LED Matrix Arcade Game Suite

LED Matrix Arcade Game Suite

ECE 3140/CS 3420 Embedded Systems Final Project

Michael Xiao (mfx2) and Nathaniel Diamond (ncd27)

Introduction

The LED Matrix arcade game suite includes several simple iconic games from our childhood made using a LED matrix and the FRDM-K64F microcontroller. These games include:

- Stacker: In stacker, the player tries to build a tower to the top of the screen by correctly timing moving blocks.
- Jump: In jump, the player avoids random incoming obstacles for as long as possible.
- Tetris: In tetris, random blocks fall from the sky and must be organized in neat rows. This is a little bit simpler than classic Tetris because we were unable to implement the rotation of blocks.

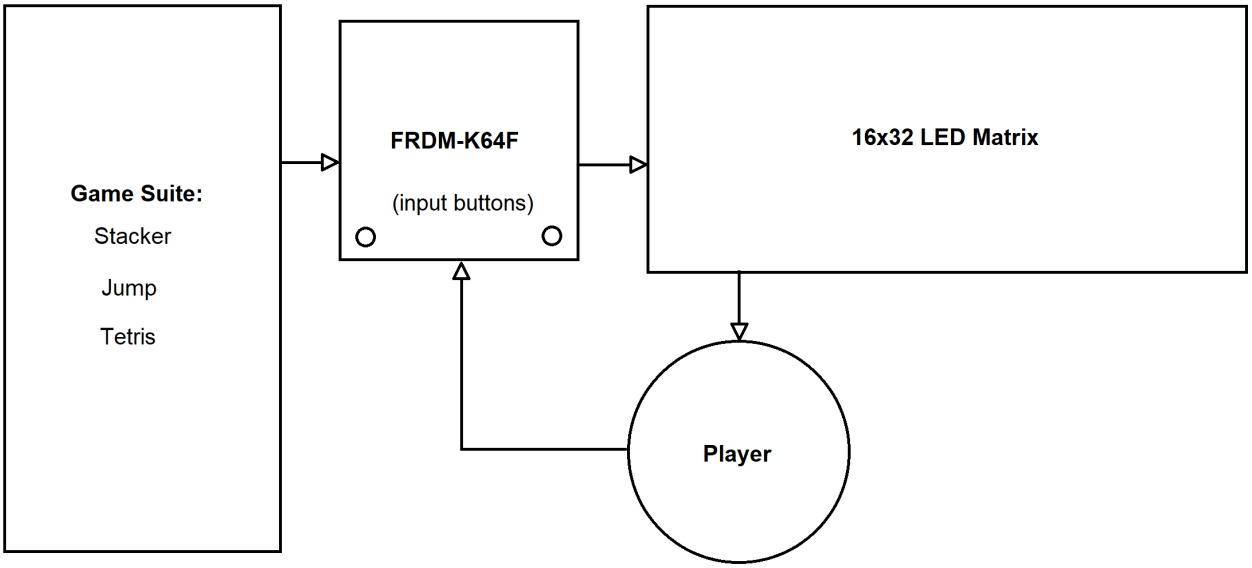
High-Level Design

Inspiration

The inspiration for our idea was the plethora of arcade games we played growing up. Stacker and Tetris were inspired from ver popular existing games while the jump game was inspired by the google chrome offline dinosaur jump game.

Major Components

The main components are simply the LED matrix, FRDM-K64F microcontroller, and of course, the player. The button inputs are the two extra buttons (SW2 and SW3) ont he microcontrollers and the GPIO pins directly control the LED matrix.

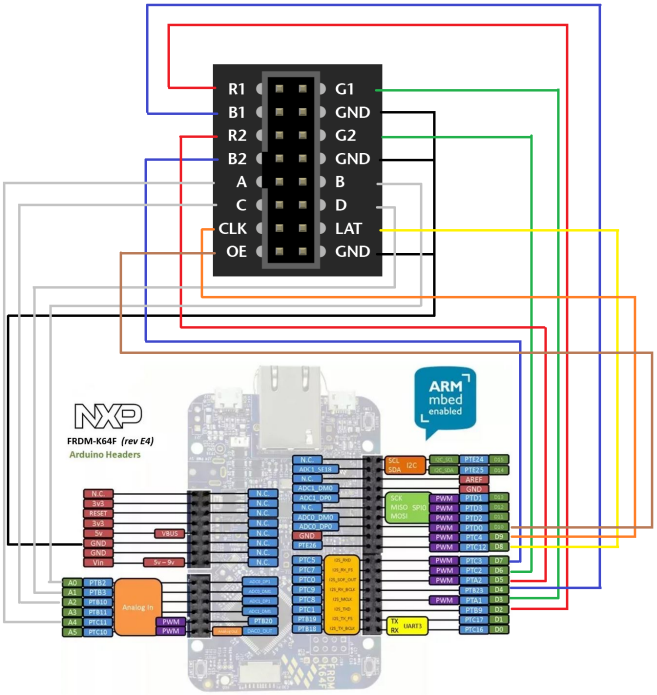


Design Considerations

The major design considerations we had to make were to think about how complex we wanted to make our game. Originally, we planned on just making one game. However, after we made a simple game, we realized we could make something more complex; hence why we ended up making an arcade suite. The games we ended up choosing to make all required minimal input (2 buttons) and had graphics that were course enough such that the LED matrix could accurately depict them well.

