May 2018

Breakout

ELEC2645 Embedded Systems Project

Ziad Abass
EL16ZA - 201041955

Table Of Contents

1. User Guide

1.1 The Gameplay Bit

Breakout is the perfect pass-time game. It starts out with 40 bricks arranged in 8 columns and 5 columns. The aim is for the user to destroy as many bricks as possible without losing all three of their lives, using just a ball.

Upon powering up the gamepad, the user is greeted with a warm welcome screen. It simply displays the game's icon and gives the instruction on how to start the game (pressing start).

The ball starts off by moving in a random direction while being able to bounce off the side and the top walls, as well as off the bar which the user controls. This bar is located at the base of the screen and is controlled by the joystick to move horizontally in one axis. The user must move the bar to collide with the ball as it is falling downwards. If the ball misses the bar and falls below its horizontal level, the user loses a life.

The user has 3 lives to begin with. The number of lives remaining are represented by the number of heart icons on the screen, as well as by the number of LEDs lit on the gamepad.

Upon colliding with a brick, the ball destroys it and rebounds off it, rewarding the user with 10 points. All bricks carry the same number of points. There are two bricks, however which bear something extra! In addition to the 10 points they give, one gives an extra life power-up, and the other gives a bar-extender power-up. The former is self-explanatory, and the latter makes the bar wider for a set period of time, making the gameplay easier for the user during that time.

The game can end for one of two reasons – either the user has lost all of their lives, or if they have managed to successfully destroy all of the bricks. In the first case, they are shown a bleak 'Game Over', and in the second a gleeful 'GG!'. In both cases, the final score is displayed for encouragement, as well as the instruction to restart the game (which is simply pressing 'Y'). Restarting the game re-draws all the bricks and starts the game again (newsflash!).

To keep the users entertained, the piezo-electric speaker is used to produce different sounds for ball collisions with different elements (walls, bricks and the bar). Other sounds are made upon losing a life, earning a power-up and losing the game.

