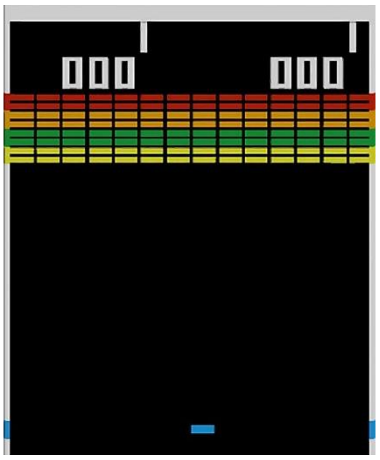
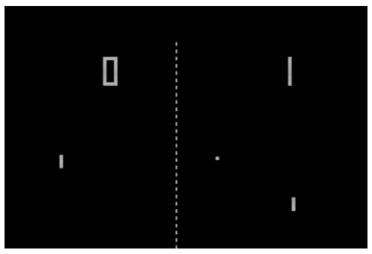
BREAKOUT REPORT

INTRODUCTION:


‘Breakout’

The basic idea, based on the brief, was to create a clone of the game ‘Breakout’, where a paddle is moved to bounce back a ball and destroy some blocks to win the game. Additionally, the player needed 3 lives, different types of blocks, 3 different levels and an available powerup. For the different levels I opted to change the number of blocks to add difficulty (the more blocks, the higher the level) while for the powerup I decided on a bonus life powerup, as I thought it would be the easiest to implement.


‘Pong’

During the first planning phase of the project, I decided I would use the pongy.c program as a starting point for my own code: based on the game ‘Pong’, the similarities with ‘Breakout’ are apparent. Provided in the labs, this code already had many features I could use, like a working display and moving ball and paddle. To achieve these aspects of the game different coding techniques were used, like structs or functions for example. I knew I needed to use these techniques for my own program, so I analyzed how they were used to create the paddle, ball etc.

The structs are essential, as they contain the different values of the objects and allow different data types, like int or double. I would need various structs: for the paddle, the powerups, the ball and the blocks. Some members like the radius, the position and the speed were already stored in the paddle and ball. I would need to add other details in the structs, like the number of lives, or the “strength” of a block. To access a struct member it can be done easily by using the pointer member operator ‘->’.

Functions are also crucial for my code: they allow to break down the program into smaller steps and can be easily called in the main function. I would be needing them for the initialization and drawing of the various objects, the collision detection, the movement, the change of direction, the render, the initialization and others.

Arrays were instead important to conserve the great number of blocks and the powerups coordinates. I also utilized a 2D array for the colours of the various blocks.

The SDL library also needs to be used for the creation of the window. After the placing of the screen and the rendering context, the checking of the user input is also done through SDL. In this case, I would need to utilize the SDL_QUIT (to exit the screen by pressing the x at the top right of the screen), SDL_MOUSEBUTTONDOWN (to click on the different levels using the mouse in the levels selection), SDL_KEYDOWN and SDL_KEYUP (to move the paddle left and right on the screen using the ‘A’ and ‘D’ keys).

OpenGl is used, together with other tasks, to draw paddle, powerups, lives, blocks and ball by using the OpenGl primitives: in this case, the GL_QUADS and GL_TRIANGLE_FAN (for the round objects).

Also, other than SDL and OpenGl, other header files must be included and defined in the code. <stdio.h> and <stdlib.h> are essential, together with <math.h> (for the creation of the round objects using sin and cos), <stdbool.h> (for the Boolean, the true or false, data type) and <time.h> (to include a way to obtain random integers).

Finally, something vital that wasn't included as heavily in the original pongy.c program was the switch case: the game features 4 different screens (the level selection, the actual game, the win and lose windows). To, in fact, switch between these different screens a number between 0 and 3 is assigned to each different case and can be called using a variable, which I called "shownScreen". This allows us to easily change between these 4 different screens in a logical way (for example the lose window is shown when the player has finished their lives).