Hardware Design

Schematic



Our original circuit design is pictured above. For our initial testing, we wired our matrix up with jumper cables. There are 16 pins to connect to the INPUT as shown.

R1, **G1**, and **B1** control the red, green, and blue LEDs on the top half of the board. You can test this by connecting any of the three to a 5V source. It should light up all the top LED pixels with the respective color. **R2**, **G2**, and **B2** do the same thing for the bottom of the board. All the R, G, and B pins are digital.

A, **B**, and **C** (**D** for 32x32) are analog inputs that select rows to light up.

The **LAT** is a latch that latches at the end of every row of data and is analog for 16x32 and digital for 32x32.

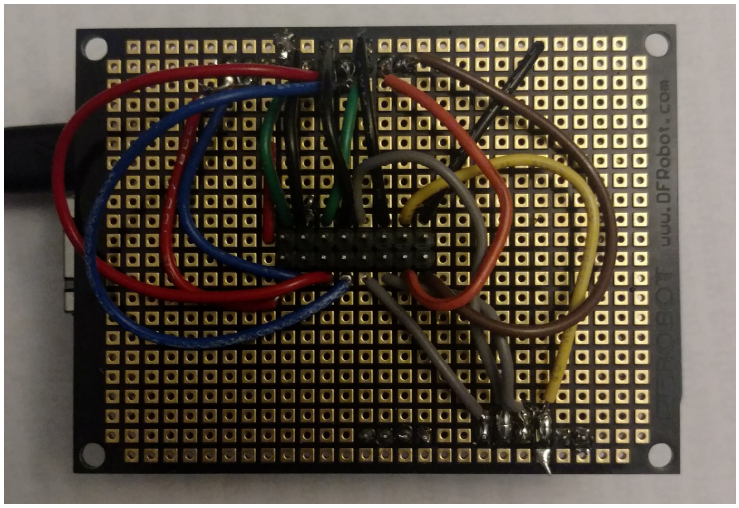
The **CLK** constantly runs and indicates the arrival of each bit of data with each rising edge.

The **OE** output enable is a digital signal that turns each row of LEDs off.

The remaining **GND** ground pins can be simply connected to the various ground ports on the board.

We wrote a more comprehensive guide to setting everything up here: Medium 16x32 RGB LED matrix panel

Shield



For a more robust construction, we soldered a custom protoboard shield to fit over the k64f. This allowed us to use the buttons on the micro controller without having to worry about jumper wires falling out. This is shown above. We used A0-A2 for row select, A3 for LAT, D10-D15 for RGB control, D9 for CLK, and D8 for OE.

Bill of Materials

Part	Cost	Link
FRDM-K64F	$0.00	
16x32 LED Matrix	$24.95	https://www.adafruit.com/product/420
100x75mm ProtoBoard Double Sided	$2.69	https://www.robotshop.com/en/100x75mm-protoboard-double-sided.html?gclid=EAIaIQobChMIt-SF1omN2wIVSeDICh1iogkIEAkYDSABE
5V power supply	$0.00	Provided by Professor Skovira

Software Design

LED matrix setup

We spent the first week and a half or so on simply getting the LED matrix working with the board. We documented our work and shared our findings here: Medium 16x32 RGB LED matrix panel

There are eleven different pins to keep track of and managing their tasks and timing prove to be very difficult. Luckily, there are a lot of libraries released by Adafruit with useful functions to update the display, draw pixels, draw lines, and draw rectangles. Unfortunately, these libraries are written to be used with an Arduino and the k64f, despite having the same pinouts, has some hardware differences that make these libraries not completely functional.

What we found to be the easiest way to make these libraries work was to make use of the MBED online compiler. From the compiler, we can make new programs by clicking NEW->NEW PROGRAM. We started by making use of the example programs in place for us and opening the “gpio example for the Freescale freedom platform” template. This introduced another very useful library for .cpp programming – the MBED library. Mbed makes it very easy to control digital and analog pins, read inputs, manage interrupts, set timers, and much more. Once we made a new program, we used IMPORT->LIBRARIES to open up a library browser. Once the library browser was open, we found the “RGB_matrix_panel” library to include in our program. After setting up in MBED's online compiler, we then ported our program to microvision with the built-in exporter. This allowed us to continue our work in the familiar microvision environment.

