Universidad de Buenos Aires
Facultad De Ingeniería
Año 2022 - 1er Cuatrimestre

Algoritmos y Programación 2

TRABAJO PRÁCTICO 2

FECHA: 11 de junio de 2022

INTEGRANTES:

Branda, Sebastian - #90058
sbranda@fi.uba.ar

Rodriguez, Franco Ezequiel - #102815
frrodriguez@fi.uba.ar

Burgos, Lautaro - #108205
lburgos@fi.uba.ar

Hinojosa Baldera , Reginaldo Salvattore - #104708
rhinojosa.uba.ar

Apellido, Nombre - #000000
email@fi.uba.ar

Índice

1. Informe **2**

2. Manual de Usuario **4**

3. Manual de Programador **6**

1. Informe

El trabajo práctico ha consistido en implementar un juego basado en turnos denominado **Batalla Campal V2.0**, en el cual al menos dos jugadores pelean entre sí con el objetivo de conquistar el tablero. El tablero consiste en una serie de casilleros en formato de tres dimensiones los cuales son ocupados por unidades, a saber, soldados, aviones o barcos. Cada jugador comienza con una cantidad de soldados determinada por los jugadores, y la intención es que a través de estas unidades, el jugador ataque al enemigo y derribe a todos sus oponentes. El juego también posee un mazo de cartas las cuales provocan distintos eventos.

Cada ataque realizado el enemigo inhabilita el casillero objetivo del ataque, y en caso de estar ocupado por un soldado (sea enemigo o no), este es eliminado.

Los casilleros poseen un tipo de terreno, y dependiendo de éste las unidades podrán situarse sobre ellos o no. Estos tipo de terrenos son:

- Tierra
- Agua
- Aire

Los jugadores se turnan entre sí, en cada turno el jugador realiza un ataque dependiendo de la cantidad y del tipo de unidades que posea, asimismo, existe la posibilidad de lanzar misiles. Además los jugadores pueden mover uno de sus soldados a posiciones de tipo tierra vecinas, teniendo la posibilidad de recorrer gran parte del terreno y, en caso de que la posicion elegida sea una ocupada por un soldado (sea enemigo o no), ambos soldados son eliminados.

Luego de realizar de mover una de sus soldados el jugador levanta una carta del mazo y se juega inmediatamente. Los tipo de cartas implementados son:

- Misil
- Nuevo avión
- Nuevo Barco
- Somnífero
- Francotirador
- Harakiri

En todo momento los jugadores pueden observar sus propias piezas desplegadas en todo el tablero, pero no así las del enemigo, las cuales permanecen ocultas. El mapa se puede observar en los archivos BMP generados por el programa.

Para la implementacion de este trabajo práctico se ha utilizado memoria dinámica de forma ardua, tanto para el tablero, como para los casilleros, los jugadores, las unidades y las cartas. También se ha utilizado la plataforma de GitHub como entorno colaborativo para el desarrollo del mismo.

Cuestionario:

1) ¿Qué es un svn?
SVN o Subversion es una herramienta de control de versiones Open Source. Es un sistema que registra los cambios realizados sobre un archivo o un conjunto de archivos a lo largo del tiempo de modo que

puedas recuperar versiones especificas mas adelante. Esto nos permite ver los cambios del codigo que suceden en el tiempo. Si existe algun problema, nos permite volver a un estado del codigo en el cual el mismo funcionaba.

2) ¿Qué es git?
Git es otro sistema de control de versiones, es el mas usado del mundo. Es distribuido, por lo cual cada uno de los desarrolladores tiene una copia del repositorio en su equipo local. Es rapido y no necesita conexion a internet. Se pueden crear varias ramas del proyecto por separado de la rama principal, y despues pueden acoplarse/unirse a la rama main del proyecto.

3) ¿Qué es Github?
Github nos permite subir nuestro proyecto a la web, colaborar con otros o usar interfaz web. Guarda nuestro proyecto, los cambios, y cada una de sus versiones.

