Fight Knights

Memoria: Desarrollo de un videojuego en Java

17 de Mayo de 2018

Alejandro Fernández Maceira

Oscar Fernández Sánchez

Adrián Palma Lima

Daniel López Moreno

Pedro Elías Cabada

Índice.

La pantalla principal.

1. Introducción e historia

Fight Knights es un videojuego de cartas en 2D por turnos con perspectiva top-down.

En la época medieval, dos caballeros luchan por ver quién es el elegido para salvar el reino de la amenaza que se avecina. Para ello, lucharán en combate singular valiéndose de sus cartas y de su estrategia.

Solo uno será el vencedor. ¿Quién se alzará con la victoria?

1.2. Metodología/Herramientas

Para la edición de las cartas, el tablero y demás elementos gráficos se han usado programas como Photoshop con licencia, Paint, Tiled y Graphicsgale. Para la creación del propio juego, la librería Slick2D del lenguaje de programación Java ha sido empleada.

1.3. Contexto

Este juego se ha creado en un contexto en el que los juegos de cartas son importantes en el mercado. Entre las inspiraciones se encuentran juegos como HearthStone de Blizzard, o Clash Royale de Supercell.

2. Diseño del videojuego

El tipo de videojuego elegido es cartas basado en fantasía medieval con perspectiva top-down. Es un videojuego con un concepto simple, pensado para el gran público, cuyas normas son claras y no requiere una gran curva de aprendizaje.

2.1. Requisitos mínimos

Los requisitos mínimos de hardware para jugar al juego no son muy elevados, cualquier ordenador actual puede ejecutarlo sin problemas de estabilidad ni cuelgues en el equipo, ya que, al no tener inteligencia artificial, el uso de CPU es escaso.

En cuanto a los requisitos software, un entorno de desarrollo compatible con Java es necesario para la ejecución, como por ejemplo NetBeans. No se necesita otro software específico, pues la librería Slick2D ya viene incluida en el propio proyecto.

2.2. Objetivo del juego

El objetivo del videojuego es ser el vencedor en el combate entre caballeros. Para ello, las cartas que se invocan en la mesa atacarán al jugador enemigo y le restarán puntos de vida en función de su ataque. Gana el jugador que consiga llevar a 0 los puntos de vida del contrario.

2.3. Lógica del juego

Los elementos del juego son:

a) Jugadores. Representados por su salud en un orbe azul, es el avatar del usuario. Solo dispone de la vida actual, y para jugar deberá usar las cartas de su mazo.
b) Unidades. Representadas por cartas, son los objetos con los que los jugadores combatirán. Cada jugador solo puede tener 5 cartas en su mano y 5 cartas en el tablero como máximo en todo momento. Cada unidad tiene características específicas, como curar vida al jugador o eliminar cartas del mazo del oponente. Sin embargo, para usar una carta, el jugador necesita energía.
c) Energía. Elemento necesario para usar cartas. Los jugadores empiezan su turno con 1 de energía, y cada turno se incrementará en uno. En cada carta está indicado el coste en energía, al usar la carta, la energía se restará del contador.
d) Tablero. Donde se colocarán las unidades de cada carta. Está representado con un estilo medieval, y en él se muestra la información de los jugadores y las cartas seleccionadas, así como botones para rendirse, salir, reiniciar o pasar turno.

El juego comienza en un turno aleatorio, puede tocarle al jugador 1 o al 2. Se mostrarán 3 cartas en el mazo, y el jugador decide lo que hacer.

La partida termina cuando un jugador quiera rendirse, salir o uno de los dos llegue a 0 puntos de vida.

2.4. Instrucciones

Para jugar, primero se selecciona una carta del mazo, en caso de haber. Si se dispone de energía, se podrá colocar en la mesa. Una vez colocada, no podrá atacar hasta el siguiente turno. Cuando se coloca una carta, se resta la energía disponible con el coste de la carta. Cada turno se añade una carta al mazo, hasta un máximo de 5, y energía, hasta un máximo de 10. Para atacar, se hace clic encima de una carta y se selecciona el objetivo: si no hay provocación activa, se puede atacar al jugador, que dispone de 20 de vida; o a una carta del tablero. Si hay provocación, solo se puede atacar a esa carta. Gana el que no se rinda o elimine al jugador contrario.