However, after portingtomicrovision, we found the example code provided by the library shown below:

example.cpp

```
/** RGBmatrixPanel is class for full color LED matrix

* @code

#include "mbed.h"

#include "RGBmatrixPanel.h" // Hardware-specific library



PinName ub1=p6;

PinName ug1=p7;

PinName ur1=p8;

PinName lb2=p9;

PinName lg2=p10;

PinName lr2=p11;

RGBmatrixPanel
matrix(ur1,ug1,ub1,lr2,lg2,lb2,p12,p13,p14,p15,p16,p17,false);

//RGBmatrixPanel(r1, g1, b1, r2, g2, b2, a, b, c, sclk, latch, oe, enable
double_buffer);



int main()

{

    matrix.begin();

    while(1) {

        // fill the screen with 'black'

        matrix.fillScreen(matrix.Color333(0, 0, 0));

        // draw a pixel in solid white

        matrix.drawPixel(0, 0, matrix.Color333(7, 7, 7));

        wait_ms(500);
```

```
        // fix the screen with green
        matrix.fillRect(0, 0, 32, 16, matrix.Color333(0, 7, 0));
        wait_ms(500);

        // draw a box in yellow
        matrix.drawRect(0, 0, 32, 16, matrix.Color333(7, 7, 0));
        wait_ms(500);

        // draw an 'X' in red
        matrix.drawLine(0, 0, 31, 15, matrix.Color333(7, 0, 0));
        matrix.drawLine(31, 0, 0, 15, matrix.Color333(7, 0, 0));
        wait_ms(500);

        // draw a blue circle
        matrix.drawCircle(7, 7, 7, matrix.Color333(0, 0, 7));
        wait_ms(500);

        // fill a violet circle
        matrix.fillCircle(23, 7, 7, matrix.Color333(7, 0, 7));
        wait_ms(500);

        // fill the screen with 'black'
        matrix.fillScreen(matrix.Color333(0, 0, 0));

        // draw some text!
        matrix.setCursor(1, 0);   // start at top left, with one pixel of
spacing
```

```
matrix.setTextSize(1);    // size 1 == 8 pixels high


// printff each letter with a rainbow color
matrix.setTextColor(matrix.Color333(7,0,0));
matrix.putc('1');
matrix.setTextColor(matrix.Color333(7,4,0));
matrix.putc('6');
matrix.setTextColor(matrix.Color333(7,7,0));
matrix.putc('x');
matrix.setTextColor(matrix.Color333(4,7,0));
matrix.putc('3');
matrix.setTextColor(matrix.Color333(0,7,0));
matrix.putc('2');


matrix.setCursor(1, 9);   // next line
matrix.setTextColor(matrix.Color333(0,7,7));
matrix.putc('*');
matrix.setTextColor(matrix.Color333(0,4,7));
matrix.putc('R');
matrix.setTextColor(matrix.Color333(0,0,7));
matrix.putc('G');
matrix.setTextColor(matrix.Color333(4,0,7));
matrix.putc('B');
matrix.setTextColor(matrix.Color333(7,0,4));
matrix.putc('*');
wait_ms(500);
```

```
        }

    }
```

This code is well commented and reading it introduced us to the functions available for us to use. If compiled and run, however, the board did not work because the clock on the Arduino and the clock of the k64f are at different speeds. Calling matrix.begin() is where the main problem occurs. Matrix.begin() calls updateDisplay() every millisecond. However, because it is designed for Arduino, the clock gets out of sync after a certain amount of time has elapsed and the board will not be able to update afterwards. If for example, you try to fill the screen with all green pixels (done so by using a for loop), it will only fill ~35 pixel before the screen “freezes” and cannot advance with further updates. We solved this problem by simply not calling matrix.begin(). This solves the timing problem but requires us to manually call updateDisplay() every ~1ms within our code so plan to work around that.

In our actual code, we included the following header files

```
#include "mbed.h"
#include "RGBmatrixPanel.h" // Hardware-specific library
```

Next, we document which pins we used in the hardware with:

```
PinName ub1=D13;
PinName ug1=D14;
PinName ur1=D15;
PinName lb2=D10;
PinName lg2=D11;
PinName lr2=D12;
```

We can then call RGBmatrixPanel() which initializes the pins. The comment below the call describes the input parameters so once again align accordingly.

```
RGBmatrixPanel matrix(ur1,ug1,ub1,lr2,lg2,lb2,A0,A1,A2,D9,A3,D8,false);
//RGBmatrixPanel(r1, g1, b1, r2, g2, b2, a, b, c, sclk, latch, oe, enable double_buffer);
```

Furthermore, the wait_ms() command also will not work – we had to create our own delay function later to accommodate not using matrix.begin(). This function is unique because it also needs to update the display very frequently to keep the current image up. This can be done with a simple for loop, using the wait_us() call from mbed library, and updateDisplay(). An example is shown below:

```
void delay(int x){

     for(int i=0;i<x;i++){

          wait_us(10);

          matrix.updateDisplay();

     }

}
```

This was our initial base of knowledge we used to build off of and from here, we learned a lot more about the included libraries to implement the games we made.

Stacker

The full code of stacker is included in the appendix below.

Stacker is our simplest game and does not have any specialized data structures. The game keeps track of several key variables:

- L - length of block
- a - row location of block
- L2 - length of previous block
- a - row location of block
- h - height of current block
- t - delay for moving

The main function is very simple and moves the current block (6x2 to start the game) back and forth across the screen. While it does this, it constantly polls for a button press on SW2. when SW2 is pressed, the **stop()** function is called.

At the beginning of the stop function, the **fall()** function is first called. In fall, a collision check is done on both the left and right sides of the current block to the previous block. This is done by comparing a with a2. If there are overhanging portions, those blocks are removed, a2 and L2 are reset, and a and L are properly adjusted.

