Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Informe Tarea 2

Bomberman

Alumno: Joaquín Cruz
Profesor: Nancy Hitschfeld.
Auxiliares: Pablo Pizarro R.
Pablo Polanco G
Mauricio Araneda H.
Ayudantes: Joaquín Torres.
Maria Jose Trujillo.
Ayudante del laboratorio: Joakin Ugalde

Fecha de realización: 26 de mayo de 2018
Fecha de entrega: 26 de mayo de 2018
Santiago, Chile

Índice de Contenidos

Lista de Figuras

Lista de Tablas

Lista de Códigos

1. Introducción

En el presente informe se da a conocer la metodologia utilizada para generar solución a la tarea 2 del curso "CC3501 Modelación y Computación Grafica para Ingenieros". Esta trata sobre la creación de un juego mediante la utilización de los software de creación grafica en dos dimensiones, Opengl y Pygame. La opción escogida es el juego conocido como "Bomberman" que consiste en un laberinto donde hay que ir destruyendo ciertas paredes para encontrar la salida mediante el uso de bombas. Para esto se utilizó el Modelo Vista Controlador (MVC) visto en clases donde la escena debia contar con ciertos elementos que podemos ver ilustrados en la figura 1

Figura 1: Super Bomberman de Super Nintendo.

Este Bomberman creado, tiene el nombre Homeroman y su tematica es el dibujo animado de los Simpsons.

2. Diseño de la Solución

Para solucionar el problema del juego, se utilizó el Modelo Vista Controlador y el modulo CC3501Utils provisto en el material docente del curso. Todos los modelos diseñados heredan de la clase Figura del modulo CC3501Utils y son dibujados por OpenGL y no Pygame. Las componentes de cada escena son las siguientes:

2.1. Modelo

2.1.1. Player

Modelo que controla las personas, esta tiene los parámetros de posición, una coloración del estilo rgb (Red Green Blue), un número (para saber si es player 1 o player 2), un parámetro interno que representa su centro, una vida que es un parametro booleano y una coloración para su camisa. Los metodos que este tiene son:

- **figura:** Metodo que dibuja al modelo player.
- **updatecenter:** Metodo que va actualizando el centro de la figura al moverse.
- **moverx:** Metodo el cual permite que el jugador se mueva en direccion derecha o izquierda. Este recibe un 1 si se desea mover hacia la derecha o un -1 hacia la izquierda. Si se desea mover más, se tiene que ingresar un numero mayor.
- **movery:** Metodo el cual permite que el objeto se mueva en direccion arriba o abajo. Este recibe un 1 si se desea mover hacia arriba o un -1 si es hacia abajo. Si se quiere mover más, basta ingresar un numero mayor o menor que 1 y -1 respectivamente.
- **getpos:** Metodo que retorna la posición del jugador como una tupla (x, y).
- **getcenter:** Metodo que entrega el centro como una tupla en coordenadas cartesianas de la forma (x, y).
- **getlife:** Metodo que indica True si el el jugador esta vivo y False si esta muerto.
- **setlife:** Metodo que recibe un booleano y lo pone como el estado de vida del jugador.
- **normalizar_camisa:** Metodo que deja el color de la camisa del jugador como el original (ó en el estado de defecto).
- **setcoloracion:** Metodo que hace una coloracion en la camisa según la tupla ingresada.

Es fácil ver que la única transformación geometrica del modelo Player es la de traslación.

2.1.2. Bombas

Para hacer las bombas se creó la clase Bombs. En esta se hizo el modelo de las bombas.
Los parámetros que tiene són un vector posición, un tiempo de creacion, un booleano vivo que dice si vive o no y un color negro en rgb. Además internamente crea un centro de la figura y un cambio que es para animaci? Los metodos que tiene son:

- **figura:** Dibuja al modelo Bombs
- **setlife:** Recibe como parametro un booleano que será el nuevo estado de vida de la bomba.
- **gettime:** Dá el tiempo en que se creo la bomba.
- **getlife:** Dá el estado de vida de la bomba.
- **getcenter:** Dá el centro como coordenadas cartesianas (x, y).
- **getpos:** Dá la posicion de la bomba como coordenadas cartesianas (x, y).
- **Cambio_change:** Dá el cambio para la animación de la bomba.
- **getcambio:** Da el valor del cambio en la animación.
- **plustime:** Le suma tiempo a el de creación de la bomba.
- **settime:** Recibe un tiempo que será el de creación.

2.1.3. Explosión

Para generar las explosiones, se tuvo que crear el la clase Explosion. Esta recibia un tiempo, un vector posición y un color por referencia naranjo.
Además se implementaron los siguientes metodos:

- **figura:** Hace el dibujo de la explosión.
- **getcenter:** Entrega el centro como coordenadas cartesianas.
- **getlife:** Entrega el estado de vida de la explosion.
- **gettime:** Entrega el tiempo en la cúal fué creada.
- **setlife:** Recibe un booleno y se lo da como valor al atributo de la vida.
- **getpos:** Entrega la posicion de la explosion en pantalla como coordenada cartesiana.

2.1.4. Enemy

Para crear los enemigos, se creó el modelo y clase Enemy que recibia como parámetros de su creación un vector posicion, un entero que representa el tipo (de lo que se iba a dibujar), un estado de vida y una tupla de colores en rgb.
Los metodos de la clase són:

- **figura:** Dibuja las primitivas del objeto segun su tipo.
- **updatecenter:** Actualiza el centro del enemigo al cual esta parado.
- **moverx:** Recibe una direccion (1 o -1) segun el lado que se quiere trasladar el enemigo en horizontal.
- **getcenter:** Entrega una tupla que representa al vector centro en coordenadas cartesianas.
- **setlife:** Recibe y establece un nuevo estado de vida para el enemigo.
- **getlife:** Entrega True si el enemigo esta vivo, y False si no lo esta.

Notemos que los metodos de los enemigos son muy parecidos a los jugadores, luego se podría hacer una clase padre que pueda hacer heredar de ciertos metodos a los enemigos y los jugadores.

2.1.5. Power

Ahora, para crear los poderes se utiliza la clase Power. Lá particularidad es que todos los power ups vienen de esta clase. El funcionamiento es a partir del creador que recibe un vector posición y un string que indica el tipo de power up que se requiere. A partir de estos se genera un indice, un centro, una coloración y una duración. Los power ups posibles son los cinco siguientes:

- **Rango de Bombas:** Aumenta en un recuadro el rango de las bombas. Pinta de color amarillo la camisa o vestido del jugador. Dura 15 segundos.
- **Atravesar Paredes:** Power up que permite teletransportarse por las paredes del centro del escenario (tanto indestructibles como destructibles). Pinta de color rojo la camisa o vestido del jugador. Dura 10 segundos.
- **Atravesar Bombas:** Power up que permite teletransportarse por las bombas, si estan estan frente una pared, se teletransportara tambien por la pared. Pinta de color azul a la camisa o vestido del jugador. Dura 10 segundos.
- **Inmortalidad:** Permite que el jugador no muera. Esta pinta de color amarillo al jugador y dura 5 segundos.
- **Cadena de Explosiones:** Provoca que las bombas que esten en la explosión de otra bomba tambien exploten. Pinta la camisa o vestido de color negro y dura 15 segundos.

Los graficos de los power up cambian segun su tipo, que será visto en la sección 3.2.5.
Finalmente, los metodos implementados en la clase son los siguientes:

- **figura:** Es el metodo que dibuja en pantalla los power ups.

