Alba Casillas Rodríguez con DNI: 76738108B
Jose Manuel Osuna Luque con DNI: 20224440B

Grupo de prácticas: martes (D1)

METODOLOGÍA DE LA PROGRAMACIÓN

PROYECTO FINAL

30 de mayo del 2019

Parte A

En la primera parte del proyecto, hemos desarrollado las clases Particula y ConjuntoParticulas.

La representación privada de la clase **Particula**:

```
class Particula{

private:

    float x;
    float y;
    float dx;
    float dy;
    Color colorForma;
    bool particulaMovil;

    void copiar(const Particula& nueva);

};
```

Utilizamos **cuatro variables float** para indicar la posición en un eje (X,Y) y su desplazamiento en dicho eje (Dx, Dy), una **variable de tipo Color** (un enumerado disponible de clase MiniWin) para representar el color de esta y un **booleano** "particulaMovil" que señala si una partícula debe ser considerada como móvil o fija, variable que nos servirá para facilitarnos la implementación de ciertos métodos.

También se ha añadido un **método "copiar"**, basándonos en la implementación explicada en clase, para evitar la repetición de código a la hora de usar el constructor de copia u el operador de asignación.

La representación privada de la clase **ConjuntoParticula**:

```
class ConjuntoParticulas{

private:

    Particula* set; //Array de particulas
    int capacidad;  //Capacidad del Array
    int utiles;     //Posiciones ocupadas
    bool movil;     //Indica si el conjunto tiene particulas moviles o fijas

    void ReservaMemoria(int capacidadC);
    void LiberaMemoria();
    void CopiarDatos(const ConjuntoParticulas& otro);
    void Redimensionar(int nuevaCapacidad);

};
```

Hacemos uso de un **array de objetos de tipo Particula**, con su correspondientes variables "**capacidad**" y "**utiles**" (dos enteros), y un booleano "movil" para indicar si el conjunto contiene partículas consideradas círculos (móviles) o rectángulos (fijas).

A su vez, incluimos varios métodos esenciales para el desarrollo del proyecto que gestionan la memoria dinámica y evitan la repetición de código.

La clase contiene métodos públicos que gestionan la agregación y borrado de partículas y así como el movimiento de estas; sin embargo, procedemos a explicar detalladamente la implementación de algunos métodos que han sido esenciales para el desarrollo de la asignatura, como los de gestión de memoria o sobrecarga de operadores:

```
void ConjuntoParticulas::ReservaMemoria(int capacidadC){
    set = new Particula[capacidadC];
}
```

Reserva memoria con un tamaño capacidadC para almacenar objetos de tipo Particula.

```
void ConjuntoParticulas::LiberaMemoria(){

    if(set) {
        delete [] set;
        set = 0;
        utiles = 0;
        capacidad = 0;
        movil = false;
    }
}
```

Inicialmente comprobamos que el array de Particulas está inicializado. Tras esto, se libera la memoria que ocupa el array y actualizamos el resto de las variables a valor 0.

```
void ConjuntoParticulas::CopiarDatos(const ConjuntoParticulas& otro){
    capacidad = otro.getCapacidad();
    utiles = otro.getUtiles();
    movil = otro.getMovil();

    //El set de la clase; el otro set a copiar, n bytes a copiar
    memcpy(set, otro.getParticulas(), capacidad*sizeof(Particula));
}
```

Para la copia de datos, pasamos un conjunto de partículas por referencia constante como argumento, modificando los valores de las variables de la clase, a los pasados por argumento. Para copiar el array de partículas utilizamos la función memcpy:

void * memcpy (void * destination, const void * source, size_t num); //Copia un bloque de memoria

```
void ConjuntoParticulas::Redimensionar(int nuevaCapacidad){

    Particula* nuevo = new Particula[nuevaCapacidad];

    for(int i = 0; i<utiles; i++){
        nuevo[i] = set[i];
    }

    delete [] set;

    capacidad = nuevaCapacidad;

    set = nuevo;

}
```

