetros lógicos de la cartas. Cada carta posee un subnodo de tipo **Button** que facilita las interacciones a través de **Eventos del Mouse**, específicamente con los clicks **Izquierdo** y **Derecho**. El click **Izquierdo** será usado para seleccionar la **Carta**, y el click **Derecho** para atacarla, habiendo previamente seleccionado otra. Existen varios métodos en la clase:

- **GenerateCardVisualBase()**:Establece el fondo y el tamaño inicial de la **Carta**.
- **MakeCard()**: Establece visualmente las propiedades de la **Carta**.
- **UpdateCardVisual()**: Actualiza visualmente los parámetros de la **Carta**, así como su representación en **Game**.

Figura 2: Ejemplo de Carta

2.3 Board

La clase **Board** se utiliza para englobar en un objeto del tipo de la clase determinadas propiedades que serán utilizadas por un jugador. En nuestro juego cada jugador cuenta con su propio **Board**, lo que le permite una mejor interacción con sus partes. Muchas acciones del juego provocan cambios en el **Board** del jugador que las ejecuta, o el del jugador contrario. Estas propiedades son:

- **Deck**: Lista de cartas en el **Deck**.
- **HandCards**: Lista de **Cartas** en la mano.
- **CardsOnBoard**: Diccionario que agrupa las **Cartas** en el campo y su respectiva posición en el mismo.
- **Graveyard**: Lista de **Cartas** en el cementerio, luego de ser destruidas.

2.4 Player

Esta clase representa un jugador, que puede ser **Humano** o **Virtual**, por lo que es la superclase de **HumanPlayer** y **VirtualPlayer**. Un **Player** se caracteriza por tener un nombre, que coincidirá con la **Corriente Política** de las **Cartas** de su **Deck**. Además tiene un **Board** como se explicó anteriormente. También, en dependencia de si es el jugador del usuario o del enemigo, tendrá una posición en el tablero general, donde desplegará sus **Cartas** de la mano y podrá invocarlas. Por último, posee una referencia al juego donde está participando, para poder acceder a sus métodos y jugar. La clase posee los métodos:

- **Summon()**: Dada una **Carta** y una posición, invoca la carta en dicha posición, removiéndola de la mano, añadiéndola al campo y llamando a la función. **MakeSummon()** de la clase **Game** para ejecutar el cambio visual.

- **Attack()**: Dadas una **Carta** atacante y una atacada, ejecuta el ataque si no pertenecen al mismo jugador, destruyendo la perdedora del combate a través de un método **DestroyCard()** de **Game**, que las mueve al cementerio.
- **DrawCards()**: Mueve una catidad especificadas de **Cartas** desde el **Deck** hacia la mano del jugador.
- **Play()**: Método virtual para ser sobreescrito por sus respectivas subclases.

2.4.1 HumanPlayer

El constructor de esta clase hereda del constructor de la clase padre **Player**

2.4.2 VirtualPlayer

El constructor de esta clase hereda del constructor de la clase padre **Player**. La clase hereda de una interfaz, en este caso **IAggressiveVirtualPlayer**, a través de la cual podrá implementar una estrategia de juego. Con esta abstracción se gana extensibilidad en cuanto a la estrategia del **Jugador Virtual**, pues para implementar otra bastaría con crear una nueva interfaz de la que herede la clase **VirtualPlayer** e implementar sus métodos. La actual interfaz **IAggressiveVirtualPlayer** posee 7 métodos, cuya implementación se encuentra en **VirtualPlayer**:

- **PlayVirtualPlayer()**: Ejecuta la estrategia
- **SortDescendingByLife()**: Ordena **Cartas** atendiendo a la propiedad **Health** de forma descendente, facilita la ejecución de la estrategia.
- **SortAscendingByLife()**: Ordena **Cartas** atendiendo a la propiedad **Health** de forma ascendente, facilita la ejecución de la estrategia.
- **SortDescendingByAttack()**: Ordena **Cartas** atendiendo a la propiedad **Attack** de forma descendente, facilita la ejecución de la estrategia.
- **SortAscendingByAttack()**: Ordena **Cartas** atendiendo a la propiedad **Attack** de forma ascendente, facilita la ejecución de la estrategia.
- **SetPossiblePositions()**: Encuentra las posiciones libres para invocar en el campo.
- **SummonAllVirtualPlayer()**: Invoca todas las **Cartas** posibles

Además se sobreescribe el método **Play()** de **Player** para que ejecute **PlayVirtualPlayer()** que contiene la estrategia.