1.2 The Graphical Bit

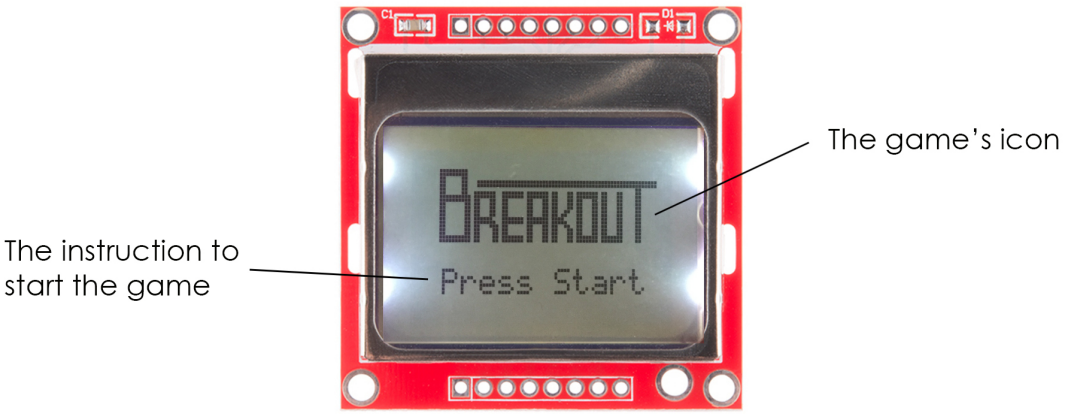


Figure 1 - The welcome screen

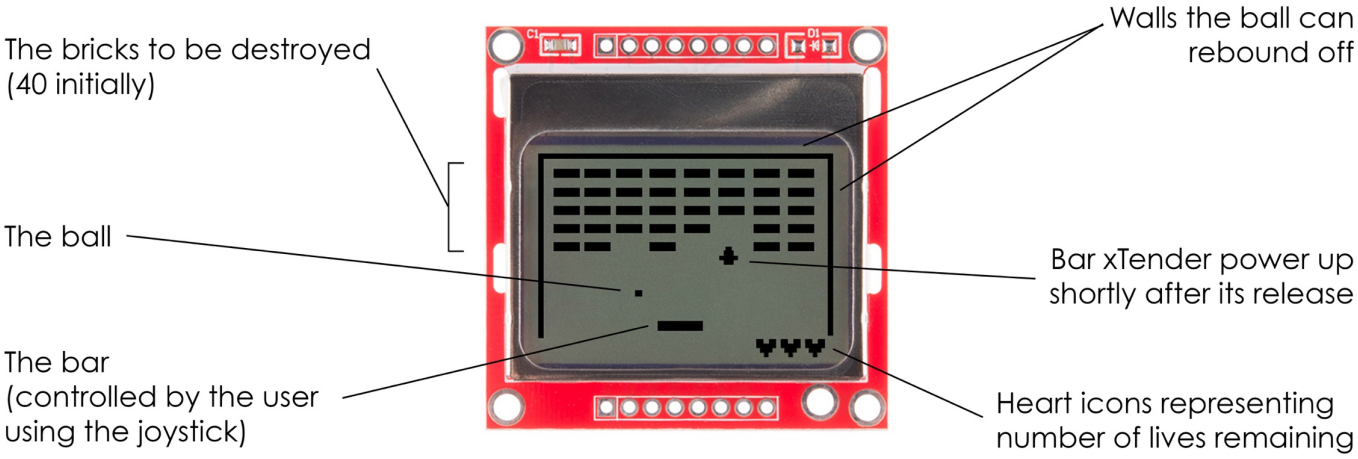


Figure 2 – The gameplay GUI

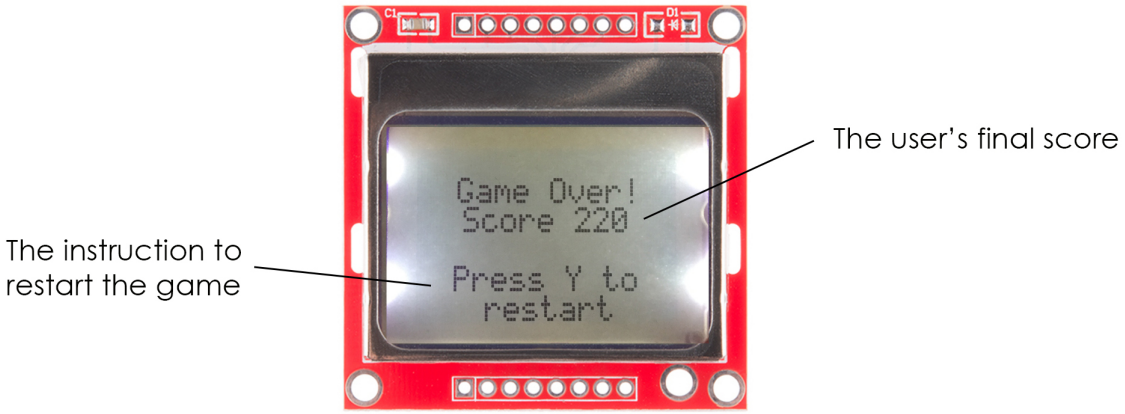


Figure 3 - The game over screen

2. Software Design

2.1 Design Rationale

Breakout may seem like a simple game when, in fact, it involves a lot of bits and pieces which had to be considered simultaneously for a smooth gaming experience. To facilitate the development of such a game, its various aspects were defined to then each be made into a class of its own.

Splitting up the game into these basic elements was the starting point of the programming, and was accomplished by picturing the expected gameplay and dissecting it into what seemed to be reasonable on-screen and off-screen components. These initially turned out to be the ball, bar and bricks, which make up the basic gameplay. A scoring element was also required to keep track of the score and the lives remaining. Afterwards, power-ups were added to the game and it seemed logical to consider them separate aspects.

A class has been created for each of the aforementioned elements, in addition to a gameplay class (acting as the game engine) which binds them all together and links them to the main file. That main file is simply in charge of calling the methods which run the game from the gameplay class.