Usaremos este método cuando no tengamos suficiente espacio de memoria, agrandando el conjunto de partículas al nuevo tamaño pasado por argumento. Para ello, necesitamos reservar memoria en un nuevo array, al cual le pasamos los valores antiguos (Por eso, el bucle for realiza iteraciones hasta “utiles” y no hasta “capacidad”). Liberamos la memoria del conjunto original y actualizamos punteros (y la variable capacidad).

```
ConjuntoParticulas& ConjuntoParticulas::operator=(const ConjuntoParticulas & nuevo){

    if(this != &nuevo){

        LiberaMemoria();
        ReservaMemoria(nuevo.getCapacidad());
        movil = nuevo.getMovil();
        CopiarDatos(nuevo);
    }

    return (*this);
}
```

La primera condición necesaria es comprobar que los punteros no sean iguales, puesto que no tendría sentido asignarse a sí mismo. Tras esto, liberamos memoria para poder reservar memoria con la capacidad del conjunto nuevo y eliminar el contenido del objeto implícito. Una vez reservada la memoria, se realiza la copia de los datos.

Se devuelve el propio objeto de la clase, con sus valores ya modificados.

```
ConjuntoParticulas ConjuntoParticulas::operator +(const ConjuntoParticulas &conjuntoP1) const{
    ConjuntoParticulas temp( (*this) );

    for(int i=0; i< conjuntoP1.getUtiles(); i++){
        temp.AgregaParticula(conjuntoP1.ObtieneParticula(i));
    }

    return (temp);

}
```

La sobrecarga del operador + consiste en sumar al objeto que hace la llamada el contenido de otro objeto pasado por argumento (el que se encuentra a la derecha del operador +).

En este caso no se devuelve una referencia, sino un objeto “temp” para evitar modificar el objeto que llama a dicho operador.

```
ostream& operator<<(ostream& salida, const ConjuntoParticulas& conjP){
    salida << "\t";

    for(int i=0; i<conjP.getUtiles(); i++){

        if(i == (conjP.getUtiles()-1) ){
            salida << conjP.ObtieneParticula(i);
        }else{
            salida << conjP.ObtieneParticula(i) << " ";
        }

    }

    salida << '\n';

    return salida;

}
```

Se realiza la sobrecarga del flujo de salida para poder mostrar (y cómo mostrarlos) por pantalla los datos del conjunto, recorriendo el vector que los contiene.

Para ello, utilizamos métodos de la clase como getUtiles() u ObtieneParticula(int) ya que este operador es una función externa a la clase (friend). Para esto, también se ha realizado la sobrecarga del flujo de salida de la clase Particula (implementado en Particula.cpp).

Parte B

La idea principal de esta parte, es la creación de una **clase Simulador** que se encargará de mover y visualizar un conjunto de partículas utilizando la biblioteca **MiniWin**.

Los métodos principales son Pintar(), Step() y Rebotes().

Para facilitar la implementación del método **Pintar()**, hemos creado varios métodos auxiliares privados.

```
void pintarParticulaCircular(const Particula& part);
```

Se encarga de pintar una partícula circular, asignándole un color y unas posiciones X e Y en el plano con un radio.

```
void pintarParticulaRectangular(const Particula& part);
```

Se encarga de pintar una partícula rectangular. En este caso tomamos como referencia el punto (X,Y), siendo este el centro del rectángulo, y para darle un espacio se le suman o restan valores a dichas coordenadas para crear un bloque:

```
rectangulo_lleno(X-(REC_DERECHA/2), Y-(REC_ABAJO/2), X+(REC_DERECHA/2), Y+((REC_ABAJO/2));
```

`void pintarConjuntoFijo();` y `void pintarConjuntoMovil();`

Formados por un bucle donde en cada iteración hace llamadas a los métodos anteriormente explicados para pintar círculos (conjuntoMovil) o rectángulos(conjuntoFijo).