- **getcenter:**Metodo que entrega el centro de la figura como coordenadas en tupla cartesiana (x, y).
- **gettomado:** Retorna True sí el power up esta recogido por el jugador.
- **tomar:** Recibe el tiempo en que el power up fue recogido por el usuario. Luego setea el tiempo de recogido como el tiempo ingresado y pone el booleano de tomado como True.
- **setlife:** Tiene como parametro un booleano que es el nuevo estado de vida del power up.
- **getlife:** True si el power up esta vivo.
- **gettime:** Entrega el tiempo en el cual fue recogido el power up.
- **getduracion:** Entrega el tiempo de duración del tipo de power up escogido.
- **gettipo:** Entrega el tipo de la clase.
- **getindex:** Indica el indice de la lista de power ups activos que le corresponde a este tipo de power up.
- **getcoloracion:** Entrega la coloracion de la camisa de Homero (o Marge) que se debe utilizar al agarrar el power up.
- **plustime:** Extiende la duracion del power en t debido a las pausas del juego.

2.1.6. Pared

Las paredes son modeladas y dibujadas mediante la clase Pared. Esta tiene un constructor que recibe un vector posición y una coloración rgb. Internamente crea el centro de la figura como otro vector. Los metodos que tiene són:

- **figura:** Metodo que dibuja la pared en pantalla.
- **getpos:** Retorna la posición como una tupla (x, y) en coordenadas cartesianas
- **getcenter:** Hace retornar el centro de la figura como tupla en cartesianas.

2.1.7. Paredes Destructibles

A diferencia de la clase pared, los destructibles (o paredes destructibles) son generados en otra clase llamada PDes. Esta es casi analoga a la anterior, pero tiene un atributo booleano de vida. Además tiene los siguientes metodos extra (más con los de Pared).

- **getlife:** Se obtiene True si el estado de vida es verdadero (si esta en escena).
- **setlife:** Metodo que recibe un nuevo estado de vida para la bomba.

2.1.8. Ganar

La posición de victoria o de ganar esta modelada en la clase Win. Esta solo recibe un vector posición y una coloracion que por defecto es de color dorado medio amarillento, pero tambien se le crea internamente el Vector de centro. Además tiene solo dos metodo, el de figura que permite dibujarlo en pantalla, y el de **getcenter** que da el centro como coordenadas cartesianas en tupla (x, y).

2.1.9. Fondo

Finalmente, el último modelo es el fondo. Este es creado en la clase Fondo. El creador recibe un vector posición por referencia en el $(0, 0)$ y el color por referencia como el gris.
Su único metodo es el figura que lo dibuja por toda la ventana.

2.2. Vista

La vista es totalmente separada. Se hizo un modelo MVC más potente (separando todos los modelos, las vistas y los controladores). La vista esta en una clase Vista que tiene un solo metodo que es el dibujar que recibe la lista de todos los objetos a dibujar más el fondo y llama para cada objeto en las listas el metodo dibujar de la clase padre Figura. El codigo es el siguiente:

Código 1: Clase Vista

```
class Vista:
    def dibujar(self,power_ups,f,l_players,l_paredes,l_destructibes,l_bombas,l_enemigos,
    l_explosiones,l_win,ancho,alto):
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        if f!=None:
            f.dibujar()
        for w in l_win:
            w.dibujar()
        for pw in power_ups:
            pw.crear()
            pw.dibujar()
        for p in l_players:
            p.crear()
            p.dibujar()
        for pared in l_paredes:
            pared.dibujar()
        for des in l_destructibes:
            des.dibujar()
        for bomba in l_bombas:
            bomba.crear()
            bomba.dibujar()
        for enemigo in l_enemigos:
            enemigo.dibujar()
        for exp in l_explosiones:
            exp.dibujar()
```

2.3. Controlador

En el controlador se dividió en dos partes. Por una tenemos el controlador que lee las acciones del usuario del juego (o los usuarios) y sú planificación. Por otro lado, y para facilitar la programación, se hizo el modulo *BUtils.py* que tiene todas las funciones que generan las acciones de los modelos de la escena. Dentro están la funciones que generan a los personajes y objetos, movilizan,desaparecen, etc... Los distintos modelos, es decir, estan todas las mecanicas de interacciones, generación de estos y limpieza de memoria del juego.
Los comandos del usuario (y sus controles) són distintos según la cantidad de jugadores.
En modo de un jugador el controlador puede leer las siguientes instrucciones:

- **Flechas:** Cambia la direccion que apunta el jugador. Si su dirección ya era la que se ingresaba, este se mueve un espacio hacia esa dirección.
- **letra a:** Pone una bomba en la direccion en que apunta el usuario.

- **letra p:** Pone pausa al juego, se detiene la musica durante este periodo.
- **letra r:** Resume el juego si estaba en pausa, además pone la musica denuevo en donde se quedó.

En todo esto, el string que dice la dirección del usuario es una variable dentro del main. Todas las variables que involucren al jugador se duplican en modo de dos jugadores (excepto la dificultad), cuyos controles són, para el primer jugador, los mismo que en el modo de un jugador exceptuando el poner bombas, que ahora se hará con la letra "Backspace" del teclado. El segundo jugador tiene los siguientes controles:

- **letras w:** Hace que el segundo jugador apunte hacia arriba o se mueva hacia arriba.
- **letra a:** Hace que el segundo jugador apunte hacia la izquierda o se mueva hacia esa dirección.
- **letras s:** Hace que el segundo jugador apunte hacia abajo o se mueva hacia abajo.
- **letra a:** Hace que el segundo jugador apunte hacia la derecha o se mueva hacia la derecha.
- **letra b:** Hace que el segundo jugador ponga una bomba en la dirección actual.

Finalmente, la forma en que el controlador sabe cuantos niveles debe jugar, en cúal dificultad quiere jugar el usuario, sí debe cambiar la dificultad y cada cuantos niveles debe hacerlo se hace mediante el uso de contadores y preguntas en consola al usuario.

3. Implementación

3.1. Planificación

Lo primero que se hizó fue una planificación por dia de lo que se debia llegar a hacer. Para esto se tienen los siguientes objetivos a lograr por dia:

- **11/05/18**:
  - Generar el laberinto de la escena con paredes solamente indestructibles.
  - Generar un jugador que se pueda mover.
  - Hacer que ese jugador choque con las paredes.
  - Hacer que pueda poner una bomba.
  - Hacer que el jugador choque con las bombas.
  - Hacer que la bomba desaparezca despues de tres segundos
  - Generar 4 enemigos en ubicaciones fijas y que sean estaticos.
- **12/05/18**:
  - Generar explosiones.
  - Hacer que las explosiones hagan su efecto de desaparecer a los enemigos.
  - Hacer las paredes destructibles.
  - Hacer que el jugador no atraviese las paredes destructibles.
  - Hacer que las explosiones destruyan las paredes destructibles.
  - Hacer el efecto de perdida al tocar enemigos y explosiones.
- **13/05/18-14/05/18**:
  - Generar movimiento aleatorio de los enemigos.
  - Generar posiciones aleatorias de los enemigos y de las paredes destructibles.
  - Generar la condicion de victoria.
  - Generar la victoria dentro debajo de un destructible al azar.
- **15/05/18-16/05/18**:
  - Mejora de graficos a las paredes indestructibles.
  - Mejora de graficos a las paredes destructibles.
  - Mejora de graficos al jugador.
  - Mejora de graficos a la posicion de ganar.
- **19/05/18**:
  - Implementación del modelo Power.

- **20/05/18**:
  - Mecanicas del modelo Power.
  - Arreglar los bugs que los power ups producian.
  - Mejorar graficos de los power ups.
  - Generar la mecanica del color de la camisa.
  - Generar al azar los power ups en escena.
  - **Añadir extra:** A partir de la aleatoriedad, añadir los niveles de dificultad.
- **21/05/18**:
  - Mejora de graficos generales (Enemigos+Player+Posición de ganar).
  - Terminar lo minimo del juego.
- **22/02/18**
  - **Añadir extra:** Power up nuevo.
  - **Añadir extra:** Multiplayer (Dibujo de Marge).
  - **Añadir extra:** Sonido.
  - **Añadir extra:** Más niveles, cuantos uno quiera jugar.