2.5 Menu

Esta clase es más amplia, contiene un gran número de funciones, por lo que haremos un resumen de las más importantes. Aquí se desarrolla la preparación del juego. Se propicia la navegación a través de un grupo de ventanas, que irán indicando los pasos para configurar la partida, mediante botones y otros nodos de **Godot** que facilitan la dinámica.
Entre los métodos más importantes están:

- **onCreateCardspressed()**: Abre el archivo **code.txt** donde se encuentra el **Código Fuente** para crear nuevas **Cartas**, nuevos **Decks** o modificar los existentes.

Figura 3: Menú Principal

```
import Decks;

PoliticalCurrent Anarquist
{

}

Card FriederichHayek
{
    CardType = "Unit";
    Rareness = "Legendary";
    Lore = "Austrian-British economist, legal theorist and philosopher who is best known for his defense of classical liberalism. Hayek shared the 1974 Nobel Memorial Prize in Economic Sciences with Gunnar Myrdal for their work on money and economic fluctuations, and the interdependence of economic, social and institutional phenomena.";
    Health = 4226;
    Attack = 4232;
    political_current = "Capitalist";
    PathToPhoto = "Hayek";
    EffectText = "Si la vida de Hayek es mayor de 3600, entonces robas una carta";
    Effect = [if(FriederichHayek.Health>=10000)
    {
        DrawCards(1);
    }
    else
    {
        if(DonaldTrump.Health>=1000)
        {
            DestroyCard();
        }
        else
        {
            DrawCards(2);
        }
    }

    ]
}
```

Figura 4: Ejemplo de Código Fuente

- **onPlayGamepressed()**: Ejecuta la función **ReadCode()** que lee y parsea el código de **code.txt**, luego pasa a la selección del **Deck** del usuario y del enemigo.
- **onSelectCardspressed()**: Una vez seleccionados ambos **Decks**, pasa a la selección de las **Cartas** del usuario, que serán desplegadas visualmente una a una.
- **ShowCardsForSelection()**: Muestra el visual de la **Carta** que puede ser seleccionada.
- **onRandompressed()**: Ejecuta la función **Random(int Amount)** que selecciona una cantidad determinada de cartas de forma aleatoria.
- **onGameModepressed()**: Una vez seleccionada la cantidad mínima de **Cartas** pasa a la selección del modo de juego (Human vs Human, Human vs Machine, Machine vs Machine).

- **onReadypressed()**: Una vez seleccionado el modo de juego, todo está listo para pasar al tablero y comenzar. Esta misma función, pero desde la clase **Game** inicializa ambos jugadores.