IMPLEMENTATION:

I will now present the gradual steps I took to advance in the creation of the code. Initially, the steps I set for myself were:
1)Adding collision detection
2)Add destroyable blocks with different strength
3)Adding lives
4)Adding the powerups that give extra lives
5)Adding the levels and different screens
To achieve these objectives, I had to overcome different and smaller tasks, which I listed from a logical and chronological standpoint:

-**ADAPTING THE PONGY.C PROGRAM**
Firstly, I needed to modify the starting program to fit my needs. Things like changing the name of the window, changing the position and size of the paddle, changing the speed and direction of the paddle, getting rid of the second paddle and changing the colors of the objects to green were easily done by changing some parameters in the already existing code. I also added a rudimentary bounce when the ball hit the paddle, but it would need refinement as sometimes the ball would get stuck inside the paddle.

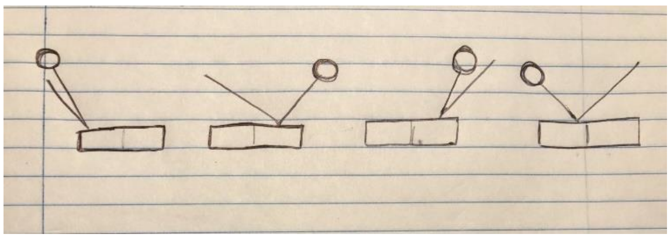
-**MAKING THE BALL ROUND**
Initially the ball was represented by a small square: to change the shape I utilized a code present in the lectures to create a round structure. Using the GL_TRIANGLE_FAN OpenGl primitive and the previously defined PI, sin and cos I was able to create a round ball formed by various triangle segments.

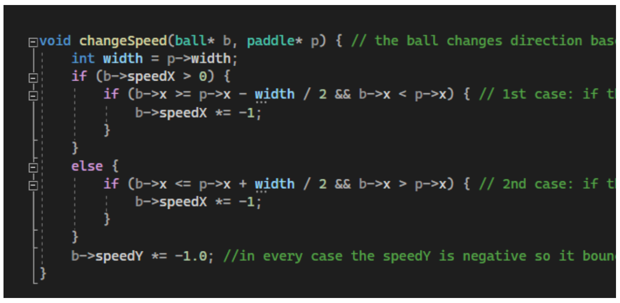
-**CREATING THE FIRST BLOCK**
To draw the first block, it was done simply by adapting the paddles creation but without the speed or movement. This was done by adapting the paddles struct, the collision detection, the drawing function and the initialization. Then, some parameters like size and position were changed to create the first block (still without the ability to bounce back the ball or to disappear after a collision).

-**IMPROVING THE BOUNCE OF THE BALL WITH OTHER OBJECTS**
As stated before, the collision detection between the paddle and the ball was already present, but it didn't count for the x axis: in fact, a bug made it so the ball would bounce before reaching the bottom of the screen, because it had reached the y axis of the paddle, and thus bouncing back. By adding the x in the collision detection, the bug was solved. This new function was also added to the block. Another problem was that the ball would collide with the same object more than once,

sometimes even going through the paddle. To obviate this a new condition was added in the updateBall function, making it so the ball would bounce back immediately after just one hit (this is where the Boolean type was needed).

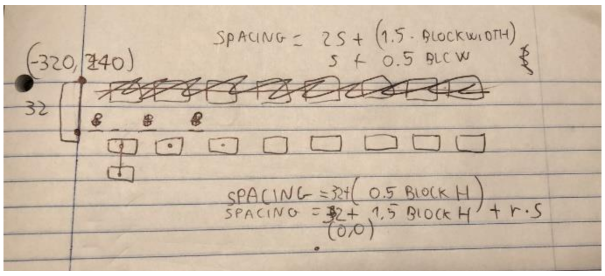
-MAKING THE BOUNCE WITH OBJECTS REALISTIC (code inspired by my brother, Davide Bressani)
The ball would always bounce back in the same direction, independently of how it hit the paddle. A new function, changeSpeed, was added to make it so the ball would bounce back differently based on how it hit the paddle, as shown in the drawing. For the blocks I had to implement a way so that the ball would bounce back also when it hit the sides: so, I created the changeSpeedBlock function. If the ball hits the block from below or from above the speedY is inverted, while if it hits the sides speedX is inverted. I also encountered a bug that would make it so the ball would go very briefly inside of a block: I wasn't able to find the source of this bug even with debugging tools, but it happens extremely rarely. It might also be a problem with my laptops' speed.