2.5. Controles

El método de control es con el ratón. Se seleccionarán los cuadros de texto haciendo clic con el ratón, al igual que las cartas y el ataque. Para pausar la partida, basta con pulsar la tecla de escape y el juego se detendrá.

2.6. Interfaz

Al ejecutar el programa, aparece una pantalla de carga. Después aparecerá un menú con la opción de pulsar start. Se clica en el cuadro de texto y empieza la partida. En el tablero hay varias opciones:

Captura de un momento del juego.

a. Reiniciar. Al pulsarse, reiniciará la partida y generará nuevas cartas para los jugadores.
b. Rendirse. Declara la victoria al jugador contrario.
c. Salir. Sale del juego.
d. Pasar turno. Termina el turno del jugador actual y empieza el del siguiente. Aumenta en 1 la energía del nuevo jugador.

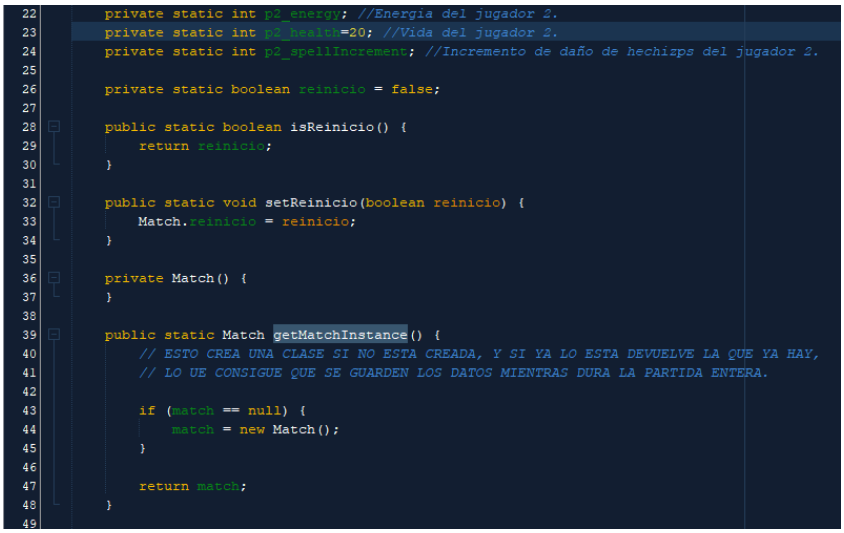
Para dar una ambientación medieval característica al juego, se reproduce un hilo musical con inspiración medieval. Si se quiere desactivar el sonido, en el menú inicial hay una opción para ello.

El juego se reproduce a 1280x720 de resolución, en modo ventana, para que sea compatible con la mayoría de pantallas actuales. En todo momento se muestran los FPS, para comprobar que todo funciona correctamente.

2.7. Patrones de diseño

Para la realización del videojuego se han empleado los siguientes patrones de diseño de videojuegos:

-Singleton. Para la creación de una sola instancia de partida. Este patrón de diseño permite que solo haya activa una instancia de la clase Match. Con esto se evita posibles pérdidas de información o escritura en instancias equivocadas. Para implementar el patrón, se ha creado un constructor privado, al que se accede mediante un método público y que no permite ser llamado más de una vez, garantizando la unidad de la instancia.



Fragmento de la clase match, donde se ve el patrón singleton.

-Factory. Para la creación de las cartas. Este patrón de diseño permite crear instancias de distintos tipos de cartas fácilmente. Se ha implementado creando un switch en el constructor de la clase Unit. Este constructor recibe un String como parámetro, el nombre de la carta, y busca en el switch el nombre que coincide. Una vez encontrado, inicializa los atributos de la clase con el valor establecido dentro del código case del switch.

```
26      public Unit(String card_name) throws SlickException {
27          switch (card_name) {
28              case "1":
29                  this.name = "Contramaestre";
30                  this.cost = 4;
31                  this.initial_damage = 3;
32                  this.initial_health = 3;
33                  this.damage = this.initial_damage;
34                  this.health = this.initial_health;
35                  this.type = "Minion";
36                  this.attackExecuted = true;
37                  this.rutaImagenTablero = new Image("res/1.png");
38                  this.rutaImagenMesa = new Image("res/r1.png");
39                  break;
40              case "2":
41                  this.name = "Caballero Sagrado";
42                  this.cost = 6;
43                  this.initial_damage = 5;
44                  this.initial_health = 6;
45                  this.damage = this.initial_damage;
46                  this.health = this.initial_health;
47                  this.type = "Minion";
48                  this.attackExecuted = true;
49                  this.guard = true;
50                  this.rutaImagenTablero = new Image("res/2.png");
51                  this.rutaImagenMesa = new Image("res/r2.png");
52                  break;
```

Fragmento del patrón Factory.