The gameplay class is considered to be the centrepiece of the whole game – its initialisation calls at the initialisation of all the other elements, and the same goes for drawing them and updating their positions. This makes the coding much more efficient, organised and easier to troubleshoot. The gameplay class is also in charge of detecting and reacting to the ball's collisions with the bar, walls and bricks, as well as the power-ups' collisions with the bar.

Perhaps the most challenging aspect of creating the game was acknowledging a ball-brick collision, rebounding the ball and finally hiding only the struck brick. This was particularly difficult since the bricks were not drawn as individual sprites, but by using a for loop incrementing x and y coordinates to display 40 equidistant bricks on the screen. (Of course the latter method reduced the lines needed to draw the bricks from 40 down to 4). Instead of having countless lines of code to check if the ball had collided with each of the 40 bricks, these ball-brick collisions were detected using maths – taking advantage of the fact that the bricks are equidistant.

The coordinates of the ball were continuously obtained, separated into x and y components and plugged into equations to find out if the ball had collided with a brick. The methods used to draw the bricks and to detect ball-brick collisions are outlined below.

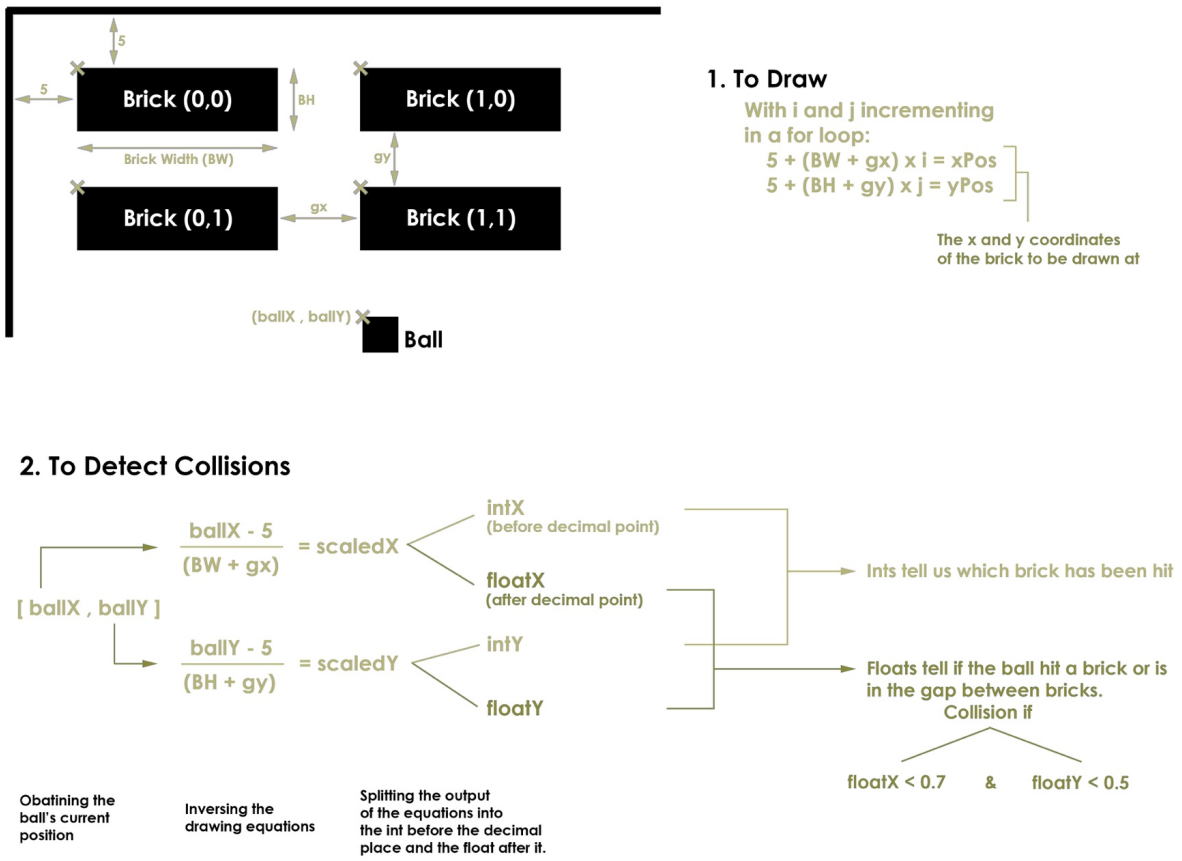


Figure 4 - Outlining how bricks are drawn & how collisions are detected

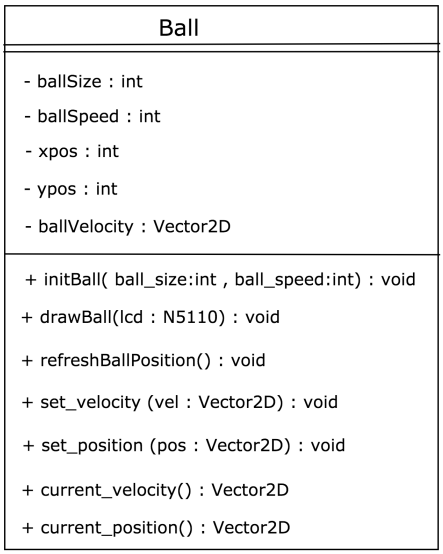