Por último, en **Pintar()**, antes de hacer la llamada a estos dos últimos métodos, se invoca al método borrar() de la clase MiniWin para eliminar cualquier dibujo que haya en la pantalla. También se hace uso del método refrescar() de MiniWin (tras pintar los conjuntos), para actualizar la pantalla. De esta forma se simula un movimiento entre las partículas.

El método **Step()** actualiza los valores de las coordenadas de las partículas móviles y gestionan el choque entre ellas. Se invoca el método **Rebotes()**, que gestiona la colisión entre las partículas móviles con las fijas.

Adicionalmente, se pedía añadir una última funcionalidad del programa a elegir entre varias opciones. Hemos elegido la opción:

4. Al presionar una tecla, se agrega una nueva partícula fija/móvil al conjunto correspondiente.

Para ello, hemos creado un método nuevo en la clase Simulador:

```
void IncluirConTecla();
```

Usando la función de MiniWin **tecla()**, capturamos la pulsación de una tecla (en nuestro caso: **ESPACIO**), de forma que cada vez que se pulse se añade una partícula nueva. Como se pedía que fuese tanto fija como móvil, hemos abordado esta condición generando un número aleatorio con la función **rand()**, de forma que si genera un cero, se agrega una partícula móvil, y si no, una fija.

Este método será invocado en el caso tres del main proporcionado por el profesor (**game.IncluirConTecla()**).

Parte opcional

Como parte opcional, hemos añadido un caso 4 al switch del main, creando un pequeño juego simulando el **juego Arkanoid** (el jugador controla una pequeña plataforma rectangular con la que se intenta impedir que una bola salga de la zona de juego, haciéndola rebotar. En la parte superior hay *ladrillos* o *bloques*, que desaparecen al ser tocados por la bola).

Al proyecto hemos añadido una variable constante (en definiciones.h) llamada "**VIDAS**", que controla el fin del juego (perdiendo), ya que de lo contrario, habremos ganado cuando no quede ningún bloque en la parte superior, finalizando a su vez el juego.

En la clase **Particula**, hemos añadido tres métodos nuevos, los cuales son una modificación de los ya implementados anteriormente.

```
void MoverArknoid(int ancho, int alto, int& muertes);
```

En este método, hemos cambiado el hecho de que si la bola supera el margen inferior de la pantalla, esta vuelva al centro (en vez de rebotar). También aumentamos una **variable "muertes"**, la cual vamos comparando con el valor de la constante VIDAS.

```
void RebotaArkanoid(Particula& otra, const int & valorAncho);
```

El rebotar ahora tiene en cuenta con qué mitad del bloque se ha chocado. De manera que si se ha chocado por la parte izquierda, realiza un rebote hacia el mismo lado , y viceversa.

En la clase **ConjuntoParticulas** se ha añadido el método:

```
void gestionarTablero();
```

El método consiste en cambiar las coordenadas de las partículas que forman los ladrillos de la parte superior del juego para adaptarlas al ancho de la pantalla dividiéndolas en filas, dejando entre partícula y partícula y al borde, un pequeño espacio.

Lo principal del método es la condición if:

```
if(contador % (getUtiles()/FILAS_ARKANOID) == 0 && contador != 0){
	contador = 0;
	y = y+(ALTO_ARKANOID/2)+5;
	x = 25;
}
```

Como en cada fila los bloques tiene que estar a la misma altura, la variable Y siempre es fija. Cuando se cumple la condición (es decir, que una fila se ha llenado), "hay que cambiar de fila" y por lo tanto, se actualizan el valor de Y, y X vuelve a valer 25 (valor que mejor se ajustaba a la pantalla) para empezar desde la parte izquierda de esta.

Para saber cuantas partículas caben en una fila, hemos usado la siguiente fórmula:

```
int nBloques = (ancho/(ANCHO_ARKANOID+5))*FILAS_ARKANOID;
```

donde ancho es el tamaño de la pantalla y ANCHO_ARKANOID es el tamaño completo de cada bloque más un espacio entre dos ladrillos. El resultado de la operación se multiplica por el número de filas deseadas.