4) ¿Qué es un valgrind?
Valgrind es un conjunto de herramientas libres que nos ayuda en la depuracion de problemas de memoria y rendimiento de programas. Tiene un punto negativo que es que al usar valgrind, tenemos una perdida notable del rendimiento del programa, el mismo puede ser entre 5 a 20 veces mas lento al usar valgrind, y el consumo de memoria es mucho mayor. Aunque el mismo no se usa constantemente, si no en situaciones determinadas cuando queremos encontrar un error particular.

2. Manual de Usuario

Batalla Campal se juega de 2 a 10 jugadores, cada uno introduce de 2 a 10 soldados en un tablero tridimensional. Se establece un tamaño de mapa, que va de 2x2 a 1000x1000 casilleros y una altura que puede ir de 2 a 20 niveles de casilleros. Una vez echo esto se elige una clase de mapa predeterminado, entre los 4 posibles, introduciendo un numero del 1 al 4. A modo de ejemplo, el juego inicia solicitando los datos previamente descriptos:

```
Bienvenidos, ingrese la cantidad de jugadores: 2
Ingrese la cantidad de soldados por jugador con la que van a jugar: 2
Ingrese el largo del mapa: 10
Ingrese el ancho del mapa: 10
Ingrese el alto del mapa: 3
En que mapa desea jugar? (1 - 4): 3
```

Luego de determinar todas las caracteristicas del mapa, se crearan archivos de extension "BMP", uno por cada nivel de altura, para cada jugador, los cuales se actualizan antes de cada accion (como puede ser mover un soldado o lanzar un misil). Cada jugador ve solo sus propias unidades en el mapa.

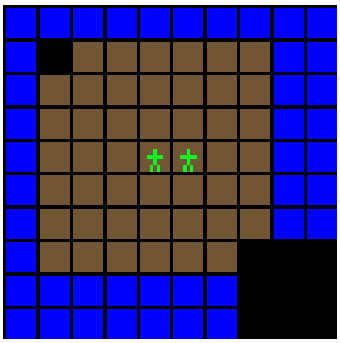

Figura 1: Dos soldados en plena batalla

Existe un mazo general, el cual contiene una cantidad de cartas igual a 4 veces la cantidad de jugadores en el juego. Las cartas del mazo estan mezcladas de forma aleatoria entre los 6 tipos de cartas que hay.

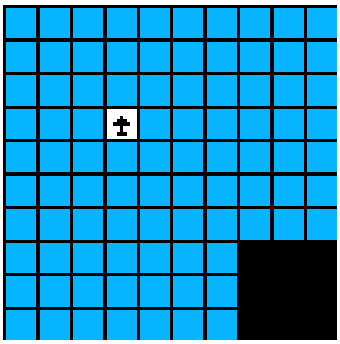

Figura 2: Avión de batalla en las alturas

Las posibles **Cartas** a levantar del mazo son:

- **Misil**: carta que agrega un misil adicional en el próximo turno al jugador que la obtenga. El area de daño de un misil es de 3x3x3 casilleros. Matando a las unidades que alcance e inhabilitando los casilleros.

- **Nuevo avión**: permite colocar un avión unicamente en un casillero que sea de aire, una vez colocado no se podrá mover. Tener un avion permite disparar 2 veces por turno.
- **Nuevo Barco**: permite colocar un barco, solamente en un casillero cuyo terreno sea agua, una vez colocado no se podrá mover. Tener un barco permite disparar un misil una vez por turno.
- **Somnífero**: el siguiente jugador pierde su turno
- **Francotirador**: permite realizar un disparo extra de forma inmediata
- **Harakiri**: aniquila automáticamente a una de las unidades del jugador que ha sacado la carta

Los casilleros del tablero pueden encontrarse en 3 estados:

- **Vacio**: significa que no hay unidades en él y esta habilitado para ser ocupado con una unidad.
- **Ocupado**: indica que hay una unidad en él.
- **Inhabilitado**: no se pueden colocar unidades en él. Un casillero se inhabilita luego de un disparo, o una pelea a muerte entre 2 soldados o un ataque de misil. En el mapa se ve de color negro.