There is an 8x5 binary array representing each brick's state – whether or not it has been destroyed yet. This array is what is used when drawing the bricks to make sure that only the ones which have not been destroyed are drawn. With each ball-brick collision, the element of the array corresponding to that brick switches from 0 to 1, indicating that the brick has been destroyed.

Another feature which wasn't simple to implement was the barXtender power up which, when collected, makes the bar wider for a set period of time. This was because setting up a waiting function which didn't affect the continuous looping of the game was a tricky job. Instead, it was made that as soon as the power-up is collected, the bar's width increases and the number of loops the main function goes through is counted. When that number reaches 80, the bar's width is returned to its regular size. This makes sure the bar is extended only for a fixed period of time.

The value for the parameters which are used to initialise the game, such as the bar's width and the bricks' dimensions, are all defined in the main.cpp file. This makes it so much easier to change any of them in the future.

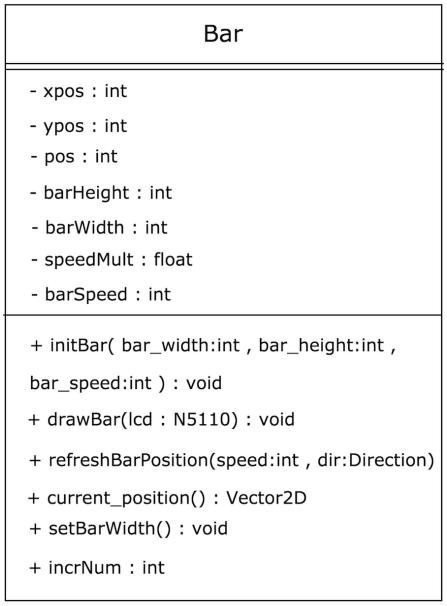
2.2 Class Diagrams

1. Ball Class



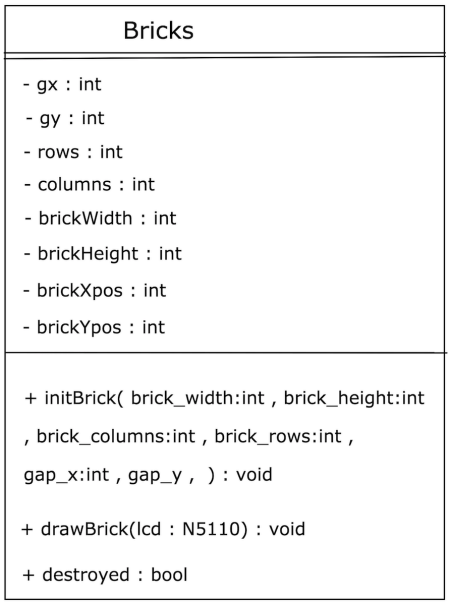
This class controls the ball's initialisation and movement. The former involves defining the ball's size, speed and starting position, as well as randomising its starting direction. As for the latter, 2 methods are used to update the ball's velocity and position respectively, and 2 others function to obtain its current velocity and position.