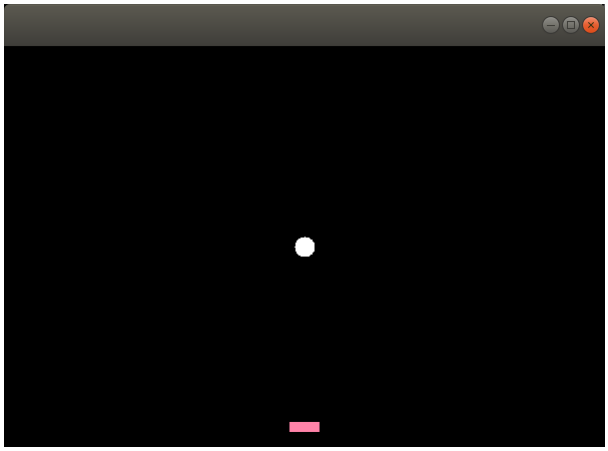
Por último, hemos añadido algunas funciones auxiliares que permiten representar el juego y gestionar los choques basándonos en los métodos del arkanoid explicados con anterioridad. Aún así, queremos destacar la función

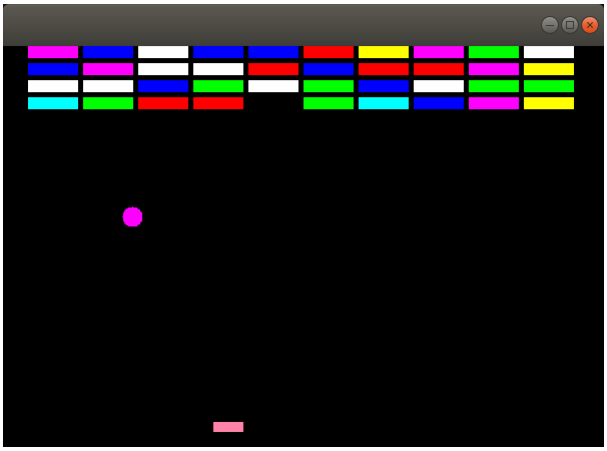
```
void moverBase(Particula& base, const int ancho);
```

que, con una idea similar a la opción elegida de añadir una partícula al pulsar una tecla, utilizamos las teclas “DERECHA” e “IZQUIERDA” del teclado para desplazar la base de un lado a otro horizontalmente sin que se sobrepasen los límites laterales de la pantalla, guardados en las variables:

```
int parteDerecha = (base.getX()+(REC_DERECHA/2))+DESPLAZAMIENTO;
int parteIzquierda = (base.getX()-(REC_DERECHA/2))-DESPLAZAMIENTO;
```

Al elegir la opción del juego (4), esto abrirá una ventana en negro con únicamente dos elementos en pantalla, una barra color coral y una pelota blanca en el centro. Para empezar a jugar será necesario que el jugador pulse la tecla RETORNO (enter, salto de línea). Al pulsarla, la pelota cambiará de color a uno rosado y el juego comenzará





