Tilt-tris Game Machine

ELEC3607 Milestone IV Report

Group 28
Alvin Ho - alho5785
James Giouroukelis - jgio2213

Abstract:
This project entails the development of an implementation of the classic game ‘tetris’. This was completed an arduino ‘DUE’ board connected to four 8x8 led matrices. The game is played in the lower 3 matrices, with the next block to spawn and the score being displayed in binary in the top matrix. The controls for this game are handled by a 3D accelerometer. The user is able to move their block left and right with the accelerometer and in addition to this, forward tilt rotates the falling block and backward tilt speeds up this block’s fall. The score displayed is calculated as the number of lines cleared. The project was heavily tested via separating sections of the code in order to desk check and playtest both in isolation and with other components of the code. This was done in order to ensure complete functionality. Some extensions to this project could consist of it having a user interface with a main menu, implementing some sort of multiplayer and having the ability to send scores to other devices via bluetooth.

Aim:

Our project aim is to create an embedded system that allows a user to play the classic game tetris on an LED matrix, using a 3D accelerometer for controls. This will be executed by tilting the accelerometer to a certain direction (i.e. left, right, away from user, towards user), which will move the falling block accordingly. Said user will also be able to see their score in binary at the top of the system in addition to the next block that will spawn. Additionally, the game difficulty will increase by decreasing the amount of time the player has to place the 'dropping' block as the game progresses. This system structure reflects the traits of an embedded system as it consists of both hardware and software, as well as a fixed set of tasks that our system needs to be able to complete.

Goals:

- Create an embedded system to simulate the game 'tetris'
- Be able to control said game
- Increase the difficulty of the game as it goes on
- Game state goes between playing the game and displaying the score

Objectives:

- Successful use of the LED dot matrices as a display for the game
- Responsive and intuitive controls using a 3D accelerometer
- Display game results in real time
- Decrease the amount of time the player has to place a block the longer the game goes
- Once game is finished, display only the score in binary for 7.5 seconds

Design:

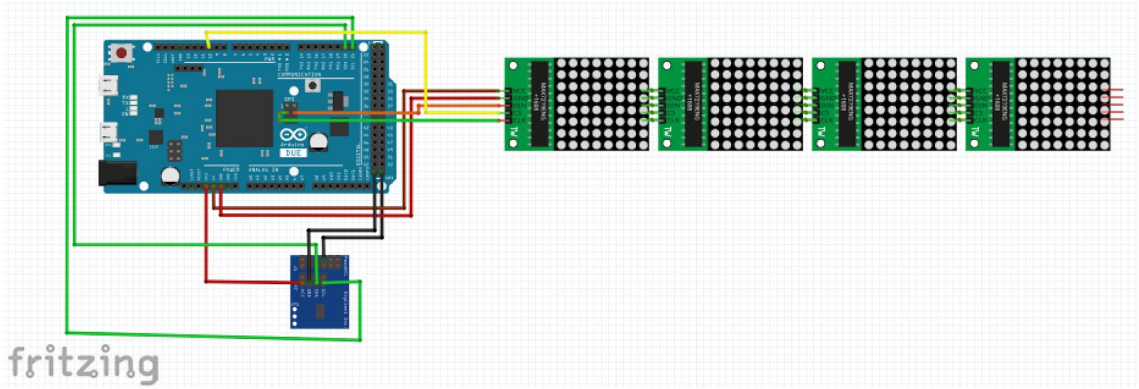
Parts included are:

- 1x Arduino Due (SAM3X)
- 1x 3D Accelerometer (Pmod ACL)
- 4x LED dot matrix connected with microcontrollers (MAX7219)
- Wires

Description of the embedded system is explained by exploring each major component as follows:

Schematic diagram - Refer to page 11 in appendix

Layout of the embedded system:



LED Dot Matrix:

The LED matrix is used for the main display. It shows the game state, score (in binary) and also the next tetris block to spawn. The use of these LED matrices is supported by the MAX7219 microcontroller which connects each of the 4 LED matrices to one another. This is done by acting as an SPI hardware interface.By utilising this microcontroller, it significantly reduced the amount of power and pins needed while also reducing the complexity in controlling the combined display.

The connection from LED matrix to Due based on wire colour is as follows:

Connection from (LED Matrix)	Connection to (Due)
VCC	5V
GND	GND
DIN	MOSI
CS	Pin 10 (SS)
CLK	SCK

Controlling LED Matrix:

With the help of the 'MD_MAX72xx' library, we were able to control each individual LED to simulate the game tetris, using the built-in methods within the said library. These methods include:

- setPoint: updates the state of an LED at a given coordinate (on if true, off if false)
- getPoint: verifies the state of an LED at a given coordinate (returns true if on, false if off)
- clear: clears all the display data on all the display devices
- update: updates the physical state of the LED matrices after any change

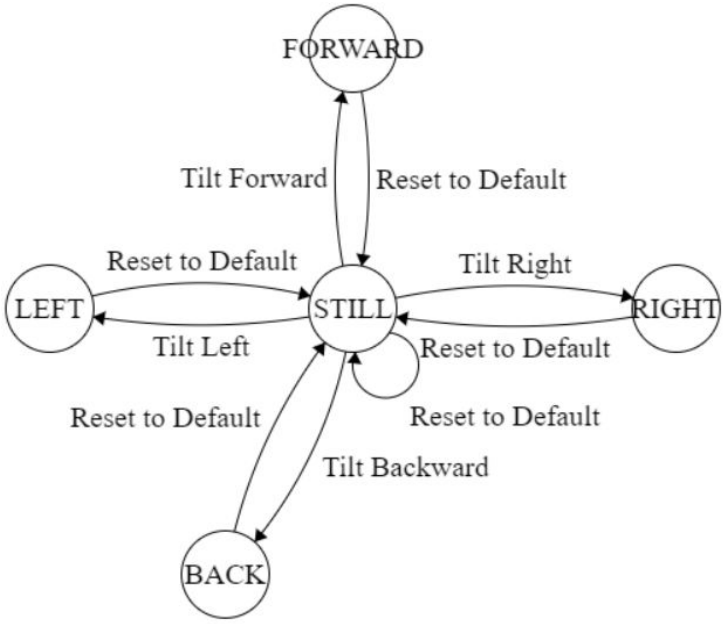
3D Accelerometer:

The 3D accelerometer was utilised in order to control the movement of the tetris blocks in the game. Raw values retrieved were in the range of 0 to 2^16 (65536) for each of the X, Y and Z axis. This created problems as the values being read were sensitive, which added difficulties in translating these values into meaningful data to result in a direction. However with the use of the built in SPI and Wire libraries for I2C communication and bit operations, the values received were easily translated and separated into five distinguishable states.