Gana el ultimo jugador que tenga soldados en el mapa. Puede ocurrir un empate en caso de que los jugadores se queden sin soldados al mismo tiempo.

3. Manual de Programador

Para el desarrollo de este juego se ha utilizado memoria dinamica. La clase que maneja y se encarga de la lógica del juego es la clase Partida. Esta clase posee a su vez punteros a objetos como han de ser el tablero, los jugadores y el mazo de cartas. El tablero se ha implementado a través de una lista de listas de lista a objetos de tipo Casillero. Esto es, dando una forma tridimensional al tablero, donde cada nodo de las listas son punteros a Casilleros.

Para el manejo de los jugadores en la partida se ha decidido implementar un arreglo de punteros a Jugador, asi como tambien para la implementacion del mazo de cartas del juego.

Cada jugador posee cierta cantidad de unidades las cuales pueden ser variables a lo largo del juego, por los cual, se ha decidido implementar una lista de unidades para tal fin. Para diferenciar cada tipo de unidad, se ha declarado un tipo enumerativo

Para el manejo del posicionamiento de las unidades en el tablero, cada nodo en las listas del tablero posee un puntero a Casillero, el cual posee un puntero tanto a Unidad como a Jugador y una Coordenada, de esta forma se determina a quien pertenece la unidad que se sitúa en el casillero. Asimismo cada objeto de tipo Unidad posee un puntero a una Coordenada, de forma tal que se puede acceder a un Casillero a partir de una Unidad de un jugador, asi como también, se puede acceder a una unidad a partir de un Casillero en particular.

Para la creacion del tablero se debe especificar las medidas correspondientes de largo, ancho y alto, asi como también uno de los cuatro mapas que se han desarrollado

```
Tablero::Tablero(unsigned int largo, unsigned int ancho, unsigned int alto, int nroMapa)
```

Para la entrada de datos por consola se han implementado los siguientes métodos de la clase Partida, los cuales se encargan de la validación de los datos de entrada a lo largo de todo el programa:

```
void Partida::pedirDatos(unsigned int& mapaLargo, unsigned int& mapaAncho, unsigned
↪  int& mapaAlto)
unsigned int Partida::ingresarNumeroYValidar(int minimo, int maximo)
```

Dentro de la clase Partida existen dos métodos importantes que instrumentan la lógica del juego, delegando las tareas a otros métodos, primeramente a saber:

```
void Partida::inicializarPartida()
```

Se encarga del pedido de datos de entrada para inicializar el tablero con las medidas indicadas, asi como también de establecer tipo de terreno de cada Casillero, de la creación del arreglo de punteros a Jugador, de la creación del mazo de cartas de forma aleatoria, y de solicitar y gestionar el posicionamento los soldados iniciales de cada jugador. El segundo método importante es:

```
void Partida::siguienteTurno()
```

