Department of Computing

CS 212: Object Oriented Programming

Class: BEEE - 13B
Fall 2022

Semester Project - BomberMan

Deadline: January 03, 2022

Attention:

- Make sure that you read and understand each and every instruction. If you have any questions or comments you are encouraged to discuss your problems with your colleagues
- Plagiarism is strongly forbidden and will be very strongly punished. If we find that you have copied from someone else or someone else has copied from you (with or without your knowledge) both of you will be punished. You will be awarded (straight zero in the project — which can eventually result in your failure) and appropriate action as recommended by the Disciplinary Committee (DC can even award a straight F in the subject) will be taken.
- Try to understand and do the project yourself even if you are not able to complete the project. Note that you will be mainly awarded on your effort not on the basis whether you have completed the project or not.
- Divide and conquer: since you have around 21 days so you are recommended to divide the complete task in manageable subtasks. We recommend to complete the drawing and design (i.e. number of classes and their relationships) phase as quickly as possible and then focus on the intelligence phase.
- Before writing even one line of code, you must design your final project. This process will require you to break down and outline your program into classes, design your data structure(s), clarify the major functionality of your program, and pseudocode important methods. After designing your program, you will find that writing the program is a much simpler process.
- No Marks will be given if you do not submit your class diagram and if you do not use the object oriented design principles you have learned during the course.
- Imagination Powers: Use your imaginative powers to make this as interesting and appealing as you can think of. An excellent solution can get you bonus marks

Goals: In this project you will build a 2D game (Bomberman – see Figure 1) using the techniques learned during the course. The major goal of this project is to consolidate the things you have learned during the course. In this respect it is requested to completely follow the principles taught to you during these past few

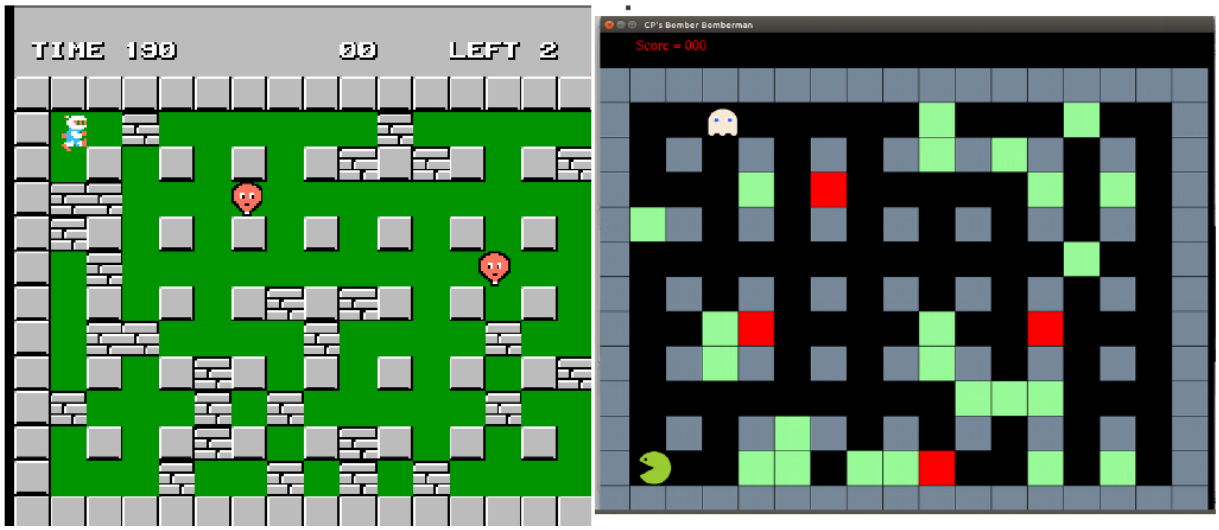

Figure 1: A screenshot of the original game (on left) and the game (on right) you have to develop.

months. Moreover, it is an excellent time to put in practice the things you haven't practiced thoroughly yet such as STL, etc.