Comprobaciones de Valgrind

A continuación se muestra una serie de capturas con los datos resultantes de la ejecución de nuestro proyecto usando la herramienta Valgrind:

```
  Opcion a probar: 1

** Prueba sobrecarga de operadores **
        Particula(263, 162, 3) Particula(690, 296, 5) Particula(410, 251, 5) Particula(172, 37, 3)

        Particula(50, 518, 5) Particula(54, 522, 5)

        Particula(50, 518, 5) Particula(54, 522, 5) Particula(266, 164, 3) Particula(695, 300, 5) Particu
la(413, 253, 5) Particula(173, 41, 3)

        Particula(50, 518, 5) Particula(54, 522, 5) Particula(266, 164, 3) Particula(695, 300, 5) Particu
la(413, 253, 5) Particula(173, 41, 3) Particula(266, 164, 3) Particula(695, 300, 5) Particula(413, 253, 5
) Particula(173, 41, 3)


^C==19554==
==19554== Process terminating with default action of signal 2 (SIGINT)
==19554==    at 0x5AC410D: __lll_lock_wait (lowlevellock.S:135)
==19554==    by 0x5ABD022: pthread_mutex_lock (pthread_mutex_lock.c:78)
==19554==    by 0x60CEEA7: xcb_writev (in /usr/lib/x86_64-linux-gnu/libxcb.so.1.1.0)
==19554==    by 0x4E7A145: _XSend (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==19554==    by 0x4E7A66B: _XReply (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==19554==    by 0x4E761CC: XSync (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==19554==    by 0x4E56E1D: XCloseDisplay (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==19554==    by 0x10DB0F: main (miniwin.cpp:740)
==19554==
==19554== HEAP SUMMARY:
==19554==     in use at exit: 52,890 bytes in 35 blocks
==19554==   total heap usage: 197 allocs, 162 frees, 138,310 bytes allocated
==19554==
==19554== 288 bytes in 1 blocks are possibly lost in loss record 22 of 30
==19554==    at 0x4C31B25: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==19554==    by 0x40134A6: allocate_dtv (dl-tls.c:286)
==19554==    by 0x40134A6: _dl_allocate_tls (dl-tls.c:530)
==19554==    by 0x5ABB227: allocate_stack (allocatestack.c:627)
==19554==    by 0x5ABB227: pthread_create@@GLIBC_2.2.5 (pthread_create.c:644)
==19554==    by 0x10D963: _maybe_call_main() (miniwin.cpp:673)
==19554==    by 0x10D9D5: _process_event() (miniwin.cpp:689)
==19554==    by 0x10DAC7: main (miniwin.cpp:733)
==19554==
==19554== LEAK SUMMARY:
==19554==    definitely lost: 0 bytes in 0 blocks
==19554==    indirectly lost: 0 bytes in 0 blocks
==19554==      possibly lost: 288 bytes in 1 blocks
==19554==    still reachable: 52,602 bytes in 34 blocks
==19554==         suppressed: 0 bytes in 0 blocks
==19554== Reachable blocks (those to which a pointer was found) are not shown.
==19554== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==19554==
==19554== For counts of detected and suppressed errors, rerun with: -v
==19554== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

```
==19572== Memcheck, a memory error detector
==19572== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==19572== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==19572== Command: ./proyectofinal
==19572==

  Opcion a probar: 2

** Prueba Conjunto Particulas **
^C==19572==
==19572== Process terminating with default action of signal 2 (SIGINT)
==19572==    at 0x5AC410D: __lll_lock_wait (lowlevellock.S:135)
==19572==    by 0x5ABD022: pthread_mutex_lock (pthread_mutex_lock.c:78)
==19572==    by 0x60CEEA7: xcb_writev (in /usr/lib/x86_64-linux-gnu/libxcb.so.1.1.0)
==19572==    by 0x4E7A145: _XSend (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==19572==    by 0x4E7A66B: _XReply (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==19572==    by 0x4E761CC: XSync (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==19572==    by 0x4E56E1D: XCloseDisplay (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==19572==    by 0x10DB0F: main (miniwin.cpp:740)
==19572==
==19572== HEAP SUMMARY:
==19572==     in use at exit: 52,890 bytes in 35 blocks
==19572==   total heap usage: 2,696 allocs, 2,661 frees, 1,164,498 bytes allocated
==19572==
==19572== 288 bytes in 1 blocks are possibly lost in loss record 22 of 30
==19572==    at 0x4C31B25: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==19572==    by 0x40134A6: allocate_dtv (dl-tls.c:286)
==19572==    by 0x40134A6: _dl_allocate_tls (dl-tls.c:530)
==19572==    by 0x5ABB227: allocate_stack (allocatestack.c:627)
==19572==    by 0x5ABB227: pthread_create@@GLIBC_2.2.5 (pthread_create.c:644)
==19572==    by 0x10D963: _maybe_call_main() (miniwin.cpp:673)
==19572==    by 0x10D9D5: _process_event() (miniwin.cpp:689)
==19572==    by 0x10DAC7: main (miniwin.cpp:733)
==19572==
==19572== LEAK SUMMARY:
==19572==    definitely lost: 0 bytes in 0 blocks
==19572==    indirectly lost: 0 bytes in 0 blocks
==19572==      possibly lost: 288 bytes in 1 blocks
==19572==    still reachable: 52,602 bytes in 34 blocks
==19572==         suppressed: 0 bytes in 0 blocks
==19572== Reachable blocks (those to which a pointer was found) are not shown.
==19572== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==19572==
==19572== For counts of detected and suppressed errors, rerun with: -v
==19572== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 200 from 1)
```