The rest of **stop()** does several checks. First, if L<1, then there are no more blocks to build with so the player loses. If h==0, then the the player has built to the very top of the screen, meaning that the player has won. Otherwise, if neither of these conditions are met, the game freezes the location of the current block and increments the speed and height of the blocks.

Jump

The full code of jump is included in the appendix below.

Inspired by the chrome offline dinosaur game, Jump is a game in which the player is a blue block that tries to jump over incoming red obstacles. Jump is also relatively simple but uses timer interrupts and polling for the input button. Jump also does not have any special data structures but does have the following key variables:

- x - x coordinate of player
- y - y coordinate of player
- l - length of player
- h - height of player
- speed - speed of the game
- score
- ox - x coordinate of obstacle
- oy - y coordinate of obstacle
- ol - length of obstacle
- oh - height of obstacle

The main game function simply starts the timer and then runs a loop to draw the static components on the screen. This timer runs continuously until it reaches the value of speed. When it does, it calls **advance()** which simply moves the obstacle forward by one pixel. When the obstacle is well behind the player, a new obstacle is created off screen with random width, height, and distance from the player. Each time advance is called, the score is also incremented.

A special **delay()** function was written to check conditions while delaying at the same time. **delay()** first calls **updateDisplay()** and then checks for collision. Collision happens when x and y are coinciding with ox and oy. When there is a collision, **lose()** is called which brings up the losing screen.

To avoid this collision, the player can jump by pressing SW2. When pressed, **jump()** is called which simply moves the player up and down and correctly repositions the x and y variables so that collision does not occur. Gravity was also taken into account to make the jump look more "natural" - this was done by slowing the delay at the arch of the jump. Each time the player jumps, the speed increases making the game more difficult. The player tries to run as long as they can before hitting an obstacle.

Tetris

The full code of tetris is included in the appendix below.

Intro:

Tetris, a game popular as it is timeless, was released on June 6th, 1984. The purpose of the game is to fit Tetrominos—various shapes consisting of four blocks—into rows that span the screen of the device on which you play. When a row is entirely fills, it clears and adds to your score.

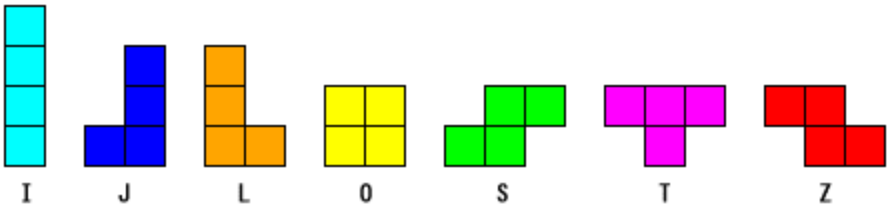


Figure of Tetrominos Depicted to the Left (Reference: https://puzzling.stackexchange.com/questions/22143/occupy-a-field-with-your-choice-of-tetromino)

Data Structures:

Enum Color- This Enum is necessary in order to implement the switch statement needed to initialize the different colored blocks and Tetrominos. There are seven colors in total: *Red, Orange, Yellow, Green, DBlue, LBlue*, and *Pink*.

Class GameBlock- This class is an abstraction of the the components of the game, be it the game board or the Tetrominos that fall on the game board. Each *GameBlock* consists of the fields *x*, *y*, *filled*, and *color*. All are integers specifying, as you may have guessed, the placement of the block on the board, whether a game piece currently inhabits the spot, and what the color of the piece is.

This class provides the functions *draw, atGround, fall, getX, getY, full, fill, clear,* and *shift*. *draw* calls on built-in functions to add a pixel to the matrix. *atGround* specifies whether the pixel is at the bottom of the matrix. *fall* moves the pixel down by one. *getX* and *getY* return the values of private fields *x* and *y*. *full* returns 1 if *filled* is 1, i.e. if a game piece inhabits the spot. *fill* sets *filled* to 1, whereas *clear* sets it to 0. *shift* moves the pixel either right or left depending on the input parameter *d*, which is an integer intended to be of value 1 or -1.

Class Tetromino- This class has one field blocks that contains the four game blocks that compose the Tetromino. Functions *draw, fall, fix*, and *shif t* behave similar to their GameBlock counterparts, except they perform the function only on the aggregate, if that is possible. fix, which has no GameBlock counterpart, adds the *GameBlock*s of the *Tetromino* to the global variable *board*, which contains the pieces that have fallen to the bottom, and then calls *clearline* to see if any lines can be cleared.

Pivotal Data Structures Relationship.

Functionality:

The main method of this game is a loop that after initialization, runs so long as the next piece to fall does not conflict with any pieces already on the board. The loop uses a timer to verify that enough time has passed to make the current *Tetromino* fall. The loop will always be responsive to the FRDM board switches, however. Since C++ allowed for object-oriented programming, the main loop was actually the most transparent part of the project. In fact, the code could be condensed further to a point where reading the game logic would be an extension of reading English. The heart of the code lies in manipulations of the game pieces, which happen in the aggregate only if they can happen on all the components. This is exemplified in the hierarchical figure shown above.

Besides the libraries necessary to support the matrix, package *ctime* was added to ensure that the random function could be properly seeded with something pseudorandom (the current time).