3.2. Proceso de realización

Dada la planificación anterior, podemos notar como el enfoque fue primero en hacer un juego funcional y luego generar graficos "bonitos". Es entonces que en el primer dia se discretizó la escena de la figura 1 para generar los movimientos (que són discretos) y las figuras correspondientes.
Para esto se hace lo siguiente, primero generemos una ventana de 800×600 (tamaño tipico 2:3). Notemos que la escena de la figura 1 muestra un laberinto de medidas 13×15, entonces la discretización de nuestra ventana viene dado por:

$$\begin{aligned} 15 \cdot \textit{medida del ancho} = 800 \Rightarrow \textit{medida del ancho} = 53{,}0 \\ 13 \cdot \textit{medida del alto} = 600 \Rightarrow \textit{medida del alto} = 46{,}0 \end{aligned} \tag{1}$$

A partir de esta discretización, podemos generar las paredes indestructibles nombradas en el modelo Pared. Estas en un principio eran cuadrados planos amarillos. En general, para utilizar las mecanicas del juego,las funciones estan en el script *BUtils.py*.

3.2.1. Mecanicas para los jugadores y bombas

Al principio para generar el choque con las paredes y bombas con el jugador se implementó el la función *chocaPared()* y *chocaBomba()*. La primera recibia un jugador, la lista de paredes, la lista de paredes destructibles, un string con la dirección y la discretizacion en x y en y. La segunda por su parte recibia un jugador, la lista de bombas, el string con la dirección y las discretizaciones. Ambas operan de la siguiente manera:

1. Primero genero un jugador auxiliar.
2. Ahora dependiendo de la dirección ingresada, muevo ese jugador auxiliar.
3. Extraigo el centro del jugador.
4. Luego se va revisando en cada lista, extraigo el centro del objeto. Si los centros estan juntos (el modulo de su resta es menor que cierta distancia). Se escogio un epsilon de 20 pixeles en cada dirección.
5. Si lo anterior es afirmativo, entonces retorno True debido a que va a chocar con una pared o bomba en esa dirección. Si no sucede nada, retorno False.

De esta forma logramos que el jugador se mueva detectando paredes y bombas sin poder atravesarlas.

Para poner una bomba, se utilizaron las mismas funciones para detectar paredes y bombas anteriormente nombradas. Es decir, si el jugador esta frente a una pared o otra bomba y apreta la tecla ”a”, este no hará nada. En cambio, si la puede poner, se implementó la función *ponerBomba()* que recibia como parametros una lista de bombas, un jugador, la direccion y el tiempo en que se hizo. Esta funciona de la siguiente manera:

1. Primero se toma la posición actual del jugador.
2. Luego según la dirección establecida, generó un objeto Bombs en la posición siguiente y con tiempo inicial el ingresado. Este se pone en la lista de bombas.

A partir de esto, se tiene que hacer su efecto de desaparición y aparac'on de una explosion.
Para lograrlo se utiliza que en cada iteración se ejecuta la función *explosion_bombas()* que recibe la lista de explosiones, la lista de paredes, la lista de bombas, un booleano llamado rango que representa si el power up esta activo o no, el tiempo actual del proceso y un sonido. Funcionando de la siguiente manera:

1. Primero se establecian las discretizaciones en x e y.
2. Para cada bomba en el arreglo ingresado, se extraia su tiempo de aparicion y se sacaba la diferencia.
3. Si esta diferencia es mayor que 3 segundos, se ejecuta el sonido y se extrae el centro de la bomba.
4. A partir del centro, se crea una explosión inicial en la misma posición de la bomba y cuatro explosiones momentaneas contiguas a esta (arriba, abajo, derecha e izquierda) en conjunto con cuatro booleanos iniciado en True que indicarán si se pueden poner o no.
5. Luego, cada pared en la lista de las paredes se ve si alguna explosion momentanea choca con la pared, si lo hace se pone su booleano en False.
6. Finalmente, se añaden a la lista de explosiones las que tengan sú booleano en True.
7. Además, se ve si esta activo el power up de rango ingresado (es decir, si es verdadero). Si lo es, se hace un proceso analogo en las nuevas explosiones. Si no, termina el proceso.

3.2.2. Mecanicas de enemigos y explosiones

A partir de la generación de explosiones generado a partir de la mecanica de explosión de las bombas, estas tenian el efecto de cadena de explosiones si se tenia el power up correspondiente a este y el de explotar cosas si las tocaba.
Para explotar cosas y hacerlas desaparecer de la escena, se implementó la función *explotar()* que recibia como parametros la lista de objetos que debiamos explotar (bombas, enemigos, jugadores) y la lista de explosiones activas. Esta función actuaba de la siguiente manera:

1. Para cada objeto en su lista y se veia en cada explosión si estos chocaban.
2. Si lo hacian, se hacia que la vida del objeto sea falsa.

Por otra parte, si el power up de la cadena de explosiones estaba activo, se activaba otra función que hacia su efecto. La función se llama *cadena_explosiones()* que recibe como parametros la lista de explosiones activas, la lista de bombas puestas, el sonido de las bombas, la lista de paredes indestructibles, un booleano de rango que representa si el power up de mayor rango activo o no y el tiempo actual. La función procede con el siguiente algoritmo:

1. Para cada explosión, se ve si estas chocan con alguna bomba.
2. Si lo hacen, se pone el tiempo de la bomba (de aparicion) en 4 segundos
3. Finalmente se llama a la función explosión bombas para generar las explosiones correspondientes.

Para los movimientos y efectos de los enemigos se implementarón mecanicas basadas en probabilidades y aleatoriedad (movimiento y generación) y choques entre el player con estos (efectos). Empezemos describiendo la primera, para esto primero situamos (preguntando por consola al usuario) el nivel de dificultad asociado a lo que quiera jugar el jugador. Luego a partir de este generaremos cierta cantidad de enemigos comprendido entre un minimo y un maximo. Estos estan dado según la siguiente tabla y son al azar:

Tabla 1: Rango de enemigos generados por nivel de dificultad
(*)Para modo multiplayer

Nivel de Dificultad	**Número minimo de enemigos**	**Número maximo de enemigos**
facil	4	4
medio	7	10
dificil	10	13
extremo	15 o 14*	15 o 14*

Luego, se tiene que el computador puede elegir entre cualquiera de sus dos diseños al azar escogiendo un numero entre 1 y 2. Si el número escogido es el 1, se produce la figura de los graficos 8 y si es 2 sé produce la figura 9. El proceso de generación de enemigos está indicado en la subsección 3.2.4.

Además de su generación, el movimiento de los enemigos tiene una probabilidad de ocurrencia asociada al nivel escogido. Esta viene dado por la siguiente tabla:

Tabla 2: Probabilidad de movimiento de los enemigos según nivel de dificultad

Nivel de dificultad	**Probabilidad de que se mueva**
facil	4 %
medio	8 %
dificil	16 %
extremo	80 %

Luego, se tiene que estos se mueven mediante el uso de la función en *BUtils,moverEnemigo()* que recibe la lista de enemigos en escena, la lista de paredes, la lista de paredes destructibles, la lista de bombas en escena, el string de dificultad y las discritizaciones en x e y.
Esta funciona de la siguiente manera:

1. Primero se fija un numero dependiendo del nivel de dificultad. Si este es facil, ese numero vale 100. Si es medio vale 50. Al nivel dificil le corresponde el 25 y finalmente si la dificutad es extrema es 5.

2. Luego se genera una lista con los strings de las cuatro posibilidades de movimiento.

3. Entonces, para cada enemigo activo, se genera una lista auxiliar con los otros y un numero al azar entre 0 y el numero mencionado en (1).