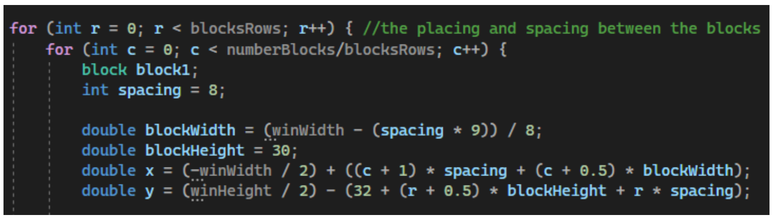




```
void changeSpeed(ball* b, paddle* p) { // the ball changes direction bas
    int width = p->width;
    if (b->speedX > 0) {
        if (b->x >= p->x - width / 2 && b->x < p->x) { // 1st case: if t
            b->speedX *= -1;
        }
    }
    else {
        if (b->x <= p->x + width / 2 && b->x > p->x) { // 2nd case: if t
            b->speedX *= -1;
        }
    }
    b->speedY *= -1.0; //in every case the speedY is negative so it boun
}
```

-CREATING THE BLOCK ARRAY
To store the great number of blocks an array had to be created. The blocksArray keeps track of the number of blocks in a level: later this number will change based on the level that was selected. Also, two for loops were created to place all the blocks in the correct position using the number of rows and columns. While setting the number of the column, of the row and the spacing between the blocks a simple formula is used to calculate the number of the pixel on which the block will be placed. Knowing the width of the screen was necessary to do so, while the width of the blocks was generated via the formula. A blank space was created at the top of the screen so that the lives could be shown on the top right.


SPACING = 2S + (1.5 · BLOCKWIDTH)
S + 0.5 BLC W
(-320, 240)
32
8
8
8
SPACING = 32+(0.5 BLOCK H)
SPACING = 32+ 1.5 BLOCK H + r·S
(0,0)



```
for (int r = 0; r < blocksRows; r++) { //the placing and spacing between the blocks
    for (int c = 0; c < numberBlocks/blocksRows; c++) {
        block block1;
        int spacing = 8;

        double blockWidth = (winWidth - (spacing * 9)) / 8;
        double blockHeight = 30;
        double x = (-winWidth / 2) + ((c + 1) * spacing + (c + 0.5) * blockWidth);
        double y = (winHeight / 2) - (32 + (r + 0.5) * blockHeight + r * spacing);
```

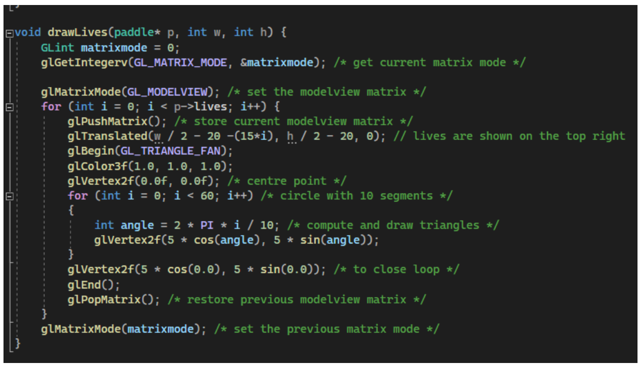
-MAKING THE BLOCKS DISAPPEAR ON COLLISION
Another member would need to be added to the block struct, the "destroyed". Since it's a Boolean data type, it can be made so the blocks are drawn or get in contact with the ball only if the "destroyed" condition is false. If a block is hit by the ball, the "destroyed" becomes true and the block disappears. This condition also needed to be appended to the blocksArray.

-THE DIFFERENT STRENGTH AND COLOURS OF THE BLOCKS
To create different types of blocks a "strength" member was assigned to the block struct. I initially opted for each block to have a random strength, but since the levels needed to be hardcoded I

decided to make it so the blocks would have an increasing strength based on the number of the row they sit in: they get stronger as they get closer to the paddle. The blocks will also change colour based on the strength: I utilized a 2D array containing the various RGB values, so that each level of strength is represented by a different colour: red for 5, orange for 4, yellow for 3, green for 2 and blue for 1.