2.6 Game

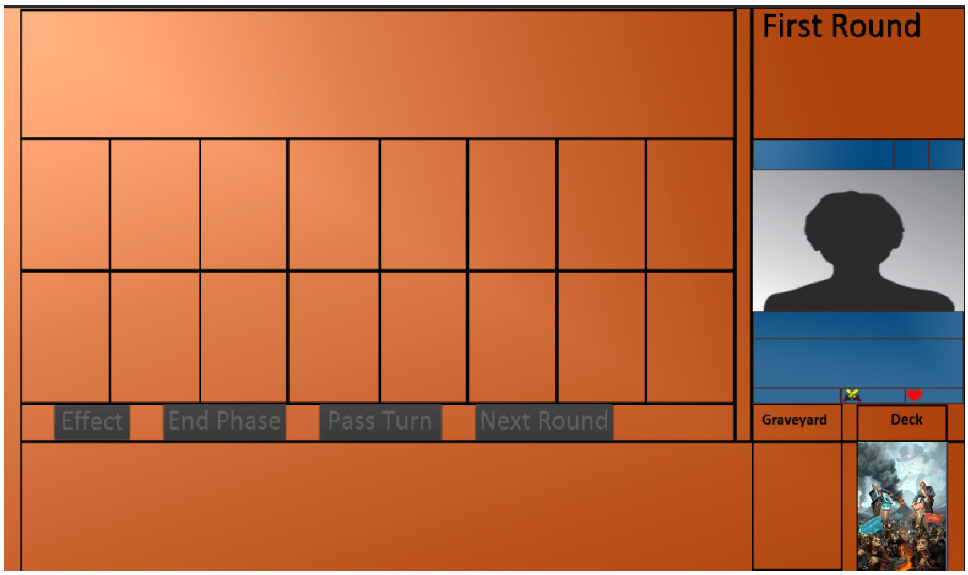

Figura 5: Tablero

Esta es la clase donde se va a desarrollar el juego. También es bastante amplia, por lo que igualmente haremos un resumen de lo más importante. Nuestro juego funciona a través de un flujo de estados o fases, y consiste en, mediante la interacción de **Cartas**, ganar 2 rondas para obtener la victria final. Una ronda esta constituida por turnos, que a su vez está constituido por fases, donde en cada fase se pueden realizar determinadas acciones. Para ganar una ronda hay que eliminar (mediante ataques y efectos) las **Cartas** del oponente o mantener la mayor vida total entre las **Cartas** del campo con respecto a las del campo rival. Primeramente se establecen las condiciones iniciales y luego se procede a interpretar y dar respuesta a las acciones de los jugadores. Para todo ello se han implementado un conjunto de métodos, que representan las acciones que se deben hacer en el momento oportuno de cada partida. Un resumen de las más importantes es el siguiente:

- **SetFields()**: Asigna las casillas del tablero que le corresponden a cada jugador.
- **SetInitialStateOfBoard()**: Establece el estado inicial y prepara el tablero para el juego.
- **CreatePlayers()**: Instancia los jugadores, pasándole los parámetros que le corresponden.
- **onDeckpressed()**: Si el juego se ha acabado, sale del juego, si no, llama a la función **Start()**, que reparte las cartas a cada jugador y comienza la fase inicial.
- **PrintCardsinRange()**: Muestra visualmente las **Cartas** de la mano de cada jugador.
- **MakeSummon()**: Ejecuta el cambio visual de invocar una **Carta**

Figura 6: Juego iniciado

- **onButtonpressed()**: Revisa si se cumplen las condiciones necesarias para invocar la **Carta** seleccionada en esa posición y llama a la función **Summon** del jugador al que pertence la carta
- **MakeDeck()**: Construye un **Deck**, a partir de la ruta hacia la carpeta donde están las cartas del **Deck**. Esta función es llamada desde **Menu** para construir el **Deck** del cual el usuario va a elegir sus **Cartas**.
- **ChangeSide()**: Indica que le toca jugar al otro jugador (el que no está jugando al ser llamada la función) y reinicia el flujo de fases.
- **DestroyCard()**: Mueve la **Carta** dada al cementerio visual y lógicamente.
- **onEndPhasepressed()**: Termina la fase actual y pasa para la siguiente, si la fase actual es la última, cambia de jugador y reinicia las fases.
- **onPassTurnpressed()**: Pasa directamente al turno de jugador rival. Si se presiona 2 veces seguidas, se termina la ronda y gana el jugador cuya sumatoria de la vida de sus **Cartas** en el campo sea mayor.
- **EndRound()**: Termina la ronda y declara un ganador de la ronda. Si es necesario, declara un ganador del juego.
- **onNextRoundpressed()**: Pasa para la próxima ronda, destruye las **Cartas** que queden en el campo de juego
- **onEffectButtonpressed()**: Se genera una fase donde se aplicarán los efectos de la carta previamente seleccionada
- **Input()**: Esta es una función especial de **Godot**. Recive y clasifica los eventos del ordenador, en este caso específicamente, los eventos de los botones del **Mouse**.

- **GetCardSelected()**: Devuelve la última **Carta** seleccionada (**null** si no hay ninguna **Carta** seleccionada).