GamePlay: The bomberman or player (You can play the game online here) must defeat enemies and reach an exit to progress through levels. Gameplay involves strategically placing down bombs, which explode in multiple directions after a certain amount of time, in order to destroy obstacles and kill enemies and other players. The player can pick up various power-ups, giving them benefits such as larger explosions or the ability to place more bombs down at a time. The player is killed if they touch an enemy or get caught up in a bomb's explosion, including their own, requiring players to be cautious of their own bomb placement. Please note that the player has to kill all the enemies and destroy obstacles before the time runs out. If the time runs out the player is killed. In a single run, there can be three players, if all are killed the game ends.

Furthermore, we will be importing the characters from Pacman, so our bomberman will be replaced by Pacman and enemies will be replaced by

Pacman's ghosts along with their behavior (the only difference will be now the goal will be to kill ghosts using bombs not by eating them).

1. Instructions

We provide a complete skeleton with all the basic drawing functions (can be located in util.h and util.cpp) needed in the project with detailed instructions and documentation. In other words all you need to know for building the game is provided. Your main task will be to understand the main design of the game and implement it. However, before proceeding with code writing you will need to install some required libraries. Please not that the provided skeleton already contains code for drawing the board, BomberMan, Bomb and the enemies.

1.1. Installing libraries on Linux (Ubuntu)

You can install libraries either from the Ubuntu software center or from the command line. We recommend the command line and provide the file "install-libraries.sh" to automate the complete installation procedure. To install libraries:

- Simply run the terminal and go to the directory which contains the downloaded file "install-libraries.sh".
- Run the command

  ```
  bash install-libraries.sh
  ```

- provide the password and wait for the libraries to be installed. If you get an error that libglew1.6-dev cannot be found, try installing an older version, such as libglew1.5-dev by issuing following on command line

  ```
  sudo apt-get install libglew1.5-dev
  ```

- If you have any other flavor of Linux. You can follow a similar procedure to install "OpenGL" libraries.

1.2. Compiling and Executing

To compile the game (skeleton) each time you will be using “g++”. However to automate the compilation and linking process we use a program “make”. Make takes as an input a file containing the names of files to compile and libraries to link. This file is named “Makefile” in the game folder and contains the details of all the libraries that the game uses and need to be linked.

So each time you need to compile and link your program (game) you will be simply calling the “make” utility in the game directory on the terminal to perform the compilation and linking.

```
make
```

That’s it if there are no errors you will have your game executable (on running you will see three shapes on your screen). Otherwise try to remove the pointed syntax errors and repeat the make procedure.

1.3. Windows Platform

A skeleton Visual Studio Project will be provided with necessary drawing functions to provide you kick start on Windows.

2. Drawing Board and Shapes

2.1. Canvas

Since we will be building 2D games, our first step towards building any game will be to define a canvas (our 2D world or 2D coordinate space in number of horizontal and vertical pixels) for drawing the game objects (in our case BomberMan,Bomb, board and bricks). For defining the canvas size you will be using (calling) the function “SetCanvas” (see below) and providing two parameters to set the drawing-world width and height in pixels.

```
/* Function sets canvas size (drawing area) in pixels...
 *  that is what dimensions (x and y) your game will have
 *  Note that the bottom-left coordinate has value (0,0)
 *  and top-right coordinate has value (width-1,height-1).
 *  To draw any object you will need to specify its location
 * */
void SetCanvasSize(int width, int height)
```

2.2. Drawing Primitives

Once we have defined the canvas our next goal will be to draw the game board and its characters using basic drawing primitives. For drawing each object we will need to specify its elementary point's locations (x & y coordinates) in 2D canvas space and its size. You will only need lines, circles, curves (toruses), rounded rectangles and triangles as drawing primitives to draw the complete board, BomberMan and enemies.

For this purpose, skeleton code already includes functions for drawing lines, circles, curves, rounded rectangles (see below), and triangles at specified locations.

Recall that a line needs two vertices (points) where as a triangle needs three vertices so to draw these primitives we will need to provide these vertices (points) locations along with the shape's color – c.f . Figure 2. Skeleton already provides a list of ≈ 140 colors which can be used for coloring different shapes – note that each color is combinations of three individual components red, green and blue.

