Programming with C/C++

Academic year: 2022-2023, Fall semester

Programming assignment

The AI BipBop

Student name: Dante Tollenaar

Student number: 2040705

1. Results

The submitted assignment provides executable code for a playable version of the BipBop game, including a working Graphical User Interface (GUI) built with FLTK. Additionally, the game can be played by the computer automatically through a hardcoded solution.

1.1 Structure

The code is distributed over four header files and a source file called ‘game.cpp’. The code is dependent on the ‘std_lib_facilities.h’ header file and a collection of fltk header and source files which were supplied in the course tutorials on GUIs. The game.cpp file is where the main function is located and where all code comes together. Three of the four header files are each dedicated to one specific class, namely the ball, the board/paddle and the brick(s). The window object is initialised in the main function and inherits completely from the FL_Window class. The last header file, called ‘mechanics.h’, consists of logic functions that handle most of the interactions between the classes while the game is running.
The hardcoded solution to the game is nested within the board header file. The algorithm starts at the same time as the game, and terminates if the user decides to play. That is, if the user hits any key on the keyboard.

1.2 Main challenges

The two main challenges were object interaction and showing the game through the GUI.
Several possible solutions for object interaction were considered, such as creating a 2D array of the window that tracks the positions of the objects, or a separate class that holds all objects and handles their interactions. The 2D array solution was quickly dropped because I did not see how that would work better than other solutions when a round object, the ball, is involved. Having a central class that holds all objects in the game was supposed to be one of the last components to be implemented, but in the end the current solution works well for the scope of the project.
In the early stage of on the project, implementing the GUI was scheduled for after the game itself. However I quickly found out that it was easier to develop the game concurrently with the GUI, so I started over and reused some of the code. Owing to this way of development, the objects of the game all inherit from FLTK.

1.3 Time organisation

The planning was divided into five weeks starting in the second half of the semester, leaving one week for slack. The first two weeks were assigned to developing object classes and game mechanics, the second two for GUI implementation and the last week for implementing the AI agent.
However, I prioritised other projects/courses and rescheduled my planning to the last two weeks before the deadline, which proved to be of too short notice for me.

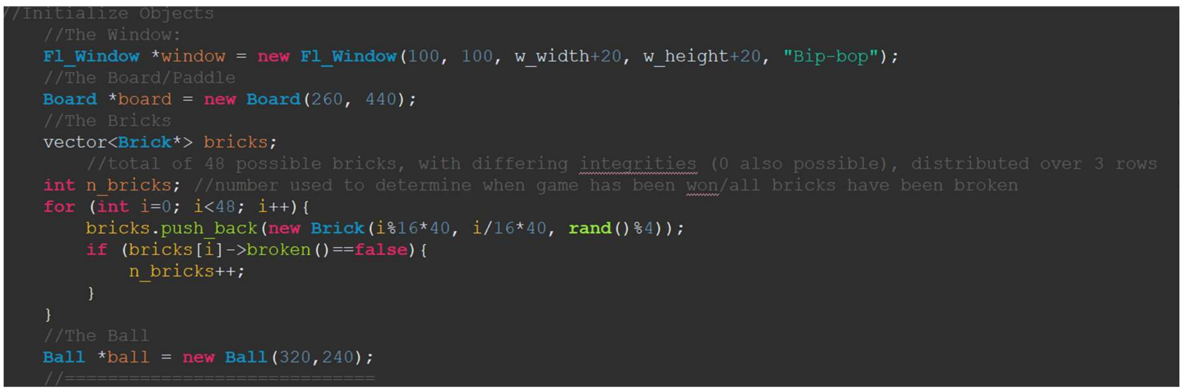
1.4 Manner of working

I developed this project in very small steps, iteratively adding code and then testing the added code. Developing the GUI and the game concurrently rewarded in this case, as it provided me with reasonably clear, visual feedback of the mechanics. I also used a separate environment for experimenting and unit testing.

2. Tasks and objectives

2.1 Window, Board, ball and bricks

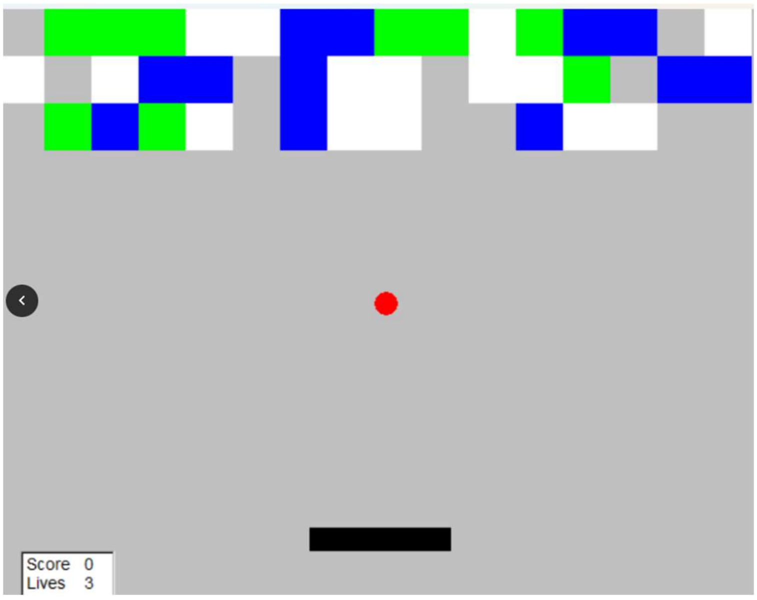
In the main functions, all the objects are initialised and their pointers are stored in variables (figure 1). Bricks are stored in a vector called bricks. There are a total of 48 places on the window reserved for generated bricks. These are distributed over three rows and sixteen columns. Each of these bricks is assigned a random integrity value in the range [0, 3]. An integrity of zero means the brick is broken. In this case the space of this brick is empty. Three rows of bricks leaves plenty of space in the upper half of the window. The number of rows can be easily adjusted by replacing the 48 in the for loop with another multiple of 16.
The ball always spawns in the middle of the window, with the board aligned underneath it. When all is initialised, the window is shown, as in figure 2.



```
//Initialize Objects
    //The Window:
    Fl_Window *window = new Fl_Window(100, 100, w_width+20, w_height+20, "Bip-bop");
    //The Board/Paddle
    Board *board = new Board(260, 440);
    //The Bricks
    vector<Brick*> bricks;
        //total of 48 possible bricks, with differing integrities (0 also possible), distributed over 3 rows
    int n_bricks; //number used to determine when game has been won/all bricks have been broken
    for (int i=0; i<48; i++){
        bricks.push_back(new Brick(i%16*40, i/16*40, rand()%4));
        if (bricks[i]->broken()==false){
            n_bricks++;
        }
    }
    //The Ball
    Ball *ball = new Ball(320,240);
    //=============================
```