4. Si este numero es menor que 4, este saca una direccion de la lista de strings con las direcciones.

5. El enemigo ve si se puede mover hacia esa posición de forma analoga a como la hacia el jugador ahora que tambien ve que no haya otro enemigo en frente.

6. Sí la respuesta es afirmativa, entonces se mueve. Si no, no lo hace.

Finalmente, para ver si un jugador choca con un enemigo, basta ver que en cada iteración el choque entre los elementos de sus listas correspondientes.

3.2.3. Mecanicas y Funcionamiento de Power Ups

El funcionamiento de los power ups tienen cuatro ambitos, su recolección, su efecto, su duración y los cambios en los graficos de los jugadores. Cada una tiene sus funciones respectivas en BUtils excepto su efecto en el juego.

Primero, los efectos en juego son controlados en el main, es decir, en la captura de eventos. Aqui para cada jugador se provee una lista de cinco booleanos que inicialmente estan todos falsos. El primero es para indicar si el power up de rango esta activo, el segundo para el power up de atravesar paredes, el tercero para atravesar bombas, el cuarto para la inmortalidad y el quinto para la cadena de explosiones. Es por esta razón que la clase Power genera, según el tipo, un indice.
Este indice indica booleano es en la lista anterior.

En segundo lugar la recolección de los power ups esta manejada en la función *obtener_pwup()* que recibe un jugador, una lista de power ups, una lista de power ups obtenidos, los power ups activos, el tiempo actual y un sonido (que por referencia es None). Entonces, a grandes rasgos, lo que hace la funcíon es ver si un power up choca con el player. Si lo hace, se dibuja en una posición fuera de la ventana de clipping (para no ser tomado dos veces) y su indice se pone como verdadero. Ademas se setea el tiempo de obtencion y con cierta probabilidad se escucha el sonido de obtencion (es una frase de homero simpson).

El cambio en los personajes es regulado por la función pwup_color() que recibe el jugador y la lista de power ups activos y la general. Esta genera que mientras un power up esté activo, se le cambie el color. Y cuando no haya ninguno activo, se vuelva al color original de la camisa o vestido.

Finalmente, su duración funciona de manera analoga a la de las bombas en juego. Es decir que si la diferencia entre el tiempo de obtencion y tiempo actual de un power up activo es mayor que su duración, su vida es False, haciendo que despues en la limpieza, se borre de la memoria.

3.2.4. Generación de niveles y limpieza de memoria

Para la generación de niveles se hicieron dos procesos, el nivel inicial en cual él o los usuarios entran y la generación de los niveles posteriores.
Los procesos son analogos y funcionan de la siguiente manera:

1. Primero se utiliza la función init del modulo *CC3501Utils.py* para iniciar pygame y OpenGL

2. Luego, pasa al proceso de carga de archivos de audio (solo en el primer nivel)

3. Ahora bien, se crea el objeto Fondo que dibuja el fondo de la escena.

4. Posteriormente se crean las paredes indestructibles, estas se guardan en un arreglo de paredes y otro de bordes (que sirve para no salirse de la escena al utilizar power ups)

5. Se crean la listas de memoria de los objetos a usar, tales como destructibles, power ups, power ups obtenidos y activos, bombas, jugadores y enemigos.
6. Despues, se crean el (o los dos) jugador(es).
7. A partir de lo anterior, lo primero que se crean son los destructibles mediante la función creaciondestructibles.
8. Más adelante, se crean los enemigos mediante la función creacionenemigos.
9. Despúes de eso, se genera la posición de victoria en algun destructible.
10. Finalmente se crean dos power ups en escena.

La creación de niveles posteriores es la misma logica desde el paso (5) peró sin los jugadores debido a que ellos ya existen.
Ahora bien, lo natural es saber como operan las funciones de creacion y estas son de manera muy similar:

1. Se identifica que objetos hay en escena, luego se crea un objeto pedido con posiciones al azar.
2. Si el objeto calza con algun objeto que no pueden estar al mismo tiempo ahi (jugador-enemigo, jugador-destructible, destructible-pared, enemigo-pared,enemigo-destructible, power up-jugador, power up-pared), se repite el proceso.
3. Este proceso acaba hasta encontrar la cantidad pedida de enemigos y destructibles en escena pedidos que son al azar.

Nota: Los power ups son generados de a uno y pueden estar bajo un destructible.

La posición de victoria toma la posición de un objeto destructible creado y se dibuja antes que este. Esto se produce con un destructible aleatorio generado con anterioridad.
Por otro lado, la cantidad de paredes destructibles a generar depende del nivel de dificultad al inicio, este se puede ver mediante un número minimo y maximo a generar. La cantidad es reflejada por la siguiente tabla:

Tabla 3: Cantidades posibles de Muros destructibles según nivel de dificultad

Dificultad escogida	**Cantidad Minima**	**Cantidad Maxima**
facil	5	20
medio	15	25
dificil	25	28
extremo	30	30

En la limpieza de memoria se usa una variación de la función limpieza en el utils de la implementación del juego ”Earth Defender” visto en clases de catedra. La función se ubica en el modulo BUtils y recibe una lista de objetos y sobre ella actua de la siguiente manera:

1. Parte tomando el tamaño del arreglo y crea otro auxiliar vacio inicialmente.

2. Luego iterando sobre el largo del arreglo siempre se hace un pop del primer elemento. Si este elemento tiene una vida en estado True se agrega al auxiliar.

3. Al finalizar el proceso anterior tenemos que el arreglo original esta vacio, entonces para cada elemento en el auxiliar, estos se añaden al original.

4. De esta forma generamos que el arreglo ingresado ahora solamente tenga cosas que estan vivas, es decir, que sean utiles para el juego.

3.2.5. Graficos del Modelos

Todos los graficos fueron hechos a partir de primitivas de OpenGL y no imagenes importadas de internet aunque si fueron dibujadas a partir de figuras encontradas en una imagen, es decir, se dibujaron a mano a partir de esta.

Fondo: Para el fondo se utilizó el siguiente dibujo:

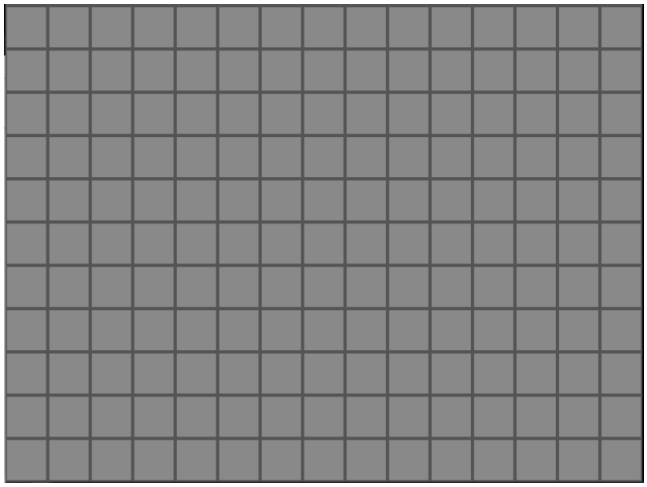

Figura 2: Fondo del juego.