Testing

Because our project is a series of games, testing was done iteratively through player testing. This was done by testing each feature as it was implemented. For example, for tetris, our first tests consisted of just seeing the different blocks on screen. Next, we tested having pieces fall. Then we tested the collision logic. etc. This made it easy to pinpoint problems as only minor changes were made between tests. Additionally, this testing methodology allowed us to really fine tune our functions to make sure they did exactly what we wanted them to do.

Results

In our project, we were successfully able to implement three games of increasing coding difficulty. We learned a lot about programming on the FRDM-K64F board and about how to look through documentation and other online resources. We felt that being able to practice this skill was really important because this is how a product is created in the real world.

Unfortunately, since our project is a series of games, quantitative data and graphs were not applicable.

We believe our product is usable because it is robust and entertaining!

Video:

Conclusions

Expectations

Overall, our design definitely met our expectations. We came in with lower expectations after running into a lot of trouble with the LED matrix. However, after we learned the ins and outs of both the matrix and mbed libraries while making stacker and jump, we realized we had a lot more resources to use. The most challenging parts of our project were probably figuring out the LED matrix and configuring classes, objects, and interrupts in our code.

Future Work

If we had more time, we would have definitely fine tuned our games. A score functionality could have definitely been added and displayed after losing. In tetris, we never managed to add a rotation feature. In jump and stacker, we definitely could add better graphics and animations. We also want to integrate some kind of GUI to select the game you play rather than having to upload them each from the laptop. Lastly, we also had a lot of ideas for other arcade games to make and would love to work on it in the future. We definitely have learned a lot about the FRDM-K64F board and the LED matrix from this project and we plan to work more with the matrix in the future as well.

Intellectual Property

The code we wrote was entirely from scratch. However, the ideas and inspiration for all four games were taken from preexisting arcade games.

Work Distribution

Michael completed most of the hardware components including setting up the LED matrix, establishing the basic matrix software, and making the protoboard shield.

Michael and Nate worked together on all of the software. However, Michael made most of the foundation for the games while Nate worked on the more unique features. Both of us have significant contributions to all of the games we made as we collaborated entirely through in-person meetings and peer programming. We also took advantage of Github. Being able to share and update code and see what parts had been changed made it easy to collaborate with the other person, and version control makes development far easier than other methods of code sharing. Both of us contributed major parts to the code and to the write-up.

Appendix

stacker.cpp

```
#include "mbed.h"

#include "RGBmatrixPanel.h" // Hardware-specific library



//pin definitions

PinName ub1=D13;

PinName ug1=D14;

PinName ur1=D15;

PinName lb2=D10;

PinName lg2=D11;

PinName lr2=D12;

RGBmatrixPanel matrix(ur1,ug1,ub1,lr2,lg2,lb2,A0,A1,A2,D9,A3,D8,false);

//RGBmatrixPanel(r1, g1, b1, r2, g2, b2, a, b, c, sclk, latch, oe, enable
double_buffer);



// input button

DigitalIn button(SW2);



int L = 6;  //size of block

int L2 = 0; //size of previous block

int a = 0;  //row number

int a2 = 0; //row number of previous block

int h = 30; //height

int t = 1000; //delay for moving
```

```
/*basic delay function
calls updateDisplay often enough to have a smooth screen display
x=4000 ~ 1 second*/
void delay(int x){
 for(int i=0;i<x;i++){
  wait_us(10);
  matrix.updateDisplay();
 }
}



/* checks to see if blocks are below
if no blocks are below:
remove blocks that are overhanging
decrease length*/
void fall(){
 //game has just started - set a2 and L2 equal to a and L
 if(h==30) {
  a2 = a;
  L2 = L;
  return;
 }
 //case where block is to right of previous block
 if(a < a2){
  //check right side
  for(int i = a; i < a2 ; i++){
   matrix.fillRect(h,i,2,1,matrix.Color333(0,0,0));
```

```
    L = L - 1;
    //readjust a
    a++;
   }
   //set new a2 and L2
   a2 = a;
   L2 = L;
   return;
  // case where block is to left of previous block
  } else if (a>a2){
   for(int i = a+L2; i > a2+L2 ; i--){
    matrix.fillRect(h,i,2,1,matrix.Color333(0,0,0));
    L = L - 1;
   }
   //set new a2 and L2
   a2 = a;
   L2 = L;
   return;
  //perfect stack
  }else{
   //set new a2 and L2
   a2 = a;
   L2 = L;
   return;
  }
 }
```

```
/* stops current moving block
called when button is pressed
calls fall()
checks win and lose conditions
stop the block and advances to next level*/
void stop() {
 //check to see if blocks are lost
 fall();
 //L=0 means no blocks left
 if(L<=0){ //loser
  matrix.fillRect(0, 0, 32, 16, matrix.Color333(0, 0, 0));
  matrix.fillRect(14,6, 4,4, matrix.Color333(3,0,0));
  //red square lose screen
    while(1) matrix.updateDisplay();
 }

  //winner - made it to the top
  if(h == 0){
    matrix.fillRect(0, 0, 32, 16, matrix.Color333(0, 0, 0));
  matrix.fillRect(14,6, 4,4, matrix.Color333(0,0,3));
  //green square win screen
    while(1) matrix.updateDisplay();
  }

  //freeze location of current block
  matrix.fillRect(h,a,2,L, matrix.Color333(1,3,3));
```

```
  delay(5000);


  h=h-2;  // increase height
  t = t - 40;  // increase speed
  return;
}



/* main game function*/
int main(){
 while(1){
  if(button == 1){
   //move to right until hits the end of the screen
   while(a<16-L){
    if(button==0) stop();
    matrix.fillRect(h,a,2,L, matrix.Color333(1,3,3));
    delay(t);
    matrix.fillRect(h,0,2,16, matrix.Color333(0, 0, 0));
    a++;
   }
   //move to left until it hits end of the screen
   while(a>0){
    if(button == 0) stop();
    matrix.fillRect(h,a,2,L, matrix.Color333(1,3,3));
    delay(t);
    matrix.fillRect(h,0,2,16, matrix.Color333(0, 0, 0));
    a--;
```

```
    }
   }else{
     stop();
   }
  }
 }
```