These states are:

- STILL: falling tetris block would not be affected (continue to fall), accelerometer is in a flat, default state and not being tilted in any direction.
- FORWARD: rotates the falling tetris block clockwise when the accelerometer is tilted within 45 to 90 degrees forward
- BACK: moves the falling tetris block downwards faster when when the accelerometer is tilted within 45 to 90 degrees backward
- LEFT: moves the falling tetris block towards the left by one LED when the accelerometer is tilted within 45 to 90 degrees to the left
- RIGHT: moves the falling tetris block towards the right by one LED when accelerometer is tilted within 45 to 90 degrees to the right

Which is represented by the FSM below:



The connection from 3D accelerometer to Due based on wire colour is as follows:

Connection from (Pmod ACL accelerometer)	Connection to (Due)
VCC	3.3V
GND	GND
MISO	GND
SDA	Pin 20 (SDA)
SCL	Pin 21 (SCL)

Pseudocode:

Import libraries
Declare accelerometer variables
Declare and allocate memory for all tetris blocks

Setup:
Start the microcontroller
Draw end of game line
Setup accelerometer
Set up and allocate memory for all rotations for every tetris block

Declare and allocate memory for a current block and a next block
Declare other global variables as needed

Gameplay State (Loop):
Check if any blocks are high enough to trigger game end
If game end is triggered, clear the board and display score (see calculation below) for 7.5s
Then reset all global variables to start game again along with redrawing the end game line
Convert the current score (which is the number of lines cleared) to binary, and light up pixels on line 6 to represent this

If this is the first iteration of the game, create initial 'next block'
Pick a random block from the linked list of blocks
Set the current block to be 'next block'
Randomise the 'next block'

Clear the top left corner of the screen
Display the 'next block' in this same section

Set leftmost column of the block to be 3 by default(Determines where the block is dropping from)
Set a delay_value to be 500ms
Iterate from row 8 (top of game board) to row 32 in order to move the dropping block downwards from the top of the screen to the bottom

Check if any lines need to be cleared (i.e. one or many full rows)
If they exist, then remove them and shift relevant lines down to fill up space

Connect the accelerometer to the due
Poll to determine when wire is available, and take values as required
Separate accelerometer data into x and y values using bitwise operations
Convert these values into a meaningful direction and set current direction to be this

If the direction was right, then check if moving the right-most LED's of the block right by one LED will cause a collision with other blocks or go out of bounds
If the move was allowed, then increment the column the block is dropping from by one

If the direction was left, then check if moving the left-most LED's of the block left by one LED will cause a collision with other blocks or go out of bounds
If the move was allowed, then decrement the column the block is dropping from by one
If the direction was back, set a 'BACK' variable to be true to deal with this later
If the direction was forward then remove the block from the display (but don't update the microcontroller to represent this yet)
After the block is removed, the block's orientation is then rotated
Attempt to redraw this new block on the display, If there any collisions or out of bounds behaviour, then make a note of this and 'break'
If the block could not be redrawn successfully, then reverse the block to its previous orientation
Redraw the block on the display

Now check for a downward collision - if a collision has occurred, then delay for 1 second and break out of the big 8 to 32 loop

If the BACK variable was true, set the delay_value to be 20ms, else set it to 500ms

Now increment the row of the block by 1 (It is incrementing from 8 to 32) in order to drop it by one LED
Redraw the end of game line
Delay the game by the delay_value minus the amount of total lines cleared times ten
Update the display

Problems encountered:
Throughout the duration of the project, we encountered some problems revolving around the game machine, which were as follows:

- Not enough or inconsistent power channeled from the Due with the old design, resulting in the change of the combined LED matrix module
- Translating sensitive accelerometer values to intuitive directions
- Software bugs as our project is more demanding in its software/code side

Testing:
In order make measurable progress and eventually reach a polished, working product, it was vital that we employed a wide range of testing strategies to ensure that no bugs were present in our code and that everything was working as intended. The main strategy we employed for each of the testing methods was to test relevant sections of the code in isolation in order to ensure they worked correctly before moving on. For example, testing that gravity works, then testing that blocks can move left/right, then testing those in unison was the progression of our testing. This was in order to modularise our code as it makes bug fixing much simpler and significantly improves code readability.

Desk Checking:
We put pen to paper in order to test the outcome of various sections of code. This is useful, as writing out the states of all relevant variables helps to find mistakes that were previously missed and fix code that was otherwise thought to be correct.

Playtesting:
We played our tetris game whenever we introduced a new block of code in order to ensure that it works properly on its own and in unison with other parts of the game.

Comparison against other implementations:
We ensured that some of the fringe case interactions of our game (e.g. slotting a falling block) work as intended by replicating other implementations of tetris that exist.

Results/Conclusion:
To conclude, we successfully managed to satisfy our aim of creating an embedded system that allows a user to play the classic game tetris on an LED matrix, using a 3D accelerometer for controls. Though the schematic and electrical setup were not the most sophisticated, the code required to complete this task consisted of a steep learning curve and was quite difficult to complete. Though, as a result of completing this project, our teamwork skills have improved and we gained invaluable experience in regards to all aspects of the creation of an embedded system.

Extensions:
Some future improvements to this project could consist of it having a user interface with a main menu, implementing networked multiplayer capabilities if there were two copies of the system, as well as local turn-based multiplayer, and finally having the ability to send scores via bluetooth to other devices.

Appendix:

Project Implementation Summary Form

Category	List of items, with web link or page number where item is described
Open source compliance Are you publishing your code as open source? Explain what license you are using, or link to where you have published your project.	https://github.com/JamJamJames98/arduino-tetris-accel - Using open source license Apache Version 2.0 - Report is uploaded along with separate schematics screenshots as well as the code
Platforms List the platforms you have used, including processor architecture (ARM, AVR, x86, etc.), programming languages (Python, C, C++, etc.), and IDEs. If you are not using Arduino Due, justify your choice of platform.	Platform: - Arduino DUE with an ARM core microcontroller architecture Programming Language: - C (Built in Arduino IDE)
Sensors and inputs List your hardware inputs (push buttons, analog sensors, I2C or SPI sensors including accelerometers, etc.) and mention on which page each input is described.	Input: - 3D accelerometer (Pmod ACL), described on page 6 & 7 - LED Dot Matrix Microcontroller (MAX7219) as an SPI Hardware Interface, described on page 5
Outputs List any mechanical, visual or other type of outputs controlled by your hardware (motors, magnets, LED, 7-Segs, LCD, etc.) and mention on which page each input is described.	Output: - 4 8x8 LED Dot Matrices connected using microcontroller MAX7219, described on page 5
Connectivity List any protocols used for communicating between your main controller and any other microcontroller excluding sensors or actuators (USART, UART over USB, Bluetooth 2, Bluetooth 4, Wi-Fi, ZigBee, etc.) and mention on which page each input is described.	Use of MAX7219 microcontroller to connect each of the LED matrices from one to another and to Arduino Due (Refer to pages 12 and 13)