Figura 7: En medio de una partida

2.7 GameStates

Esta clase no se encuentra en un script en la carpeta raíz (en States.cs la carpeta src) pero es fundamental para la ejecución correcta del juego. Es la encargada de regular los estados del juego que garantizarán el debido flujo de la partida. Principalmente contiene las las siguientes sublclases que representan fases:

- **MainPhase1**: En esta fase es posible realizar invocaciones y efectos
- **BattlePhase**: Es posible atacar las **Cartas** del oponente.
- **MainPhase2**: Nuevamente se pueden realizar invocaciones y efectos, luego de saber el resultado de la batalla.
- **EffectPhase**: Se genera cuando un jugador intenta ejecutar el efecto de una **Carta** y almacena todos los efectos de la carta que van a ser ejecutados.

3. Sobre el Lenguaje

El lenguaje diseñado para el juego es muy similar a **C#**. Para el mismo utilizamos una de las estructuras de programación más conocidas en el ámbito de la creación de lenguajes de programación: **Árboles de Sintaxis Abstracta (AST)**. Un **AST** es una representación de árbol de la estructura sintáctica simplificada del **Código Fuente** (en nuestro caso el que se encuentra en **code.txt**) escrito en cierto lenguaje de programación. Este árbol está constituido por nodos, que están representados por una clase abstracta llamada **ASTNode**, dentro del intérprete **AST**. De estos nodos heredarán clases como **ColdWarProgram**, que representa nuestro programa a la hora del parseo, donde estarán guardados los **Errores**, las **Corrientes Políticas** y las **Cartas** que se vayan generando.

3.1 Lexer

El **Lexer** es el componente del **Compilador** o **Intérprete** que identifica los símbolos escritos en un programa como válidos. Es el que reconoce las letras, palabras y símbolos. Nosotros creamos una clase **Token** que nos facilitará el trabajo con el **Lexer**. El **Lexer** leerá todos los símbolos para **Tokenizarlos**, convirtiéndolos en una lista de **Tokens**. Cada **Token** tendrá un tipo, un valor y una localización. La lista de **Tokens** se le pasa al **Parser**, que tiene una función que retorna un objeto de tipo **ColdWarProgram** (nodo principal de nuestro **AST**, posee las **Cards**, las **PoliticalCurrents**, los **Print** y los **EffectExpression**) y que se encarga de parsearla.

3.2 Parser

El **Parser**, como se dijo antes, será el encargado de, a partir del **TokenString** generado, leer los **Tokens** y generar **Cards**, **PoliticalCurrents** e interpretar condicionales de manera correcta. Por ejemplo, podrá leer y generar una **Carta** con esos valores, siempre y cuando cumpla con los requerimientos siguientes:

3.2.1 Sobre la creación de Cartas

Para crear **Cartas** existe la palabra clave **Card**, la cual debe ir seguida de un espacio y el nombre que se le quiera asignar. Luego, entre llaves, se escriben todas las características de la **Carta**. Se esperan 8 líneas de código con la siguiente estructura$< Keyword >=< Expression >$;. **Expression** debe ser un parámetro de tipo string encerrado entre comillas o una operación aritmética simple o compuesta. **Keyword** debe ser una de las palabras clave del lenguaje, que no deben repetirse y son las siguientes:

- **CardType** = ” $<$ *type* $>$;” Se especifica el tipo de **Carta**. Las únicas opciones son **Unidad** y **Política**. Debe escribirse entre comillas.
- **Rareness**= ” $<$ *rareness* $>$;” Se especifica la rareza de la **Carta**. Las únicas opciones posibles son **Legendaria** y **Común**. Debe escribirse entre comillas.
- **Lore**: = ” $<$ *lore* $>$ ”; Un breve texto que explica la historia de la carta. Debe escribirse entre comillas, ya que es un atributo de texto.
- **Health**: = $<$ *double* $>$;
- **Attack**: = $<$ *double* $>$; Números que representan la vida y ataque de la **Carta** con la que luchará en el tablero. En las fichas de tipo **Política** no se deben configurar estos campos.