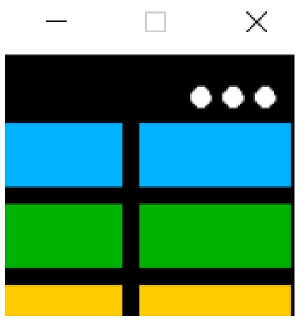
-THE LIVES SYSTEM AND HOW IT'S SHOWN ON SCREEN

The lives were assigned to the paddle struct, as it represents the player. When initialized, the lives are set to 3. Next, the collision and bounce with the boundaries of the screen, which were already implemented in the pongy.c program, was modified so that if the ball hit the bottom of the screen the "restart" if statement present in the updateBall function would be set to true. The "restart", if set to true, makes it so the ball and the paddle are put back to their initial position and a life is removed. Initially if the lives arrived at zero the game would close, but this feature was only temporary before the win and lose windows were added. Finally, to show the lives on screen a drawLives function was added, where based on the number of lives left some white balls would remain on the top right of the screen, unless a life is lost. The spacing between these balls is calculated with an operation and placed using glTranslated.



```c
void drawLives(paddle* p, int w, int h) {
    GLint matrixmode = 0;
    glGetIntegerv(GL_MATRIX_MODE, &matrixmode); /* get current matrix mode */

    glMatrixMode(GL_MODELVIEW); /* set the modelview matrix */
    for (int i = 0; i < p->lives; i++) {
        glPushMatrix(); /* store current modelview matrix */
        glTranslated(w / 2 - 20 -(15*i), h / 2 - 20, 0); // lives are shown on the top right
        glBegin(GL_TRIANGLE_FAN);
        glColor3f(1.0, 1.0, 1.0);
        glVertex2f(0.0f, 0.0f); /* centre point */
        for (int i = 0; i < 60; i++) /* circle with 10 segments */
        {
            int angle = 2 * PI * i / 10; /* compute and draw triangles */
            glVertex2f(5 * cos(angle), 5 * sin(angle));
        }
        glVertex2f(5 * cos(0.0), 5 * sin(0.0)); /* to close loop */
        glEnd();
        glPopMatrix(); /* restore previous modelview matrix */
    }
    glMatrixMode(matrixmode); /* set the previous matrix mode */
}
```



-PLACING THE POWERUPS INSIDE BLOCKS (code inspired by my brother Davide Bressani)

The idea is that powerups will randomly spawn when a block is destroyed, they will fall towards the bottom and grant an extra life if they are touched by the paddle. Firstly, a new powerup struct was created and inserted inside the block struct. An array is instead needed to store the number of the block from which the powerups will spawn from. Another function, the "appendNoDuplicates", makes it so only one powerup can spawn from one block. Finally, if the spawn condition is set to true the powerups will be initialized, and they will look like the lives on the top right of the screen.

-THE POWERUPS FALL DOWN

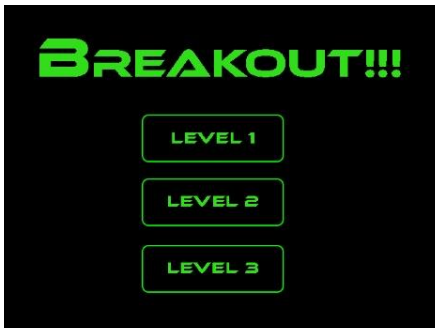
The powerup initialization function sets the speed to -60, so that the powerup falls towards the bottom since it's a negative speed. Also, the "destroyed" Boolean condition was added to the powerups struct.