2.3. Drawing Primitives

Initially it might seem drawing and managing the board is extremely difficult however this difficulty can be overcome using a very simple trick of divide and conquer. The trick revolve around the idea of the board being split into tiles. "Tile" or "cell" in this context refers to an 8 × 8 – you can use any tile size as you wish – pixel square on the screen. BomberMan's screen resolution is 224 × 288, so this gives us a total board size of 28 × 36 tiles. Since we will be working independent of pixel units, we can define tile size in our coordinates units. For instance we can divide the board in 8 × 81 units So drawing and managing the board will require these two steps:

```
1    // Drawing functions provided in the skeleton code
2
3   /* To draw a triangle we need three vertices with each
4   *    vertex having 2-coordinates [x, y] and a color for the
5   *    triangle.
6   *    This function takes 4 arguments first three arguments
7   *    (3 vertices + 1 color) to draw the triangle with the
8   *    given color.
9   * */
10 void DrawTriangle(int x1, int y1, int x2, int y2, int x3,
11                    int y3,float color[]/*three
12                                component color vector*/)
13
14 // Function draws a circle of given radius and color at the
15 // given point location (sx,sy).
16 void DrawCircle(float x, float y, float radius,
17                                float*color);
18
19 // Function draws a circular curve of given radius
20 void Torus2d(int x /*Starting position x*/,
21              int y /*Starting position Y*/,
22              float angle, // starting angle in degrees
23              float length, // length of arc in degrees, >0
24              float radius, // inner radius, >0
25              float width,              // width of torus, >0
26              unsigned int samples=60,// number of circle samples, >=3
27              float *color = NULL);
28
29 // Function draws a line between point P1(x1,y1) and P2(x2,y2)
30 // of given width and colour
31 void DrawLine(int x1, int y1, int x2, int y2,
32                     int lwidth = 3, float *color =NULL);
33
34 // Function draws a rectangle with rounded corners
35 at given x,y coordinates
36 void DrawRoundRect(float x, float y, float width,
37                                float height,
38                                float* color = 0,
39                                float radius = 0.0/*corner radius*/);
40// Function draws a string at given x,y coordinates
41 void DrawString(int x, int y, const string& str, float * color = NULL);
```

Figure 2: A set of functions for drawing primitive shapes

1. Splitting the board in tiles.
2. Finding and storing what shape to draw in each tile.

Once we have divided the boards into tiles our job reduces to finding what lies in each tile i.e. what primitive shape we need to draw in each tile. We can record this information in an offline table and then can loop over this table to draw each primitive. We can further simplify our task by defining an enum environment to assign constant names (integers) to these primitives and then build table of these primitives. Figure 4 shows an example on how can we draw some part of the board using a 2D offline table. Complete code can be found in the skeleton.

Following similar lines we have drawn the complete board. Note that your system must follow object oriented design principles. Remember that you can do your drawing only in the **GameDisplay()** function, that is only those objects will be drawn on the canvas that are mentioned inside the **GameDisplay** function. This **GameDisplay** function is automatically called by the graphics library whenever the contents of the canvas (window) will need to be drawn i.e. when the window is initially opened, and likely when the window is raised above other windows and previously obscured areas are exposed, or when **glutPostRedisplay()** is explicitly called.

In short, **GameDisplay** function is called automatically by the library and all the things inside it are drawn. However whenever you need to redraw the canvas you can explicitly call the **GameDisplay()** function by calling the function **glutPostRedisplay()**. For instance, you will calling the **GameDisplay** function whenever you want to animate (move) your objects; where first you will set the new positions of your objects and then call the **glutPostRedisplay()** to redraw the objects at their new positions. Also see the documentation of Timer function.

```
1 // A simple example of board
2 enum BoardParts {
3          NILL, // Prohibitive Empty space
4          TLC, // Left corner top
5          TRC, //Right corner top
6          BLC, // Left corner bottom
7          BRC, //Right corner bottom
8          HL, // Horizontal line
9          VL, // Vertical line
10         PEBB, // Pebbles
11 };
12 const int BOARD_X = 10;
13 const int BOARD_Y = 5;
14 static int board_array[BOARD_Y][BOARD_X] = {
15 { PEBB, PEBB, PEBB, BRC, BLC, PEBB, VL, VL, PEBB, PEBB},
16 { PEBB, PEBB, PEBB, VL, VL, PEBB, PEBB, PEBB, PEBB, PEBB },
17 { PEBB, PEBB, PEBB, VL, VL, PEBB, PEBB, PEBB, PEBB, PEBB },
18 { BRC, HL, HL, TLC, TRC, HL, HL, HL, HL, BLC },
19 { TRC, HL, HL, HL, HL, HL, HL, HL, HL, TLC } };
```