No se ha implementado una base de datos al ser un juego que no necesita guardar datos del jugador y solo permite jugar partidas rápidas.

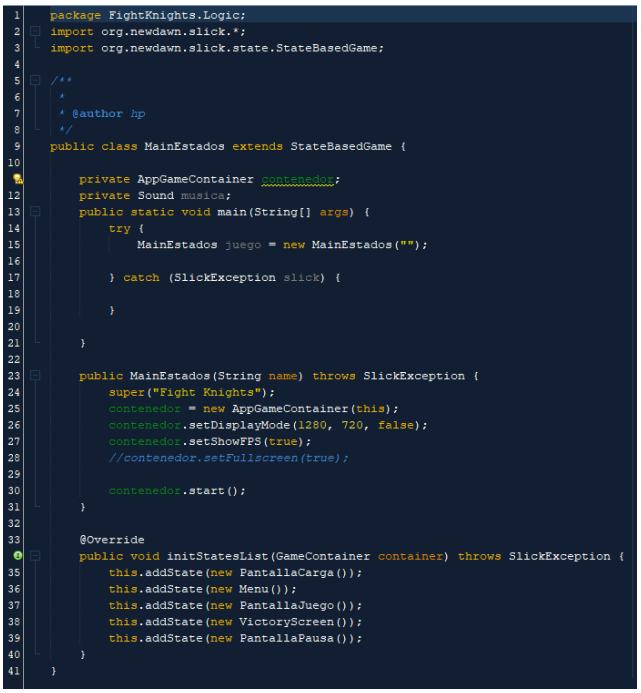
La detección de colisiones que posee el juego es la del ratón con los elementos de la pantalla. Para realizarla, se obtienen las coordenadas del ratón en cada momento, y luego se comparan con las deseadas, por ejemplo, un botón. Si coincide y el ratón está pulsado, se produce la colisión y se ejecuta la acción seleccionada. A continuación, un ejemplo del código.

```
91
92      @Override
        public void render(GameContainer container, StateBasedGame game, Graphics g) throws SlickException
94      {
95
96          tablero.draw();
97          pasarTurno.draw(1037, 540, 200, 70);
98          rendicion.draw(23, 500, 200, 50);
99          reinicio.draw(23, 550, 200, 50);
100         salir.draw(23, 600, 200, 50);
101
102         dibujarCartas();
103         dibujarEnTablero();
104
105         g.drawString(partida.getP_turn()+"", 500, 500);
106     }
107     @Override
        public void update(GameContainer container, StateBasedGame game, int delta) throws SlickException
109     {
110         ratonPulsado = raton.isMouseButtonDown(0);
111         ratonPress = raton.isMousePressed(Input.MOUSE_LEFT_BUTTON);
112         ratonSobreCartas();
113         ratonSobreBoton();
114         botonPulsado(game);
115         dragAndDrop();
116     }
117
118     public void ratonSobreBoton()
119     {
120         reinicioOver = ((raton.getMouseX()>=23&&raton.getMouseY()>=550)&&(raton.getMouseX()<=223&&raton.getMouseY()<=600));
121         rendicionOver = ((raton.getMouseX()>=23&&raton.getMouseY()>500)&&(raton.getMouseX()<=223&&raton.getMouseY()<=550));
122         salirOver = ((raton.getMouseX()>=23&&raton.getMouseY()>=600)&&(raton.getMouseX()<=223&&raton.getMouseY()<=650));
123         pasarTurnoOver = ((raton.getMouseX()>=1037&&raton.getMouseY()>540)&&(raton.getMouseX()<=1237&&raton.getMouseY()<=610));
124     }
125
126     public void botonPulsado(StateBasedGame game) throws SlickException
127     {
128         if(ratonPress)
129         {
130             //Reinicia la partida
131             if(reinicioOver)
```

Colisión entre el ratón y los botones.