Graphical User Interface List any GUI used for interfacing users (local LCD, Phone app, Desktop app, Web app, Web dashboard, etc.) and mention on which page each input is described.	Our users are interfaced as they are either presented with a screen to play the game on, or a screen representing the score of the last played game in an infinite loop (FSM). (Refer to pages 6 and 7 for pseudocode)
Algorithms / Logic List substantial control algorithms (state-machines, real-time operating system, filesystems, feedback algorithms, mathematical transformations, etc.) and mention on which page each input is described.	State-machines: - States between gameplay and score display, described on pages 6 and 7 - States to control the game based on direction, described on pages 6 and 7 Mathematical transformations: - Bitwise operations to read values from accelerometer and also to convert score from decimal to binary, described on pages 6 and 7 (Refer to pages 6 and 7 for pseudocode)
Physical case and other mechanical consideration List what casing or mechanical systems have been used in your project (3d prints, pipes, cardboard mechanics, etc.) and mention on which page each input is described.	No physical case or other mechanical considerations

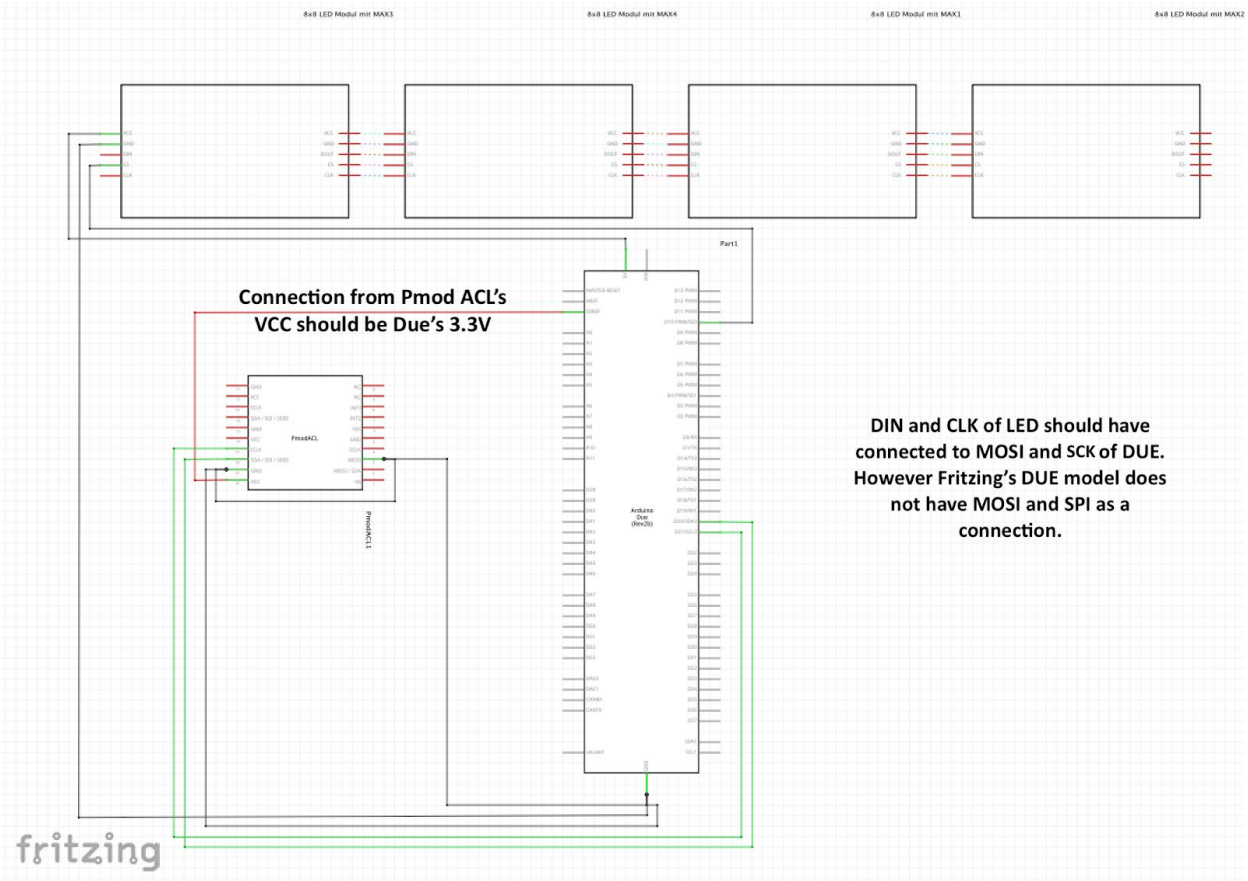
References (Used only for code):

Arduino - SPI. (2018). Retrieved from https://www.arduino.cc/en/Reference/SPI

Arduino - Wire. (2018). Retrieved from https://www.arduino.cc/en/reference/wire

Migliacio, M., & Wong, A. (2018). Using the Pmod ACL with Arduino Uno. Retrieved from https://www.hackster.io/56491/using-the-pmod-acl-with-arduino-uno-35d564

Schematic Diagram:



Full Code:

```
//Copyright to use this implementation
//Open Source Initiative License
/*
 *    Copyright 2018 James Giouroukelis

   Licensed under the Apache License, Version 2.0 (the "License");
   you may not use this file except in compliance with the License.
   You may obtain a copy of the License at

     http://www.apache.org/licenses/LICENSE-2.0

   Unless required by applicable law or agreed to in writing, software
   distributed under the License is distributed on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
```

```
   See the License for the specific language governing permissions and
   limitations under the License.
 *
 */


//Copyright for MAX72xx library
/*
 * # MAX72xx LED Matrix Display Library

The library implements functions that allow the MAX72xx to be used for LED
matrices,
allowing the programmer to use the LED matrix as a pixel device,displaying
graphics elements much like any other pixel addressable display.
The library still retains flexibility for device level control, should the
developer require, through the use of overloaded class methods.
[Library Documentation](https://majicdesigns.github.io/MD_MAX72XX/)

Copyright
Copyright (C) 2012-18 Marco Colli. All rights reserved.
This library is free software; you can redistribute it and/or modify it
under the terms of the GNU Lesser General Public License
as published by the Free Software Foundation; either version 2.1 of the
License, or (at your option) any later version.
This library is distributed in the hope that it will be useful, but WITHOUT
ANY WARRANTY;
without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE. See the GNU Lesser General Public License for more
details.
You should have received a copy of the GNU Lesser General Public License
along with this library;
if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth
Floor, Boston, MA 02110-1301 USA

*
*/
#include <MD_MAX72xx.h>


/*
 * these are built in imports, no copyright/use terms are needed
 */
```

```
//https://www.arduino.cc/en/Reference/SPI
#include <SPI.h>

//https://www.arduino.cc/en/reference/wire
#include <Wire.h>

#define MAX_DEVICES 4
#define CLK_PIN   ICSP-3
#define DATA_PIN  ICSP-4
#define CS_PIN    10

MD_MAX72XX mx = MD_MAX72XX(CS_PIN, MAX_DEVICES); // SPI hardware interface

//ACCELEROMETER CODE
// code taken from
//https://www.hackster.io/56491/using-the-pmod-acl-with-arduino-uno-35d564
// Declaration of the address of the module
#define ADXL345_Adresse 0x53 // ADXL345 address
#define POWER_CTL 0x2D // Power Control register
#define DATA_FORMAT 0x31 // Data Format register
#define DATAX0 0x32 // LSB axe X
#define DATAX1 0x33 // MSB axe X
#define DATAY0 0x34 // LSB axe Y
#define DATAY1 0x35 // MSB Y
#define DATAZ0 0x36 // LSB axe Z
#define DATAZ1 0x37 // bMSB Z

// Configuration of the module
#define ADXL345_Precision2G 0x00
#define ADXL345_Precision4G 0x01
#define ADXL345_Precision8G 0x02
#define ADXL345_Precision16G 0x03
#define ADXL345_ModeMesure 0x08

byte buffer[6]; // storage of data of the module
int i = 0;
double xValue;
double yValue;

//node struct for each block
struct node {
```

```
  int height;
  int width;
  struct node* next;
  struct node* prev;
  struct orientation* shape; // current shape of the block
};

//orientation struct, allows each node to rotate
struct orientation {
  int arr[10][10]; // this is the new shape->arr, this is the array to
render so: curr->shape->arr[][] instead of shape->arr[][]
  int arr_height; // height of current shape
  int arr_width; // width of current shape
  struct orientation* rotate; // this is a next, so to go to next shape we
call: curr->shape->rotate (THIS IS ALREADY DONE IN FORWARD OPERATION)
  struct orientation* reverse; // reverse rotation
  struct orientation* default_shape; // spawning orientation
};

//all our blocks
struct node* block_4 = (node*)malloc(sizeof *block_4);
struct node* block_2_2 = (node*)malloc(sizeof *block_2_2);
struct node* block_3_1 = (node*)malloc(sizeof *block_3_1);
struct node* block_Z = (node*)malloc(sizeof *block_Z);
struct node* block_S = (node*)malloc(sizeof *block_S);
struct node* block_L = (node*)malloc(sizeof *block_L);
struct node* block_J = (node*)malloc(sizeof *block_J);

void setup() {
  //start up the combined led matrix controls
  mx.begin();
  mx.update(true);
  draw_end_line();

  //ACCELEROMETER SETUP
  // code taken from

//https://www.hackster.io/56491/using-the-pmod-acl-with-arduino-uno-35d564
  Serial.begin(9600); // initialization of serial communication
  Wire.begin (); // initialization of I2C communication
  Wire.beginTransmission (ADXL345_Adresse); // configuration of the module
  Wire.write (DATA_FORMAT);
```

```
  Wire.write (ADXL345_Precision4G);
  Wire.endTransmission ();
  Wire.beginTransmission (ADXL345_Adresse);
  Wire.write (POWER_CTL);
  Wire.write (ADXL345_ModeMesure);
  Wire.endTransmission ();

   /*
    *
    * NEW BLOCK
    *
    */

  // block 4
  block_4->height = 4;
  block_4->width = 1;
  block_4->next = block_2_2;
  block_4->prev = block_J;

  // 2 different orientations of block_4
  /*
   * [[1],
   *  [1],
   *  [1],
   *  [1]]
   */
  struct orientation* block_4_shape_1 =
(orientation*)malloc(sizeof(orientation));
  block_4_shape_1->arr_height = 4;
  block_4_shape_1->arr_width = 1;
  for (int i = 0; i < block_4_shape_1->arr_height; i++) {
    for (int j = 0; j < block_4_shape_1->arr_width; j++) {
      block_4_shape_1->arr[i][j] = 1;
    }
  }

  /*
   * [[1, 1, 1, 1]]
   */
  struct orientation* block_4_shape_2 =
(orientation*)malloc(sizeof(orientation));
  block_4_shape_2->arr_height = 1;
```

```
  block_4_shape_2->arr_width = 4;
  for (int i = 0; i < block_4_shape_2->arr_height; i++) {
    for (int j = 0; j < block_4_shape_2->arr_width; j++) {
      block_4_shape_2->arr[i][j] = 1;
    }
  }

  // assigning circular dependency
  block_4_shape_1->default_shape = block_4_shape_1;
  block_4_shape_2->default_shape = block_4_shape_1;
  block_4_shape_1->rotate = block_4_shape_2;
  block_4_shape_1->reverse = block_4_shape_2;
  block_4_shape_2->rotate = block_4_shape_1;
  block_4_shape_2->reverse = block_4_shape_1;
  block_4->shape = block_4_shape_1;

  /*
   *
   * NEW BLOCK
   *
   */

  // block 2_2
  block_2_2->height = 2;
  block_2_2->width = 2;
  block_2_2->next = block_3_1;
  block_2_2->prev = block_4;

  // 1 orientation of block_2_2
  /*
   * [[1, 1],
   *  [1, 1]]
   */
  struct orientation* block_2_2_shape_1 =