Figure 1 Object initialization in main function

2.2 Movement of the ball



Score 0
Lives 3

Figure 2 Window upon initialization. The colors of the bricks correspond to their integrity, which are blue (3), green (2) and white (1).

The movement of the ball is determined by its direction. At initialization, the ball gets a random direction between 245 and 295. This way it moves in the general direction of the board when the game starts. In the interest of making collision detection easier, the ball can only move in four directions, which goes almost unnoticed in-game, because of the large scale of the window. To determine which way to move, the starting coordinates of the ball have been

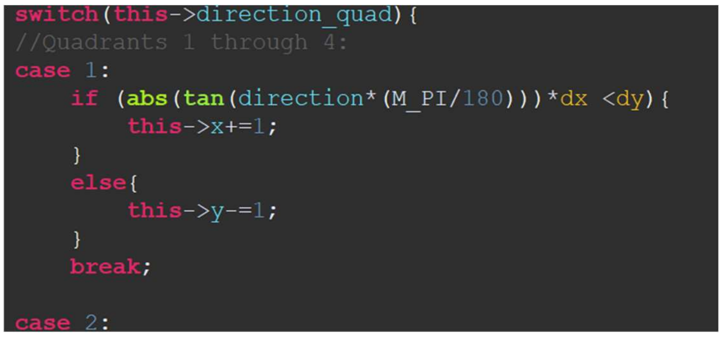
stored in the variables x0 and y0. These are used to calculate the length of the legs of the right triangle between the starting coordinates and current coordinates. The tangent function is used to determine which move to make. The tangent function works with radians, so the direction is multiplied with pi divided by 180 to convert it to radians. For the code implementation, see figure 3.
At the end of the main function, a statement is implemented which checks whether the ball has gone too far down. When the ball dies, it resets in the middle of the board with a new direction. A life is lost when this happens.

2.3 Movement of the board

The board object uses FL events to determine which keys are pressed during the game. The A and D keys correspond to left and right, respectively. When pressed, the x coordinate of the board is adjusted by five into the requested direction.

2.4 Gameplay

Three functions, located in the mechanics header file, have been developed to detect and handle collisions between the ball and the walls, the ball and the board, and the ball and the bricks. The functions all use the x and y coordinates of the ball and the object to decide whether they collide, whereafter they use the direction of the ball to calculate the redirection of the bounce.
The outer sides of the board are slanted, so that the ball bounces off these sides with a slight correction of thirty degrees. On any other surfaces, the redirection is a mirroring of the previous direction in either the y-axis (when the surface is vertical) or x-axis (when the surface is horizontal).



```
switch(this->direction_quad){
//Quadrants 1 through 4:
case 1:
    if (abs(tan(direction*(M_PI/180)))*dx <dy){
        this->x+=1;
    }
    else{
        this->y-=1;
    }
    break;

case 2:
```

2.5 AI agent

Since I had some trouble with my time management, I decided to focus on other tasks of this assignment. Therefore, my smart solution is a hardcoded solution.
It is an algorithm, nested within the handle method of the board class, that tracks the x coordinate of the ball. The board then moves parallel to the ball.
The solution is ensured to yield a win, seeing that the ball cannot die. However it may not be the fastest possible win.

3. Challenges

3.1 Solved challenges

Solving the maths behind the bounce redirection took a while for me. Writing it out on paper and drawing helped. A big mistake in solving this problem was my false predisposition, that the tan function from the math header takes degrees as input, which is not the case.

4. Discussion

4.1 Bugs

The collision functions are still not waterproof. In some cases, the ball does not collide with the bricks, because it hits them on the corners. Additional code should be implemented that can detect collisions on the ball when those collisions are not fully vertical or horizontal.

4.2 Improvements

The hardcoded solution could be improved by choosing a target and aiming for that target, by using the slanted sides of the board.

The collision detection system for the ball and bricks can be made more efficient. In the current version of the game, the vector of bricks is iterated over at every tick, to check for collisions. Sorting the bricks based on coordinates (in an 2D array maybe) and starting search in the proximity of the ball could improve efficiency.