Figure 3: Example code for generating some section of the board.

2.4. Drawing Enemy

An enemy can be easily drawn by splitting it into its elementary parts. Figure 4 shows the division of the enemy into “tiles”. Once given this configuration we can draw an enemy using circles and a rounded rectangle. If you want a more realistic depiction you can use

ellipses to draw legs and eyes outer regions, recall from the conic section that an ellipse is nothing more than a non-uniform scaled version of the circle. From the given description first you will find the dimensions of the enemy. We have draw the enemy exactly via this procedure, please see the function **DrawEnemy**.

2.5. Drawing Bomberman

Drawing BomberMan is much easier than drawing enemies. Specifically, a BomberMan can be drawn by using the same technique used to draw a circle. The only difference here is that you will not draw the vertices in the range [157.5◦, 202.5◦]. Moreover you can add a circle of black color as its eyes.

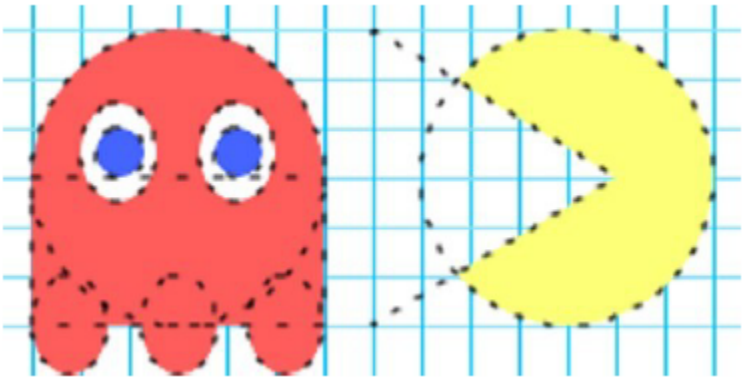

Figure 4: Representation of enemy and BomberMan in elementary parts with grid layout.

2.6. Interaction with the Game

For the interaction with your game you will be using arrow keys on your keyboard (you can use mouse and other keys as well). Each key on your keyboard have associated ASCII code. You will be making using of these ASCII codes to check which key is pressed and will take appropriate action corresponding to the pushed key. E.g . to move the BomberMan right you will check for the pressed key if the pressed key is left arrow you will move the BomberMan left (change its position). Keyboard keys are divided in two main groups: printable characters (such as a, b, tab, etc.) and non-printable ones (such as arrow keys, ctrl, etc.). Graphics library will call your corresponding registered functions whenever any printable and non-printable key from your keyboard is pressed. In the skeleton code we have registered two different functions (see below) to graphics library. These two functions are called whenever either a printable or non-printable ASCII key is pressed (see the skeleton for complete documentation). Your main tasks here is to add all the necessary functionality needed to make the game work.

```
1  /*This function is called (automatically by library)
2  * whenever any non-printable key (such as up-arrow,
3  * down-arraw) is pressed from the keyboard
4  *
5  * You will have to add the necessary code here
6  * when the arrow keys are pressed or any other key
7  * is pressed...
8  * This function has three argument variable key contains
9  * the ASCII of the key pressed, while x and y tells the
10 * program coordinates of mouse pointer when key was
11 * pressed.
12 * */
13 void NonPrintableKeys(int key, int x, int y)
14
15 /* This function is called (automatically by library)
16 * whenever any printable key (such as x,b, enter, etc.)
17 * is pressed from the keyboard
18 * This function has three argument variable key contains
19 * the ASCII of the key pressed, while x and y tells the
20 * program coordinates of mouse pointer when key was
21 * pressed.
22 * */
23 void PrintableKeys(unsigned char key, int x, int y)
```

3. GamePlay

The basic gameplay rules are listed in the following subsections.