(orientation*)malloc(sizeof(orientation));
  block_2_2_shape_1->arr_height = 2;
  block_2_2_shape_1->arr_width = 2;
  for (int i = 0; i < block_2_2_shape_1->arr_height; i++) {
    for (int j = 0; j < block_2_2_shape_1->arr_width; j++) {
      block_2_2_shape_1->arr[i][j] = 1;
    }
  }
```

```
  // assigning circular dependency
  block_2_2_shape_1->default_shape = block_2_2_shape_1;
  block_2_2_shape_1->rotate = block_2_2_shape_1;
  block_2_2_shape_1->reverse = block_2_2_shape_1;
  block_2_2->shape = block_2_2_shape_1;

  /*
   *
   * NEW BLOCK
   *
   */

  // block 3_1
  block_3_1->height = 2;
  block_3_1->width = 3;
  block_3_1->next = block_Z;
  block_3_1->prev = block_2_2;

  // 4 different orientations of block_3_1
  /*
   * [[0, 1, 0],
   *  [1, 1, 1]]
   */
  struct orientation* block_3_1_shape_1 =
(orientation*)malloc(sizeof(orientation));
  block_3_1_shape_1->arr_height = 2;
  block_3_1_shape_1->arr_width = 3;
  for (int i = 0; i < block_3_1_shape_1->arr_height; i++) {
    for (int j = 0; j < block_3_1_shape_1->arr_width; j++) {
      if (i == 0 && j == 0 || i == 0 && j == 2) {
        block_3_1_shape_1->arr[i][j] = 0;
      } else {
        block_3_1_shape_1->arr[i][j] = 1;
      }
    }
  }

  /*
   * [[1, 0],
   *  [1, 1],
   *  [1, 0]]
```

```
   */
  struct orientation* block_3_1_shape_2 =
(orientation*)malloc(sizeof(orientation));
  block_3_1_shape_2->arr_height = 3;
  block_3_1_shape_2->arr_width = 2;
  for (int i = 0; i < block_3_1_shape_2->arr_height; i++) {
    for (int j = 0; j < block_3_1_shape_2->arr_width; j++) {
      if (i == 0 && j == 1 || i == 2 && j == 1) {
        block_3_1_shape_2->arr[i][j] = 0;
      } else {
        block_3_1_shape_2->arr[i][j] = 1;
      }
    }
  }

  /*
   * [[1, 1, 1],
   *  [0, 1, 0]]
   */
  struct orientation* block_3_1_shape_3 =
(orientation*)malloc(sizeof(orientation));
  block_3_1_shape_3->arr_height = 2;
  block_3_1_shape_3->arr_width = 3;
  for (int i = 0; i < block_3_1_shape_3->arr_height; i++) {
    for (int j = 0; j < block_3_1_shape_3->arr_width; j++) {
      if (i == 1 && j == 0 || i == 1 && j == 2) {
        block_3_1_shape_3->arr[i][j] = 0;
      } else {
        block_3_1_shape_3->arr[i][j] = 1;
      }
    }
  }

  /*
   * [[0, 1],
   *  [1, 1],
   *  [0, 1]]
   */
  struct orientation* block_3_1_shape_4 =
(orientation*)malloc(sizeof(orientation));
  block_3_1_shape_4->arr_height = 3;
  block_3_1_shape_4->arr_width = 2;
```

```
for (int i = 0; i < block_3_1_shape_4->arr_height; i++) {
  for (int j = 0; j < block_3_1_shape_4->arr_width; j++) {
    if (i == 0 && j == 0 || i == 2 && j == 0) {
      block_3_1_shape_4->arr[i][j] = 0;
    } else {
      block_3_1_shape_4->arr[i][j] = 1;
    }
  }
}

// assigning circular dependency
block_3_1_shape_1->default_shape = block_3_1_shape_1;
block_3_1_shape_2->default_shape = block_3_1_shape_1;
block_3_1_shape_3->default_shape = block_3_1_shape_1;
block_3_1_shape_4->default_shape = block_3_1_shape_1;
block_3_1_shape_1->rotate = block_3_1_shape_2;
block_3_1_shape_1->reverse = block_3_1_shape_4;
block_3_1_shape_2->rotate = block_3_1_shape_3;
block_3_1_shape_2->reverse = block_3_1_shape_1;
block_3_1_shape_3->rotate = block_3_1_shape_4;
block_3_1_shape_3->reverse = block_3_1_shape_2;
block_3_1_shape_4->rotate = block_3_1_shape_1;
block_3_1_shape_4->reverse = block_3_1_shape_3;
block_3_1->shape = block_3_1_shape_1;

/*
 *
 * NEW BLOCK
 *
 */

// block Z
block_Z->height = 2;
block_Z->width = 3;
block_Z->next = block_S;
block_Z->prev = block_3_1;

// 2 different orientations of block_Z
/*
 *    [[1, 1, 0],
 *     [0, 1, 1]]
 */
```

```
  struct orientation* block_Z_shape_1 =
(orientation*)malloc(sizeof(orientation));
  block_Z_shape_1->arr_height = 2;
  block_Z_shape_1->arr_width = 3;
  for (int i = 0; i < block_Z_shape_1->arr_height; i++) {
    for (int j = 0; j < block_Z_shape_1->arr_width; j++) {
      if (i == 0 && j == 2 || i == 1 && j == 0) {
        block_Z_shape_1->arr[i][j] = 0;
      } else {
        block_Z_shape_1->arr[i][j] = 1;
      }
    }
  }

  /*
   *    [[0, 1],
   *     [1, 1],
   *     [1, 0]]
   */
  struct orientation* block_Z_shape_2 =
(orientation*)malloc(sizeof(orientation));
  block_Z_shape_2->arr_height = 3;
  block_Z_shape_2->arr_width = 2;
  for (int i = 0; i < block_Z_shape_2->arr_height; i++) {
    for (int j = 0; j < block_Z_shape_2->arr_width; j++) {
      if (i == 0 && j == 0 || i == 2 && j == 1) {
        block_Z_shape_2->arr[i][j] = 0;
      } else {
        block_Z_shape_2->arr[i][j] = 1;
      }
    }
  }

  // assigning circular dependency
  block_Z_shape_1->default_shape = block_Z_shape_1;
  block_Z_shape_2->default_shape = block_Z_shape_1;
  block_Z_shape_1->rotate = block_Z_shape_2;
  block_Z_shape_1->reverse = block_Z_shape_2;
  block_Z_shape_2->rotate = block_Z_shape_1;
  block_Z_shape_2->reverse = block_Z_shape_1;
  block_Z->shape = block_Z_shape_1;
```

```
  /*
   *
   * NEW BLOCK
   *
   */

  // block S
  block_S->height = 2;
  block_S->width = 3;
  block_S->next = block_L;
  block_S->prev = block_Z;

  // 2 different orientations of block_S
  /*
   *    [[0, 1, 1],
   *     [1, 1, 0]]
   */
  struct orientation* block_S_shape_1 =
(orientation*)malloc(sizeof(orientation));
  block_S_shape_1->arr_height = 2;
  block_S_shape_1->arr_width = 3;
  for (int i = 0; i < block_S_shape_1->arr_height; i++) {
    for (int j = 0; j < block_S_shape_1->arr_width; j++) {
      if (i == 0 && j == 0 || i == 1 && j == 2) {
        block_S_shape_1->arr[i][j] = 0;
      } else {
        block_S_shape_1->arr[i][j] = 1;
      }
    }
  }

  /*
   *    [[1, 0],
   *     [1, 1],
   *     [0, 1]]
   */
  struct orientation* block_S_shape_2 =
(orientation*)malloc(sizeof(orientation));
  block_S_shape_2->arr_height = 3;
  block_S_shape_2->arr_width = 2;
  for (int i = 0; i < block_S_shape_2->arr_height; i++) {
    for (int j = 0; j < block_S_shape_2->arr_width; j++) {
```

```
      if (i == 0 && j == 1 || i == 2 && j == 0) {
        block_S_shape_2->arr[i][j] = 0;
      } else {
        block_S_shape_2->arr[i][j] = 1;
      }
    }
  }

  // assigning circular dependency
  block_S_shape_1->default_shape = block_S_shape_1;
  block_S_shape_2->default_shape = block_S_shape_1;
  block_S_shape_1->rotate = block_S_shape_2;
  block_S_shape_1->reverse = block_S_shape_2;
  block_S_shape_2->rotate = block_S_shape_1;
  block_S_shape_2->reverse = block_S_shape_1;
  block_S->shape = block_S_shape_1;

  /*