- **PoliticalCurrent**: = " $<$ *politicalcurrent* $>$;" Se especifica la **Corriente Política** a la que pertenece la **Carta**. Debe elegirse entre los ya existentes o creados en el código. Debe escribirse entre comillas.
- **PathToPhoto**: = " $<$ *pathtophoto* $>$;" Pequeño texto que permite buscar en la base de datos de imágenes una foto adecuada para la **Carta**. Debe escribirse entre comillas. Puede ser solo un nombre, un apellido o algo que se refiera a la carta.
- **EffectText**: = " $<$ $effecttext$ $>$;" Un pequeño texto que explica el efecto de la carta. Debe estar encerrado entre comillas.
- **Effect**: = [$<$ *codestring* $>$]; Debe escribirse entre corchetes. El efecto se escribirá en lenguaje de código, que puede contener lo mismo que el programa exterior, con la adición de las funciones del efecto. Nota: Si no desea agregar un efecto a la carta a crear, estos dos últimos campos se pueden omitir.

Todas las propiedades de las cartas, excepto **Effect**, pueden ser escritas en cualquier orden siempre y cuando se conserve **Effect** como último campo, elaborado en su correcto formato.
En la versión actual del juego hay 8 efectos básicos. Estos se explican a continuación:

- **DestroyCard()**: Se elige una **Carta** en el campo para ser destruida.
- **DrawCards(**$<$ *int* $>$**)**: Obligatorio pasar un parámetro entero. Saca ese número de **Cartas** especificadas del **Deck**.
- **AddCardToDeck(**$<$ *Card* $>$**)**:
- **AddCardToBoard(**$<$ *Card* $>$**)**: Añade al mazo o tablero respectivamente una **Carta** correctamente diseñada dentro del espacio de efecto.
- **IncreaseHealth(**$<$ *int* $>$, $<$ *conditionaleffect* $>$**)**:
- **DecreaseHealth(**$<$ *int* $>$, $<$ *conditionaleffect* $>$**)**:
- **IncreaseAttack(**$<$ *int* $>$, $<$ *conditionaleffect* $>$**)**:
- **DecreaseAttack(**$<$ *int* $>$, $<$ *conditionaleffect* $>$**)**: Obligatorio pasarles un parámetro entero y la condición de efecto es opcional. Disminuye y aumenta la vida y el ataque respectivamente por la cantidad ingresada como parámetro. Si no se garantiza una condición de efecto, el jugador puede elegir la **Carta** que recibirá la acción. Si se especifica, entonces será automático de acuerdo con los detalles, que se pueden resumir de la siguiente manera:
  - **minHealth**:
  - **maxHealth**:
  - **minAttack**:
  - **maxAttack**: Estas condiciones eligen automáticamente como objetivo del efecto, entre las cartas del tablero, la que tiene menor o mayor salud o ataque respectivamente.

3.3 Expression

Luego de **Parsear** sin errores corresponde **Evaluar**. **Evaluar** es buscar todas las **Expressions** y asignarles un valor correcto con respecto a la entrada del usuario. Para ello se utilizan las **Expressions**

3.4 Expression

Definimos varios tipos de **Expressions**:

- **BinaryExpression**: add, sub, mul, div, POO, NumericComparerBool, SimpleBool.
- **AtomExpression**: Boolean, Number, Text.

Existe otro tipo de **Expression** llamada **EffectExpression**, que es la encargada de tener toda la información necesaria con respecto a un **Efecto** y evaluarlo.

3.5 CheckSemantic

Teniendo los valores de las expresiones, comprueba que son los correspondientes. Por ejemplo, si en el espacio de **Health** se escribió un int, esatría correcto, y todo lo contrario si se escribió un string o algún otro tipo. Igualmente reporta como incorrecto escribir una una **PoliticalCurrent** que no existe en el actual **Contexto**.

Nota: Es posible interpretar operaciones booleanas y aritméticas complejas. Al terminar este preceso sin errores reportados, se serializa la **Carta** en un archivo .json.

References

[1] Katrib, M. *Empezar a programar. Un enfoque multiparadigma con **C#**.* Editorial UH, 2020.

[2] Godot Docs