3.1. Bomberman

If BomberMan runs into an enemy, he loses a life, and the enemies and BomberMan are all reset to their original starting locations. If BomberMan collides with a gift or energizer, he eats the item, which disappears from the board and updates the score.

3.2. Bricks

BomberMan board consists of different types of bricks.
Solid Bricks: These bricks are part of environment and will not break by bomb blast.
Green Bricks: these are Breakable bricks and will break if they come in the destruction radius of bomb. Breaking this brick earns 100 points and can be broken instantaneously.
Red Bricks: These are Special bricks will not only break themselves, they will also break one extra brick on each of their side also. Breaking this brick earns 200 points however this can be broken after 50 seconds. There can be no more than 10 Red Bricks in the board.

3.3. Bomb

BomberMan can place a bomb wherever it moves on the board. Once the bomb is placed, it blasts after 1 second. When bomb blasts,it destroys all breakable bricks found in the destruction radius of 1 brick. Blast will also destroy any enemy or BombeMan itself, if they are present in that radius.

3.4. Enemies

The enemies move according to a “Breadth First Search” algorithm is outlined later. The enemies’ behavior changes (each one has a different behavior), and BomberMan is able to kill them.

3.5. Enemies Behavior

The enemies navigate the board in three different modes:

Chase: the enemies chase BomberMan
Scatter: the enemies move towards the corners of the map
Frightened: the enemies move randomly

During normal gameplay, the enemies alternate between Chase mode and Scatter mode. The length of each mode is up to you, but we suggest alternating between 20 seconds of Chase mode and 7 seconds of Scatter mode. These two modes use a BFS algorithm in order for the enemies to reach their targets as quickly as possible.
The enemies turn around (180➦) when they change their mode.

3.5.1. Enemies Personalities

In BomberMan all the enemies have different personalities and thus behave differently. You are required to implement the personalities listed below:
Blinky starts one row above BomberMan
Pinky starts three rows away from BomberMan but its behavior is random(It does not follow BFS for movement)
Inky starts one row away from BomberMan when 30% of the breakable bricks are destroyed
Clyde starts one row away from BomberMan when 30% of the breakable bricks are remaining

3.5.2. Target Locations

In Scatter mode, each enemy's target should be a different corner of the board. In Chase mode, the enemies should all have targets that are relative to BomberMan's current location. In both modes, the targets are allowed to be walls.
Here are some suggestions for target locations:

- BomberMan's location
- 2 spaces to the right of BomberMan's location
- 4 spaces above BomberMan's location
- 3 spaces to the left and 1 space down from BomberMan's location

3.5.3. Bread First Search for the Enemies

When a enemy needs to make a decision about where to turn, it must choose the direction that will bring it to its target the fastest. You will be implementing a "Breadth First Search (BFS)" algorithm to determine this optimal direction.
The general idea behind BFS is to first search all the squares neighboring you, then expand and search the squares neighboring your neighbors, then expand again, and so on, until the entire maze has been searched. A BFS guarantees that a target will be reached first by the shortest path. Note that we need to somehow mark which squares have already been visited, so we don't loop around the maze forever!
Note how different this search is from what you did in the first assignment where you tried to solve the maze. In that problem you did a Depth First Search (DFS) (implicitly) using recursion. In DFS you go along one path and keep on going until you find the solution. If you reach a dead-end you backtrack to the nearest junction and go along an un-explored path systematically. BFS is fairly easy to understand and is used to find the shortest path between two points. The two points being the position of the enemy and its destination. In contrast to DFS, the BFS takes one step in each possible direction in the hope to reach the destination. In the example in Figure 6-2 you are exploring 2 squares at the same time while in Figure 6-3 the search is expanded to 4 candidate paths. Finally one of these paths finds the BomberMan and that is the one the enemy should take.
Note that each square has four neighbors, i.e. four different search directions but the enemy can only check the squares that are not walls or are not already marked by the same enemy's search. When the target of a enemy is a square that is not reachable, change the destination to a reachable square nearest to the actual destination. This will be handy in the cases where the destination is one of the corners of the board or the calculated destination (e.g . 2 spaces to the right of BomberMan's location) lies
in the wall or outside the board. To successfully implement the BFS you can use the queue data structure which works like a queue in real life. Elements go in at one end and leave the queue through the other (First In First Out, FIFO). Look it up and you'll know more. We'll also discuss it during the lectures next week when we start STL.