jump.cpp

```
#include "mbed.h"
#include "RGBmatrixPanel.h" // Hardware-specific library

//pin setups
PinName ub1=D13;
PinName ug1=D14;
PinName ur1=D15;
PinName lb2=D10;
PinName lg2=D11;
PinName lr2=D12;
RGBmatrixPanel matrix(ur1,ug1,ub1,lr2,lg2,lb2,A0,A1,A2,D9,A3,D8,false);
//RGBmatrixPanel(r1, g1, b1, r2, g2, b2, a, b, c, sclk, latch, oe, enable
double_buffer);

//input button
DigitalIn button(SW2);

int x = 2;  // x coordinate of player
int y = 9;  // y coordinate of player
int l = 2;  // length of player
int h = 5;  // width of player
int speed = 50;  // speed of game: lower = faster
int score = 0;
int ox = 50; // x coordinate of obstacle
int oy = 10; // y coordinate of obstacle
int ol = 3;  // length of obstacle
int oh = 4;  // height of obstacle

Timer timer;  // timer for interrupt

/*basic wait function
calls updateDisplay() frequently enough for smooth display
x=4000 ~ 1 second*/
void delay1(int x){
 for(int i=0;i<x;i++){
  wait_us(10);
  matrix.updateDisplay();
 }
}

void displayScore(){

}

/*lose function
freezes screen to show collision
```

```
flashes score
goes to lose screen (red square)*/
void lose(){
 //freeze screen briefly
 delay1(4000);

 //clear screen for lose screen
 matrix.fillRect(0, 0, 32, 16, matrix.Color333(0, 0, 0));
 matrix.drawChar(1, 0, (char)50, matrix.Color333(2, 2, 2),
matrix.Color333(0, 0, 0), 1);
 matrix.drawChar(7, 0, 'c', matrix.Color333(2, 2, 2), matrix.Color333(0, 0,
0), 1);
 matrix.drawChar(13,0, 'o', matrix.Color333(2, 2, 2), matrix.Color333(0, 0,
0), 1);
 matrix.drawChar(19, 0, 'r', matrix.Color333(2, 2, 2), matrix.Color333(0,
0, 0), 1);
 matrix.drawChar(25, 0, 'e', matrix.Color333(2, 2, 2), matrix.Color333(0,
0, 0), 1);

 matrix.fillRect(14,6, 4,4, matrix.Color333(3,0,0));
  while(1) matrix.updateDisplay();
}

/*interrupt function
called whenever timer is above speed
moves the obstacle forward
creates new randomized obstacle when off screen*/
void advance(){
 //clear screen
 matrix.fillRect(ox,oy,ol,oh,matrix.Color333(0,0,0));
 //increment ox and score
 ox --;
 score++;
 // move obstacle
 matrix.fillRect(ox,oy,ol,oh,matrix.Color333(3,0,0));
 // if obstacle off screen, generate random new one
 if (ox<-5){
  ol = (rand()%3)+1;
  ox=32+(rand()%15);
  oh= (rand()%4)+2;
  oy=14-oh;
 }
 return;
}
/*modified delay function
checks interrupts and lose condition*/
void delay(int x){
 for(int i=0;i<x;i++){
  //wait_us(10);
  matrix.updateDisplay();
  //check to see if timer has reached speed
  int t = timer.read_ms();
  if(t >speed) {
```

```
   timer.reset();
   advance();
  }

  //check collision for lose condition
  if(ox<4 && ox>1 && oy<y+h) lose();
  }
}
/*player jump function
called whenever button is pressed*/
void jump(){
 matrix.fillRect(x,y,l,h,matrix.Color333(0,0,0));
 //every time player jumps, the game gets faster
 speed = speed - 2;
 //jump up and down from y = 9 to 1
 for (int i = 9 ; i>0 ; i--){
  matrix.fillRect(x,i,l,h,matrix.Color333(0,3,0));
  //include effects of gravity
  delay(400-25*i);
  matrix.fillRect(x,i,l,h,matrix.Color333(0,0,0));
  //update player y coordinate
  y = i;
 }
 for (int i = 1 ; i<10 ; i++){
  matrix.fillRect(x,i,l,h,matrix.Color333(0,3,0));
  delay(400-30*i);
  matrix.fillRect(x,i,l,h,matrix.Color333(0,0,0));
  y=i;
 }
 return;
}


/* main game function
*/
int main(){
 timer.start();

 while(1){
  matrix.drawLine(0,14,31,14,matrix.Color333(0,0,3));  //grass
  matrix.fillRect(x,y,l,h,matrix.Color333(0,3,0));  // player sprite
  matrix.updateDisplay();
  delay(10);
  // check button to jump
  if(button == 0) jump();

 }
}
```