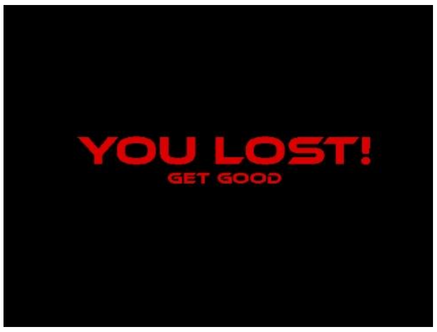
-THE POWERUPS TOUCH THE PADDLE AND GIVE AN EXTRA LIFE

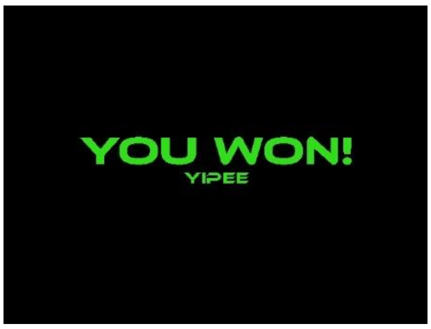
If the "destroyed" is set to true, it means that the powerup has touched the paddle, and similarly to the functioning of the blocks, it will disappear. Finally, if the "destroyed" is set to true an extra life will be added and will appear in the top right of the screen.

-THE SWITCH CASE FOR THE DIFFERENT SCREENS

As stated earlier, a switch case is necessary for the change between the various screens. After assigning the variable shownScreen to the switch, this variable can be changed to show a different window. Initially I had created a loadMedia function that would load a bmp image directly on the screen, but that would only function on Windows and not on Linux. So, thanks to Eike Andersons assistance, I was able to implement two different functions: renderImage, which creates a quad OpenGl primitive that covers the entire screen, and createTexture, which projects the bmp images as a texture on the quad. Unfortunately, a problem occurred with the "you lost" screen, as it displayed a completely opposite RGB value from the actual image: it was blue instead of red. To solve this problem, I simply changed the colour of the image to blue so that it would display as red. Case 0 initializes the menu. For now, this screen is not very useful, as it shows the level selection which has not been implemented yet. The main game is present in case 1, while case 2 and 3 use renderImage and createTexture again for the loss and win screens. The shownScreen variable also changes based on what happens in the game: for the 'you lose' window to appear the number of lives must have reached zero, while for the 'you win' one the variable blocksDestroyed must match the actual number of blocks at the start of the game. The lost and win screens will appear for 3 seconds (or 3000 milliseconds in this case), and then the player will be redirected to the level selection menu. This was done initially by using the Sleep command, but further investigation revealed that c doesn't allow the sleep command to be used, and it only worked because Windows doesn't follow his own programming language guidelines, thus allowing illegal syntax to be used. This is the same reason why the loadMedia function couldn't work on Linux. The Sleep command was thus changed to the SDL_delay command.







These images were made in Photoshop

-**THE LEVELS SYSTEM AND MENU**

Other switch cases are used for when the paddle keys are pressed, but also when a level needs to be selected. Firstly, I found the level buttons coordinates in Photoshop and by using the SDL_MOUSEBUTTONDOWN command I made it so that if the player left clicks on the right position he's redirected to the chosen level. All the level options activate the shownScreen number one, thus redirecting the player to the game. The difference between the levels comes from the init function, which I will analyze in the next section.