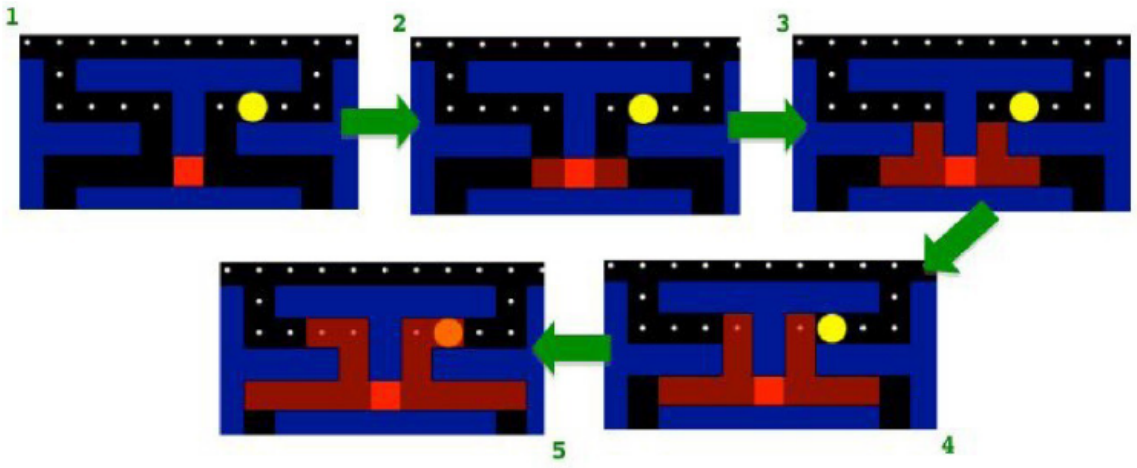

Figure 6: Example of BFS path starting at the enemy's location and searching for BomberMan's location, examining all possible neighbors at each step..

3.6. Scoring

Whenever BomberMan eats a dot, energizer, or enemy, the score should be updated as follows:

- Destroying a brick +50 points
- Gift +100 points
- Energizer: +100 points
- Enemy: +200 points

3.7. Winning and Losing

The game ends either when all of the bricks are destroyed and energizers are eaten or when BomberMan loses all of his lives. When the game ends, the enemies and BomberMan should all stop moving, and keyboard input should be disabled. The game should be paused when p is pressed and should resume on pressing p again.

4. Marks Distribution

- Submission of Design (on paper): 2 points
- Rendering and movement of enemies,bomb and BomberMan: 2 points

- Shortest path through BFS: 2 points
- Scoring: 1 point
- Misc gameplay: 3 points

The 2 points for design will only be awarded if you do some implementation. Submitting only the design will not earn you any marks. You can only use one global pointer. No other global variable is allowed.

5. Minimum Requirements

To get any viable portion of the above marks you will need to fullfill following minimum requirements.

- Complete object oriented design with Inheritance, Polymorphism, Templates, etc. Note that you are not allowed to declare any global variables except a single pointer, any other global variable define by you will lead to serious penalty. Avoid global functions as well.
- BomberMan can move around the board
- BomberMan collides with enemies and places bomb
- Enemies chase BomberMan using a working BFS algorithm.

6. Extra Credit (Bonus)

Getting a bonus in this assignment is fairly easy once you have completed the functionality above. A fully working solution can itself be a candidate for some extra credit. Moreover you can use your imagination to add interesting features to earn you bonus marks e.g . a new level, keeping track of the highest score, implementing different enemy personalities, more power-ups (or speed-up for BomberMan) etc.

Good Luck :)