Pared: Para la pared, el grafico fue una repeticion del siguiente patron:

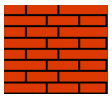

Figura 3: Pared Indestructible

Entonces el laberinto es visto de la siguiente forma:

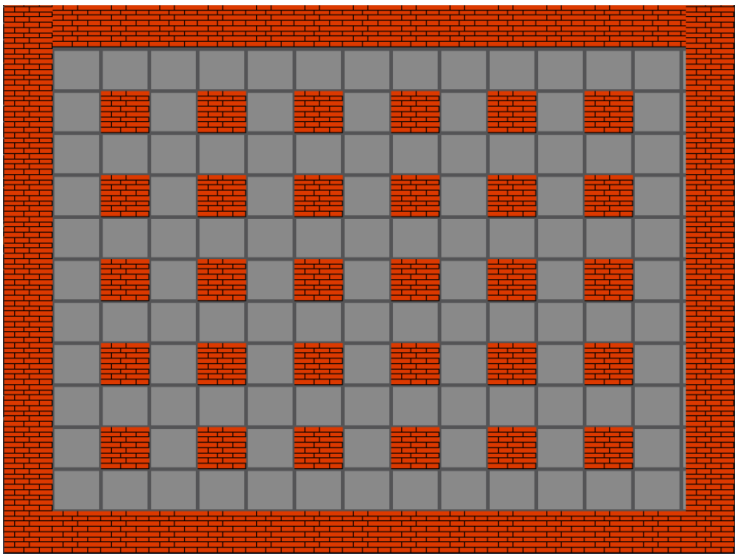

Figura 4: Laberinto base del juego

Destructibles: Para los destructibles se utilizó la siguiente representación:

Figura 5: Destructibles o Paredes destructibles

Player 1: El grafico del primer jugador es el de Homero Simpson, que tiene el siguiente dibujo:

Figura 6: Homero Simpson, Jugador 1

Player 2: El grafico que representa al segundo jugador es el de Marge Simpson, que fue dibujada con la forma:

Figura 7: Marge Simpson, Jugador 2

Enemigos: Hay dos dibujos para los enemigos que son escogidos al azar mediante el proceso descrito anteriormente.Su representación es Rafa Gorgori y Ned Flanders cuyos dibujos son respectivamente:

Figura 8: Rafa Gorgori, primer modelo para enemigo.

Figura 9: Ned Flanders, segundo modelo para enemigo.

Bombas: Las bombas tienen el siguiente dibujo:

Figura 10: Dibujo de las bombas

Explosiones: Para las explosiones se utilizó el siguiente dibujo:

Figura 11: Explosiones en el juego

Victoria: Dada la tematica del juego, la salida o posición de victoria esta dada por una cerveza. Su dibujo es:

Figura 12: Dibujo del espacio de victoria

Power Ups: Todos los Power Ups son una dona excepto que tienen cinco graficos distintos que son los distintos frosting. Estos son:

Figura 13: Power up rango de explosiones

Figura 14: Power up Atravesar Paredes.

Figura 15: Power up Atravesar Bombas

Figura 16: Power up Inmortalidad

Figura 17: Power up Cadena de Explosiones

No se hicieron animaciones, excepto para las explosiones. La animacion de explosión es la siguiente secuencia:

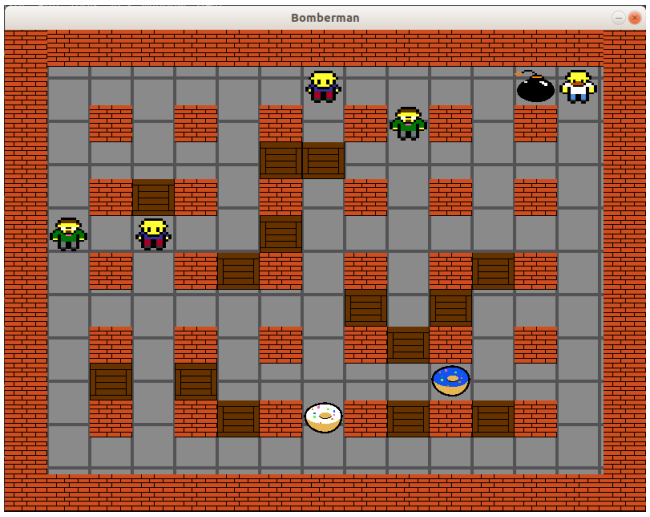

Figura 18: Proceso de poner una bomba

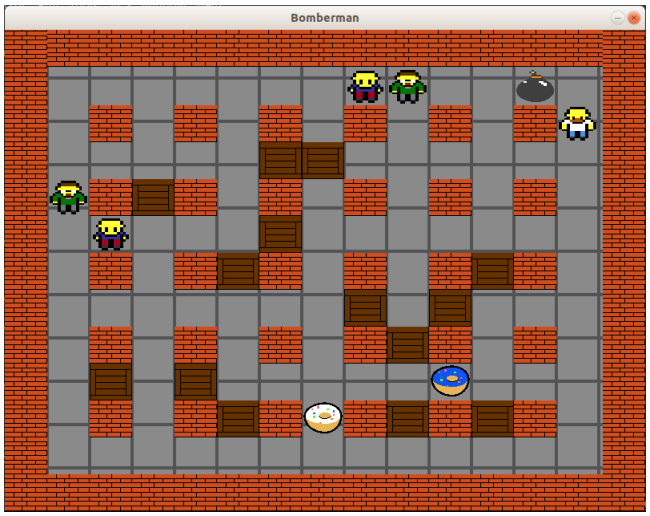

Figura 19: Bomba en menos de un segundo de explotar

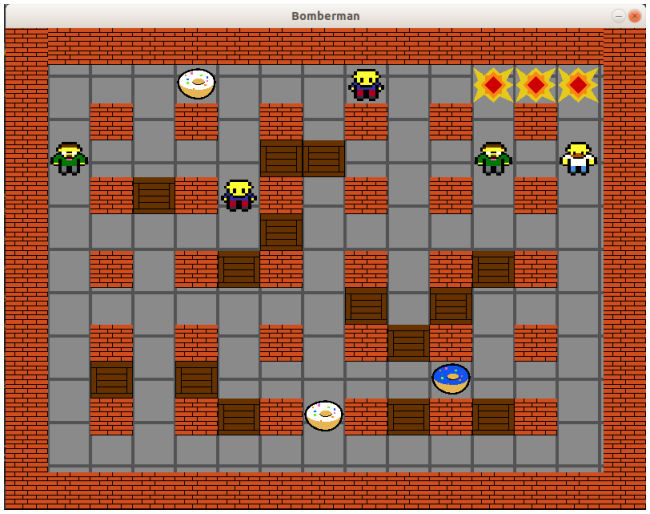

Figura 20: Explosion de la bomba

3.3. Extras

Los extras implementados son los niveles de dificultad ya mencionados, la mayor cantidad de power ups que los necesarios, una mayor cantidad de niveles (se puede jugar hasta el mayor entero posible en un computador que es 2.147.483.647 de niveles), la inclusión de dos jugadores y la musica. La musica es una version de 8 bits de la cancion ”Mr. Blue Sky” del grupo ”Electric Light Orchestra”. Los sonidos emitidos son archivos .wav de ciertas escenas de los simpson o ciertas frases tipicas que están en ingles.

4. Dificultades Encontradas

Durante el transcurso del programa, se encontraron varias dificultades solucionables y otras que no pudieron ser solucionadas.

1. Al concentrarse solamente en las mecanicas primero y luego los dibujos, algunos dibujos provocaron muchos bugs que fueron solucionados (traspazar paredes fue lo más recurrente que pasaba).

2. Se confió mucho del modulo *CC3501Utils.py* que se dio en material docente. Este principalmente trabajaba con OpenGL (que desde el principio siempre resultó comodo para trabajar dado a lo que habiamos visto en el curso hasta esos momentos).Esto produjo que más avanzado en el proyecto se quizo hacer sprites o cosas más sofisticadas pero OpenGL no era la forma más facil de hacerlo sino que pygame proveia funciones más comodas. Estas no eran compatibles con OpenGL y el proyecto estaba muy avanzado para hacer todo en pygame, dado que se tenia que cambiar todo. Una de las cosas que se pensó fue el uso de sprites y texturas, además de puntajes en pantalla o animaciones de movimiento.