```
==19579== Memcheck, a memory error detector
==19579== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==19579== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==19579== Command: ./proyectofinal
==19579==

  Opcion a probar: 3

**************  Probando Simulador ****************
terminate called without an active exception
==19579==
==19579== Process terminating with default action of signal 6 (SIGABRT)
==19579==    at 0x5D10E97: raise (raise.c:51)
==19579==    by 0x5D12800: abort (abort.c:79)
==19579==    by 0x52008B6: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25)
==19579==    by 0x5206A05: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25)
==19579==    by 0x5206A40: std::terminate() (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25)
==19579==    by 0x5206449: __gxx_personality_v0 (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25)
==19579==    by 0x58AB647: ??? (in /lib/x86_64-linux-gnu/libgcc_s.so.1)
==19579==    by 0x58ABC3B: _Unwind_ForcedUnwind (in /lib/x86_64-linux-gnu/libgcc_s.so.1)
==19579==    by 0x5AC3F0F: __pthread_unwind (unwind.c:121)
==19579==    by 0x5E3BA5E: __pthread_unwind (forward.c:202)
==19579==    by 0x5E02734: __libc_enable_asynccancel (cancellation.S:81)
==19579==    by 0x5DE86D4: writev (writev.c:26)
==19579==
==19579== HEAP SUMMARY:
==19579==     in use at exit: 54,810 bytes in 39 blocks
==19579==   total heap usage: 2,118 allocs, 2,079 frees, 188,658 bytes allocated
==19579==
==19579== 288 bytes in 1 blocks are possibly lost in loss record 24 of 34
==19579==    at 0x4C31B25: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==19579==    by 0x40134A6: allocate_dtv (dl-tls.c:286)
==19579==    by 0x40134A6: _dl_allocate_tls (dl-tls.c:530)
==19579==    by 0x5ABB227: allocate_stack (allocatestack.c:627)
==19579==    by 0x5ABB227: pthread_create@@GLIBC_2.2.5 (pthread_create.c:644)
==19579==    by 0x10D963: _maybe_call_main() (miniwin.cpp:673)
==19579==    by 0x10D9D5: _process_event() (miniwin.cpp:689)
==19579==    by 0x10DAC7: main (miniwin.cpp:733)
==19579==
==19579== LEAK SUMMARY:
==19579==    definitely lost: 0 bytes in 0 blocks
==19579==    indirectly lost: 0 bytes in 0 blocks
==19579==      possibly lost: 288 bytes in 1 blocks
==19579==    still reachable: 54,522 bytes in 38 blocks
==19579==         suppressed: 0 bytes in 0 blocks
==19579== Reachable blocks (those to which a pointer was found) are not shown.
==19579== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==19579==
==19579== For counts of detected and suppressed errors, rerun with: -v
==19579== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 502 from 1)
```

```
==19600== Memcheck, a memory error detector
==19600== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==19600== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==19600== Command: ./proyectofinal
==19600==