tetris.cpp

```
#include <iostream>
#include "mbed.h"
#include "RGBmatrixPanel.h" // Hardware-specific library
#include <ctime>

PinName ug1=D13;
PinName ub1=D14;
PinName ur1=D15;
PinName lg2=D10;
PinName lb2=D11;
PinName lr2=D12;
RGBmatrixPanel matrix(ur1,ug1,ub1,lr2,lg2,lb2,A0,A1,A2,D9,A3,D8,false);
//RGBmatrixPanel(r1, g1, b1, r2, g2, b2, a, b, c, sclk, latch, oe, enable
double_buffer);


DigitalIn button(SW2);
DigitalIn button2(SW3);

enum Color {Red, Orange, Yellow, Green, DBlue, LBlue, Pink};

uint16_t getColor(Color c);

class GameBlock {
    private:
        int x,y;
        int filled;
        uint16_t color;

    public:
        GameBlock() {
            x = 0;
            y = 0;
            filled = 0;
            color = matrix.Color333(0,0,0);
        }

        GameBlock(int a, int b, uint16_t c) {
            x = a;
            y = b;
            color = c;
            filled = 1;
        }

        void draw() {
            if(filled) {
```

```
                matrix.drawPixel(x,y,color);
            }
        }

        void fall() {
            x++;
        }

        int atGround() {
            if( x == 31) {
                return 1;
            }
            return 0;
        }

        int getX() {
            return x;
        }

        int getY() {
            return y;
        }
        int full() {
            return filled;
        }
        void fill() {
            filled = 1;
        }
        void clear() {
            filled = 0;
        }
        void shift(int d) {
            y = (y+d+16)%16;
        }
    };

GameBlock board[32][16];

void clearline() {
    int clear = 0;
    for(int i = 1; i < 32; i ++) {
        clear = 1;
        for(int j = 0; j < 16; j ++) {
            if(!board[i][j].full()) {
                clear = 0;
            }
        }
        if(clear) {
            for(int j = 0; j < 16; j ++) {
                board[i][j].clear();
                for(int k = i; k > 0; k --) {
                    board[k][j] = board[k-1][j];
                    board[k][j].fall();
```

```
                }
            }
        }
    }
}

class Tetromino {
    private:
        GameBlock blocks[4];
        Tetromino(){}

    public:

        Tetromino(Color c) {
        switch(c) {
            case Red:
                blocks[0] = GameBlock(1,7,getColor(c));
                blocks[1] = GameBlock(1,8,getColor(c));
                blocks[2] = GameBlock(0,8,getColor(c));
                blocks[3] = GameBlock(0,9,getColor(c));
                break;
            case Orange:
                blocks[0] = GameBlock(1,7,getColor(c));
                blocks[1] = GameBlock(1,8,getColor(c));
                blocks[2] = GameBlock(1,9,getColor(c));
                blocks[3] = GameBlock(0,9,getColor(c));
                break;
            case Yellow:
                blocks[0] = GameBlock(1,7,getColor(c));
                blocks[1] = GameBlock(1,8,getColor(c));
                blocks[2] = GameBlock(0,8,getColor(c));
                blocks[3] = GameBlock(0,7,getColor(c));
                break;
            case Green:
                blocks[0] = GameBlock(0,7,getColor(c));
                blocks[1] = GameBlock(0,8,getColor(c));
                blocks[2] = GameBlock(1,8,getColor(c));
                blocks[3] = GameBlock(1,9,getColor(c));
                break;
            case LBlue:
                blocks[0] = GameBlock(0,7,getColor(c));
                blocks[1] = GameBlock(1,7,getColor(c));
                blocks[2] = GameBlock(2,7,getColor(c));
                blocks[3] = GameBlock(3,7,getColor(c));
                break;
            case DBlue:
                blocks[0] = GameBlock(0,7,getColor(c));
                blocks[1] = GameBlock(1,7,getColor(c));
                blocks[2] = GameBlock(1,8,getColor(c));
                blocks[3] = GameBlock(1,9,getColor(c));
                break;
            case Pink:
                blocks[0] = GameBlock(1,7,getColor(c));
```

```
        blocks[1] = GameBlock(1,8,getColor(c));
        blocks[2] = GameBlock(0,8,getColor(c));
        blocks[3] = GameBlock(1,9,getColor(c));
        break;
    }
}

void draw() {
    for (int i = 0; i < 4; i ++) {
        blocks[i].draw();
    }
    matrix.updateDisplay();
}

int fall() {
    for (int i = 0; i < 4; i ++) {
        GameBlock * b = &(blocks[i]);
        int x = b->getX();
        int y = b->getY();
        if(b->atGround()) {
            fix();
            return 0;
        }
        if(board[x+1][y].full()) {
            fix();
            return 0;
        }
    }
    for (int i = 0; i < 4; i ++) {
        blocks[i].fall();
    }
    return 1;
}

void fix() {
    for(int i = 0; i < 4; i++) {
        GameBlock * b = &(blocks[i]);
        int x = b->getX();
        int y = b->getY();
        board[x][y] = *b;
    }
    clearline();