3. En un momento determinado se pensó y construyo en un power up que escogia entre los existentes un efecto al azar. El power up no operaba de la manera esperada y provocaba dos bugs que no se pudieron corregir. El primero era que la camisa nunca volvia a su color original y el segundo era que te volvias inmortal para siempre a pesar de que escogia otro poder. Es así como la mejor solución a este problema fue finalmente sacarlo del juego.

4. La descritización del movimiento es segun recuadro, lo cual para hacer una pseudo animacion no se encontró la forma de hacerlo pero si como comenzarla (cambiar el paso de los avanzes del personaje). Esto provocó que al momento de intentar hacer algo parecido, se produjieron muchos bugs con lo que ya se tenia hasta ese momento.

5. El proceso de generación en niveles de dificultad dificiles puede llegar a durar mucho tiempo, entonces es posible que al abrir el juego no se genere nada. Este fue corregido bajando la cantidad de cosas a generar pero aveces sigue ocurriendo debido a que aun existe la probabilidad de que ciertos objetos tenga las mismas propiedades y no sean compatibles,entonces solamente se redujo a ocurrencias raras de estás situaciones.

6. En el modo de dos jugadores, como el juego no esta optimizado, se puede llegar a saturar a partir de la cantidad de eventos que toma Pygame.

7. El no poder mantener una tecla para los movimientos de los personajes, aun no se sabe como se hace.

5. Aprendizajes Obtenidos y Conclusiones Finales

En conclusión, se aprendió la practica de generar juegos en dos dimensiones, de su funcionamiento mediante el modulo de pygame y el uso de OpenGL. Además, esto me llevó a investigar ciertos usos de los programas y de como planificar las cosas para que ocurran a la primera vez teniendo resultados esperados en el juego "Homeroman". Por otro lado, se pudo poner en practica el uso de probabilidades de eventos dentro del sistema Lo que no se pudo generar fué el uso de joysticks dado el tiempo de planificación y la falta de uno, no podia ver si lo que podia implementar estaba bien o mal.
Finalmente y para un futuro, se debe utilizar en su mayoria Pygame más que OpenGL debido a las funciones que esté provee son más sencillas de utilizar que las de OpenGL.
Un caso interesante a implementar seria un menú que dijera todo lo que la consola pregunta en un inicio, animaciones y una mejor descritización de la pantalla, ademas de los extra que no se pudieron implementar y la optimización del funcionamiento de esté.

Anexo A. Codigos

Código A.1: Clase Figura

```
# Clase generica para crear figuras de openGL con posicion y color.
# se usa self.dibujar() para dibujar en la escena.
# self.figura define las primitivas que tiene.
class Figura:
    def __init__(self, pos: Vector, rgb=(1.0, 1.0, 1.0)):
        self.pos = pos
        self.color = rgb
        self.lista = 0
        self.crear()

    def crear(self):
        self.lista = glGenLists(1)
        glNewList(self.lista, GL_COMPILE)

        self.figura()

        glEndList()

    def dibujar(self):
        glPushMatrix()

        glColor3fv(self.color)
        glTranslatef(self.pos.x, self.pos.y, 0.0)
        glCallList(self.lista)

        glPopMatrix()

    def figura(self):
        pass
```

Código A.2: Función chocaPared

```
def chocaPared(p,l_pared,l_destructibles,direccion,dx,dy):
    """
    True si esta chocando una pared
    :param p: Player or Enemy
    :param l_pared: List (listas de paredes)
    :param direccion: str
    :param dx: float
    :param dy: float
    :return: boolean
    """
    (x,y)=p.getcenter()
    epsilon=20
    if direccion=="arriba":
        y+=dy
    if direccion=="abajo":
```

```
        y-=dy
    if direccion=="izquierda":
        x-=dx
    if direccion=="derecha":
        x+=dx
    for j in l_pared:
        (z,w)=j.getcenter()
        if abs(z-x)<epsilon and abs(w-y)<epsilon:
            return True
    for j in l_destructibles:
        (z,w)=j.getcenter()
        if abs(z-x)<epsilon and abs(w-y)<epsilon:
            return True
    return False
```

Código A.3: Función chocaBomba

```
def chocaBomba(p,l_bombas,direccion,dx,dy):
    """
    Ve si el jugador o enemigo va a chocar con una bomba (o apunta hacia ella)
    :param p: Player or Enemy
    :param l_bombas: list
    :param direccion: str
    :param dx: num
    :param dy: num
    :return: boolean
    """
    (x,y)=p.getcenter()
    epsilon=20
    if direccion=="arriba":
        y+=dy
    if direccion=="izquierda":
        x-=dx
    if direccion=="abajo":
        y-=dy
    if direccion=="derecha":
        x+=dx
    for bomba in l_bombas:
        (z,w)=bomba.getcenter()
        if abs(z-x)<epsilon and abs(w-y)<epsilon:
            return True
    return False
```

Código A.4: Función ponerBomba

```
def ponerBomba(l_bombas,jugador,direccion,t):
    """
    pone una bomba en la direccion en que se mira
    :param l_bombas: List
    :param jugador: Player
    :param direccion: str
    :param t: float
```

```
    :return: none
    """
    (px,py)=jugador.getpos()
    if direccion=="derecha":
        l_bombas.append(Bombs(Vector(px+53.0,py),t))
    if direccion=="izquierda":
        l_bombas.append(Bombs(Vector(px-53.0,py),t))
    if direccion=="arriba":
        l_bombas.append(Bombs(Vector(px,py+46.0),t))
    if direccion=="abajo":
        l_bombas.append(Bombs(Vector(px,py-46.0),t))
```

Código A.5: Función explosion_bombas

```
def explosion_bombas(l_explosiones,l_bombas,l_paredes,rango,t_a,sonido=None):
    """
    Ve que bombas deben explotar y genera las explosiones correspondientes
    :param l_explosiones: list
    :param l_bombas: list
    :param l_paredes: list
    :param rango: boolean
    :param t_a: num
    :param sonido: wav or None
    :return: None
    """
    dx=53.0
    dy=46.0
    for bomba in l_bombas:
        t0=bomba.gettime()
        dt=t_a-t0
        if dt>=3000.0:
            if sonido!=None:
                pygame.mixer.Sound.play(sonido)
            (x,y)=bomba.getcenter()
            xp=x-dx/2
            yp=y-dy/2
            e=Explosion(t_a,Vector(xp,yp))
            (e_arriba,ar)=(Explosion(t_a,Vector(xp,yp+dy)),True)
            (e_izq,iz)=(Explosion(t_a,Vector(xp-dx,yp)),True)
            (e_abajo,aba)=(Explosion(t_a,Vector(xp,yp-dy)),True)
            (e_der,der)=(Explosion(t_a,Vector(xp+dx,yp)),True)
            for pared in l_paredes:
                if chocar(e_arriba,pared):
                    ar=False
                if chocar(e_izq,pared):
                    iz=False
                if chocar(e_abajo,pared):
                    aba=False
                if chocar(e_der,pared):
                    der=False
            l_explosiones.append(e)
            if ar:
```

```
                l_explosiones.append(e_arriba)
            if iz:
                l_explosiones.append(e_izq)
            if aba:
                l_explosiones.append(e_abajo)
            if der:
                l_explosiones.append(e_der)
            #si tengo el power up:
            if rango:
                if ar:
                    (p_arriba,ar2)=(Explosion(t_a,Vector(xp,yp+2*dy)),True)
                if iz:
                    (p_izq,iz2)=(Explosion(t_a,Vector(xp-2*dx,yp)),True)
                if aba:
                    (p_aba,aba2)=(Explosion(t_a,Vector(xp,yp-2*dy)),True)
                if der:
                    (p_der,der2)=(Explosion(t_a,Vector(xp+2*dx,yp)),True)
                for pared in l_paredes:
                    if ar:
                        if chocar(p_arriba,pared):
                            ar2 = False
                    if iz:
                        if chocar(p_izq, pared):
                            iz2 = False
                    if aba:
                        if chocar(p_aba, pared):
                            aba2 = False
                    if der:
                        if chocar(p_der, pared):
                            der2 = False
                if ar:
                    if ar2:
                        l_explosiones.append(p_arriba)
                if iz:
                    if iz2:
                        l_explosiones.append(p_izq)
                if aba:
                    if aba2:
                        l_explosiones.append(p_aba)
                if der:
                    if der2:
                        l_explosiones.append(p_der)
            bomba.setlife(False)

        if dt>=1000.0 and not bomba.getcambio():
            bomba.color=(63.0/255,63.0/255,63.0/255)
            bomba.Cambio_change()
```