2. Bar Class



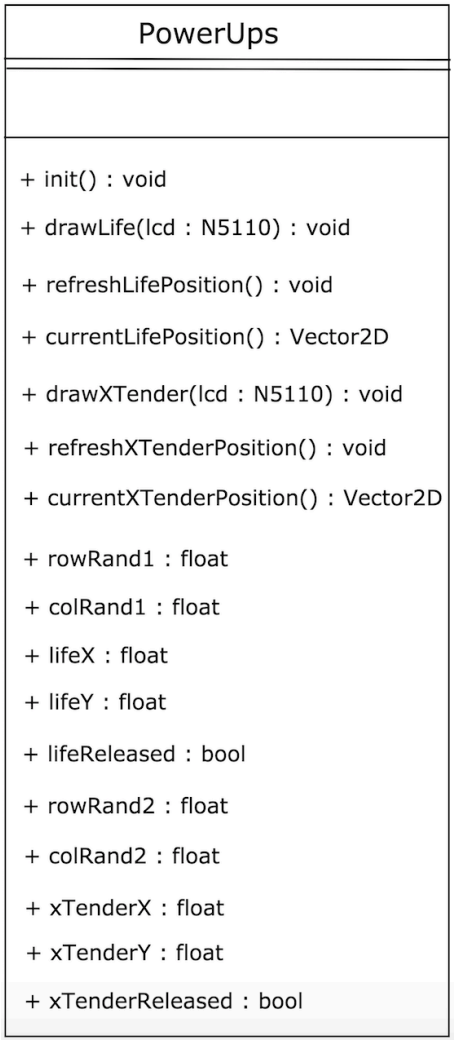
Here, the bar which the user controls is initialised to set its dimensions and speed. Also, it is drawn and its position is updated depending on the user's control of the joystick. There is also a method to obtain the bar's current position to be used for collision detection.

3. Bricks Class



Firstly, this class initialises the bricks by defining the dimensions of the individual bricks as well as the number of rows and columns of bricks to be drawn on the screen. Secondly, it draws the bricks on the LCD and keeps track of which bricks have been destroyed and which have not yet been.

4. PowerUps class



This class adds 2 power-ups to the game (an extra life and a bar extender), each hidden in a different brick. For each one, this class draws its icon after the special brick is hit, updates its positions to make it fall down, and provides the icon's current position to be used to detect its collision with the bar.

5. Score Class

Score
+ init(lives:int) : void + didHitBrick(gamepad:Gamepad) : void + lostLife(gamepad:Gamepad) : void + display(lcd:N5110 , gamepad:Gamepad) : void + gameOver(lcd:N5110 , gamepad:Gamepad) : void + livesRemaining : int + currentScore : int

The score class holds the current score as well as the number of lives remaining. In it is the method responsible for adding onto the score when a brick is struck, and one which deducts from the number of lives when the ball misses the bar. The class also displays on the LCD the number of lives remaining and defines what happens when the game ends.

6. Gameplay Class

Gameplay
- keepBallOnScreen(gamepad : Gamepad) : void - ballBrickCollision(gamepad : Gamepad) : void - barBallCollision(gamepad : Gamepad) : void - ballMissedBar(gamepad : Gamepad) : void - hideStruckBrick(xBallPosInt:int, yBallPosInt int, ballPosFraction:Vector2D, gamepad:Gamepad) : void - specialLifeBrickStruck(lcd : N5110) : void - specialXTenderBrickStruck(lcd : N5110) : void - barCollectedExtraLife(gamepad : Gamepad) : void - barCollectedWiderBar(gamepad : Gamepad) : void - di : Direction - ma : float - ballPosition : Vector2D - ballVelocity : Vector2D - barPosition : Vector2D - lifePos : Vector2D - xTenderPos : Vector2D - currentLoop : int - stopLookingForLife : bool - stopLookingForXTender : bool - ballPosFraction : Vector2D - scaledBallPos : Vector2D
+ initGame(ball_size:int , ball_speed:int , bar_width:int , bar_height:int , bar_speed:float , gap_x:int , gap_y:int , brick_width:int , brick_height:int , brick_columns:int , brick_rows:int , lives:int) : void + refreshPositions(gamepad:Gamepad , lcd:N5110) : void + drawItems(lcd : N5110, gamepad : Gamepad) : void

The gameplay class is the game's engine – it calls at the initialisation and drawing methods of all the other classes. It is also in charge of acknowledging and reacting to the ball's collision with the walls, bar and bricks, as well as the power-ups' collision with the bar.