3. Implementación

El juego está basado en estados, cada ventana es un estado diferente, y se ejecuta desde la clase MainEstados. Los métodos básicos render, update e init se han implementado. En init se inicializan las variables necesarias en cada clase, en update se actualiza la posición del ratón y si se ha hecho clic o no, y en render se generan las imágenes de las cartas, del tablero y demás elementos visuales.



```java
package FightKnights.Logic;
import org.newdawn.slick.*;
import org.newdawn.slick.state.StateBasedGame;

/**
 *
 * @author hp
 */
public class MainEstados extends StateBasedGame {

    private AppGameContainer contenedor;
    private Sound musica;
    public static void main(String[] args) {
        try {
            MainEstados juego = new MainEstados("");

        } catch (SlickException slick) {

        }

    }

    public MainEstados(String name) throws SlickException {
        super("Fight Knights");
        contenedor = new AppGameContainer(this);
        contenedor.setDisplayMode(1280, 720, false);
        contenedor.setShowFPS(true);
        //contenedor.setFullscreen(true);

        contenedor.start();
    }

    @Override
    public void initStatesList(GameContainer container) throws SlickException {
        this.addState(new PantallaCarga());
        this.addState(new Menu());
        this.addState(new PantallaJuego());
        this.addState(new VictoryScreen());
        this.addState(new PantallaPausa());
    }
}
```

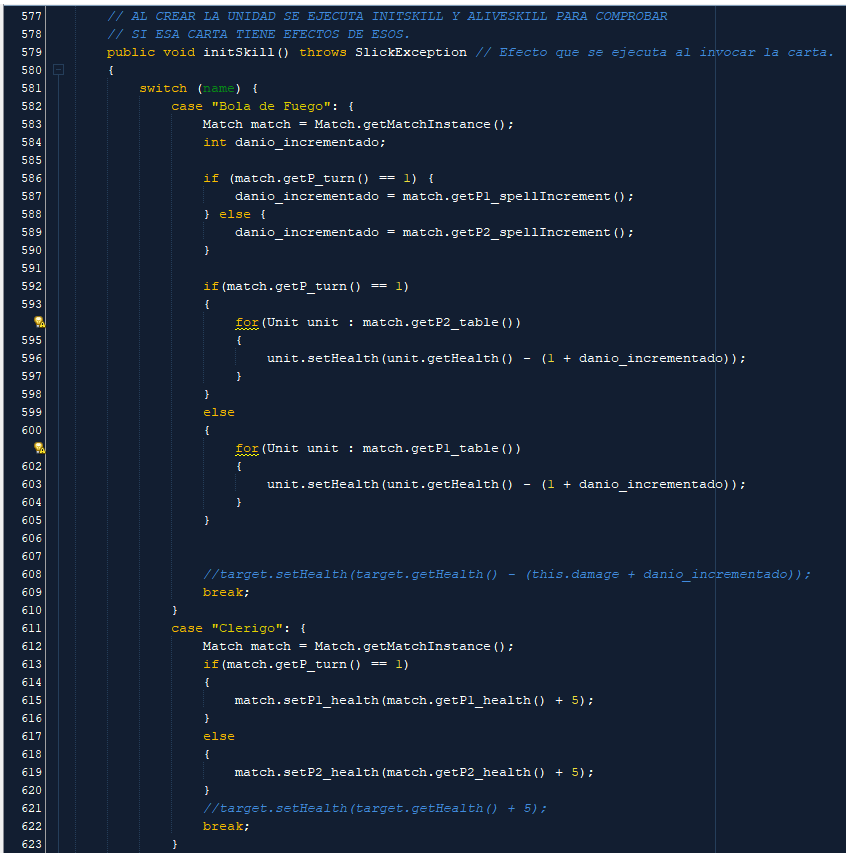
La clase MainEstados.

La clase MainEstados contiene la definición de cada pantalla/estado y el método main desde el que se ejecutará la aplicación. También se define el tamaño de pantalla y la resolución.

La clase PantallaCarga representa la primera pantalla que verá el jugador. Contiene el logo del equipo y un indicador para comprobar el estado de la carga del juego. No hay ninguna interacción con el jugador en esta pantalla. De esta clase se pasa a la siguiente, Menú.