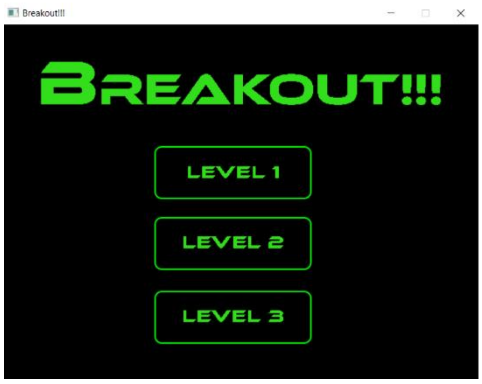
-**THE INITIALIZATION AFTER A LOSS OR A WIN**

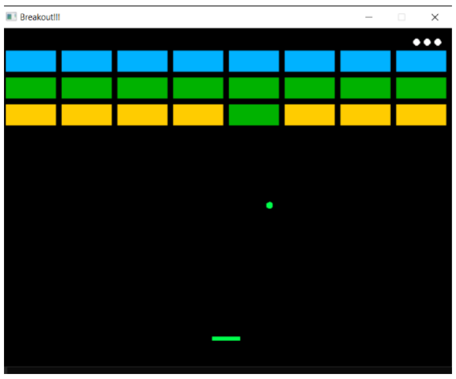
The init function is where all the other initialization functions are contained. This makes it so that if the init function is called the game is reset after a win or a loss. Additionally, some of these functions' parameters are the number of blocks and number of rows: this means that, since the init function is used in the level selection, I was able to decide the number of blocks in each of the 3 levels. 24 blocks for the first one, 32 for the second and 40 for the third so that the levels have an increasing level of difficulty.

To summarize all these steps, I listed the structs, functions etc., their use and when they're utilized:
-**Header files**: SDL, OpenGl, stdio.h, stdlib.h, math, stdbool, time
Essential for the functioning of the program
-**Structs**: paddle, powerup, ball, block
For storing the values of the different objects
-**initializePaddle**, **initializePowerup**, **initializeBall**, **initializeBlock** functions:
They give a value to the structs parameters, initializePowerup is contained in initializeBlock
-**powerupXpaddle** function:
The collision detection between powerup and paddle
-**updatePowerup** function:
Function that gives an extra life when the powerup is destroyed
-**updatePaddle** function:
Calculates the next position and makes sure the paddle doesn't go beyond the boundaries
-**ballXpaddle**, **ballXblock** functions:
Collision detection between the ball and the paddle and blocks
-**changeSpeed**, **changeSpeedBlock** functions:
They cause the change of direction of the ball when it hits the paddle or a block
-**updateBall** function:
Collision detection with boundaries, restart activates if the ball hits the bottom, contains ballXpaddle, changeSpeed, ballXblock, changeSpeedBlock functions to update the strength of the blocks and to make sure the ball hits objects only once
-**drawBall**, **drawPaddle**, **drawBlock, drawLives** functions:
To draw the objects on screen using OpenGl (the powerups are included in the drawBlock)
-**render** function:
Renders the objects on screen, contains the 'draw' functions
-**renderImage, createTexture** functions:
Create a quad on screen and projects the images in a bmp format as a texture
-**appendNoDuplicates** function:
So that the powerups don't end up in the same block
-**init** function:
Initializes the game. Contains appendNoDuplicates, initializeBlock, initializeBall, initializePaddle functions
-**main**:
Contains the creation of the window and the main switch case. The switch case contains the renderImage and createTexture functions for the screens, while for the actual game the updatePowerup, updatePaddle, updateBall and render functions are used.

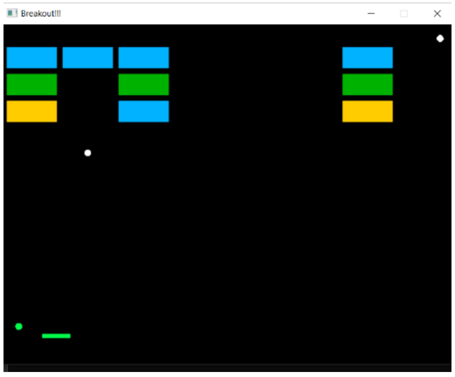
RESULTS:

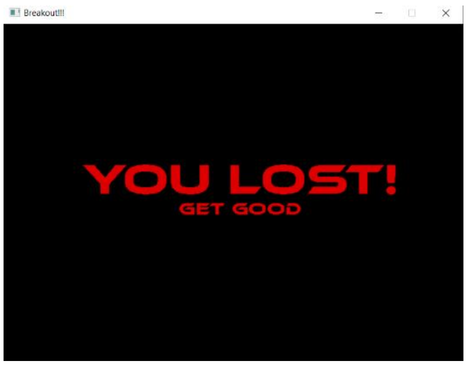
When the game is booted up for the first time it should show a level selection screen. Left clicking on level 1 starts the game: 24 blocks of various strengths (symbolized by their colour) need to be destroyed to win. The paddle is controlled by the player and is used to bounce back a ball against the blocks, destroying them. 3 lives are given to the player, as seen in the top right of the screen.