   *
   * NEW BLOCK
   *
   */

  // block L
  block_L->height = 2;
  block_L->width = 3;
  block_L->next = block_J;
  block_L->prev = block_S;

  // 4 different orientations of block_L
  /*
   * [[0, 0, 1],
   *  [1, 1, 1]]
   */
  struct orientation* block_L_shape_1 =
(orientation*)malloc(sizeof(orientation));
  block_L_shape_1->arr_height = 2;
  block_L_shape_1->arr_width = 3;
  for (int i = 0; i < block_L_shape_1->arr_height; i++) {
    for (int j = 0; j < block_L_shape_1->arr_width; j++) {
      if (i == 0 && j == 0 || i == 0 && j == 1) {
        block_L_shape_1->arr[i][j] = 0;
```

```
      } else {
        block_L_shape_1->arr[i][j] = 1;
      }
    }
  }

  /*
   * [[1, 0],
   *  [1, 0],
   *  [1, 1]]
   */
  struct orientation* block_L_shape_2 =
(orientation*)malloc(sizeof(orientation));
  block_L_shape_2->arr_height = 3;
  block_L_shape_2->arr_width = 2;
  for (int i = 0; i < block_L_shape_2->arr_height; i++) {
    for (int j = 0; j < block_L_shape_2->arr_width; j++) {
      if (i == 0 && j == 1 || i == 1 && j == 1) {
        block_L_shape_2->arr[i][j] = 0;
      } else {
        block_L_shape_2->arr[i][j] = 1;
      }
    }
  }

  /*
   * [[1, 1, 1],
   *  [1, 0, 0]]
   */
  struct orientation* block_L_shape_3 =
(orientation*)malloc(sizeof(orientation));
  block_L_shape_3->arr_height = 2;
  block_L_shape_3->arr_width = 3;
  for (int i = 0; i < block_L_shape_3->arr_height; i++) {
    for (int j = 0; j < block_L_shape_3->arr_width; j++) {
      if (i == 1 && j == 1 || i == 1 && j == 2) {
        block_L_shape_3->arr[i][j] = 0;
      } else {
        block_L_shape_3->arr[i][j] = 1;
      }
    }
  }
```

```
  /*
   * [[1, 1],
   *  [0, 1],
   *  [0, 1]]
   */
  struct orientation* block_L_shape_4 =
(orientation*)malloc(sizeof(orientation));
  block_L_shape_4->arr_height = 3;
  block_L_shape_4->arr_width = 2;
  for (int i = 0; i < block_L_shape_4->arr_height; i++) {
    for (int j = 0; j < block_L_shape_4->arr_width; j++) {
      if (i == 1 && j == 0 || i == 2 && j == 0) {
        block_L_shape_4->arr[i][j] = 0;
      } else {
        block_L_shape_4->arr[i][j] = 1;
      }
    }
  }

  // assigning circular dependency
  block_L_shape_1->default_shape = block_L_shape_1;
  block_L_shape_2->default_shape = block_L_shape_1;
  block_L_shape_3->default_shape = block_L_shape_1;
  block_L_shape_4->default_shape = block_L_shape_1;
  block_L_shape_1->rotate = block_L_shape_2;
  block_L_shape_1->reverse = block_L_shape_4;
  block_L_shape_2->rotate = block_L_shape_3;
  block_L_shape_2->reverse = block_L_shape_1;
  block_L_shape_3->rotate = block_L_shape_4;
  block_L_shape_3->reverse = block_L_shape_2;
  block_L_shape_4->rotate = block_L_shape_1;
  block_L_shape_4->reverse = block_L_shape_3;
  block_L->shape = block_L_shape_1;

  /*
   *
   * NEW BLOCK
   *
   */

  // block J
```

```
  block_J->height = 2;
  block_J->width = 3;
  block_J->next = block_4;
  block_J->prev = block_L;

  // 4 different orientations of block_J
  /*
   * [[1, 0, 0],
   *  [1, 1, 1]]
   */
  struct orientation* block_J_shape_1 =
(orientation*)malloc(sizeof(orientation));
  block_J_shape_1->arr_height = 2;
  block_J_shape_1->arr_width = 3;
  for (int i = 0; i < block_J_shape_1->arr_height; i++) {
    for (int j = 0; j < block_J_shape_1->arr_width; j++) {
      if (i == 0 && j == 1 || i == 0 && j == 2) {
        block_J_shape_1->arr[i][j] = 0;
      } else {
        block_J_shape_1->arr[i][j] = 1;
      }
    }
  }

  /*
   * [[1, 1],
   *  [1, 0],
   *  [1, 0]]
   */
  struct orientation* block_J_shape_2 =
(orientation*)malloc(sizeof(orientation));
  block_J_shape_2->arr_height = 3;
  block_J_shape_2->arr_width = 2;
  for (int i = 0; i < block_J_shape_2->arr_height; i++) {
    for (int j = 0; j < block_J_shape_2->arr_width; j++) {
      if (i == 1 && j == 1 || i == 2 && j == 1) {
        block_J_shape_2->arr[i][j] = 0;
      } else {
        block_J_shape_2->arr[i][j] = 1;
      }
    }
  }
```

```
  /*
   * [[1, 1, 1],
   *  [0, 0, 1]]
   */
  struct orientation* block_J_shape_3 =
(orientation*)malloc(sizeof(orientation));
  block_J_shape_3->arr_height = 2;
  block_J_shape_3->arr_width = 3;
  for (int i = 0; i < block_J_shape_3->arr_height; i++) {
    for (int j = 0; j < block_J_shape_3->arr_width; j++) {
      if (i == 1 && j == 0 || i == 1 && j == 1) {
        block_J_shape_3->arr[i][j] = 0;
      } else {
        block_J_shape_3->arr[i][j] = 1;
      }
    }
  }

  /*
   * [[0, 1],
   *  [0, 1],
   *  [1, 1]]
   */
  struct orientation* block_J_shape_4 =
(orientation*)malloc(sizeof(orientation));
  block_J_shape_4->arr_height = 3;
  block_J_shape_4->arr_width = 2;
  for (int i = 0; i < block_J_shape_4->arr_height; i++) {
    for (int j = 0; j < block_J_shape_4->arr_width; j++) {
      if (i == 0 && j == 0 || i == 1 && j == 0) {
        block_J_shape_4->arr[i][j] = 0;
      } else {
        block_J_shape_4->arr[i][j] = 1;
      }
    }
  }

  // assigning circular dependency
  block_J_shape_1->default_shape = block_J_shape_1;
  block_J_shape_2->default_shape = block_J_shape_1;
  block_J_shape_3->default_shape = block_J_shape_1;
```

```
  block_J_shape_4->default_shape = block_J_shape_1;
  block_J_shape_1->rotate = block_J_shape_2;
  block_J_shape_1->reverse = block_J_shape_4;
  block_J_shape_2->rotate = block_J_shape_3;
  block_J_shape_2->reverse = block_J_shape_1;
  block_J_shape_3->rotate = block_J_shape_4;
  block_J_shape_3->reverse = block_J_shape_2;
  block_J_shape_4->rotate = block_J_shape_1;
  block_J_shape_4->reverse = block_J_shape_3;
  block_J->shape = block_J_shape_1;

  //seed for random function later on
  randomSeed(analogRead(9));
}

/*
 * GLOBAL VARIABLES
 */

struct node* curr = (node*)malloc(sizeof *curr);
struct node* next_block = (node*)malloc(sizeof *next_block);
int delay_value = 500;
int first_run = 0;
int lines_finished = 0;

void loop(void) {
  check_game_end();

  if (first_run == 0) {
    create_first_block();
  }

  //set the new block that is dropping to be the old next_block
  curr = next_block;

  //randomise next_block to be a new block
  randomise_next_block();

  //the next_block is always displayed in the top left
  //of the screen (the 5x4 section of pixels)
  //clear this to prepare it for the new next_block
  for (int i = 0; i < 5; i++) {
```

```
    for (int j = 0; j < 4; j ++) {
      mx.setPoint(j, i, false);
    }
  }

  //display the next_block
  for (int i = 0; i < next_block->height; i++) {