La clase Menú es el menú principal del juego, donde el jugador elegirá empezar la partida. Contiene el logo del juego. Para capturar el ratón se emplea el método isMousePressed() de la clase Input de Slick2D. Si el ratón está pulsado y se encuentra dentro de las coordenadas del cuadro de texto dibujado, entonces empieza la partida. No empezará hasta que se pulse.

La clase Unit representa una unidad. El constructor es un switch en el que se inicializan las cartas según su número. También contiene métodos que emplean las cartas como initSkill, la habilidad que tienen al ponerse en la mesa; attackSkill, la habilidad al atacar; o deathSkill, la habilidad al morir. Según la carta, estas habilidades realizan una acción u otra.



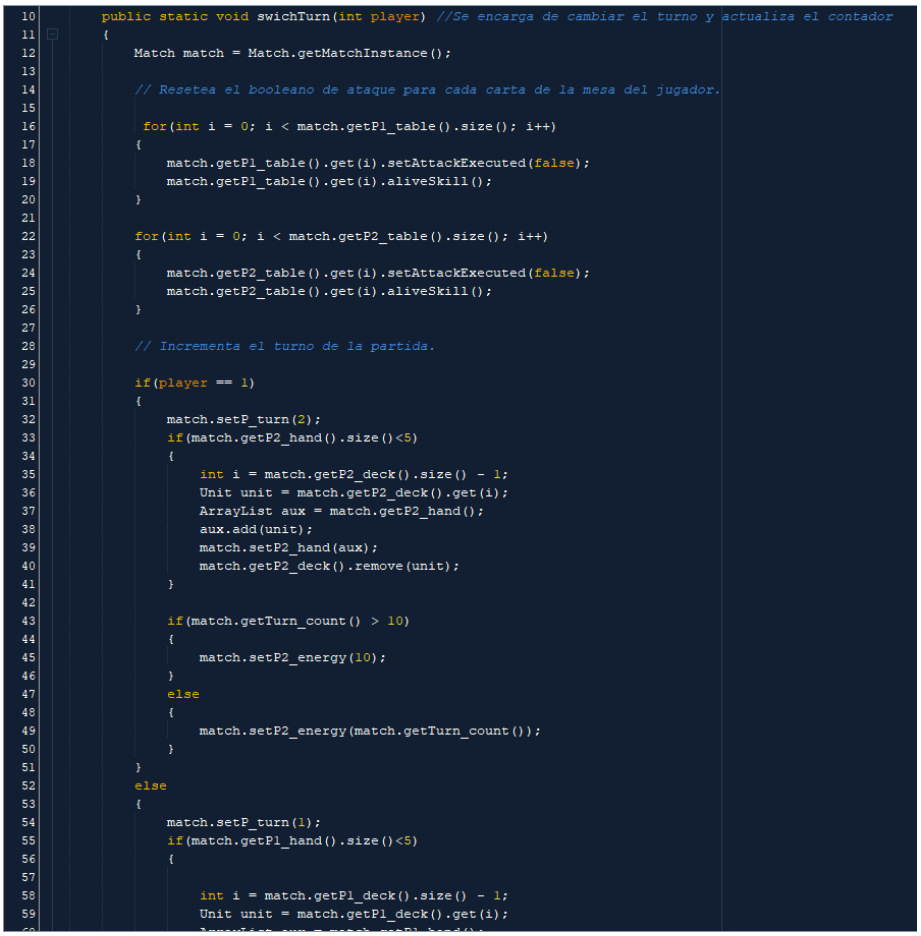
```
577         // AL CREAR LA UNIDAD SE EJECUTA INITSKILL Y ALIVESKILL PARA COMPROBAR
578         // SI ESA CARTA TIENE EFECTOS DE ESOS.
579         public void initSkill() throws SlickException // Efecto que se ejecuta al invocar la carta.
580         {
581             switch (name) {
582                 case "Bola de Fuego": {
583                     Match match = Match.getMatchInstance();
584                     int danio_incrementado;
585 
586                     if (match.getP_turn() == 1) {
587                         danio_incrementado = match.getP1_spellIncrement();
588                     } else {
589                         danio_incrementado = match.getP2_spellIncrement();
590                     }
591 
592                     if(match.getP_turn() == 1)
593                     {
595                         for(Unit unit : match.getP2_table())
596                         {
597                             unit.setHealth(unit.getHealth() - (1 + danio_incrementado));
598                         }
599                     }
600                     else
602                         for(Unit unit : match.getP1_table())
603                         {
604                             unit.setHealth(unit.getHealth() - (1 + danio_incrementado));
605                         }
606                     }
607 
608                     //target.setHealth(target.getHealth() - (this.damage + danio_incrementado));
609                     break;
610                 }
611                 case "Clerigo": {
612                     Match match = Match.getMatchInstance();
613                     if(match.getP_turn() == 1)
614                     {
615                         match.setP1_health(match.getP1_health() + 5);
616                     }
617                     else
618                     {
619                         match.setP2_health(match.getP2_health() + 5);
620                     }
621                     //target.setHealth(target.getHealth() + 5);
622                     break;
623                 }
```