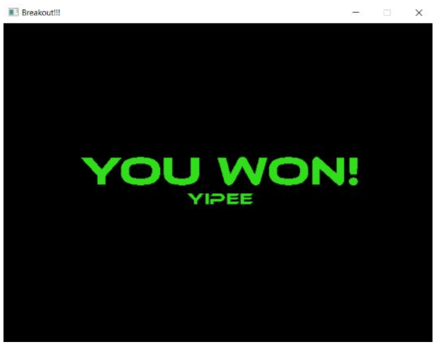


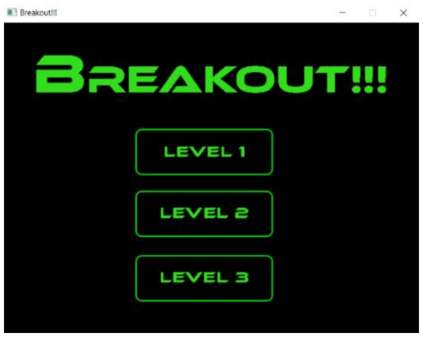
2 random blocks will spawn a powerup once destroyed. The powerup is a ball, resembling the lives, that falls towards the paddle. If the paddle touches the powerup the player receives an extra life. Instead, the player loses a life when the ball passes the paddle and touches the bottom of the screen. When all the lives are lost the lost screen will appear for 3 seconds and then the player will be redirected to the level selection menu.



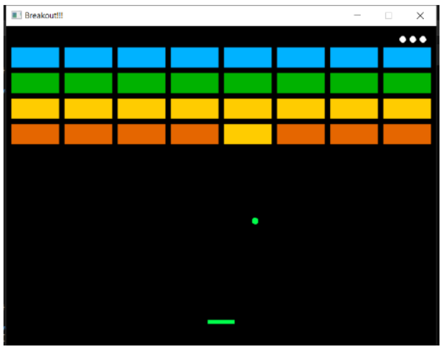


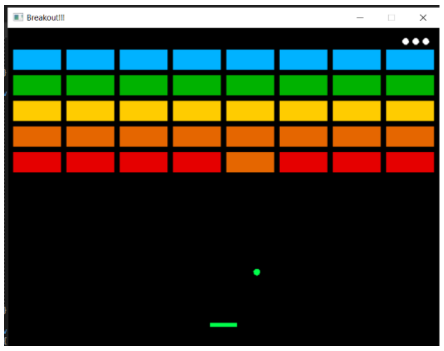
If the player succeeds in destroying all the blocks the win screen will appear for 3 seconds before the player is redirected to the level selection menu again.





Level 2 will have 32 blocks and 3 powerups, while level 3 will have 40 blocks and 4 powerups. This is so the game has an increasing level of difficulty. To exit the game the only thing necessary is to press the x button on the top right of the screen.





References:
1)Breakout Atari image, Available from: https://spectrum.ieee.org/atari-breakout#toggle-gdpr
2)Pong image, Available from: https://it.wikipedia.org/wiki/Pong
3)Eike Anderson, lab 9, lecture 17, lecture 12, Available from: https://brightspace.bournemouth.ac.uk/d2l/le/content/300113/viewContent/1757210/View
4)Davide Bressani, Computer Science student at York University, general assistance
5)Boolean type header file in c, Available from: https://www.educative.io/answers/what-is-boolean-in-c
6)Generating a random number in c, Available from: https://www.geeksforgeeks.org/generating-random-number-range-c/
7)Collision detection, Available from: https://happycoding.io/tutorials/processing/collision-detection
8)Loading images with SDL, Available From: https://wiki.libsdl.org/SDL_image/IMG_Load
9)SDL mouse commands, Available from: https://stackoverflow.com/questions/35165716/sdl-mouse-click
10)Sleep function (on Windows), Available from: https://stackoverflow.com/questions/3379139/sleep-function-in-windows-using-c
11)Resizing an array in C, Available from: https://stackoverflow.com/questions/12917727/resizing-an-array-in-c
12)Rgb colour wheel, Available from: https://www.colorspire.com/rgb-color-wheel/