    for (int j = 0; j < next_block->width; j++) {
      if (next_block->shape->arr[i][j] == 1) {
        mx.setPoint(j+1, i+1, true);
      }
    }
  }

  bool hit = false;
  int block_dropping_column = 3;

  //this loop will drop the block from the top (row 8) to the bottom (row
32)
  //unless there is a collision (which will be explored later)
  for (int i = 8; i < 32; i++) {
   //check if any lines need to be cleareds
   if (i == 8) {
     check_cleared_lines();
   }

   //setup accelerometer value for this row
   String control_direction = check_control();

   //setup booleans which are needed for later
   bool able_move_right = false;
   bool able_move_left = false;
   bool valid_move = true;
   bool change_delay = false;
   bool rotate_block = false;

   //if accelerometer value was "RIGHT"
   if (control_direction == "RIGHT") {
     //need to check if every right-most block can move right one pixel
     for (int h = 0; h < curr->height; h++) {
       //i-h > 7 -> ensuring the block has finished spawning AND
       //curr->shape->arr[h][curr->width - 1] == 1 -> ensure that the pixel
```

```
we are dealing with is a '1' in the shape array
       //mx.getPoint(block_dropping_column+curr->width, i-h) == true -> if
the pixel to the right of the above ^ is lit up (therefore it cannot move
right)
       //block_dropping_column + curr->width > 7 -> this will be off the
screen
       if (i-h > 7 && (curr->shape->arr[h][curr->width - 1] == 1) &&
(mx.getPoint(block_dropping_column + curr->width, i-h) == true) ||
block_dropping_column + curr->width > 7) {
         //ensure this move is not allowed
         valid_move = false;
         break;
       }
     }
     //if the move was allowed, allow the right movement
     //increment the column of dropping (moves the block right)
     if (valid_move == true) {
       able_move_right = true;
       block_dropping_column++;
     }
   } else if (control_direction == "LEFT") {
     //need to check if every left-most block can move left one pixel
     for (int h = 0; h < curr->height; h++) {
       //i-h > 7 -> ensuring the block has finished spawning AND
       //curr->shape->arr[h][0] == 1 -> ensure that the pixel we are
dealing with is a '1' in the shape array
       //mx.getPoint(block_dropping_column - 1, i-h) == true -> if the
pixel to the left of the above ^ is lit up (therefore it cannot move left)
       //block_dropping_column - 1 < 0 -> this will be off the screen
       if (i-h > 7 && (curr->shape->arr[h][0] == 1) &&
(mx.getPoint(block_dropping_column - 1, i-h) == true ||
block_dropping_column - 1 < 0)) {
         //ensure this move is not allowed
         valid_move = false;
         break;
       }
     }
     //if the move was allowed, allow the left movement
     //decrement the column of dropping (moves the block left)
     if (valid_move == true) {
       able_move_left = true;
       block_dropping_column--;
```

```
     }
    } else if (control_direction == "BACK") {
      //inform the program to reduce the delay amount to drop the block
faster
      change_delay = true;
    } else if (control_direction == "FORWARD") {
      bool can_rotate = true;
      //remove block from the board, it will trigger this check which we
dont want
      int row_counter = 0;
      for (int k = curr->height-1; k >= 0; k--) {
        for (int j = 0; j < curr->width; j++) {
          //if value in the shape array is '1'
          if (curr->shape->arr[k][j] == 1) {
            //ensure we are out of spawn
            if (i-row_counter > 7) {
              //turn off pixel
              //k is decrementing so we cannot use it for height values
              //we use row_counter instead
              mx.setPoint(block_dropping_column + j, i-row_counter-1,
false);
            }
          }
        }
        row_counter = row_counter + 1;
      }

      //change the orientation of the block (rotate it)
      curr->shape = curr->shape->rotate;
      curr->width = curr->shape->arr_width;
      curr->height = curr->shape->arr_height;

      //now attempt to redraw the block in new orientation
      row_counter = 0;
      for (int k = curr->height-1; k >= 0; k--) {
        for (int j = 0; j < curr->width; j++) {
          if (curr->shape->arr[k][j] == 1) {
            if (i-row_counter > 7) {
              //mx.getPoint(block_dropping_column + j, i-row_counter-1) ==
true -> if any of the pixels we try to draw to are already lit
              //or if we go out of bounds
              if (mx.getPoint(block_dropping_column + j, i-row_counter-1)
```

```
== true || block_dropping_column + j > 7 || block_dropping_column + j < 0)
{
                can_rotate = false;
                break;
              }
            }
          }
        }
        if (can_rotate == false) {
          break;
        }
        row_counter = row_counter + 1;
      }
      //if we could not redraw the block in new orientation, then reverse
the rotation
      if (can_rotate == false) {
        curr->shape = curr->shape->reverse;
        curr->width = curr->shape->arr_width;
        curr->height = curr->shape->arr_height;
      }

      //now redraw the block
      int row_counter = 0;
      for (int k = curr->height-1; k >= 0; k--) {
        for (int j = 0; j < curr->width; j++) {
          //if value in the shape array is '1'
          if (curr->shape->arr[k][j] == 1) {
            //ensure we are out of spawn
            if (i-row_counter > 7) {
              //turn off pixel
              //k is decrementing so we cannot use it for height values
              //we use row_counter instead
              mx.setPoint(block_dropping_column + j, i-row_counter-1,
true);
            }
          }
        }
        row_counter = row_counter + 1;
      }
    }

    //check for a downward collsion
```

```
    //iterate through the width of the block
    for (int j = 0; j < curr->width; j++) {
      //counter, this finds the lowest most lit pixel for every row in the
block
      int counter = 0;
      while(curr->shape->arr[curr->height-counter-1][j] == 0) {
        counter = counter + 1;
      }
      //if the next block is filled i.e. there will be a collision
      //(curr->shape->arr[curr->height-counter-1][j] == 1) -> the index of
the block is filled
      //mx.getPoint(block_dropping_column + j, i-counter) -> the next pixel
below it
      //i-counter > 7 -> ensuring the block is out of the spawn point