Código A.6: Función explotar

```
def explotar(l_objetos,l_exp):
    """
```

```
    Ve que objetos son alcanzados por la explosion y los mata
    :param l_objetos: list
    :param l_exp: list
    :return: None
    """
    for obj in l_objetos:
        for exp in l_exp:
            if chocar(obj,exp):
                obj.setlife(False)
```

Código A.7: Función cadena_explosiones

```
def cadena_explosiones(l_explosiones,l_bombas,sonido,l_paredes,rango,t_a):
    """
    Ve si hay que hacer una cadena de explosiones con el power up
    :param l_explosiones: list
    :param l_bombas: list
    :param sonido: wav or None
    :param l_paredes: list
    :param rango:boolean
    :param t_a: num
    :return: boolean
    """
    respuesta=False
    for explosion in l_explosiones:
        for bomba in l_bombas:
            if chocar(explosion,bomba):
                bomba.settime(4000)
                respuesta=True
    explosion_bombas(l_explosiones,l_bombas,l_paredes,rango,t_a,sonido)
    return respuesta
```

Código A.8: Función moverEnemigo

```
def moverEnemigo(l_enemigos,l_pared,l_destructibles,l_bombas,dificultad,dx,dy):
    """
    Mueve los enemigos a posiciones adyacentes aleatorias
    :param enemigo: Enemy
    :param l_pared: list
    :param l_destructibles: list
    :param dx: float
    :param dy: float
    :return: none
    """
    n=100
    if dificultad=="medio":
        n=50
    if dificultad=="dificil":
        n=25
    if dificultad=="extremo":
        n=5
    direcciones = ["arriba", "abajo", "izquierda", "derecha"]
```

```
    for enemigo in l_enemigos:
        aux=[]
        for e in l_enemigos:
            if e!=enemigo:
                aux.append(e)
        j=rand.randint(0,n)
        if j<4:
            dire=direcciones[j]
            if not chocaPared(enemigo,l_pared,l_destructibles,dire,dx,dy) and not
    chocaBomba(enemigo,l_bombas,dire,dx,dy) and not frenteEnemigo(enemigo,aux,dire,dx,dy)
    :
                if dire=="arriba":
                    enemigo.movery(1)
                if dire=="abajo":
                    enemigo.movery(-1)
                if dire=="izquierda":
                    enemigo.moverx(-1)
                if dire=="derecha":
                    enemigo.moverx(1)
```

Código A.9: Función obtener_pwup

```
def obtener_pwup(p: Player,l_power_ups,l_obtenidos,l_activados,t_a,sound=None):
    """
    Mecanica para obtener un power up
    :param p: Player
    :param l_power_ups: list
    :param l_obtenidos: list
    :param l_activados: list
    :param t_a: num
    :param sound: wav or None
    :return: None
    """
    prob=rand.randint(0,100)
    for pw in l_power_ups:
        if chocar(p,pw):
            if prob <= 40 and sound != None:
                pygame.mixer.Sound.play(sound)
            pw.tomar(t_a)
            l_obtenidos.append(pw)
            i=pw.getindex()
            l_activados[i]=True
```

Código A.10: Función pwup_color

```
def pwup_color(p: Player,l_power_ups,l_activados):
    """
    Cambia el color de la camisa o el vestido
    :param p: Player
    :param l_power_ups: list
    :param l_activados: list
    :return: None
```

```
    """
    for pw in l_power_ups:
        i=pw.getindex()
        if l_activados[i]:
            (r,g,b)=pw.getcoloracion()
            p.setcoloracion((r,g,b))
    inactivos=True
    for efecto in l_activados:
        if efecto:
            inactivos=False
    if inactivos:
        p.normalizar_camisa()
```

Código A.11: Función limpiar

```
def limpiar(arr):
    """
    Borra los elementos que no estan en escena (o no estan vivos)
    :param arr: List
    :return: none
    """
    n=len(arr)
    aux=[]
    for i in range(n):
        elemento=arr.pop(0)
        if elemento.getlife():
            aux.append(elemento)
    for b in aux:
        arr.append(b)
```