3. Testing

3.1 Automated Low-Level Tests

TEST NAME	Bar_movement_test
DESCRIPTION	Check that the bar moves correctly when the joystick does.
COMPONENTS TESTED	• Bar::current_position • Bar::refreshBarPosition
DETAILED TEST PROCEDURE	• Initialise the bar with a width of 9 at the position (42,40) • Set the user input to have a direction of WEST and a magnitude of 20 (consider that there is a speed multiplier of 0.5). • Read bar's current position • Set the user input to have a direction of EAST and a magnitude of 50. • Read bar's current position
EXPECTED RESULTS	[32 , 40] [57 , 40]
STATUS	PASSED
ACTUAL RESULTS	[32 , 40] - PASS [57 , 40] - PASS
ANALYSIS	The bar was initially moving in one direction but not the other. When writing the code, the magnitude of the velocity was accidentally negated to move the bar in the opposite direction. It should not be negated as only the direction applied (West or East) is in charge of whether the bar goes left or right. The magnitude must remain positive.

TEST NAME	Ball_movement_test
DESCRIPTION	Check that the ball moves correctly when a velocity is applied.
COMPONENTS TESTED	• Ball::set_position • Ball::set_velocity • Ball::refreshBallPosition • Ball::currentPosition
DETAILED TEST PROCEDURE	• Initialise the ball, placing it at (42,40). • Set the starting position of the ball to be (10,10). • Read the ball's current position. • Set a velocity vector of [5, 6], and apply it to the ball. • Refresh the ball's position • Read ball's new position
EXPECTED RESULTS	[10 , 10] [15 , 16]
STATUS	PASSED
ACTUAL RESULTS	[10 , 10] - PASS [15 , 16] - PASS
ANALYSIS	The ball was being placed in desired positions easily and accurately, its current position could be obtained correctly and its position could be updated flawlessly.

3.2 High-Level Tests

TEST NAME	Destroyed_brick_test
DESCRIPTION	Checks that a brick struck by the ball is destroyed and is no longer displayed on LCD.
COMPONENTS TESTED	• Ball • Gameplay • Bricks
TEST 'STORY'	1. **Given** that the ball is moving towards a certain brick • **When** the ball collides with the brick, • **Then** only the brick which is hit should disappear and the ball should rebound. 2. **Given** that the ball is moving towards the bricks' region • **When** the ball moves between the bricks • **Then** the ball should continue in its path and no bricks should disappear
STATUS	PASSED
ACTUAL RESULTS	1. Behaviour was as expected 2. Behaviour was as expected
ANALYSIS	This indicates that the mathematical method used to detect ball-brick collisions is functioning well in accurately determining if the ball is in a brick's region or in an empty space.

TEST NAME	Game_over_test
DESCRIPTION	Checks that the game ends when all lives are lost or when all the bricks are destroyed
COMPONENTS TESTED	• Gameplay • Score
TEST 'STORY'	**1. Given** that there is one life remaining, • **When** that final life is lost, • **Then** the gameplay should stop, and the game over screen should be displayed **2. Given** that there is one undestroyed brick remaining, • **When** that final brick is hit, • **Then** the gameplay should stop, and the game over screen should be displayed
STATUS	PASSED
ACTUAL RESULTS	1. Behaviour was as expected 2. Behaviour was as expected
ANALYSIS	Displaying something different on the screen does not stop the actual gameplay, i.e. the ball would still update its position and virtually hit bricks even if the user cannot see this. The solution was to place the code in the gameplay's 'drawItems' and 'refreshPositions' within if statements dependant on the number of lives remaining and the total score. This makes sure the gaming elements are only drawn and refreshed if there are one or more lives remaining, and if the score had not yet reached 400 (i.e. all bricks destroyed).