      if ((curr->shape->arr[curr->height-counter-1][j] == 1) &&
mx.getPoint(block_dropping_column + j, i-counter) == true && i-counter > 7)
{
        hit = true;
        break;
      }
    }
    //check if a collision has occured
    if (hit == true) {
      delay(1000);
      break;
    }
    //change the delay value to be 20 when the user called move "BACK"
    //else reset it
    if (change_delay == true) {
      delay_value = 20;
    } else {
      delay_value = 500;
    }

    //move the whole block down now
    //all collisions/rotations have been checked and cleared by now
    //no need to check them again
    int row_counter = 0;
    for (int k = curr->height-1; k >= 0; k--) {
      for (int j = 0; j < curr->width; j++) {
        if (curr->shape->arr[k][j] == 1) {
          if (i+row_counter > 7) {
```

```
            mx.setPoint(block_dropping_column + j, i-row_counter, true);
            if (control_direction == "LEFT" && able_move_left) {
              mx.setPoint(block_dropping_column + j+1, i-1-row_counter,
false);
            } else if (control_direction == "RIGHT" && able_move_right) {
              mx.setPoint(block_dropping_column + j-1, i-1-row_counter,
false);
            } else {
              mx.setPoint(block_dropping_column + j, i-1-row_counter,
false);
            }
          }
        }
      }
      row_counter = row_counter + 1;
    }
    draw_end_line();
    if (i == 31 && control_direction == "BACK") {
      delay(1000);
    } else {
      delay(delay_value-(lines_finished*10));
    }
    mx.update();
  }
}

void create_first_block() {
  //increment first run to ensure it doesnt run again
  first_run = first_run + 1;

  //initialise next_block as a block, it doesnt matter which one
  next_block = block_S;
  randomise_next_block();
}

void randomise_next_block() {
  //the arr in each shape is declared to be arr[10][10]
  //we only ever use the first 1-4 of each index, meaning
  //the remaining 5-9 are 'junk' and will spew out huge random
  //integers when called, which is VERY useful for a truly random function

  //do ->next a random amount of times to get a random block
```

```
  for (int i = 0; i < random(abs(next_block->shape->arr[random(0,
9)][random(0, 9)])); i++) {
    next_block = next_block->next;
  }
  next_block->shape = next_block->shape->default_shape;
  next_block->width = next_block->shape->arr_width;
  next_block->height = next_block->shape->arr_height;
}

void check_game_end() {
  //check if any of the LED's in the 'end game row' are lit up
  for (int i = 0; i < 8; i++) {
    //if (above comment) is lit up
    if (mx.getPoint(i, 8) == true) {
      //clear the board and score
      mx.clear();
      clear_score();
      display_score();

      //print score to console
      Serial.println("Score:");
      Serial.println(lines_finished);

      //this shows the score for 7.5 seconds
      mx.update();
      delay(7500);

      //reset global variables for a new game
      lines_finished = 0;
      first_run = 0;

      //setup new game
      clear_score();
      draw_end_line();
    }
  }
}

void clear_score() {
  for (int i = 0; i < 8; i++) {
      mx.setPoint(i, 5, false);
  }
```

```
}

void draw_end_line() {
  for (int i = 0; i < 8; i++) {
      mx.setPoint(i, 7, true);
  }
}

void check_cleared_lines() {
  int rows_cleared = 0;
  //check if any row is full
  for (int i = 8; i < 32; i++) {
    bool full_row = true;
    for (int j = 0; j < 8; j++) {
      if (mx.getPoint(j, i) == false) {
        full_row = false;
        break;
      }
    }

    //if a row is full, clear it and increment rows_cleared
    if (full_row == true) {
      for (int j = 0; j < 8; j++) {
        mx.setPoint(j, i, false);
      }
      rows_cleared = rows_cleared + 1;
    }
  }

  //increment global variable
  lines_finished = lines_finished + rows_cleared;

  //update screen to show new board
  delay(50);
  mx.update();

  //increment from bottom row to top of screen
  int starting_line = 31;
  for (int i = 31; i >= 8; i--) {
    bool empty_row = true;
    //check if a row is empty
    for (int j = 0; j < 8; j++) {
```

```
      if (mx.getPoint(j, i) == true) {
        empty_row = false;
        break;
      }
    }

    //if row was NOT empty
    if (empty_row == false) {
      //int starting_line starts at the bottom (line 31)
      //for each NON empty line, it is decremented
      //this variable is used to fill up the board from the bottom up
      //it ignores empty lines and hence clears the lines as required
      for (int j = 0; j < 8; j++) {
        if (mx.getPoint(j, i) == true) {
          mx.setPoint(j, starting_line, true);
        } else {
          mx.setPoint(j, starting_line, false);
        }
      }
      starting_line = starting_line - 1;
    }
  }

  //clear up everything else
  for (int i = starting_line; i >= 8; i--) {
    for (int j = 0; j < 8; j++) {
      mx.setPoint(j, i, false);
    }
  }
  clear_score();
  display_score();
  mx.update();
}

void display_score() {
  //determine the score in binary
  int binaryBitsCount = lines_finished * 8;
  char str[binaryBitsCount + 1];
  itoa(lines_finished, str, 2);

  //display score on the blank screen
  int counter = 0;
```

```
  for (int i = binaryBitsCount; i >= 0; i--) {
    if (str[i] == '0') {
      mx.setPoint(7-counter, 5, false);
      counter = counter + 1;
    }
    if (str[i] == '1') {
      mx.setPoint(7-counter, 5, true);
      counter = counter + 1;
    }
  }
}

String check_control() {
  // code taken from

//https://www.hackster.io/56491/using-the-pmod-acl-with-arduino-uno-35d564
  Wire.beginTransmission(ADXL345_Adresse);
  Wire.write(DATAX0);
  Wire.endTransmission();
  Wire.beginTransmission(ADXL345_Adresse);
  Wire.requestFrom(ADXL345_Adresse, 6); // Recovery of the 6 components
  i=0;

  while (Wire.available()) {
    buffer[i] = Wire.read();
    i++;
  }
  Wire.endTransmission();
  xValue=(buffer[1] << 8) | buffer[0]; // Development of the 3 components
  yValue=(buffer[3] << 8) | buffer[2];

  if ((xValue > 65500 || xValue < 100) && (yValue > 65400 && yValue <
65500)) {
    return "RIGHT";
  } else if ((xValue > 65500 || xValue < 100) && yValue < 100) {
    return "LEFT";
  } else if ((xValue < 120 && xValue > 50) && (yValue > 65500 || yValue <
150)) {
    return "FORWARD";
  } else if ((xValue > 65400 && xValue < 65520) && (yValue > 65500 ||
yValue < 100)) {
    return "BACK";
```

```
  } else {
    return "STILL";
  }
}
```