Fragmento del método initSkill.

En el fragmento superior puede verse la estructura seguida para la habilidad inicial de las cartas. Por ejemplo, la bola de fuego resta uno de vida a todas las unidades enemigas teniendo en cuenta el nivel de hechizos, como puede se observa en las llamadas a setHealth con los valores establecidos.

La clase Match emplea el patrón de diseño singleton, creando una única instancia de la partida, con todos los valores necesarios para su funcionamiento: los mazos de cada jugador, representados con un arrayList, el turno, la vida de los jugadores, su energía, las unidades en el tablero, etc. También se incluyen métodos get y set de cada atributo.

La clase GameMethods contiene los métodos necesarios para el correcto funcionamiento de las cartas y sus acciones. Un método es cambiar turno, en el que se aumenta la energía del jugador siguiente, se suma una carta a su mano y cambia el turno al siguiente. Otro método es executeAttack, donde se comprueban los valores necesarios para que una carta ataque a otra, teniendo en cuenta si existe un efecto de provocación activo en ese momento. Para ello, se resta la vida de la carta objetivo en tantos puntos como ataque tenga la carta atacante.

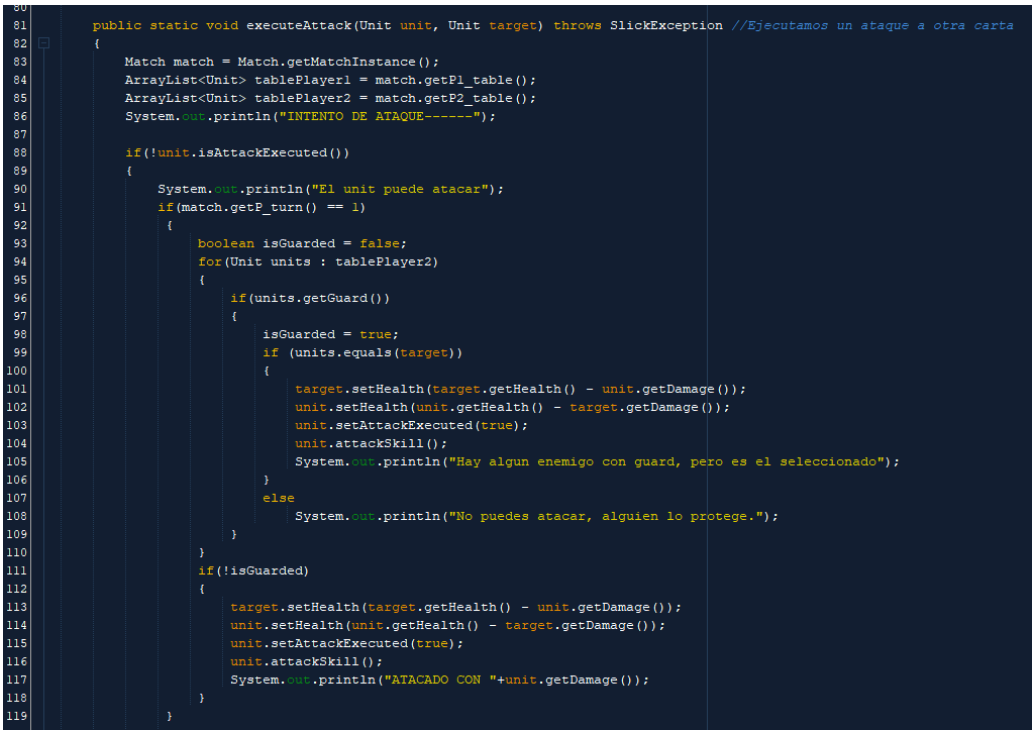


```
10      public static void swichTurn(int player) //Se encarga de cambiar el turno y actualiza el contador
11      {
12          Match match = Match.getMatchInstance();
13
14          // Resetea el booleano de ataque para cada carta de la mesa del jugador.
15
16           for(int i = 0; i < match.getP1_table().size(); i++)
17          {
18              match.getP1_table().get(i).setAttackExecuted(false);
19              match.getP1_table().get(i).aliveSkill();
20          }
21
22          for(int i = 0; i < match.getP2_table().size(); i++)
23          {
24              match.getP2_table().get(i).setAttackExecuted(false);
25              match.getP2_table().get(i).aliveSkill();
26          }
27
28          // Incrementa el turno de la partida.
29
30          if(player == 1)
31          {
32              match.setP_turn(2);
33              if(match.getP2_hand().size()<5)
34              {
35                  int i = match.getP2_deck().size() - 1;
36                  Unit unit = match.getP2_deck().get(i);
37                  ArrayList aux = match.getP2_hand();
38                  aux.add(unit);
39                  match.setP2_hand(aux);
40                  match.getP2_deck().remove(unit);
41              }
42
43              if(match.getTurn_count() > 10)
44              {
45                  match.setP2_energy(10);
46              }
47              else
48              {
49                  match.setP2_energy(match.getTurn_count());
50              }
51          }
52          else
53          {
54              match.setP_turn(1);
55              if(match.getP1_hand().size()<5)
56              {
57
58                  int i = match.getP1_deck().size() - 1;
59                  Unit unit = match.getP1_deck().get(i);
```