El cual se encarga del manejo de turnos, en los cuales cada jugador puede realizar los disparos que tiene disponibles, lanzar misiles, mover unidades y sacar una carta del mazo de cartas, en cada etapa se exporta el tablero para que el usuario pueda ver en las correspondientes imagenes bitmap de su tablero, escondiendo las unidades del enemigo. Asimismo se chequea luego de cada acción si es que las condiciones de finalización del juego se han terminado.

```
void Partida::siguienteTurno(){
    unsigned int nroJugadorEnTurno = this->getTurno() % this->getCantidadJugadores();
    cout << endl << "[***]\tTurno del Jugador #" << nroJugadorEnTurno;
    this->realizarDisparosJugador(nroJugadorEnTurno,
    ↪  this->jugadores[nroJugadorEnTurno]->getCantidadDisparosDisponibles());
```

```
5       if(this->haTerminado() == true){
6           return;
7       }
8       this->lanzarMisilesJugador(nroJugadorEnTurno,
        ↪  this->jugadores[nroJugadorEnTurno]->getCantidadMisilesDisponibles());
9       if(this->haTerminado() == true){
10          return;
11      }
12      this->moverUnidad(nroJugadorEnTurno);
13      if(this->haTerminado() == true){
14          return;
15      }
16      this->sacarCartaDelMazo(nroJugadorEnTurno);
17      if(this->haTerminado() == true){
18          return;
19      }
20      this->exportarTablero(nroJugadorEnTurno);
21      this->turno++;
22  }
```

Esta condición de finalización está dada por la cantidad de soldados que los jugadores tienen. Si existe como máximo un jugador con soldados, entonces el juego ha terminado. Esto es, porque existe sólo un jugador con soldados, o los soldados de todos los jugadores han muerto.

```
1   bool Partida::haTerminado(){
2       if(this->getCantidadJugadoresConSoldados() <= 1){
3           cout << endl << "El juego ha terminado!";
4           return true;
5       }
6       return false;
7   }
```

En caso de que todos los soldados de un jugador hayan muerto, el resto de las unidades (aviones y barcos) huyen con gran temor del campo de batalla por medio del siguiente método

```
1   void Partida::jugadorEmprendeRetirada(unsigned int nroJugador){
2       if(this->jugadores[nroJugador] == NULL){
3           return;
4       }
5       Unidad* unidad = this->jugadores[nroJugador]->buscarUnidad(1);
6       Coordenada* coordenada;
7       Casillero* casillero;
8       while(unidad != NULL){
9           coordenada = unidad->getPosicion();
10          if(!coordenada){
11              throw "ERROR EN COORDENADA AL EMPRENDER RETIRADA";
12          }
13          casillero = this->tablero->getCasillero(coordenada->getLargo(),
            ↪  coordenada->getAncho(), coordenada->getAlto());
14          if(!casillero){
15              throw "ERROR EN CASILLERO AL EMPRENDER RETIRADA";
16          }
17          this->jugadores[nroJugador]->removerUnidad(unidad);
18          casillero->inhabilitar();
```

```
19      }
20      cout << endl << "Al morir todos sus soldados, las unidades del jugador #" <<
        ↪  nroJugador << " han emprendido retirada";
21  }
```

Para la generación de archivos de tipo BMP, se ha implementado el siguiente método:

```
1  void Partida::exportarTablero(unsigned int nroJugador);
```

El cual a partir del número de jugador indicado, recorre completamente todos los casilleros del tablero, indicando aquellos que estan inhabilitados, y en caso de que estén ocupados por unidades enemigas, se muestran al jugador como si estuviesen vacias, esto es, mostrando el tipo de terreno del casillero.

En más detalle, primeramente crea todos los modelos BMP a ser utilizados (tablero, unidades y terreno). Lee todo el tablero, casillero por casillero en ancho, largo y alto mediante el metodo **Tablero::getCasillero()** que devuelve un puntero a casillero.
Con ese puntero a Casillero se obtienen todos los datos del casillero mediante sus diferentes metodos, estado mediante **getEstado()**, terreno mediante **getTipoDeTerreno()** y la coordenada de este mediante **getCoordenada()**.

Una vez que lee el estado del casillero, si este está inhabilitado, copia el modelo de terreno inhabilitado a ese casillero mediante **copiarBMPAlMapa()**. Si el casillero está ocupado, se revisa que tipo de unidad es la que ocupa ese casillero y mediante **copiarBMPAlMapa()** se agrega su modelo correspondiente al BMP del tablero.