}

int conflicts() {
    for(int i = 0; i < 4; i++) {
        GameBlock * b = &(blocks[i]);
        int x = b->getX();
        int y = b->getY();
        if(board[x][y].full()) {
            return 1;
        }
    }
```

```
            return 0;
        }
        void shift(int d) {
            for(int i = 0; i < 4; i ++) {
                GameBlock * b = &(blocks[i]);
                int x = b->getX();
                int y = b->getY();
                if((board[x][(y+d+16)%16].full())) {
                    return;
                }
            }
            for(int i = 0; i < 4; i ++) {
                blocks[i].shift(d);
            }
        }
    };

uint16_t getColor(Color c) {
    switch (c) {
        case Red:
            return matrix.Color333(7,0,0);
        case Orange:
            return matrix.Color333(7,1,0);
        case Yellow:
            return matrix.Color333(7,3,0);
        case Green:
            return matrix.Color333(0,7,0);
        case LBlue:
            return matrix.Color333(0,7,7);
        case DBlue:
            return matrix.Color333(0,0,7);
        case Pink:
            return matrix.Color333(7,0,7);
    }
    return NULL;
}

void delay(int x){
    for(int i=0;i<x;i++){
        wait_us(10);
        matrix.updateDisplay();
    }
}

/*lose function
freezes screen to show collision
flashes score
goes to lose screen (red square)*/
void lose(){

    //clear screen for lose screen
    delay(10000);
    matrix.fillRect(0, 0, 32, 16, matrix.Color333(0, 0, 0));
```

```
    matrix.fillRect(14,6, 4,4, matrix.Color333(3,0,0));
    while(1) matrix.updateDisplay();
}

int main(){
    Timer timer;
    timer.start();
    int fallTime = 100; //fall speed in milliseconds
    int pastTime = timer.read_ms();
    Color colors[7] = {Red, Orange, Yellow, Green, LBlue, DBlue, Pink};
    matrix.fillScreen(matrix.Color333(0,0,0));
    srand(time(0));
    Tetromino t = Tetromino(colors[rand()%7]);
    t.draw();
    while (1) {
        int currTime = timer.read_ms();
        if (currTime - pastTime > fallTime) {
            pastTime = currTime;
            if(!t.fall()) {
                t = Tetromino(colors[rand()%7]);
                if(t.conflicts()) {
                    break;
                }
                matrix.fillScreen(matrix.Color333(0,0,0));

            } else {
                matrix.fillScreen(matrix.Color333(0,0,0));
                t.draw();
            }
            //draw board
            for(int i = 0; i < 32; i ++) {
                for(int j = 0; j < 16; j ++) {
                    //if(&(board[i][j]) != NULL) {
                        board[i][j].draw();
                    //}
                }
            }
        } else {
            delay(1);
        }
        if(button == 0) {
            t.shift(-1);
            delay(500);
            matrix.fillScreen(matrix.Color333(0,0,0));
            t.draw();
            //draw board
            for(int i = 0; i < 32; i ++) {
                for(int j = 0; j < 16; j ++) {
                    //if(&(board[i][j]) != NULL) {
                        board[i][j].draw();
                    //}
                }
```

```
            }
        }
        if(button2 == 0) {
            t.shift(1);
            delay(500);
            matrix.fillScreen(matrix.Color333(0,0,0));
            t.draw();
            //draw board
            for(int i = 0; i < 32; i ++) {
                for(int j = 0; j < 16; j ++) {
                    //if(&(board[i][j]) != NULL) {
                        board[i][j].draw();
                    //}
                }
            }
        }
    }
    lose();
}
```

References

We used two very helpful libraries in our code:

The mbedlibraryprovidedfunctionswithGPIO that made programming our board much easier. Additionally, it had helpful features such as a timer and analog control.

https://os.mbed.com/users/mbed_official/code/mbed/

The other library was hardware specific to the 16x32 matrix panel we used. The library provided functions to update the display, draw pixels, draw shapes, and draw characters. There were a few edits that had to be made to the library to make it compatible with the k64f and are described in our peripheral document.

https://os.mbed.com/teams/EIC_mbed/code/RGB_matrix_Panel/

With both of these libraries, we made use of the MBED online compiler to set up the librariesonlineandthenexportintomicrovision.

This tutorial also did a good job introducing the hardware and what each pin does. We based our peripheral document hardware section on this page

https://learn.adafruit.com/32x16-32x32-rgb-led-matrix

Peripherals

The main peripheral used in the project was the 32x16 LED matrix produced by Adafruit.

We wrote a comprehensive guide to get the hardware setup and software started found here:

Medium 16x32 RGB LED matrix panel