  Opcion a probar: 4

**************  Probando Arkanoid ***************
^C==19600==
==19600== Process terminating with default action of signal 2 (SIGINT)
==19600==    at 0x5AC09F3: futex_wait_cancelable (futex-internal.h:88)
==19600==    by 0x5AC09F3: __pthread_cond_wait_common (pthread_cond_wait.c:502)
==19600==    by 0x5AC09F3: pthread_cond_wait@@GLIBC_2.3.2 (pthread_cond_wait.c:655)
==19600==    by 0x60CE951: ??? (in /usr/lib/x86_64-linux-gnu/libxcb.so.1.1.0)
==19600==    by 0x60D006E: ??? (in /usr/lib/x86_64-linux-gnu/libxcb.so.1.1.0)
==19600==    by 0x60D01EE: xcb_wait_for_reply64 (in /usr/lib/x86_64-linux-gnu/libxcb.so.1.1.0)
==19600==    by 0x4E7A6E7: _XReply (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==19600==    by 0x4E761CC: XSync (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==19600==    by 0x4E56E1D: XCloseDisplay (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==19600==    by 0x10DB0F: main (miniwin.cpp:740)
==19600==
==19600== HEAP SUMMARY:
==19600==     in use at exit: 79,134 bytes in 148 blocks
==19600==   total heap usage: 3,777 allocs, 3,629 frees, 266,470 bytes allocated
==19600==
==19600== 288 bytes in 1 blocks are possibly lost in loss record 26 of 40
==19600==    at 0x4C31B25: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==19600==    by 0x40134A6: allocate_dtv (dl-tls.c:286)
==19600==    by 0x40134A6: _dl_allocate_tls (dl-tls.c:530)
==19600==    by 0x5ABB227: allocate_stack (allocatestack.c:627)
==19600==    by 0x5ABB227: pthread_create@@GLIBC_2.2.5 (pthread_create.c:644)
==19600==    by 0x10D963: _maybe_call_main() (miniwin.cpp:673)
==19600==    by 0x10D9D5: _process_event() (miniwin.cpp:689)
==19600==    by 0x10DAC7: main (miniwin.cpp:733)
==19600==
==19600== LEAK SUMMARY:
==19600==    definitely lost: 0 bytes in 0 blocks
==19600==    indirectly lost: 0 bytes in 0 blocks
==19600==      possibly lost: 288 bytes in 1 blocks
==19600==    still reachable: 78,846 bytes in 147 blocks
==19600==         suppressed: 0 bytes in 0 blocks
==19600== Reachable blocks (those to which a pointer was found) are not shown.
==19600== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==19600==
==19600== For counts of detected and suppressed errors, rerun with: -v
==19600== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 808 from 1)
```

A lo largo del desarrollo del proyecto, nos hemos encontrado con alguna dificultades:

Comprender el uso del método “rectangulo_lleno(izq, arr, der, abajo);”, ya que no sabíamos cómo calcular las variables derecha y abajo. A raíz de esto, nos surgió otro problema: la colisión entre una partícula móvil y una fija no era del todo correcta (ya que en parte atravesaba el rectángulo), resuelto al modificar el valor del factor que multiplica al umbral al comprobar si han chocado.

Otro de nuestros mayores problemas se basaron en el uso de la herramienta Valgrind qué, a pesar de que fueron escasos los errores, no sabíamos comprender el error que indicaba, ya que era nuestra primera vez usándola. Por ejemplo, tuvimos un error en el constructor por parámetros del simulador, ya que dimos por hecho que las variables “moviles” y “fija” (objetos de ConjuntoParticulas) existían; cuya forma correcta es:

```
Simulador::Simulador(const ConjuntoParticulas&conjMovil, const ConjuntoParticulas& conjFijo, const int& ventanaAncho, const int& ventanaAlto) : moviles(conjMovil), fijas(conjFijo){ … }
```

Como indicación, declaramos la variable particulaMovil (bool) para evitar hacer varios métodos que controlen el rebote, ya que no podíamos tratar el choque entre dos partículas móviles y el de una móvil con una fija de la misma manera; tratándolo todo en un mismo método.

Nota de autoevaluación: Notable.