Fragmento del método switchTurn.

En la imagen superior se aprecia el método para cambiar turno, en el que primero se evita que las cartas ataquen y luego se comprueba el turno actual para actualizarlo y añadir la energía al nuevo jugador, así como una nueva carta si no está lleno su mazo ya.

También está en esta clase el método para atacar al jugador, en el que se pulsa la carta atacante y se selecciona el orbe azul del contrario. Después se resta tanta vida como ataque tenga la carta. El último método es invokeCard, donde se comprueba que el jugador tiene energía, y después se elimina la carta del mazo y se coloca en el tablero.



```
    public static void executeAttack(Unit unit, Unit target) throws SlickException //Ejecutamos un ataque a otra carta
    {
        Match match = Match.getMatchInstance();
        ArrayList<Unit> tablePlayer1 = match.getP1_table();
        ArrayList<Unit> tablePlayer2 = match.getP2_table();
        System.out.println("INTENTO DE ATAQUE------");

        if(!unit.isAttackExecuted())
        {
            System.out.println("El unit puede atacar");
            if(match.getP_turn() == 1)
             {
                boolean isGuarded = false;
                for(Unit units : tablePlayer2)
                {
                    if(units.getGuard())
                    {
                        isGuarded = true;
                        if (units.equals(target))
                        {
                            target.setHealth(target.getHealth() - unit.getDamage());
                            unit.setHealth(unit.getHealth() - target.getDamage());
                            unit.setAttackExecuted(true);
                            unit.attackSkill();
                            System.out.println("Hay algun enemigo con guard, pero es el seleccionado");
                        }
                        else
                            System.out.println("No puedes atacar, alguien lo protege.");
                    }
                }
                if(!isGuarded)
                {
                    target.setHealth(target.getHealth() - unit.getDamage());
                    unit.setHealth(unit.getHealth() - target.getDamage());
                    unit.setAttackExecuted(true);
                    unit.attackSkill();
                    System.out.println("ATACADO CON "+unit.getDamage());
                }
             }
```

Fragmento de executeAttack.

En este método se comprueba si una unidad del tablero puede ejecutar un ataque. En caso afirmativo, se ve si hay alguna carta con efecto de provocación. Si la hay, se actualiza la vida de la carta objetivo a la que tenía menos el nivel de daño de la carta atacante. Si llega a 0, se elimina. Además, una carta solo puede atacar una vez por turno, así que se pone a true el booleano que regula este comportamiento.