Anexo B. Versiones del juego

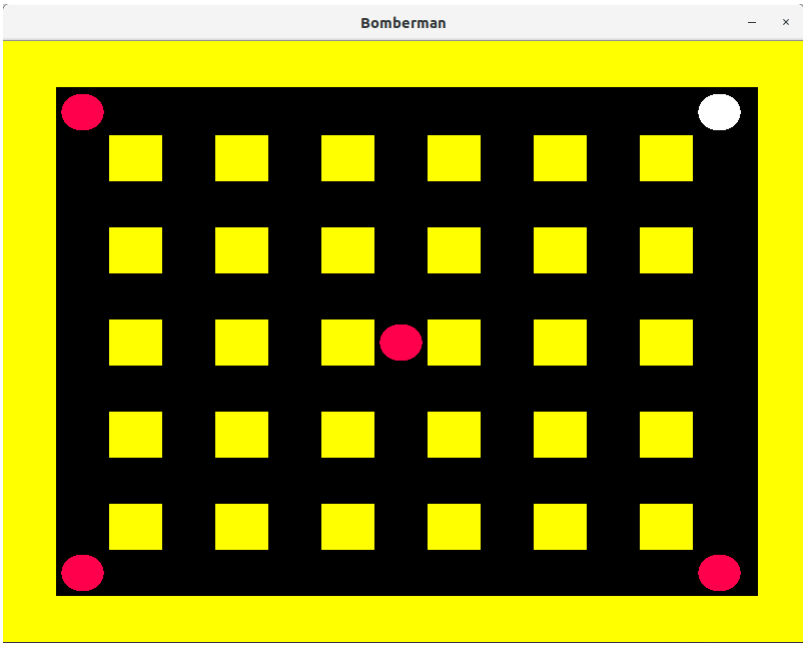

Figura B.1: Primera Versión del juego. Hecha el **11/05/18**

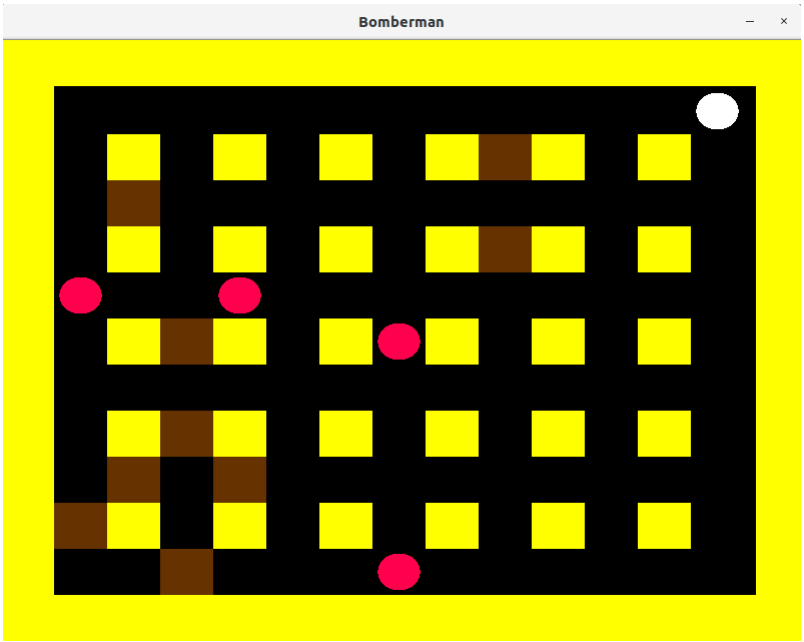

Figura B.2: Segunda Versión del juego. Hecha el **12/05/18**

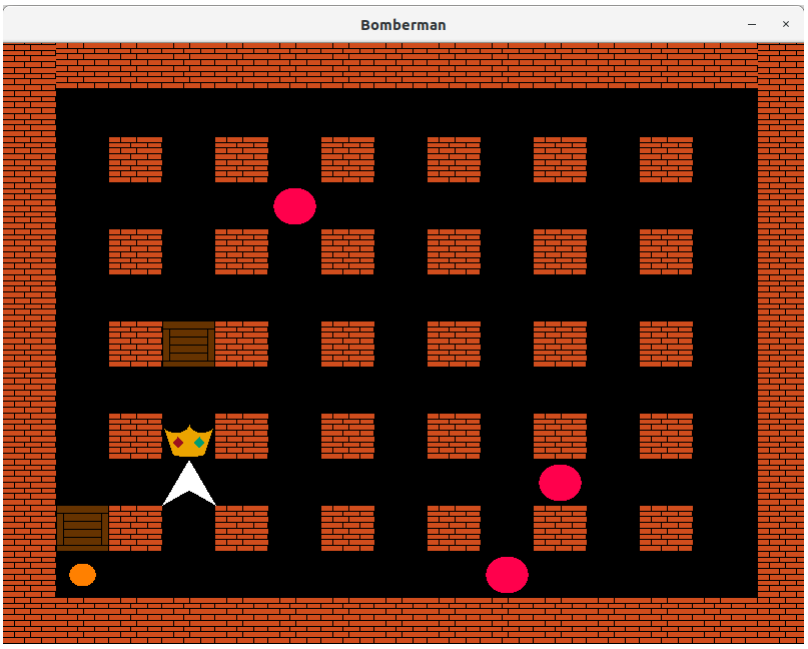

Figura B.3: Tercera Versión del juego. Hecha el **15/05/18**

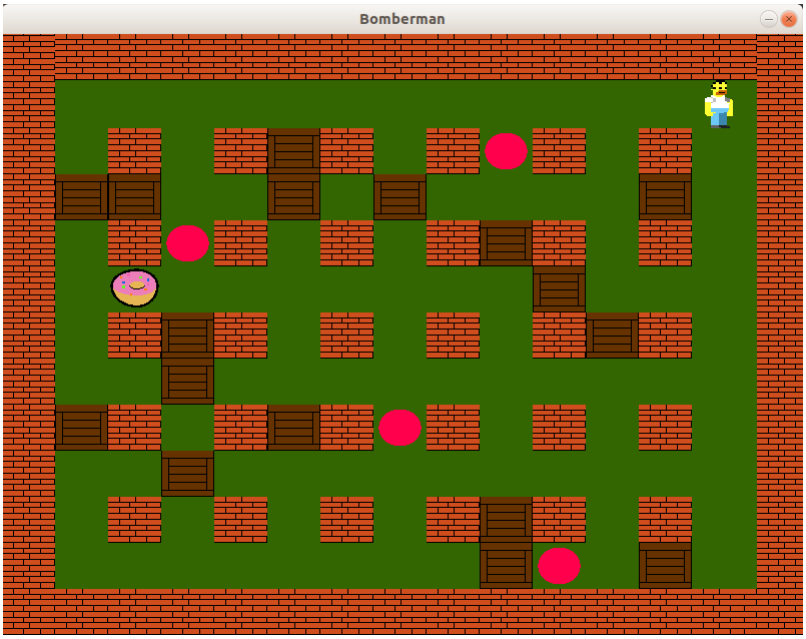

Figura B.4: Cuarta Versión del juego en modo facil. Hecha el **20/05/18**

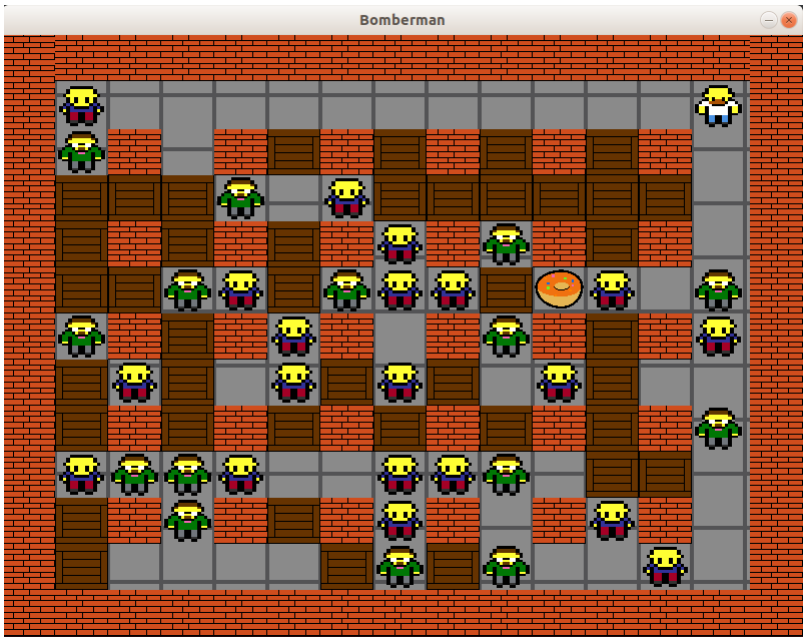

Figura B.5: Quinta Versión del juego en modo extremo. Hecha el **21/05/18**

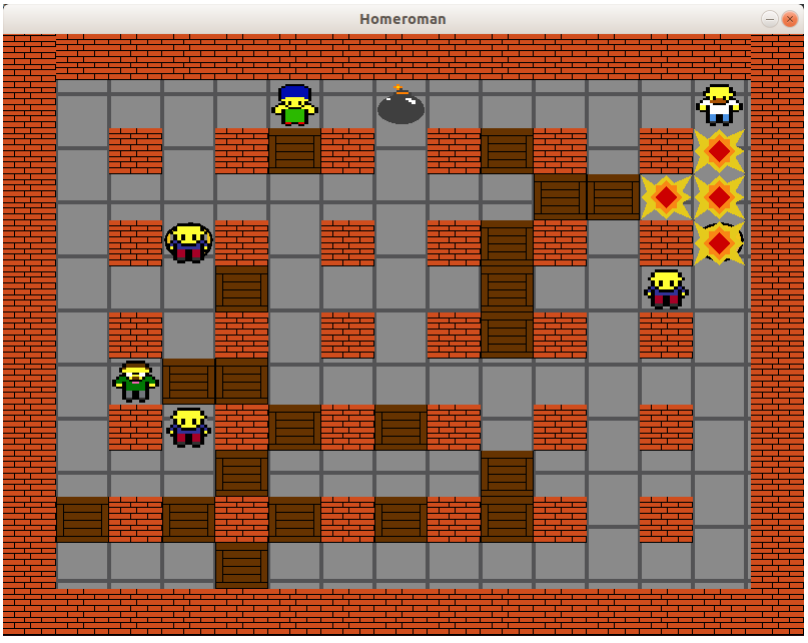

Figura B.6: Sexta Versión del juego en modo dos jugadores. Hecha el **22/05/18**