En caso de que el casillero se encuentre vació, se determina qué tipo de terreno posee el casillero y luego mediante **copiarBMPAlMapa()** se agrega el modelo correspondiente al mapa.
El BMP del tablero se exporta finalmente mediante el metodo **WriteToFile()** perteneciente a la libreria **EasyBMP**. El nombre de los diferentes archivos BMP que se exportan (1 por cada nivel de altura del mapa) va a depender del jugador que sea, se realiza una conversion del numero del jugador y del nivel del mapa, a tipo de dato char , para poder utilizarlo como parametro al método **WriteToFile()**, los archivos van tener nombre del tipo "jugador<NumeroJugador>altura<NumeroNivelAltura>.bmp".

Para la implementación de los distintos tipos de casilleros se ha creado una nueva librería o header llamada **visuales.hpp**. La cual posee las siguientes funciones:

```
1  void dibujarCasilleros(BMP &Tablero);
```

Procedimiento que se encarga de dibujar las lineas que separan cada uno de los casilleros. Lo hace cada 10 pixeles, por lo tanto cada casillero mide visualmente 9x9 pixeles. Este procedimiento se vale de dos sub-funciones llamadas **dibujarCasillerosHorizontal()** y **dibujarCasillerosVertical()**, que cada uno como su nombre indica se encarga uno de dibujar las lineas horizontales y las verticales respectivamente.

El siguiente procedimiento se encarga de establecer los valores basicos del tablero BMP.

```
1  void crearBMPTablero(BMP &TableroVisual,unsigned int ancho,unsigned int largo);
```

Establece su tamaño y dibuja sus casilleros mediante **dibujarCasilleros()**. Para pasar de los valores de ancho y largo que recibe por parametro a su adaptación en pixeles. Se pasan esos parámetros a la función **adaptarMedidas()**, que multiplica ambos parametros por 10. Una vez adaptados sus valores, se los asigna al BMP del tablero mediante **establecerMedidas()**, que a su vez se vale al metodo **setSize()** de **".EasyBMP"** para hacerlo.

```
1 void pintarArea(BMP &Tablero,int xInicial,int xFinal,int yInicial,int yFinal,int
  ↪  r,int g,int b);
```

Como su nombre lo indica, esta función se encarga de pintar un área especificada de un objeto BMP del color designado mediante sus parámetros. Pinta desde xInicial a xFinal ,y desde yInicial hasta yFinal, del color que se forme de la combinacion de los parametros r,g y b, que se le asignan a cada pixel dentro del area especificada al objeto BMP.

```
1 void crearBMPUnidades(BMP &Soldado,BMP &Barco,BMP &Avion);
```

Esta función se encarga de asignar a todos los BMP de las unidades sus respectivos modelos de 9x9 pixeles, lo hace mediante las funciones **crearBMPSoldado()**, **crearBMPAvion()** y **crearBMP-Barco()**.

```
1 void crearBMPTiposTerreno(BMP &Tierra,BMP &Agua,BMP &Aire,BMP &TInhabilitado);
```

Procedimiento que se encarga de asignarle a todos los BMP que son para los casilleros vacios, sus respectivos modelos de 9x9 pixeles cada uno. El terreno de un casillero puede ser del tipo:

- Agua (Color azul)
- Tierra(Color marron)
- Aire(Color celeste)
- Terreno inhabilitado (Color negro)

Esto es realizado mediante las funciones **crearBMPAgua()**, **crearBMPAire()**, **crearBMPTierra()** y **crearBMPTInhabilitado()**.

```
1 void copiarBMPAlMapa(BMP& Mapa,BMP Modificacion,int xInicial,int yInicial);
```

Esta función se ha implementado para copiar un objeto BMP llamado modificacion (una unidad o un tipo de terreno) a su casillero equivalente en pixeles, los valores xInicial y yInicial son la coordenada x e y del casillero a donde se debe copiar el BMP. Para adaptar esos valores a pixeles se usa la funcion **adaptarValoresBMP()**.