La clase PantallaJuego es donde se desarrolla la acción del videojuego. Aquí se crean y se usan todos los métodos y atributos de las clases anteriores, para el correcto desarrollo de la partida. Lo más importante es la captura de coordenadas del ratón. Según la posición en la que se encuentre, se realiza una acción u otra. También se realiza la lógica de los botones y su acción, como reiniciar, pausar o salir. Para usar una carta, primero se comprueba si el ratón está sobre esa carta, luego se obtienen datos de la carta y se arrastra al tablero, automáticamente se colocará en una posición disponible, o en ninguna si no hay espacio. Si se desplaza el cursor sobre las cartas del mazo, se ampliará la imagen para ver sus características. Dentro del tablero, cada unidad tiene a sus lados unos orbes azul y rojo que representan la vida y el ataque, respectivamente. Luego se captura la zona de actuación del ratón, y se actúa según la entrada del jugador. Si pulsa en la carta y luego en el orbe de vida del contrario, se atacará. Una carta solo puede atacar una vez por turno, y si hay alguna carta con efecto de provocación deberá ser atacada. Cuando la partida acaba, bien porque el jugador eligió rendirse, o porque la vida de alguno llegó a 0, se pasa a la siguiente clase.

También se ha creado un método que permite desplazar la carta seleccionada hasta el tablero. Para ello, se obtienen las posiciones x e y del cursor y de la carta y se va dibujando por donde pasa el ratón, dando sensación de movimiento. Este método es draganddrop. Para la primera carta sería:

```
179
180     public void dragAndDrop() throws SlickException
181     {
182         if(ratonPulsado)
183         {
184             if((raton.getMouseX()>=x1&&raton.getMouseX()<=x1+110)&&(raton.getMouseY()>=y1&&raton.getMouseY()<=y1+150))
185             {
186                 if(distanciaX == 0 && distanciaY == 0)
187                 {
188                     distanciaX = raton.getMouseX() - x1;
189                     distanciaY = raton.getMouseY() - y1;
190                 }
191                 x1 = raton.getMouseX() - distanciaX;
192                 y1 = raton.getMouseY() - distanciaY;
193 
194                 if(partida.getP1_hand().size() >= 1)
195                 {
196                     invocacionPosible = true;
197                     if(partida.getP_turn() == 1)
198                     {
199                         unidadSeleccionada = partida.getP1_hand().get(0);
200                     }
201                     else
202                     {
203                         unidadSeleccionada = partida.getP2_hand().get(0);
204                     }
205                 }
206             }
```

Fragmento del método dragAndDrop.

La clase VictoryScreen se muestra cuando se cumple una de las circunstancias anteriores. Contiene la información sobre quién es el ganador de la partida y dos botones, uno para salir, y otro para volver a jugar.

La pantalla de victoria.

La clase PantallaPausa se muestra cuando, dentro del juego principal, un jugador pulsa la tecla escape del teclado. La partida se detendrá y se reanudará una vez que se haga clic en cualquier parte de la pantalla.

3.1. Estructura del proyecto

El proyecto se divide en varios paquetes. Por un lado, están los relativos a la librería de Slick2D, que no se han modificado. Por otro se encuentran los paquetes FightKnights, en los que se han incluido las clases creadas para el desarrollo del videojuego.

Para una mayor facilidad en la localización de los recursos, todas las imágenes y el audio se han incluido en el paquete res. El último paquete es fonts, que contiene las fuentes empleadas para la visualización de los diversos textos.

4. Conclusión

Fight Knights es un juego de cartas medieval desarrollado en Java con el motor Slick2D y que permite a dos jugadores en el mismo ordenador jugar por turnos. Tiene una interfaz amigable y una gran carga lógica y matemática.

5. Bibliografía y fuentes de apoyo

Para la parte de código se ha consultado el manual de Java de Oracle y los foros de Stack Overflow, así como el manual de Slick2D y los videos de YouTube del usuario Makigas.

En la parte de diseño artístico, las cartas han sido creadas por nosotros desde cero, cortesía de Adrián Palma Lima. El tablero también lo hemos creado nosotros, gracias a Pedro Elías Cabada. La música la hemos cogido del artista Kevin Mc Leod, sin copyright. El estilo de letra es Hellgrazer de la página www.dafont.com.

6. División del trabajo

Adrián se ha encargado del código y del diseño de las cartas. Pedro se ha encargado de los diseños de las cartas, el tablero y demás elementos gráficos. Alejandro se ha encargado de la memoria. Todos hemos ayudado con el código de las clases y la implementación.